

BridgeControl.Tests.csproj

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net10.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
    <IsPackable>false</IsPackable>
    <IsTestProject>true</IsTestProject>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Mvc.Testing" Version="10.0.12" />
    <PackageReference Include="Jint" Version="4.16.2" />
    <PackageReference Include="Microsoft.NET.Test.Sdk" Version="17.14.1" />
    <PackageReference Include="xunit" Version="2.9.3" />
    <PackageReference Include="xunit.runner.visualstudio" Version="3.1.4" PrivateAssets="all" />
  </ItemGroup>
  <ItemGroup>
    <Using Include="Xunit" />
    <Content Include="Fixtures\ps1Widget.schema.json" CopyToOutputDirectory="PreserveNewest" />
    <Content Include="..\Documentation\Specifications\BridgeControl_API_Documentation_Final.docx"
Link="Acceptance\api.docx" CopyToOutputDirectory="PreserveNewest" />
    <Content Include="..\Documentation\Decisions\0031-requirement-api-map.json"
Link="Acceptance\requirements.json" CopyToOutputDirectory="PreserveNewest" />
    <Content Include="..\Documentation\Acceptance\requirements-evidence.json"
Link="Acceptance\evidence.json" CopyToOutputDirectory="PreserveNewest" />
    <ProjectReference Include="..\BridgeControl\BridgeControl.csproj" />
  </ItemGroup>
</Project>
```

Fixtures/ps1Widget.schema.json

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "ps1Widget configuration",
  "description": "Draft 2020-12 authoring schema for ps1Widget config.json. Properties are optional; use canonical property and enum casing. Unknown properties are rejected.",
  "type": "object",
  "additionalProperties": false,
  "properties": {
    "$schema": {
      "type": "string",
      "description": "Optional editor schema association."
    },
    "ScriptSources": {
      "type": "array",
      "items": {
        "$ref": "#/$defs/scriptSource"
      }
    },
    "ProjectDirectories": {
      "type": "array",
      "items": {
        "$ref": "#/$defs/path"
      }
    },
    "deprecated": true,
    "description": "Legacy project-source configuration. Prefer ScriptSources."
  }
}
```

```

},
"ProjectDirectoryDisplayNames": {
  "type": "object",
  "additionalProperties": {
    "type": "string"
  },
  "deprecated": true,
  "description": "Legacy project-source configuration. Prefer ScriptSources."
},
"ProjectDirectoryExtensions": {
  "type": "object",
  "additionalProperties": {
    "$ref": "#/$defs/extensions"
  },
  "deprecated": true,
  "description": "Legacy project-source configuration. Prefer ScriptSources."
},
"ProjectSourceOptionsByPath": {
  "type": "object",
  "additionalProperties": {
    "$ref": "#/$defs/sourceOptions"
  },
  "deprecated": true,
  "description": "Legacy project-source configuration. Prefer ScriptSources."
},
"OpenDirectories": {
  "type": "array",
  "items": {
    "$ref": "#/$defs/path"
  }
},
"OpenDirectoryDisplayNames": {
  "type": "object",
  "additionalProperties": {
    "type": "string"
  }
},
"DisabledOpenDirectories": {
  "type": "array",
  "items": {
    "$ref": "#/$defs/path"
  }
},
"WebsiteLinks": {
  "$ref": "#/$defs/websiteLinks"
},
"KeyboardNavigation": {
  "$ref": "#/$defs/keyboardNavigation"
},
"CollapsedSections": {
  "type": "array",
  "items": {
    "type": "string"
  }
},
"StartWithWindows": {
  "type": "boolean"
},
"RetractFabs": {
  "type": "boolean"
},
"FabRetractionPercent": {

```

```

        "type": "integer",
        "minimum": 20,
        "maximum": 80,
        "default": 80
    },
    "FabRetractionDurationMilliseconds": {
        "type": "integer",
        "minimum": 0,
        "maximum": 2000,
        "default": 200
    },
    "FabRetractionDelayMilliseconds": {
        "type": "integer",
        "minimum": 0,
        "maximum": 5000,
        "default": 500
    },
    "Position": {
        "$ref": "#/$defs/position"
    }
},
"$defs": {
    "path": {
        "type": "string",
        "pattern": "\\S",
        "description": "File or directory path; supports environment variables, * and ** wildcards,
and {UserStartup}, {CommonStartup}, {AppDirectory}, {UserConfigDirectory} tokens. Paths need not
exist during validation."
    },
    "extensions": {
        "type": "array",
        "items": {
            "type": "string",
            "pattern": "^(?:\\*?\\.|)?[\\.\\.\\s\\\\\\\\][^\\\\s\\\\\\\\]*$",
            "description": "Custom extensions are allowed, e.g. .ps1, ps1 or *.ps1."
        }
    },
    "sourceOptions": {
        "type": "object",
        "additionalProperties": false,
        "properties": {
            "IsEnabled": {
                "type": "boolean",
                "default": true
            },
            "UseRecursiveWildcard": {
                "type": "boolean",
                "default": false
            },
            "UseWildcard": {
                "type": "boolean",
                "default": false
            },
            "IncludeContainingFolderName": {
                "type": "boolean",
                "default": false
            }
        }
    },
    "scriptSource": {
        "type": "object",
        "additionalProperties": false,

```

```

    "properties": {
      "Path": {
        "$ref": "#/$defs/path"
      },
      "DisplayName": {
        "type": "string"
      },
      "Extensions": {
        "$ref": "#/$defs/extensions"
      },
      "Options": {
        "$ref": "#/$defs/sourceOptions"
      }
    },
    "required": [
      "Path"
    ]
  },
  "offset": {
    "anyOf": [
      {
        "type": "number"
      },
      {
        "type": "string",
        "pattern": "^-?(?:0|[1-9][0-9]*) (?:\\. [0-9]+)?(?:[eE] [+]?[0-9]+)?$"
      },
      {
        "type": "null"
      }
    ],
    "description": "Offset in device-independent pixels. Null selects the automatic position.  
Numeric strings are supported for compatibility."
  },
  "position": {
    "type": "object",
    "additionalProperties": false,
    "properties": {
      "Offset": {
        "$ref": "#/$defs/offset"
      },
      "Snap": {
        "enum": [
          "Left",
          "Right",
          "Top",
          "Bottom"
        ]
      },
      "Scroll": {
        "$ref": "#/$defs/offset",
        "deprecated": true,
        "description": "Legacy alias for Offset in full settings only."
      }
    }
  },
  "websiteNode": {
    "type": "object",
    "additionalProperties": false,
    "properties": {
      "Id": {
        "type": "string",

```

```

    "description": "Stable identifier; generated when omitted."
  },
  "Kind": {
    "enum": [
      "Folder",
      "Link"
    ]
  },
  "Name": {
    "type": "string",
    "pattern": "\\S"
  },
  "Url": {
    "type": "string"
  },
  "BrowserExecutablePath": {
    "type": "string",
    "description": "Empty inherits the parent browser or system default."
  },
  "Children": {
    "type": "array",
    "items": {
      "$ref": "#/$defs/websiteNode"
    }
  }
},
"required": [
  "Kind",
  "Name"
],
"allOf": [
  {
    "if": {
      "properties": {
        "Kind": {
          "const": "Link"
        }
      }
    },
    "required": [
      "Kind"
    ]
  },
  "then": {
    "required": [
      "Url"
    ],
    "properties": {
      "Url": {
        "type": "string",
        "format": "uri",
        "pattern": "^([hH]([tT]([pP]([sS]?://([^\s/?#]+)(?:[/?#][^\s]*)?)?$"
      },
      "Children": {
        "maxItems": 0
      }
    }
  },
  "else": {
    "properties": {
      "Url": {
        "const": ""
      }
    }
  }
]

```

```

    }
  }
}
]
},
"websiteLinks": {
  "type": "object",
  "additionalProperties": false,
  "properties": {
    "DefaultBrowserExecutablePath": {
      "type": "string",
      "description": "Empty selects the system default browser."
    },
    "QuickAddShortcut": {
      "$ref": "#/$defs/shortcut"
    },
    "Items": {
      "type": "array",
      "items": {
        "$ref": "#/$defs/websiteNode"
      }
    }
  }
},
"key": {
  "type": "string",
  "enum": [
    "Cancel",
    "Back",
    "Tab",
    "LineFeed",
    "Clear",
    "Return",
    "Enter",
    "Pause",
    "Capital",
    "CapsLock",
    "KanaMode",
    "HangulMode",
    "JunjaMode",
    "FinalMode",
    "HanjaMode",
    "KanjiMode",
    "Escape",
    "ImeConvert",
    "ImeNonConvert",
    "ImeAccept",
    "ImeModeChange",
    "Space",
    "Prior",
    "PageUp",
    "Next",
    "PageDown",
    "End",
    "Home",
    "Left",
    "Up",
    "Right",
    "Down",
    "Select",
    "Print",
    "Execute",

```

```
"Snapshot",
"PrintScreen",
"Insert",
"Delete",
"Help",
"D0",
"D1",
"D2",
"D3",
"D4",
"D5",
"D6",
"D7",
"D8",
"D9",
"A",
"B",
"C",
"D",
"E",
"F",
"G",
"H",
"I",
"J",
"K",
"L",
"M",
"N",
"O",
"P",
"Q",
"R",
"S",
"T",
"U",
"V",
"W",
"X",
"Y",
"Z",
"Apps",
"Sleep",
"NumPad0",
"NumPad1",
"NumPad2",
"NumPad3",
"NumPad4",
"NumPad5",
"NumPad6",
"NumPad7",
"NumPad8",
"NumPad9",
"Multiply",
"Add",
"Separator",
"Subtract",
"Decimal",
"Divide",
"F1",
"F2",
"F3",
```

```
"F4",
"F5",
"F6",
"F7",
"F8",
"F9",
"F10",
"F11",
"F12",
"F13",
"F14",
"F15",
"F16",
"F17",
"F18",
"F19",
"F20",
"F21",
"F22",
"F23",
"F24",
"NumLock",
"Scroll",
"BrowserBack",
"BrowserForward",
"BrowserRefresh",
"BrowserStop",
"BrowserSearch",
"BrowserFavorites",
"BrowserHome",
"VolumeMute",
"VolumeDown",
"VolumeUp",
"MediaNextTrack",
"MediaPreviousTrack",
"MediaStop",
"MediaPlayPause",
"LaunchMail",
"SelectMedia",
"LaunchApplication1",
"LaunchApplication2",
"Oem1",
"OemSemicolon",
"OemPlus",
"OemComma",
"OemMinus",
"OemPeriod",
"Oem2",
"OemQuestion",
"Oem3",
"OemTilde",
"AbntC1",
"AbntC2",
"Oem4",
"OemOpenBrackets",
"Oem5",
"OemPipe",
"Oem6",
"OemCloseBrackets",
"Oem7",
"OemQuotes",
"Oem8",
```

```

        "Oem102",
        "OemBackslash",
        "ImeProcessed",
        "System",
        "OemAttn",
        "DbeAlphanumeric",
        "OemFinish",
        "DbeKatakana",
        "OemCopy",
        "DbeHiragana",
        "OemAuto",
        "DbeSbcsChar",
        "OemEnlw",
        "DbeDbcsChar",
        "OemBackTab",
        "DbeRoman",
        "Attn",
        "DbeNoRoman",
        "CrSel",
        "DbeEnterWordRegisterMode",
        "ExSel",
        "DbeEnterImeConfigureMode",
        "EraseEof",
        "DbeFlushString",
        "Play",
        "DbeCodeInput",
        "Zoom",
        "DbeNoCodeInput",
        "NoName",
        "DbeDetermineString",
        "Pal",
        "DbeEnterDialogConversionMode",
        "OemClear",
        "DeadCharProcessed"
    ]
},
"shortcut": {
    "type": "string",
    "pattern":
"^(?:(:?Ctrl|Control|Alt|Shift|Win|Windows)\\s+)(?:Cancel|Back|Tab|LineFeed|Clear|Return|Enter|Pa
use|Capital|CapsLock|KanaMode|HangulMode|JunjaMode|FinalMode|HanjaMode|KanjiMode|Escape|ImeConver
t|ImeNonConvert|ImeAccept|ImeModeChange|Space|Prior|PageUp|Next|PageDown|End|Home|Left|Up|Right|D
own|Select|Print|Execute|Snapshot|PrintScreen|Insert|Delete|Help|D0|D1|D2|D3|D4|D5|D6|D7|D8|D9|A|
B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|Apps|Sleep|NumPad0|NumPad1|NumPad2|NumPad3|NumP
ad4|NumPad5|NumPad6|NumPad7|NumPad8|NumPad9|Multiply|Add|Separator|Subtract|Decimal|Divide|F1|F2|
F3|F4|F5|F6|F7|F8|F9|F10|F11|F12|F13|F14|F15|F16|F17|F18|F19|F20|F21|F22|F23|F24|NumLock|Scroll|B
rowserBack|BrowserForward|BrowserRefresh|BrowserStop|BrowserSearch|BrowserFavorites|BrowserHome|V
olumeMute|VolumeDown|VolumeUp|MediaNextTrack|MediaPreviousTrack|MediaStop|MediaPlayPause|LaunchMa
il|SelectMedia|LaunchApplication1|LaunchApplication2|Oem1|OemSemicolon|OemPlus|OemComma|OemMinus|
OemPeriod|Oem2|OemQuestion|Oem3|OemTilde|AbntC1|AbntC2|Oem4|OemOpenBrackets|Oem5|OemPipe|Oem6|Oem
CloseBrackets|Oem7|OemQuotes|Oem8|Oem102|OemBackslash|ImeProcessed|System|OemAttn|DbeAlphanumeric
|OemFinish|DbeKatakana|OemCopy|DbeHiragana|OemAuto|DbeSbcsChar|OemEnlw|DbeDbcsChar|OemBackTab|Dbe
Roman|Attn|DbeNoRoman|CrSel|DbeEnterWordRegisterMode|ExSel|DbeEnterImeConfigureMode|EraseEof|DbeF
lushString|Play|DbeCodeInput|Zoom|DbeNoCodeInput|NoName|DbeDetermineString|Pal|DbeEnterDialogConv
ersionMode|OemClear|DeadCharProcessed)$",
    "description": "Canonical shortcut: one or more modifiers followed by a WPF key name, e.g.
Ctrl+Alt+L. Runtime also accepts case-insensitive names and whitespace."
},
"keyboardNavigation": {
    "type": "object",
    "additionalProperties": false,
    "properties": {

```

```

        "ActivationShortcut": {
            "$ref": "#/$defs/shortcut"
        },
        "UpKey": {
            "$ref": "#/$defs/key"
        },
        "DownKey": {
            "$ref": "#/$defs/key"
        },
        "LeftKey": {
            "$ref": "#/$defs/key"
        },
        "RightKey": {
            "$ref": "#/$defs/key"
        },
        "ActivateKey": {
            "$ref": "#/$defs/key"
        }
    },
    "description": "Navigation keys must resolve to five distinct keys, including defaults for
omitted properties. This cross-property rule is checked by the application."
}
}
}

```

Infrastructure/AdjustableTimeProvider.cs

```

namespace BridgeControl.Tests.Infrastructure;

/// <summary>
/// Supplies a controllable clock for tests involving expiry and ordered events.
/// </summary>
public sealed class AdjustableTimeProvider : TimeProvider
{
    public TimeSpan Offset { get; set; }

    /// <summary>
    /// Returns the current simulated test time.
    /// </summary>
    public override DateTimeOffset GetUtcNow() => DateTimeOffset.UtcNow + Offset;
}

```

Infrastructure/AgentTest.cs

```

using System.Net;
using BridgeControl.DTOS.AgentDTOS.AgentContracts;

namespace BridgeControl.Tests.Infrastructure;

/// <summary>
/// Provides shared setup and operations for agent tests.
/// </summary>
public abstract class AgentTest : ConfigurationTest
{
    /// <summary>
    /// Requests a test agent enrollment credential.
    /// </summary>
    protected async Task<string> EnrollAsync(Guid customer)
    {
    }
}

```

```

        {
            using var response = await SendAsync(HttpMethod.Post, "/api/v1/agent-enrollments",
new { CustomerId = customer });
            Assert.Equal(HttpStatusCode.Created, response.StatusCode);
            Assert.True(response.Headers.CacheControl?.NoStore);
            return (await ReadAsync(response)).GetProperty("enrollmentCode").GetString()!;
        }
        /// <summary>
        /// Registers a test agent using an enrollment credential.
        /// </summary>
        protected Task<HttpResponseMessage> RegisterAsync(string code) =>
SendAsync(HttpMethod.Post, "/api/v1/agent/register",
            new { EnrollmentCode = code, MachineName = " Test machine ", AgentVersion = "1.0"
});
        /// <summary>
        /// Creates an enrolled agent fixture for the test scenario.
        /// </summary>
        protected async Task<AgentRegistrationResponse> AgentAsync(Guid customer)
        {
            using var response = await RegisterAsync(await EnrollAsync(customer));
            Assert.Equal(HttpStatusCode.Created, response.StatusCode);
            var body = await ReadAsync(response);
            Assert.Equal(new[] { "agentId", "credential", "expiresAt" },
body.EnumerateObject().Select(x => x.Name).Order());
            return new(body.GetProperty("agentId").GetGuid(),
body.GetProperty("credential").GetString()!, body.GetProperty("expiresAt").GetDateTimeOffset());
        }
        /// <summary>
        /// Obtains an access token for the test agent.
        /// </summary>
        protected async Task<string> TokenAsync(string credential)
        {
            using var response = await SendAsync(HttpMethod.Post, "/api/v1/agent/auth/token",
new { Credential = credential });
            Assert.Equal(HttpStatusCode.OK, response.StatusCode);
            return (await ReadAsync(response)).GetProperty("accessToken").GetString()!;
        }
        /// <summary>
        /// Creates a product for the test scenario.
        /// </summary>
        protected async Task<Guid> ProductAsync()
        {
            using var response = await SendAsync(HttpMethod.Post, "/api/v1/products", new {
Name = "Product" }); Assert.Equal(HttpStatusCode.Created, response.StatusCode);
            return (await ReadAsync(response)).GetProperty("id").GetGuid();
        }
    }
}

```

Infrastructure/ApiFactory.cs

```

using System.Security.Cryptography;
using BridgeControl.Data;
using BridgeControl.Options;
using BridgeControl.Services.Platform;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.AspNetCore.Mvc.Testing;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Infrastructure;
using Microsoft.Extensions.Configuration;

```

```

using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.DependencyInjection.Extensions;
using Microsoft.Extensions.Hosting;
using Serilog;

namespace BridgeControl.Tests.Infrastructure;

// The public API runs in memory. All configuration is local to this factory, including a
disposable signing key.
/// <summary>
/// Hosts the API with isolated test configuration and database dependencies.
/// </summary>
public sealed class ApiFactory(string connection, string environment = "Development", int
loginLimit = 20, bool configurationWorker = false, string? archiveRoot = null, bool stateWorker =
false, Action<IServiceCollection>? configure = null)
    : WebApplicationFactory<Program>
{
    private readonly HttpAcceptanceEvidence acceptanceEvidence = new();
    public AdjustableTimeProvider Clock { get; } = new();
    public string SigningKey { get; } =
Convert.ToBase64String(RandomNumberGenerator.GetBytes(32));
    public string TenantSigningKey { get; } =
Convert.ToBase64String(RandomNumberGenerator.GetBytes(32));
    public string AgentSigningKey { get; } =
Convert.ToBase64String(RandomNumberGenerator.GetBytes(32));
    public string PrivateRoot { get; } = Path.Combine(Path.GetTempPath(),
"BridgeControl_Tests_" + Guid.NewGuid().ToString("N"));
    public string ArchiveRoot => archiveRoot ?? Path.Combine(PrivateRoot, "archives");
    public string InvitationDirectory => Path.Combine(PrivateRoot, "invitations");

    /// <summary>
    /// Creates and initializes the test API host.
    /// </summary>
    protected override IHost CreateHost(IHostBuilder builder)
    {
        builder.ConfigureHostConfiguration(configuration =>
configuration.AddInMemoryCollection(new Dictionary<string, string?>
{
            ["ConnectionStrings:PlatformDatabase"] = environment == "Development" ?
connection : "Server=sql.example.test;Database=Unused;Integrated
Security=True;Encrypt=True;TrustServerCertificate=False",
            ["ConnectionStrings:VendorServer"] = environment == "Development" ?
connection : "Server=sql.example.test;Database=Unused;Integrated
Security=True;Encrypt=True;TrustServerCertificate=False"
        }));
        return base.CreateHost(builder);
    }

    /// <summary>
    /// Overrides API configuration and dependencies for the test host.
    /// </summary>
    protected override void ConfigureWebHost(IWebHostBuilder builder)
    {
        builder.UseEnvironment(environment);
        builder.ConfigureServices(services =>
        {
            services.AddSingleton<IStartupFilter>(acceptanceEvidence);
            // In-memory HTTP fixtures intentionally coexist with offline recovery
commands against disposable databases.
            foreach (ServiceDescriptor descriptor in services.Where(x => x.ServiceType
== typeof(IHostedService) && x.ImplementationType == typeof(DatabaseHostLeaseService)).ToArray())
            { services.Remove(descriptor); }
        }
    }
}

```

```

        services.PostConfigure<DatabaseOperationsOptions>(options =>
        {
            options.BackupRoot = Path.Combine(PrivateRoot, "backups");
options.ActiveKeyId = "test";
            options.EncryptionKeys = new() { ["test"] = SigningKey };
        });
        services.AddSingleton<TimeProvider>(Clock);
        services.PostConfigure<HostingProfileOptions>(options => {
options.PublicOrigin = "https://localhost"; options.DataProtectionKeyDirectory =
Path.Combine(PrivateRoot, "keys"); options.LogDirectory = Path.Combine(PrivateRoot, "logs"); });
        services.PostConfigure<OperationalHistoryOptions>(options =>
options.WorkerEnabled = false);
        services.RemoveAll<DbContextOptions<PlatformDbContext>>();
        services.RemoveAll<IDbContextOptionsConfiguration<PlatformDbContext>>();
        services.AddDbContext<PlatformDbContext>(options =>
options.UseSqlServer(connection));
        services.PostConfigure<PlatformAuthenticationOptions>(options =>
        {
            options.SigningKey = SigningKey;
            options.LoginRequestsPerMinute = loginLimit;
        });
        services.AddSerilog(new
LoggerConfiguration().MinimumLevel.Fatal().CreateLogger(), dispose: true);
        services.PostConfigure<TenantAuthenticationOptions>(options => {
options.SigningKey = TenantSigningKey; options.LoginRequestsPerMinute = loginLimit; });
        services.PostConfigure<AgentAuthenticationOptions>(options =>
options.SigningKey = AgentSigningKey);
        services.PostConfigure<EffectiveConfigurationOptions>(options => {
options.WorkerEnabled = configurationWorker; options.PollSeconds = 1; });
        services.PostConfigure<StateReconciliationOptions>(options => {
options.WorkerEnabled = stateWorker; options.PollSeconds = 1; });
        services.AddSingleton<IDataProtectionProvider>(new
EphemeralDataProtectionProvider());
        services.PostConfigure<VendorProvisioningOptions>(options =>
        {
            options.StorageRoot = ArchiveRoot;
            options.InvitationDirectory = InvitationDirectory;
        });
        configure?.Invoke(services);
    });
}

/// <summary>
/// Creates an HTTPS test client with the requested authentication behavior.
/// </summary>
public HttpClient CreateApiClient() => CreateClient(new()
{
    BaseAddress = new Uri("https://localhost"),
    AllowAutoRedirect = false,
    HandleCookies = false
});

/// <summary>
/// Asynchronously releases the resources owned by this instance.
/// </summary>
public override async ValueTask DisposeAsync()
{
    await base.DisposeAsync();
    await acceptanceEvidence.SaveAsync();
    string root = Path.GetFullPath(PrivateRoot);
    string parent = Path.GetFullPath(Path.GetTempPath());

```

```

        if (Path.GetDirectoryName(root)?.TrimEnd(Path.DirectorySeparatorChar) !=
parent.TrimEnd(Path.DirectorySeparatorChar)
        || !Path.GetFileName(root).StartsWith("BridgeControl_Tests_",
StringComparison.Ordinal))
        {
            throw new InvalidOperationException("Refusing to remove a directory outside
this test's temporary storage.");
        }
        if (Directory.Exists(root)) { Directory.Delete(root, recursive: true); }
    }
}

```

Infrastructure/ConfigurationTest.cs

```

using System.Net;
using System.Net.Http.Headers;
using System.Net.Http.Json;
using System.Text;
using System.Text.Json;
using BridgeControl.Models.Vendor;
using Microsoft.EntityFrameworkCore;

namespace BridgeControl.Tests.Infrastructure;

/// <summary>
/// Provides shared setup and operations for configuration tests.
/// </summary>
public abstract class ConfigurationTest : VendorTest
{
    protected Guid VendorId;
    protected string AdminToken = "";
    protected const string Password = "configuration-test-password";
    protected const string Schema =
        """{"$schema":"https://json-schema.org/draft/2020-12/schema","type":"object","properties":{"port":
{"type":"integer","minimum":1,"maximum":65535},"label":{"type":"string"},"tags":{"type":"array",
"items":{"type":["string","null"]}}},"additionalProperties":false}""";
    protected string ArchiveRoot => Path.Combine(Factory.ArchiveRoot, VendorId.ToString("N"));
    /// <summary>
    /// Creates the accounts and resources required by the test scenario.
    /// </summary>
    protected async Task PrepareAsync()
    {
        await SignInAsync(); VendorId = await CreateVendorAsync(); string invitation =
await InviteAsync(VendorId);
        using var response = await Client.PostAsJsonAsync("/api/v1/auth/activate", new {
Token = invitation, Password }); Assert.Equal(HttpStatusCode.NoContent, response.StatusCode);
        AdminToken = await LoginAsync("invited@example.test");
    }
    /// <summary>
    /// Authenticates the test user and returns the resulting session data.
    /// </summary>
    protected async Task<string> LoginAsync(string email)
    {
        using var login = await Client.PostAsJsonAsync("/api/v1/auth/login", new {
VendorName = "Test vendor", Email = email, Password });
        Assert.Equal(HttpStatusCode.OK, login.StatusCode); return (await
ReadAsync(login)).GetProperty("accessToken").GetString()!;
    }
    /// <summary>
    /// Sends the test HTTP request with the supplied method, credentials, and payload.

```

```

    /// </summary>
    protected async Task<HttpResponseBodyMessage> SendAsync(HttpMethod method, string path,
object? body = null, string? token = null)
    {
        using var request = new HttpRequestMessage(method, path);
request.Headers.Authorization = new AuthenticationHeaderValue("Bearer", token ?? AdminToken);
        if (body is not null) { request.Content = JsonContent.Create(body); }
        return await Client.SendAsync(request);
    }
    /// <summary>
    /// Creates the product catalog used by configuration tests.
    /// </summary>
    protected async Task<(Guid Product, Guid Version)> CatalogAsync(bool installer = false,
string schema = Schema)
    {
        using var product = await SendAsync(HttpMethod.Post, "/api/v1/products", new {
Name = "Product" }); Assert.Equal(HttpStatusCode.Created, product.StatusCode);
        Guid productId = (await ReadAsync(product)).GetProperty("id").GetGuid(); Guid
versionId = await VersionAsync(productId, "1", schema);
        if (installer) { using var upload = await UploadAsync(productId, versionId,
"installer", MsiFixture.Create(Factory.PrivateRoot)); Assert.Equal(HttpStatusCode.OK,
upload.StatusCode); }
        return (productId, versionId);
    }
    /// <summary>
    /// Creates a product version for the test scenario.
    /// </summary>
    protected async Task<Guid> VersionAsync(Guid product, string label, string schema =
Schema)
    {
        using var version = await SendAsync(HttpMethod.Post,
$/api/v1/products/{product}/versions", new { Version = label });
Assert.Equal(HttpStatusCode.Created, version.StatusCode);
        Guid id = (await ReadAsync(version)).GetProperty("id").GetGuid();
        using var upload = await UploadAsync(product, id, "schema",
Encoding.UTF8.GetBytes(schema)); Assert.Equal(HttpStatusCode.OK, upload.StatusCode); return id;
    }
    /// <summary>
    /// Uploads the supplied artifact through the test API.
    /// </summary>
    protected async Task<HttpResponseBodyMessage> UploadAsync(Guid product, Guid version, string
kind, byte[] bytes)
    {
        using var request = new HttpRequestMessage(HttpMethod.Put,
$/api/v1/products/{product}/versions/{version}/{kind}");
request.Headers.Authorization = new AuthenticationHeaderValue("Bearer",
AdminToken);
        var body = new MultipartFormDataContent(); body.Add(new ByteArrayContent(bytes),
"file", kind == "installer" ? "setup.msi" : "schema.json");
        request.Content = body; return await Client.SendAsync(request);
    }
    /// <summary>
    /// Writes configuration values through the test API.
    /// </summary>
    protected Task<HttpResponseBodyMessage> SetAsync(string path, Guid version, object values,
string? token = null)
    => SendAsync(HttpMethod.Put, path, new { ProductVersionId = version, Values =
values }, token);
    /// <summary>
    /// Publishes the test product version through the API.
    /// </summary>
    protected Task<HttpResponseBodyMessage> PublishAsync(Guid product, Guid version)

```

```

=> SendAsync(HttpMethod.Post,
$/api/v1/products/{product}/versions/{version}/publish");
/// <summary>
/// Creates a customer for the test scenario.
/// </summary>
protected async Task<Guid> CustomerAsync(string name = "Customer")
{
    using var customer = await SendAsync(HttpMethod.Post, "/api/v1/customers", new {
Name = name }); Assert.Equal(HttpStatusCode.Created, customer.StatusCode);
    return (await ReadAsync(customer)).GetProperty("id").GetGuid();
}
/// <summary>
/// Creates an installation for the test scenario.
/// </summary>
protected async Task<Guid> InstallationAsync(Guid customer, Guid product)
{
    await using var db = await OpenVendorAsync(VendorId);
    Installation installation = new() { CustomerId = customer, ProductId = product,
Name = "Installation" }; db.Installations.Add(installation); await db.SaveChangesAsync(); return
installation.Id;
}
/// <summary>
/// Creates a user for the test scenario.
/// </summary>
protected async Task<(Guid Id, string Token)> UserAsync(string email, string roleName,
Guid? customer = null)
{
    await using var db = await OpenVendorAsync(VendorId); Guid role = await
db.Roles.Where(x => x.Name == roleName).Select(x => x.Id).SingleAsync();
    using var created = await SendAsync(HttpMethod.Post, "/api/v1/users", new { Email
= email, CustomerId = customer, RoleIds = new[] { role } }); Assert.Equal(HttpStatusCode.Created,
created.StatusCode);
    Guid id = (await ReadAsync(created)).GetProperty("id").GetGuid(); Guid activation
= await db.AccountActivations.Where(x => x.UserId == id && x.UsedAt == null && x.RevokedAt ==
null).Select(x => x.Id).SingleAsync();
    string message = await
File.ReadAllTextAsync(Path.Combine(Factory.InvitationDirectory, $"{activation:N}.eml"));
    string credential = message.Split('\n').Single(x => x.StartsWith("Activation
token: ", StringComparison.Ordinal))[18..].Trim();
    using var activated = await Client.PostAsJsonAsync("/api/v1/auth/activate", new {
Token = credential, Password }); Assert.Equal(HttpStatusCode.NoContent, activated.StatusCode);
    return (id, await LoginAsync(email));
}
}

```

Infrastructure/EffectiveConfigurationTest.cs

```

using BridgeControl.Services.Vendor;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;

namespace BridgeControl.Tests.Infrastructure;

/// <summary>
/// Provides shared setup and operations for effective configuration tests.
/// </summary>
public abstract class EffectiveConfigurationTest : ConfigurationTest
{
    /// <summary>
    /// Generates effective configuration for the test installation and version.

```

```

        /// </summary>
        protected Task<Guid> GenerateAsync(Guid installation, Guid version, bool selectTarget =
false)
        =>
        Services.GetRequiredService<EffectiveConfigurationGenerator>().GenerateAsync(VendorId,
installation, version, selectTarget);
        /// <summary>
        /// Processes queued effective-configuration work for the test vendor.
        /// </summary>
        protected async Task DrainAsync()
        {
            for (int i = 0; i < 100; i++)
            {
                await using var scope = Factory.Services.CreateAsyncScope();
                await
scope.ServiceProvider.GetRequiredService<EffectiveConfigurationProcessor>().ProcessVendorAsync(Ve
ndorId);

                await using var db = await OpenVendorAsync(VendorId);
                if (!await db.ConfigurationSourceChanges.AnyAsync(x => x.ProcessedAt ==
null))

                    && !await db.EffectiveConfigurationSignals.AnyAsync(x =>
x.ProcessedAt == null)

                    && !await db.EffectiveConfigurationWork.AnyAsync(x => x.Status ==
"Pending")) { return; }
            }
            Assert.Fail("Recalculation did not drain within the bounded test run.");
        }
    }
}

```

Infrastructure/FrozenApiContract.cs

```

using System.IO.Compression;
using System.Text.RegularExpressions;
using System.Xml.Linq;

namespace BridgeControl.Tests.Infrastructure;

/// <summary>Reads the original Word contract, independently of application metadata and
generated OpenAPI.</summary>
public static class FrozenApiContract
{
    /// <summary>
    /// Represents operation data.
    /// </summary>
    public sealed record Operation(string Method, string Path);
    public static string Path => System.IO.Path.Combine(AppContext.BaseDirectory,
"Acceptance", "api.docx");
    private static readonly XNamespace Word =
"http://schemas.openxmlformats.org/wordprocessingml/2006/main";
    /// <summary>
    /// Loads the frozen API contract used by acceptance tests.
    /// </summary>
    private static XDocument Read()
    { using ZipArchive archive = ZipFile.OpenRead(Path); using Stream stream =
archive.GetEntry("word/document.xml")!.Open(); return XDocument.Load(stream); }
    /// <summary>
    /// Reads the approved operations from the frozen API contract.
    /// </summary>
}

```

```

        public static Operation[] Operations() => Read().Descendants(Word + "tr").Select(row =>
row.Elements(Word + "tc").Select(cell => string.Concat(cell.Descendants(Word + "t").Select(x =>
x.Value))).ToArray())
        .Where(cells => cells.Length >= 2 && Regex.IsMatch(cells[0],
"^ (GET|POST|PUT|PATCH|DELETE)$") && cells[1].StartsWith("/api/v1/", StringComparison.Ordinal))
        .Select(cells => new Operation(cells[0],
cells[1].Split('?')[0])).Distinct().OrderBy(x => x.Path).ThenBy(x => x.Method).ToArray();
        /// <summary>
        /// Reads the approved DTO field sets from the frozen API contract.
        /// </summary>
        public static Dictionary<string, string[]> DtoFields()
        {
            string text = string.Join("\n", Read().Descendants(Word + "p").Select(p =>
string.Concat(p.Descendants(Word + "t").Select(x => x.Value))));
            return Regex.Matches(text, @"public sealed record (\w+) (?<T>)?\((.*?)\);",
RegexOptions.Singleline)
                .ToDictionary(match => match.Groups[1].Value, match =>
match.Groups[2].Value.Split(',').Select(field => field.Trim().Split(' ',
StringSplitOptions.RemoveEmptyEntries).Last())
                .Select(field => char.ToLowerInvariant(field[0]) +
field[1..])).Order().ToArray();
        }
    }
}

```

Infrastructure/HttpAcceptanceEvidence.cs

```

using System.Collections.Concurrent;
using System.Text.Json;
using System.Text.RegularExpressions;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Routing;

namespace BridgeControl.Tests.Infrastructure;

/// <summary>Optional safe test evidence: route templates and status counts only, never
credentials, IDs or payloads.</summary>
public sealed class HttpAcceptanceEvidence : IStartupFilter
{
    private readonly ConcurrentDictionary<(string Method, string Path, int Status), int>
counts = new();
    /// <summary>
    /// Configures where HTTP acceptance evidence is recorded.
    /// </summary>
    public Action<IApplicationBuilder> Configure(Action<IApplicationBuilder> next) => app =>
    {
        app.Use(async (context, continuation) =>
        {
            await continuation(context);
            if (context.GetEndpoint() is RouteEndpoint endpoint &&
endpoint.RoutePattern.RawText is string route)
            {
                string path = "/" + Regex.Replace(route.TrimStart('/'),
@"\{([^\:]+):[^\}]+\}", "{$1}");
                if (path.StartsWith("/api/v1/", StringComparison.Ordinal))
                { counts.AddOrUpdate((context.Request.Method, path,
context.Response.StatusCode), 1, (_, count) => count + 1); }
            }
        })
    }
}

```

```

        });
        next(app);
    };
    /// <summary>
    /// Saves the captured HTTP acceptance evidence for the test scenario.
    /// </summary>
    public async Task SaveAsync()
    {
        string? directory =
Environment.GetEnvironmentVariable("BRIDGECONTROL_ACCEPTANCE_RESULTS");
        if (string.IsNullOrEmpty(directory)) { return; }
        Directory.CreateDirectory(directory);
        await using FileStream output = new(Path.Combine(directory,
Guid.NewGuid().ToString("N") + ".json"), FileMode.CreateNew);
        await JsonSerializer.SerializeAsync(output, counts.OrderBy(x =>
x.Key.Path).ThenBy(x => x.Key.Method).ThenBy(x => x.Key.Status)
        .Select(x => new { method = x.Key.Method, path = x.Key.Path, status =
x.Key.Status, count = x.Value }));
    }
}

```

Infrastructure/InstallationStateTest.cs

```

using System.Net;
using BridgeControl.Services.Vendor;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;

namespace BridgeControl.Tests.Infrastructure;

/// <summary>
/// Provides shared setup and operations for installation state tests.
/// </summary>
public abstract class InstallationStateTest : EffectiveConfigurationTest
{
    /// <summary>
    /// Represents state fixture data.
    /// </summary>
    protected sealed record StateFixture(Guid Product, Guid Version, Guid Customer, Guid
Installation, Guid Agent, string Token, Guid Configuration, Guid ConfigurationArtifact, Guid
Installer, Guid License, string Hash);

    /// <summary>
    /// Creates an enrolled agent fixture for the test scenario.
    /// </summary>
    protected async Task<(Guid Agent, string Token)> AgentAsync(Guid customer)
    {
        using var enrollment = await SendAsync(HttpMethod.Post,
"/api/v1/agent-enrollments", new { CustomerId = customer }); Assert.Equal(HttpStatusCode.Created,
enrollment.StatusCode);

        using var registered = await SendAsync(HttpMethod.Post, "/api/v1/agent/register",
new { EnrollmentCode = (await ReadAsync(enrollment)).GetProperty("enrollmentCode").GetString(),
MachineName = "Machine", AgentVersion = "1" }); Assert.Equal(HttpStatusCode.Created,
registered.StatusCode);

        var registration = await ReadAsync(registered);
        using var login = await SendAsync(HttpMethod.Post, "/api/v1/agent/auth/token", new
{ Credential = registration.GetProperty("credential").GetString() });
        Assert.Equal(HttpStatusCode.OK, login.StatusCode);

        return (registration.GetProperty("agentId").GetGuid(), (await
ReadAsync(login)).GetProperty("accessToken").GetString());
    }
}

```

```

    /// <summary>
    /// Builds the installation-state fixture required by the test.
    /// </summary>
    protected async Task<StateFixture> StateAsync()
    {
        await PrepareAsync(); var catalog = await CatalogAsync(installer: true); Guid
customer = await CustomerAsync(); var agent = await AgentAsync(customer);
        using var registered = await SendAsync(HttpMethod.Post,
"/api/v1/agent/installations", new { ProductId = catalog.Product, Name = "Service" },
agent.Token); Assert.Equal(HttpStatusCode.Created, registered.StatusCode);
        Guid installation = (await ReadAsync(registered)).GetProperty("id").GetGuid();
        using var published = await PublishAsync(catalog.Product, catalog.Version);
Assert.Equal(HttpStatusCode.OK, published.StatusCode);
        Guid effective = await GenerateAsync(installation, catalog.Version);
        using var license = await SendAsync(HttpMethod.Post, "/api/v1/licenses", new {
CustomerId = customer, ProductId = catalog.Product, LicenseReference = "State test", ValidFrom =
DateTimeOffset.UtcNow.AddDays(-1), MaxInstallations = 10 }); Assert.Equal(HttpStatusCode.Created,
license.StatusCode);
        Guid licenseId = (await ReadAsync(license)).GetProperty("id").GetGuid();
        using var assigned = await SendAsync(HttpMethod.Put,
$"/api/v1/installations/{installation}/license", new { LicenseId = licenseId });
Assert.Equal(HttpStatusCode.NoContent, assigned.StatusCode);
        await using var db = await OpenVendorAsync(VendorId); var config = await
db.EffectiveConfigurationRevisions.Include(x => x.FileArtifact).SingleAsync(x => x.Id ==
effective);
        var version = await db.ProductVersions.SingleAsync(x => x.Id == catalog.Version);
        return new(catalog.Product, catalog.Version, customer, installation, agent.Agent,
agent.Token, effective, config.FileArtifactId, version.PublishedInstallerArtifactId!.Value,
licenseId, config.FileArtifact.Sha256);
    }
    /// <summary>
    /// Sets desired state for the test installation through the API.
    /// </summary>
    protected Task<HttpResponseMessage> SetDesiredAsync(StateFixture state, Guid? version =
null, Guid? configuration = null, string? token = null)
    => SendAsync(HttpMethod.Put,
$"/api/v1/installations/{state.Installation}/desired-state", new { ProductVersionId = version ??
state.Version, EffectiveConfigurationRevisionId = configuration ?? state.Configuration }, token);
    /// <summary>
    /// Submits installation-state evidence for the test agent.
    /// </summary>
    protected Task<HttpResponseMessage> ReportAsync(StateFixture state, string status,
DateTimeOffset at, Guid? version = null, Guid? configuration = null, string? hash = null)
    => SendAsync(HttpMethod.Put,
$"/api/v1/agent/installations/{state.Installation}/reported-state", new {
InstalledProductVersionId = version, EffectiveConfigurationRevisionId = configuration,
ConfigurationHash = hash, Status = status, ReportedAt = at }, state.Token);
    /// <summary>
    /// Processes pending installation-state reconciliation for the test vendor.
    /// </summary>
    protected async Task ReconcileAsync()
    {
        await using var scope = Factory.Services.CreateAsyncScope(); await
scope.ServiceProvider.GetRequiredService<StateReconciliationProcessor>().ProcessVendorAsync(Vendo
rId);
    }
}

```

Infrastructure/LicensingTest.cs

```

using System.Net;

namespace BridgeControl.Tests.Infrastructure;

/// <summary>
/// Provides shared setup and operations for licensing tests.
/// </summary>
public abstract class LicensingTest : AgentTest
{
    /// <summary>
    /// Assigns a test license to an installation.
    /// </summary>
    protected Task<HttpResponseMessage> AssignAsync(Guid installation, Guid license, string?
token = null)
    => SendAsync(HttpMethod.Put, $"/api/v1/installations/{installation}/license", new
{ LicenseId = license }, token);
    /// <summary>
    /// Creates a license with the test's requested terms.
    /// </summary>
    protected async Task<Guid> LicenseAsync(Guid customer, Guid product, int? capacity = null,
DateTimeOffset? from = null, DateTimeOffset? until = null)
    {
        using var response = await SendAsync(HttpMethod.Post, "/api/v1/licenses", new
        {
            CustomerId = customer,
            ProductId = product,
            LicenseReference = " Reference ",
            ValidFrom = from ?? DateTimeOffset.UtcNow.AddDays(-1),
            ValidUntil = until,
            MaxInstallations = capacity
        });
        Assert.Equal(HttpStatusCode.Created, response.StatusCode);
        var body = await ReadAsync(response); Assert.Equal("Reference",
body.GetProperty("licenseReference").GetString());
        Assert.Equal(new[] { "customerId", "id", "licenseReference", "maxInstallations",
"productId", "status", "validFrom", "validUntil" }, body.EnumerateObject().Select(x =>
x.Name).Order());
        return body.GetProperty("id").GetGuid();
    }
    /// <summary>
    /// Creates the managed installation and licensing fixture for a test.
    /// </summary>
    protected async Task<(Guid Installation, string AgentToken)> ManagedAsync(Guid customer,
Guid product, string name = "Service")
    {
        var agent = await AgentAsync(customer); string token = await
TokenAsync(agent.Credential);
        using var response = await SendAsync(HttpMethod.Post,
"/api/v1/agent/installations", new { ProductId = product, Name = name, InstalledVersion = "1" },
token);
        Assert.Equal(HttpStatusCode.Created, response.StatusCode); return ((await
ReadAsync(response)).GetProperty("id").GetGuid(), token);
    }
}

```

Infrastructure/MigrationTest.cs

```

using System.Net;
using System.Text.Json;
using BridgeControl.DTOs.Vendor;

```

```

using BridgeControl.Models.Vendor;
using BridgeControl.Services.Vendor;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;

namespace BridgeControl.Tests.Infrastructure;

/// <summary>
/// Provides shared setup and operations for migration tests.
/// </summary>
public abstract class MigrationTest : EffectiveConfigurationTest
{
    protected const string OpenSchema =
        """{"$schema":"https://json-schema.org/draft/2020-12/schema","type":"object"}""";
    /// <summary>
    /// Creates an authenticated tenant actor for a migration test.
    /// </summary>
    protected async Task<TenantActor> ActorAsync()
    {
        await using var db = await OpenVendorAsync(VendorId); var user = await
db.UserAccounts.SingleAsync(x => x.Email == "invited@example.test");
        var session = await db.AuthenticationSessions.Where(x => x.AccountId == user.Id &&
x.RevokedAt == null).OrderByDescending(x => x.CreatedAt).FirstAsync();
        return new(VendorId, user.Id, session.Id);
    }
    /// <summary>
    /// Creates a configuration migration definition for the test versions.
    /// </summary>
    protected async Task<Guid> DefinitionAsync(Guid product, Guid from, Guid to, params
MigrationRuleRequest[] rules)
    {
        using var response = await SendAsync(HttpMethod.Post,
$"/api/v1/products/{product}/configuration-migrations", new { FromProductVersionId = from,
ToProductVersionId = to, Rules = rules });
        Assert.Equal(HttpStatusCode.Created, response.StatusCode); return (await
ReadAsync(response)).GetProperty("id").GetGuid();
    }
    /// <summary>
    /// Creates a migration review for the test installation.
    /// </summary>
    protected async Task<ConfigurationMigrationReview> ReviewAsync(Guid installation, Guid
target)
    {
        await using var scope = Factory.Services.CreateAsyncScope(); return await
scope.ServiceProvider.GetRequiredService<ConfigurationMigrationService>().ReviewAsync(await
ActorAsync(), installation, target, default);
    }
    /// <summary>
    /// Applies the selected migration review in the test scenario.
    /// </summary>
    protected async Task<EffectiveConfigurationResponse>
ApplyAsync(ConfigurationMigrationReview review, params Guid[] approvals)
    {
        await using var scope = Factory.Services.CreateAsyncScope(); return await
scope.ServiceProvider.GetRequiredService<ConfigurationMigrationService>().ApplyReviewedAsync(wai
t ActorAsync(), review.InstallationId, review.TargetProductVersionId, review.Id, approvals,
default);
    }
    /// <summary>
    /// Deserializes the migration plan saved in the selected review.
    /// </summary>

```

```

        protected static ConfigurationMigrationPlan Plan(ConfigurationMigrationReview review) =>
        JsonSerializer.Deserialize<ConfigurationMigrationPlan>(review.PlanJson)!;
    }

```

Infrastructure/MsiFixture.cs

```

using System.Runtime.InteropServices;

namespace BridgeControl.Tests.Infrastructure;

/// <summary>A native MSI database fixture. It is created/read as data and never
installed.</summary>
internal static class MsiFixture
{
    /// <summary>
    /// Builds a minimal MSI package for installer validation tests.
    /// </summary>
    public static byte[] Create(string directory)
    {
        string path = Path.Combine(directory, Guid.NewGuid().ToString("N") + ".msi");
        uint database = 0, summary = 0;
        try
        {
            Check(MsiOpenDatabase(path, new IntPtr(3), out database));
            Execute(database, "CREATE TABLE `Property` (`Property` CHAR(72) NOT NULL,
`Value` CHAR(0) LOCALIZABLE PRIMARY KEY `Property`)");
            foreach (var (name, value) in new Dictionary<string, string>
            {
                ["ProductCode"] = Guid.NewGuid().ToString("B").ToUpperInvariant(),
                ["UpgradeCode"] = Guid.NewGuid().ToString("B").ToUpperInvariant(),
                ["ProductName"] = "BridgeControl test fixture",
                ["ProductVersion"] = "1.0.0",
                ["Manufacturer"] = "Tests",
                ["ProductLanguage"] = "1033"
            }) { Execute(database, $"INSERT INTO `Property` (`Property`, `Value`)
VALUES ('{name}', '{value}')"); }
            foreach (string definition in new[]
            {
                "`Directory` (`Directory` CHAR(72) NOT NULL, `Directory_Parent`
CHAR(72), `DefaultDir` CHAR(255) NOT NULL PRIMARY KEY `Directory`)",
                "`Component` (`Component` CHAR(72) NOT NULL, `ComponentId`
CHAR(38), `Directory_` CHAR(72) NOT NULL, `Attributes` SHORT NOT NULL, `Condition` CHAR(255),
`KeyPath` CHAR(72) PRIMARY KEY `Component`)",
                "`Feature` (`Feature` CHAR(38) NOT NULL, `Feature_Parent` CHAR(38),
`Title` CHAR(64), `Description` CHAR(255), `Display` SHORT, `Level` SHORT NOT NULL, `Directory_`
CHAR(72), `Attributes` SHORT NOT NULL PRIMARY KEY `Feature`)",
                "`FeatureComponents` (`Feature_` CHAR(38) NOT NULL, `Component_`
CHAR(72) NOT NULL PRIMARY KEY `Feature_`, `Component_`)",
                "`File` (`File` CHAR(72) NOT NULL, `Component_` CHAR(72) NOT NULL,
`FileName` CHAR(255) NOT NULL, `FileSize` LONG NOT NULL, `Version` CHAR(72), `Language` CHAR(20),
`Attributes` SHORT, `Sequence` SHORT NOT NULL PRIMARY KEY `File`)",
                "`Media` (`DiskId` SHORT NOT NULL, `LastSequence` SHORT NOT NULL,
`DiskPrompt` CHAR(64), `Cabinet` CHAR(255), `VolumeLabel` CHAR(32), `Source` CHAR(72) PRIMARY KEY
`DiskId`)",
                "`InstallExecuteSequence` (`Action` CHAR(72) NOT NULL, `Condition`
CHAR(255), `Sequence` SHORT PRIMARY KEY `Action`)"
            }) { Execute(database, "CREATE TABLE " + definition); }
            Check(MsiGetSummaryInformation(database, null, 4, out summary));
            Check(MsiSummaryInfoSetProperty(summary, 9, 30, 0, IntPtr.Zero,
Guid.NewGuid().ToString("B").ToUpperInvariant()));

```

```

        Check(MsiSummaryInfoSetProperty(summary, 7, 30, 0, IntPtr.Zero,
"Intel;1033"));
        Check(MsiSummaryInfoSetProperty(summary, 14, 3, 200, IntPtr.Zero, null));
        Check(MsiSummaryInfoPersist(summary)); Check(MsiDatabaseCommit(database));
    }
    finally { if (summary != 0) { MsiCloseHandle(summary); } if (database != 0) {
MsiCloseHandle(database); } }
    try { return File.ReadAllBytes(path); }
    finally { File.Delete(path); }
}
/// <summary>
/// Executes a SQL statement against the test MSI database.
/// </summary>
private static void Execute(uint database, string sql)
{
    uint view = 0;
    try { Check(MsiDatabaseOpenView(database, sql, out view));
Check(MsiViewExecute(view, 0)); }
    finally { if (view != 0) { MsiCloseHandle(view); } }
}
/// <summary>
/// Fails test fixture creation when a native MSI operation fails.
/// </summary>
private static void Check(uint result) => Assert.Equal(0U, result);
/// <summary>
/// Opens a Windows Installer database through the native MSI API.
/// </summary>
[DllImport("msi.dll", CharSet = CharSet.Unicode, EntryPoint = "MsiOpenDatabaseW")] private
static extern uint MsiOpenDatabase(string path, IntPtr persist, out uint database);
/// <summary>
/// Creates a native MSI database view for a SQL query.
/// </summary>
[DllImport("msi.dll", CharSet = CharSet.Unicode, EntryPoint = "MsiDatabaseOpenViewW")]
private static extern uint MsiDatabaseOpenView(uint database, string query, out uint view);
/// <summary>
/// Executes a native MSI database view with the supplied record.
/// </summary>
[DllImport("msi.dll")] private static extern uint MsiViewExecute(uint view, uint record);
/// <summary>
/// Commits changes to a native MSI database.
/// </summary>
[DllImport("msi.dll")] private static extern uint MsiDatabaseCommit(uint database);
/// <summary>
/// Releases a native Windows Installer handle.
/// </summary>
[DllImport("msi.dll")] private static extern uint MsiCloseHandle(uint handle);
/// <summary>
/// Opens the summary information stream of an MSI package.
/// </summary>
[DllImport("msi.dll", CharSet = CharSet.Unicode, EntryPoint =
"MsiGetSummaryInformationW")] private static extern uint MsiGetSummaryInformation(uint database,
string? path, uint update, out uint summary);
/// <summary>
/// Sets a property in an MSI summary information stream.
/// </summary>
[DllImport("msi.dll", CharSet = CharSet.Unicode, EntryPoint =
"MsiSummaryInfoSetPropertyW")] private static extern uint MsiSummaryInfoSetProperty(uint summary,
uint property, uint type, int value, IntPtr time, string? text);
/// <summary>
/// Persists changes to an MSI summary information stream.
/// </summary>
[DllImport("msi.dll")] private static extern uint MsiSummaryInfoPersist(uint summary);

```

```
}
```

Infrastructure/SqlServerTest.cs

```
using BridgeControl.Data;
using BridgeControl.Models.Platform;
using BridgeControl.Services.Platform;
using Microsoft.Data.SqlClient;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;

namespace BridgeControl.Tests.Infrastructure;

// xUnit creates a fresh instance per test, so each test gets its own database and application.
/// <summary>
/// Provides shared setup and operations for sql server tests.
/// </summary>
public abstract class SqlServerTest : IAsyncLifetime
{
    private readonly string databaseName = "BridgeControl_Test_" +
Guid.NewGuid().ToString("N");
    private AsyncServiceScope scope;
    protected ApiFactory Factory { get; private set; } = null!;
    protected HttpClient Client { get; private set; } = null!;
    protected PlatformDbContext Database { get; private set; } = null!;
    protected IServiceProvider Services => scope.ServiceProvider;

    /// <summary>
    /// Creates and initializes the isolated test databases.
    /// </summary>
    public async Task InitializeAsync()
    {
        string connection =
$"Server=(localdb)\\MSSQLLocalDB;Database={databaseName};Integrated
Security=True;TrustServerCertificate=True";
        Factory = new ApiFactory(connection);
        Client = Factory.CreateApiClient();
        scope = Factory.Services.CreateAsyncScope();
        Database = Services.GetRequiredService<PlatformDbContext>();
        CheckTestDatabase();
        await Database.Database.MigrateAsync();
    }

    /// <summary>
    /// Asynchronously releases the resources owned by this instance.
    /// </summary>
    public async Task DisposeAsync()
    {
        try
        {
            if (Database is not null)
            {
                CheckTestDatabase();
                List<SoftwareVendor> vendors = await
Database.SoftwareVendors.AsNoTracking().Include(vendor => vendor.Database)
                .Where(vendor => vendor.Database != null).ToListAsync();
                foreach (SoftwareVendor vendor in vendors)
                {
                    if (vendor.Database!.DatabaseName !=
VendorInfrastructure.DatabaseName(vendor.Id))

```

```

        {
            throw new InvalidOperationException("Refusing to
remove a database not generated for this test's vendor.");
        }
        await using VendorDbContext tenant =
Services.GetRequiredService<IVendorInfrastructure>().OpenDatabase(vendor);
        SqlConnectionStringBuilder connection =
new(tenant.Database.GetConnectionString());
        if (connection.DataSource != "(localdb)\\MSSQLLocalDB" ||
connection.InitialCatalog != VendorInfrastructure.DatabaseName(vendor.Id))
        {
            throw new InvalidOperationException("Refusing to
remove a vendor database outside the test LocalDB instance.");
        }
        await tenant.Database.EnsureDeletedAsync();
    }
    await Database.Database.EnsureDeletedAsync();
}
}
finally
{
    if (Database is not null) { await scope.DisposeAsync(); }
    Client?.Dispose();
    if (Factory is not null) { await Factory.DisposeAsync(); }
}
}

/// <summary>
/// Rejects database targets that do not satisfy the test isolation requirements.
/// </summary>
private void CheckTestDatabase()
{
    SqlConnectionStringBuilder connection =
new(Database.Database.GetConnectionString());
    if (connection.InitialCatalog != databaseName || connection.DataSource !=
"(localdb)\\MSSQLLocalDB")
    {
        throw new InvalidOperationException("Refusing to use a database other than
this test's temporary LocalDB database.");
    }
}
}
}

```

Infrastructure/VendorTest.cs

```

using System.Net;
using System.Net.Http.Headers;
using System.Net.Http.Json;
using System.Text.Json;
using BridgeControl.Data;
using BridgeControl.Models.Platform;
using BridgeControl.Services.Platform;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;

namespace BridgeControl.Tests.Infrastructure;

/// <summary>
/// Provides shared setup and operations for vendor tests.
/// </summary>

```

```

public abstract class VendorTest : SqlServerTest
{
    /// <summary>
    /// Authenticates the test platform administrator.
    /// </summary>
    protected async Task SignInAsync()
    {
        await
Services.GetRequiredService<PlatformAdminSetup>().CreateAsync("platform@example.test",
"1234567890");
        using HttpResponseMessage login = await
Client.PostAsJsonAsync("/api/v1/platform/auth/login", new { Email = "platform@example.test",
Password = "1234567890" });
        Assert.Equal(HttpStatusCode.OK, login.StatusCode);
        JsonElement body = await ReadAsync(login);
        Client.DefaultRequestHeaders.Authorization = new
AuthenticationHeaderValue("Bearer", body.GetProperty("accessToken").GetString());
    }

    /// <summary>
    /// Provisions a vendor for the test scenario.
    /// </summary>
    protected async Task<Guid> CreateVendorAsync(string name = "Test vendor")
    {
        using HttpResponseMessage response = await
Client.PostAsJsonAsync("/api/v1/platform/vendors", new { Name = name });
        Assert.Equal(HttpStatusCode.Created, response.StatusCode);
        return (await ReadAsync(response)).GetProperty("id").GetGuid();
    }

    /// <summary>
    /// Opens the isolated vendor database used by the test.
    /// </summary>
    protected async Task<VendorDbContext> OpenVendorAsync(Guid id)
    {
        SoftwareVendor vendor = await
Database.SoftwareVendors.AsNoTracking().Include(vendor => vendor.Database).SingleAsync(vendor =>
vendor.Id == id);
        return Services.GetRequiredService<IVendorInfrastructure>().OpenDatabase(vendor);
    }

    /// <summary>
    /// Creates a vendor administrator invitation for the test.
    /// </summary>
    protected async Task<string> InviteAsync(Guid vendorId, string email =
"invited@example.test")
    {
        using HttpResponseMessage response = await
Client.PostAsJsonAsync($"{"/api/v1/platform/vendors/{vendorId}/administrators", new { Email = email
}});
        Assert.Equal(HttpStatusCode.Created, response.StatusCode);
        JsonElement body = await ReadAsync(response);
        Assert.Equal(new[] { "createdAt", "customerId", "email", "id", "roles", "status"
}, body.EnumerateObject().Select(property => property.Name).Order());
        await using VendorDbContext database = await OpenVendorAsync(vendorId);
        Guid id = (await database.AccountActivations.AsNoTracking().SingleAsync(activation
=> activation.RevokedAt == null && activation.UsedAt == null)).Id;
        string message = await
File.ReadAllTextAsync(Path.Combine(Factory.InvitationDirectory, $"{id:N}.eml"));
        return message.Split('\n').Single(line => line.StartsWith("Activation token: ",
StringComparison.Ordinal))[18..].Trim();
    }
}

```

```

    /// <summary>
    /// Deserializes the test HTTP response as the requested type.
    /// </summary>
    protected static async Task<JsonElement> ReadAsync(HttpResponseMessage response)
    {
        using JsonDocument document = JsonDocument.Parse(await
response.Content.ReadAsStringAsync());
        return document.RootElement.Clone();
    }
}

```

Integration/AgentCommunicationTests.cs

```

using System.Net;
using System.Net.Http.Headers;
using System.Net.Http.Json;
using BridgeControl.Models.Vendor;
using BridgeControl.Tests.Infrastructure;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Infrastructure;
using Microsoft.EntityFrameworkCore.Migrations;
using Agent = BridgeControl.Models.AgentModels.Agent;

namespace BridgeControl.Tests.Integration;

/// <summary>
/// Verifies agent communication behavior and regression scenarios.
/// </summary>
[Trait("Category", "Integration")]
public class AgentCommunicationTests : AgentTest
{
    /// <summary>
    /// Verifies that migration preserves legacy installations without assigning an agent.
    /// </summary>
    [Fact]
    public async Task Migration_PreservesLegacyInstallationsWithoutAssigningAnAgent()
    {
        await PrepareAsync(); Guid customer = await CustomerAsync(); Guid product = await
ProductAsync(); Guid id = Guid.NewGuid();
        await using var db = await OpenVendorAsync(VendorId);
        IMigrator migrator = db.GetService<IMigrator>();
        await migrator.MigrateAsync(db.Database.GetMigrations().Single(x =>
x.EndsWith("_AddConfigurationRevisionsAndPublishing", StringComparison.Ordinal)));
        await db.Database.ExecuteSqlInterpolatedAsync($"INSERT INTO Installations (Id,
CustomerId, ProductId, Name) VALUES ({id}, {customer}, {product}, N'Legacy')");
        DateTimeOffset before = DateTimeOffset.UtcNow.AddSeconds(-1);
        await migrator.MigrateAsync();
        Installation installation = await db.Installations.SingleAsync(x => x.Id == id);
        Assert.Null(installation.AgentId); Assert.Equal("Legacy", installation.Name);
        Assert.Equal("Active", installation.Status);
        Assert.InRange(installation.RegisteredAt, before,
DateTimeOffset.UtcNow.AddSeconds(1)); Assert.Null(installation.LastSeenAt);
        Assert.Empty(await db.Agents.ToArrayAsync());
        Assert.False(db.Database.HasPendingModelChanges());
    }
    /// <summary>
    /// Verifies that installation registration is persistent idempotent and cannot claim
    another agent or legacy installation.
    /// </summary>

```

```

[Fact]
public async Task
InstallationRegistration_IsPersistentIdempotent_AndCannotClaimAnotherAgentOrLegacyInstallation()
{
    await PrepareAsync(); Guid customer = await CustomerAsync(); Guid product = await
ProductAsync();
    var agent = await AgentAsync(customer); string token = await
TokenAsync(agent.Credential);
    var request = new { ProductId = product, Name = " Service ", InstalledVersion =
"1.2" };
    HttpResponseMessage[] responses = await Task.WhenAll (SendAsync(HttpMethod.Post,
"/api/v1/agent/installations", request, token), SendAsync(HttpMethod.Post,
"/api/v1/agent/installations", request, token));
    Assert.All(responses, x => Assert.Equal(HttpStatusCode.Created, x.StatusCode));
    Guid id = (await ReadAsync(responses[0])).GetProperty("id").GetGuid();
    Assert.Equal(id, (await ReadAsync(responses[1])).GetProperty("id").GetGuid());
    Assert.Equal(new[] { "agentId", "customerId", "id", "lastSeenAt", "name",
"productId", "registeredAt", "status" }, (await
ReadAsync(responses[0])).EnumerateObject().Select(x => x.Name).Order());
    foreach (var response in responses) { response.Dispose(); }
    await using (ApiFactory restarted = new(Database.Database.GetConnectionString(!))
    {
        using HttpClient client = restarted.CreateApiClient(); using var login =
await client.PostAsJsonAsync("/api/v1/agent/auth/token", new { agent.Credential });
        Assert.Equal(HttpStatusCode.OK, login.StatusCode);
        client.DefaultRequestHeaders.Authorization = new
AuthenticationHeaderValue("Bearer", (await
ReadAsync(login)).GetProperty("accessToken").GetString());
        using var retry = await
client.PostAsJsonAsync("/api/v1/agent/installations", request); Assert.Equal(id, (await
ReadAsync(retry)).GetProperty("id").GetGuid());
    }
    var other = await AgentAsync(customer); string otherToken = await
TokenAsync(other.Credential);
    using (var conflict = await SendAsync(HttpMethod.Post,
"/api/v1/agent/installations", request, otherToken)) { Assert.Equal(HttpStatusCode.Conflict,
conflict.StatusCode); }
    using (var list = await SendAsync(HttpMethod.Get, "/api/v1/agent/installations",
token: otherToken)) { Assert.Equal(0, (await
ReadAsync(list)).GetProperty("totalCount").GetInt32()); }
    await InstallationAsync(customer, product);
    using (var legacy = await SendAsync(HttpMethod.Post,
"/api/v1/agent/installations", new { ProductId = product, Name = "Installation" }, token)) {
        Assert.Equal(HttpStatusCode.Conflict, legacy.StatusCode); }
    using (var absent = await SendAsync(HttpMethod.Post,
"/api/v1/agent/installations", new { ProductId = Guid.NewGuid(), Name = "Unknown" }, token)) {
        Assert.Equal(HttpStatusCode.NotFound, absent.StatusCode); }
    await using var db = await OpenVendorAsync(VendorId);
    var stored = await db.Installations.SingleAsync(x => x.Id == id);
    Assert.Equal("1.2", stored.InstalledVersion); Assert.Equal(agent.AgentId, stored.AgentId);
    stored.Status = "Disabled"; await db.SaveChangesAsync();
    using (var disabled = await SendAsync(HttpMethod.Post,
"/api/v1/agent/installations", request, token)) { Assert.Equal(HttpStatusCode.Conflict,
disabled.StatusCode); }
    using var hidden = await SendAsync(HttpMethod.Get, "/api/v1/agent/installations",
token: token); Assert.Equal(0, (await ReadAsync(hidden)).GetProperty("totalCount").GetInt32());
    }
}

/// <summary>
/// Verifies that heartbeats deduplicate and keep latest status with bounded history.
/// </summary>
[Fact]

```

```

    public async Task Heartbeats_DeduplicateAndKeepLatestStatus_WithBoundedHistory()
    {
        await PrepareAsync(); var agent = await AgentAsync(await CustomerAsync()); string
token = await TokenAsync(agent.Credential);
        DateTimeOffset now = DateTimeOffset.UtcNow; var latest = new { AgentVersion = "2",
Status = "Healthy", ReportedAt = now };
        for (int i = 0; i < 2; i++)
        { using var response = await SendAsync(HttpMethod.Post, "/api/v1/agent/heartbeat",
latest, token); Assert.Equal(HttpStatusCode.NoContent, response.StatusCode); }
        using (var old = await SendAsync(HttpMethod.Post, "/api/v1/agent/heartbeat", new {
AgentVersion = "1", Status = "Old", ReportedAt = now.AddMinutes(-1) }, token)) {
Assert.Equal(HttpStatusCode.NoContent, old.StatusCode); }
        using (var conflict = await SendAsync(HttpMethod.Post, "/api/v1/agent/heartbeat",
new { AgentVersion = "1", Status = "Other", ReportedAt = now }, token)) {
Assert.Equal(HttpStatusCode.Conflict, conflict.StatusCode); }
        using (var future = await SendAsync(HttpMethod.Post, "/api/v1/agent/heartbeat",
new { AgentVersion = "3", Status = "Future", ReportedAt = now.AddHours(1) }, token)) {
Assert.Equal(HttpStatusCode.BadRequest, future.StatusCode); }
        using var detail = await SendAsync(HttpMethod.Get,
$"/api/v1/agents/{agent.AgentId}"); var body = await ReadAsync(detail);
        Assert.Equal("2", body.GetProperty("agentVersion").GetString());
Assert.Equal("Healthy", body.GetProperty("reportedStatus").GetString()); Assert.Equal("Active",
body.GetProperty("status").GetString());
        using var history = await SendAsync(HttpMethod.Get,
$"/api/v1/agents/{agent.AgentId}/heartbeats?pageSize=1&page=2"); var page = await
ReadAsync(history);
        Assert.Equal(2, page.GetProperty("totalCount").GetInt32());
Assert.Single(page.GetProperty("items").EnumerateArray());
        using var invalidPage = await SendAsync(HttpMethod.Get,
$"/api/v1/agents/{agent.AgentId}/heartbeats?pageSize=101");
Assert.Equal(HttpStatusCode.BadRequest, invalidPage.StatusCode);
        await using var db = await OpenVendorAsync(VendorId); Assert.Equal(2, await
db.AgentHeartbeats.CountAsync());
    }

    /// <summary>
    /// Verifies that human agent reads use customer technician and installation assignments.
    /// </summary>
    [Fact]
    public async Task HumanAgentReads_UseCustomerTechnicianAndInstallationAssignments()
    {
        await PrepareAsync(); Guid customer = await CustomerAsync(); Guid otherCustomer =
await CustomerAsync("Other");
        var agent = await AgentAsync(customer); var other = await
AgentAsync(otherCustomer); Guid product = await ProductAsync();
        string token = await TokenAsync(agent.Credential);
        using var registered = await SendAsync(HttpMethod.Post,
"/api/v1/agent/installations", new { ProductId = product, Name = "Service" }, token);
        Guid installation = (await ReadAsync(registered)).GetProperty("id").GetGuid();
        var admin = await UserAsync("customer@example.test", "CustomerAdministrator",
customer);
        var user = await UserAsync("user@example.test", "CustomerUser", customer);
        var technician = await UserAsync("tech@example.test", "VendorTechnician");
        using (var denied = await SendAsync(HttpMethod.Post, "/api/v1/agent-enrollments",
new { CustomerId = customer }, admin.Token)) { Assert.Equal(HttpStatusCode.Forbidden,
denied.StatusCode); }
        using (var visible = await SendAsync(HttpMethod.Get, "/api/v1/agents", token:
admin.Token)) { Assert.Equal(1, (await ReadAsync(visible)).GetProperty("totalCount").GetInt32());
}

        foreach (string path in new[] { $"/api/v1/agents/{other.AgentId}",
$"/api/v1/agents/{other.AgentId}/heartbeats" })

```

```

        { using var hidden = await SendAsync(HttpMethod.Get, path, token: admin.Token);
Assert.Equal(HttpStatusCode.NotFound, hidden.StatusCode); }
        foreach (string restricted in new[] { user.Token, technician.Token })
        { using var hidden = await SendAsync(HttpMethod.Get, "/api/v1/agents", token:
restricted); Assert.Equal(0, (await ReadAsync(hidden)).GetProperty("totalCount").GetInt32()); }
        using (var grant = await SendAsync(HttpMethod.Put,
$/api/v1/users/{user.Id}/installation-access", new { InstallationIds = new[] { installation }
})) { Assert.Equal(HttpStatusCode.NoContent, grant.StatusCode); }
        using (var grant = await SendAsync(HttpMethod.Put,
$/api/v1/users/{technician.Id}/customer-access", new { CustomerIds = new[] { customer } })) {
Assert.Equal(HttpStatusCode.NoContent, grant.StatusCode); }
        foreach (string allowed in new[] { user.Token, technician.Token })
        { using var visible = await SendAsync(HttpMethod.Get, "/api/v1/agents", token:
allowed); Assert.Equal(1, (await ReadAsync(visible)).GetProperty("totalCount").GetInt32()); }
        using (var revoke = await SendAsync(HttpMethod.Put,
$/api/v1/users/{user.Id}/installation-access", new { InstallationIds = Array.Empty<Guid>() } )) {
Assert.Equal(HttpStatusCode.NoContent, revoke.StatusCode); }
        using var revoked = await SendAsync(HttpMethod.Get,
$/api/v1/agents/{agent.AgentId}", token: user.Token); Assert.Equal(HttpStatusCode.NotFound,
revoked.StatusCode);
    }

    /// <summary>
    /// Verifies that database rejects agent installation customer mismatch.
    /// </summary>
    [Fact]
    public async Task Database_RejectsAgentInstallationCustomerMismatch()
    {
        await PrepareAsync(); Guid customer = await CustomerAsync(); var agent = await
AgentAsync(customer); Guid other = await CustomerAsync("Other"); Guid product = await
ProductAsync();
        await using var db = await OpenVendorAsync(VendorId);
        db.Installations.Add(new Installation { CustomerId = other, AgentId =
agent.AgentId, ProductId = product, Name = "Forged" });
        await Assert.ThrowsAsync<DbUpdateException>(() => db.SaveChangesAsync());
    }
}

```

Integration/AgentIdentityTests.cs

```

using System.Net;
using System.Net.Http.Headers;
using System.Net.Http.Json;
using System.Security.Cryptography;
using System.Text;
using BridgeControl.DTOS.AgentDTOS.AgentContracts;
using BridgeControl.Services.Vendor;
using BridgeControl.Tests.Infrastructure;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using BridgeControl.Services.AgentServices;

namespace BridgeControl.Tests.Integration;

/// <summary>
/// Verifies agent identity behavior and regression scenarios.
/// </summary>
[Trait("Category", "Integration")]
public class AgentIdentityTests : AgentTest
{

```

```

    /// <summary>
    /// Verifies that tokens are capped by credential expiry and reject another active vendor
locator.
    /// </summary>
    [Fact]
    public async Task Tokens_AreCappedByCredentialExpiry_AndRejectAnotherActiveVendorLocator()
    {
        await PrepareAsync(); var agent = await AgentAsync(await CustomerAsync());
        await using var db = await OpenVendorAsync(VendorId);
        DateTimeOffset expiry = DateTimeOffset.UtcNow.AddMinutes(2);
        await db.AgentCredentials.ExecuteUpdateAsync(s => s.SetProperty(x => x.ExpiresAt,
expiry));
        using var token = await SendAsync(HttpMethod.Post, "/api/v1/agent/auth/token", new
{ agent.Credential }); Assert.Equal(HttpStatusCode.OK, token.StatusCode);
        var body = await ReadAsync(token); Assert.Equal(expiry,
body.GetProperty("expiresAt").GetDateTimeOffset());
        var jwt = new
System.IdentityModel.Tokens.Jwt.JwtSecurityTokenHandler().ReadJwtToken(body.GetProperty("accessTo
ken").GetString());
        Assert.Equal("BridgeControl.Agent", Assert.Single(jwt.Audiences));
        Assert.Equal("agent", jwt.Claims.Single(x => x.Type == "identity_type").Value);
        Assert.True(jwt.ValidTo <= expiry.UtcDateTime);
        Guid otherVendor = await CreateVendorAsync("Other vendor"); await
InviteAsync(otherVendor, "other@example.test");
        using var wrongVendor = await SendAsync(HttpMethod.Post,
"/api/v1/agent/auth/token", new { Credential = agent.Credential.Replace(VendorId.ToString("N"),
otherVendor.ToString("N")) });
        Assert.Equal(HttpStatusCode.Unauthorized, wrongVendor.StatusCode);
    }

    /// <summary>
    /// Verifies that token endpoint counts failed attempts across route variants.
    /// </summary>
    [Fact]
    public async Task TokenEndpoint_CountsFailedAttemptsAcrossRouteVariants()
    {
        await PrepareAsync(); var agent = await AgentAsync(await CustomerAsync());
        string invalid = agent.Credential[..^43] + new string('A', 43);
        for (int i = 0; i < 60; i++)
        {
            using var denied = await SendAsync(HttpMethod.Post,
"/api/v1/agent/auth/token", new { Credential = invalid });
            Assert.Equal(HttpStatusCode.Unauthorized, denied.StatusCode);
        }
        using var limited = await SendAsync(HttpMethod.Post, "/API/V1/AGENT/AUTH/TOKEN/",
new { agent.Credential });
        Assert.Equal(HttpStatusCode.TooManyRequests, limited.StatusCode);
        Assert.NotNull(limited.Headers.RetryAfter);
    }

    /// <summary>
    /// Verifies that enrollment is atomic single use stores only hashes and separates human
tokens.
    /// </summary>
    [Fact]
    public async Task Enrollment_IsAtomicSingleUse_StoresOnlyHashes_AndSeparatesHumanTokens()
    {
        await PrepareAsync(); Guid customer = await CustomerAsync(); string code = await
EnrollAsync(customer);
        HttpResponseMessage[] attempts = await Task.WhenAll(RegisterAsync(code),
RegisterAsync(code));

```

```

        Assert.Single(attempts, x => x.StatusCode == HttpStatusCode.Created);
Assert.Single(attempts, x => x.StatusCode == HttpStatusCode.Unauthorized);
        var body = await ReadAsync(attempts.Single(x => x.IsSuccessStatusCode)); string
credential = body.GetProperty("credential").GetString();
        foreach (var response in attempts) { response.Dispose(); }
        await using var db = await OpenVendorAsync(VendorId);
        var enrollment = await db.AgentEnrollments.SingleAsync(); var stored = await
db.AgentCredentials.SingleAsync(); var agent = await db.Agents.SingleAsync();
        Assert.NotNull(enrollment.UsedAt); Assert.Equal(agent.Id, enrollment.AgentId);
Assert.Equal(customer, agent.CustomerId); Assert.Equal("Test machine", agent.MachineName);
        Assert.Equal(SHA256.HashData(Encoding.UTF8.GetBytes(code.Split('.')[3])),
enrollment.SecretHash);
        Assert.Equal(SHA256.HashData(Encoding.UTF8.GetBytes(credential.Split('.')[3])),
stored.SecretHash);
        Assert.InRange(stored.ExpiresAt - stored.CreatedAt, TimeSpan.FromDays(90),
TimeSpan.FromDays(90).Add(TimeSpan.FromSeconds(1)));
        string token = await TokenAsync(credential);
        using var humanRoute = await SendAsync(HttpMethod.Get, "/api/v1/agents", token:
token); Assert.Equal(HttpStatusCode.Unauthorized, humanRoute.StatusCode);
        using var agentRoute = await SendAsync(HttpMethod.Get,
"/api/v1/agent/installations"); Assert.Equal(HttpStatusCode.Unauthorized, agentRoute.StatusCode);
        using var platformRoute = await Client.GetAsync("/api/v1/agent/installations");
Assert.Equal(HttpStatusCode.Unauthorized, platformRoute.StatusCode);
        using var managed = await SendAsync(HttpMethod.Get, "/api/v1/agent/installations",
token: token); Assert.Equal(HttpStatusCode.OK, managed.StatusCode);
        Assert.DoesNotContain(credential, string.Join(" ", await
db.VendorAuditLogs.Select(x => x.Context).ToArrayAsync()));
    }

    /// <summary>
    /// Verifies that enrollment rejects expired revoked wrong purpose and altered locators.
    /// </summary>
    [Fact]
    public async Task Enrollment_RejectsExpiredRevokedWrongPurposeAndAlteredLocators()
    {
        await PrepareAsync(); Guid customer = await CustomerAsync(); string code = await
EnrollAsync(customer);
        await using var db = await OpenVendorAsync(VendorId);
        var enrollment = await db.AgentEnrollments.SingleAsync();
        foreach (string invalid in new[] { " " + code,
code.Replace(VendorId.ToString("N"), Guid.NewGuid().ToString("N")), code.Replace("bce1", "bca1"),
code[..^43] + new string('A', 43) })
        { using var response = await RegisterAsync(invalid);
Assert.Equal(HttpStatusCode.Unauthorized, response.StatusCode); }
        enrollment.ExpiresAt = DateTimeOffset.UtcNow.AddMinutes(-1); await
db.SaveChangesAsync();
        using (var expired = await RegisterAsync(code)) {
Assert.Equal(HttpStatusCode.Unauthorized, expired.StatusCode); }
        enrollment.ExpiresAt = DateTimeOffset.UtcNow.AddMinutes(15); enrollment.RevokedAt
= DateTimeOffset.UtcNow; await db.SaveChangesAsync();
        using (var revoked = await RegisterAsync(code)) {
Assert.Equal(HttpStatusCode.Unauthorized, revoked.StatusCode); }
        enrollment.RevokedAt = null; enrollment.Purpose = "Other"; await
db.SaveChangesAsync();
        using (var wrong = await RegisterAsync(code)) {
Assert.Equal(HttpStatusCode.Unauthorized, wrong.StatusCode); }
        Assert.Empty(await db.Agents.ToArrayAsync());
    }

    /// <summary>
    /// Verifies that operator rotation and revocation invalidate existing tokens and
    credential survives host restart.

```

```

    /// </summary>
    [Fact]
    public async Task
    OperatorRotationAndRevocation_InvalidatesExistingTokens_AndCredentialSurvivesHostRestart()
    {
        await PrepareAsync(); var agent = await AgentAsync(await CustomerAsync()); string
        oldToken = await TokenAsync(agent.Credential);
        await using (ApiFactory restarted = new(Database.Database.GetConnectionString(!))
        {
            using HttpClient client = restarted.CreateApiClient();
            using var response = await
            client.PostAsJsonAsync("/api/v1/agent/auth/token", new { agent.Credential });
            Assert.Equal(HttpStatusCode.OK, response.StatusCode);
            client.DefaultRequestHeaders.Authorization = new
            AuthenticationHeaderValue("Bearer", (await
            ReadAsync(response)).GetProperty("accessToken").GetString());
            using var read = await client.GetAsync("/api/v1/agent/installations");
            Assert.Equal(HttpStatusCode.OK, read.StatusCode);
        }
        AgentRegistrationResponse rotated = (await
        Services.GetRequiredService<AgentCredentialOperator>().ChangeAsync(VendorId, agent.AgentId,
        true));
        using (var old = await SendAsync(HttpMethod.Get, "/api/v1/agent/installations",
        token: oldToken)) { Assert.Equal(HttpStatusCode.Unauthorized, old.StatusCode); }
        using (var old = await SendAsync(HttpMethod.Post, "/api/v1/agent/auth/token", new
        { agent.Credential })) { Assert.Equal(HttpStatusCode.Unauthorized, old.StatusCode); }
        string token = await TokenAsync(rotated.Credential);
        await Services.GetRequiredService<AgentCredentialOperator>().ChangeAsync(VendorId,
        agent.AgentId, false);
        using var denied = await SendAsync(HttpMethod.Get, "/api/v1/agent/installations",
        token: token); Assert.Equal(HttpStatusCode.Unauthorized, denied.StatusCode);
        await using var db = await OpenVendorAsync(VendorId); Assert.Equal(2, await
        db.AgentCredentials.CountAsync(x => x.RevokedAt != null));
        Assert.Equal(1, await db.Agents.CountAsync()); Assert.Equal(2, await
        db.VendorAuditLogs.CountAsync(x => x.ActorType == "Operator"));
    }

    /// <summary>
    /// Verifies that every request rechecks live lifecycle.
    /// </summary>
    [Theory]
    [InlineData("credential-expiry")]
    [InlineData("credential-purpose")]
    [InlineData("agent")]
    [InlineData("customer")]
    [InlineData("vendor")]
    public async Task EveryRequest_RechecksLiveLifecycle(string boundary)
    {
        await PrepareAsync(); Guid customer = await CustomerAsync(); var agent = await
        AgentAsync(customer); string token = await TokenAsync(agent.Credential);
        string unused = await EnrollAsync(customer);
        await using var db = await OpenVendorAsync(VendorId);
        switch (boundary)
        {
            case "credential-expiry": await db.AgentCredentials.ExecuteUpdateAsync(s =>
            s.SetProperty(x => x.ExpiresAt, DateTimeOffset.UtcNow.AddSeconds(-1))); break;
            case "credential-purpose": await db.AgentCredentials.ExecuteUpdateAsync(s
            => s.SetProperty(x => x.Purpose, "Other")); break;
            case "agent": await db.Agents.ExecuteUpdateAsync(s => s.SetProperty(x =>
            x.Status, "Disabled")); break;
        }
    }

```

```

        case "customer": using (var disabled = await SendAsync(HttpMethod.Post,
$/api/v1/customers/{customer}/deactivate")) { Assert.Equal(HttpStatusCode.NoContent,
disabled.StatusCode); } break;
        case "vendor": using (var disabled = await
Client.PostAsync($"/api/v1/platform/vendors/{VendorId}/deactivate", null)) {
Assert.Equal(HttpStatusCode.NoContent, disabled.StatusCode); } break;
    }
    using (var denied = await SendAsync(HttpMethod.Get, "/api/v1/agent/installations",
token: token)) { Assert.Equal(HttpStatusCode.Unauthorized, denied.StatusCode); }
    using (var denied = await SendAsync(HttpMethod.Post, "/api/v1/agent/auth/token",
new { agent.Credential })) { Assert.Equal(HttpStatusCode.Unauthorized, denied.StatusCode); }
    if (boundary is "customer" or "vendor")
    {
        using var enabled = boundary == "customer" ? await
SendAsync(HttpMethod.Post, $"/api/v1/customers/{customer}/reactivate") : await
Client.PostAsync($"/api/v1/platform/vendors/{VendorId}/reactivate", null);
        Assert.Equal(HttpStatusCode.NoContent, enabled.StatusCode);
        using var replay = await RegisterAsync(unused);
        Assert.Equal(HttpStatusCode.Unauthorized, replay.StatusCode);
        using var denied = await SendAsync(HttpMethod.Get,
"/api/v1/agent/installations", token: token); Assert.Equal(HttpStatusCode.Unauthorized,
denied.StatusCode);

        Assert.NotNull((await db.AgentCredentials.SingleAsync()).RevokedAt);
    }
}

/// <summary>
/// Verifies that registration audit failure rolls back enrollment and identity.
/// </summary>
[Fact]
public async Task Registration_AuditFailureRollsBackEnrollmentAndIdentity()
{
    await PrepareAsync(); string code = await EnrollAsync(await CustomerAsync());
    await using var db = await OpenVendorAsync(VendorId);
    await db.Database.ExecuteSqlRawAsync("CREATE TRIGGER RejectAgentAudit ON
VendorAuditLogs AFTER INSERT AS BEGIN THROW 51000, 'test audit failure', 1; END");
    using (var failed = await RegisterAsync(code)) {
        Assert.Equal(HttpStatusCode.InternalServerError, failed.StatusCode); }
    Assert.Null((await db.AgentEnrollments.SingleAsync()).UsedAt); Assert.Empty(await
db.Agents.ToArrayAsync()); Assert.Empty(await db.AgentCredentials.ToArrayAsync());
    await db.Database.ExecuteSqlRawAsync("DROP TRIGGER RejectAgentAudit");
    using var retried = await RegisterAsync(code);
    Assert.Equal(HttpStatusCode.Created, retried.StatusCode);
}

/// <summary>
/// Verifies that enrollment and exchange limits count failures and share case variants.
/// </summary>
[Fact]
public async Task EnrollmentAndExchangeLimits_CountFailures_AndShareCaseVariants()
{
    await PrepareAsync(); Guid customer = await CustomerAsync();
    for (int i = 0; i < 10; i++)
    {
        using var failed = await SendAsync(HttpMethod.Post,
"/api/v1/agent-enrollments", new { CustomerId = Guid.NewGuid() });
        Assert.Equal(HttpStatusCode.NotFound, failed.StatusCode);
        using var invalid = await SendAsync(HttpMethod.Post,
"/api/v1/agent/register", new { EnrollmentCode = "invalid", MachineName = "Machine", AgentVersion
= "1" }); Assert.Equal(HttpStatusCode.Unauthorized, invalid.StatusCode);
    }
}

```

```

        using var limited = await SendAsync(HttpMethod.Post, "/API/V1/AGENT-ENROLLMENTS/",
new { CustomerId = customer }); Assert.Equal(HttpStatusCode.TooManyRequests, limited.StatusCode);
Assert.NotNull(limited.Headers.RetryAfter);
        using var exchange = await SendAsync(HttpMethod.Post, "/API/V1/AGENT/REGISTER/",
new { EnrollmentCode = "invalid", MachineName = "Machine", AgentVersion = "1" });
Assert.Equal(HttpStatusCode.TooManyRequests, exchange.StatusCode);
Assert.NotNull(exchange.Headers.RetryAfter);
    }
}

```

Integration/ApiContractAcceptanceTests.cs

```

using System.Security.Cryptography;
using System.Text.Json;
using System.Text.RegularExpressions;
using BridgeControl.Tests.Infrastructure;
using Microsoft.AspNetCore.Routing;
using Microsoft.Extensions.DependencyInjection;

namespace BridgeControl.Tests.Integration;

/// <summary>
/// Verifies api contract acceptance behavior and regression scenarios.
/// </summary>
public sealed class ApiContractAcceptanceTests
{
    /// <summary>
    /// Verifies that delivery evidence preserves all 110 stories priorities and scope
    exceptions.
    /// </summary>
    [Fact]
    public async Task DeliveryEvidencePreservesAll110StoriesPrioritiesAndScopeExceptions()
    {
        using var original = JsonDocument.Parse(await
File.ReadAllTextAsync(Path.Combine(AppContext.BaseDirectory, "Acceptance",
"requirements.json")));
        using var evidence = JsonDocument.Parse(await
File.ReadAllTextAsync(Path.Combine(AppContext.BaseDirectory, "Acceptance", "evidence.json")));
        var stories =
evidence.RootElement.GetProperty("stories").EnumerateArray().ToDictionary(x =>
x.GetProperty("id").GetString());
        Assert.Equal(110, stories.Count);
        Assert.Equal(103, stories.Values.Count(x => x.GetProperty("priority").GetString()
== "Must Have"));
        Assert.Equal(7, stories.Values.Count(x => x.GetProperty("priority").GetString() ==
"Should Have"));
        foreach (var story in
original.RootElement.GetProperty("stories").EnumerateArray())
        {
            var delivered = stories[story.GetProperty("id").GetString()!];
            foreach (string field in new[] { "title", "priority" }) {
Assert.Equal(story.GetProperty(field).GetString(), delivered.GetProperty(field).GetString());
            Assert.Equal(story.GetProperty("resolution").GetString(),
delivered.GetProperty("scopeResolution").GetString());
            string status = delivered.GetProperty("deliveryStatus").GetString();
            if
(story.GetProperty("resolution").GetString()!.Contains("scope-exception",
StringComparison.Ordinal)) { Assert.Contains("scope-exception", status); }
            Assert.NotEmpty(delivered.GetProperty("testClasses").EnumerateArray());
            foreach (var test in delivered.GetProperty("testClasses").EnumerateArray())

```

```

        {
Assert.NotNull(typeof(ApiContractAcceptanceTests).Assembly.GetType("BridgeControl.Tests.Integrati
on." + test.GetString())); }
        }
        Assert.Contains("capacity-acceptance-open",
stories["M8-S06"].GetProperty("deliveryStatus").GetString());
    }
    private const string Connection =
"Server=(localdb)\\MSSQLLocalDB;Database=BridgeControl_Unused;Integrated
Security=True;TrustServerCertificate=True";
    /// <summary>
    /// Verifies that open API matches approved documented operations with complete metadata.
    /// </summary>
    [Fact]
    public async Task OpenApiMatchesApprovedDocumentedOperations_WithCompleteMetadata()
    {
        Assert.Equal("472FEC025C5CD1CC0714ADA59E132B7827D655F158276FBB763339AB2F007FE6",
Convert.ToHexString(SHA256.HashData(await File.ReadAllBytesAsync(FrozenApiContract.Path))));
        await using ApiFactory factory = new(Connection); using HttpClient client =
factory.CreateApiClient();
        using JsonDocument api = JsonDocument.Parse(await
client.GetStringAsync("/openapi/v1.json"));
        var expected = FrozenApiContract.Operations(); Assert.Equal(121, expected.Length);
        // Inspect the actual router as well as OpenAPI, so hidden or duplicate routes
cannot escape this comparison.
        var registered =
factory.Services.GetRequiredService<EndpointDataSource>().Endpoints.Of<RouteEndpoint>()
.Where(endpoint =>
endpoint.RoutePattern.RawText?.TrimStart('/').StartsWith("api/v1/", StringComparison.Ordinal) ==
true)
        .SelectMany(endpoint =>
endpoint.Metadata.GetRequiredMetadata<HttpMethodMetadata>().HttpMethods.Select(method =>
new FrozenApiContract.Operation(method, "/" +
Regex.Replace(endpoint.RoutePattern.RawText!.TrimStart('/'), @"\"{([^\:]+):[^\}]+\}", "{${1}}")))
        .OrderBy(operation => operation.Path).ThenBy(operation =>
operation.Method).ToArray();
        Assert.Equal(expected.OrderBy(x => x.Path).ThenBy(x => x.Method), registered);
        var actual =
api.RootElement.GetProperty("paths").EnumerateObject().SelectMany(path =>
path.Value.EnumerateObject().Select(operation => new
FrozenApiContract.Operation(operation.Name.ToUpperInvariant(), path.Name))).ToArray();
        Assert.Equal(expected.OrderBy(x => x.Path).ThenBy(x => x.Method), actual.OrderBy(x
=> x.Path).ThenBy(x => x.Method));
        HashSet<string> ids = [];
        foreach (var path in api.RootElement.GetProperty("paths").EnumerateObject())
        {
            foreach (var operation in path.Value.EnumerateObject())
            {
                string label = operation.Name + " " + path.Name;
                Assert.True(operation.Value.TryGetProperty("operationId", out var
id) && !string.IsNullOrWhiteSpace(id.GetString()) && ids.Add(id.GetString()), label + " lacks a
unique stable operation ID.");
                Assert.True(operation.Value.TryGetProperty("summary", out var
summary) && !string.IsNullOrWhiteSpace(summary.GetString()), label + " lacks a summary.");
                Assert.True(operation.Value.TryGetProperty("tags", out var tags) &&
tags.GetArrayLength() > 0, label + " lacks a tag.");
                Assert.Contains("Authorization:",
operation.Value.GetProperty("description").GetString());

Assert.Contains(operation.Value.GetProperty("responses").EnumerateObject(), response =>
response.Name.StartsWith('2'));

```

```

Assert.True(operation.Value.GetProperty("responses").TryGetProperty("400", out _), label + "
lacks validation/transport errors.");
    }
    string json = api.RootElement.ToString();
    foreach (string secret in new[] { "connectionSecretRef", "accessSecretRef",
"storageIdentifier", "passwordHash", "protectedToken", "tokenHash", "databaseName" })
    { Assert.DoesNotContain("'" + secret + "'", json,
StringComparison.OrdinalIgnoreCase); }
    Assert.Contains("section 4.1",
api.RootElement.GetProperty("paths").GetProperty("/api/v1/platform/account/me").GetProperty("get"
).GetProperty("description").GetString());
    }
    /// <summary>
    /// Verifies that generated dto fields match every explicit frozen record without adding
infrastructure fields.
    /// </summary>
    [Fact]
    public async Task
GeneratedDtoFieldsMatchEveryExplicitFrozenRecord_WithoutAddingInfrastructureFields()
    {
        await using ApiFactory factory = new(Connection); using HttpClient client =
factory.CreateApiClient();
        using JsonDocument api = JsonDocument.Parse(await
client.GetStringAsync("/openapi/v1.json")); JsonElement schemas =
api.RootElement.GetProperty("components").GetProperty("schemas");
        Dictionary<string, string> aliases = new()
        {
            ["VendorResponse"] = "VendorDto",
            ["ProductResponse"] = "ProductDto",
            ["ProductVersionResponse"] = "ProductVersionDto",
            ["FileArtifactResponse"] = "FileArtifactDto"
        };
        foreach (var (name, fields) in FrozenApiContract.DtoFields())
        {
            if (name == "PagedResponse") { continue; }
            string actualName = aliases.GetValueOrDefault(name, name);
            Assert.True(schemas.TryGetProperty(actualName, out JsonElement schema),
"Missing documented DTO " + name + "; generated schemas: " + string.Join(", ",
schemas.EnumerateObject().Select(x => x.Name)));
            Assert.Equal(fields,
schema.GetProperty("properties").EnumerateObject().Select(x => x.Name).Order());
        }
    }
    /// <summary>
    /// Verifies that multipart binary pagination and no content responses describe the actual
wire contract.
    /// </summary>
    [Fact]
    public async Task
MultipartBinaryPaginationAndNoContentResponsesDescribeTheActualWireContract()
    {
        await using ApiFactory factory = new(Connection); using HttpClient client =
factory.CreateApiClient();
        using JsonDocument api = JsonDocument.Parse(await
client.GetStringAsync("/openapi/v1.json")); var paths = api.RootElement.GetProperty("paths");
        foreach (string kind in new[] { "installer", "schema" })
        {
            var upload =
paths.GetProperty("/api/v1/products/{productId}/versions/{versionId}/" +
kind).GetProperty("put");

```

```

        var schema =
upload.GetProperty("requestBody").GetProperty("content").GetProperty("multipart/form-data").GetProperty("schema");
        Assert.Equal("binary",
schema.GetProperty("properties").GetProperty("file").GetProperty("format").GetString());
        var download =
paths.GetProperty("/api/v1/products/{productId}/versions/{versionId}/" + kind +
"/content").GetProperty("get");
        Assert.Equal("binary",
download.GetProperty("responses").GetProperty("200").GetProperty("content").EnumerateObject().First().Value.GetProperty("schema").GetProperty("format").GetString());
    }
    foreach (string route in new[] { "/api/v1/agent/logs", "/api/v1/agent/heartbeat"
})
    {
        var responses =
paths.GetProperty(route).GetProperty("post").GetProperty("responses");
Assert.True(responses.TryGetProperty("204", out _)); Assert.False(responses.TryGetProperty("200",
out _)); }

Assert.True(paths.GetProperty("/api/v1/agent/installations").GetProperty("post").GetProperty("responses").TryGetProperty("201", out _));
        var parameters =
paths.GetProperty("/api/v1/products").GetProperty("get").GetProperty("parameters");
        Assert.Contains(parameters.EnumerateArray(), x =>
x.GetProperty("name").GetString() == "pageSize" &&
x.GetProperty("schema").GetProperty("default").GetInt32() == 25);
    }
}

```

Integration/ApidentityAcceptanceTests.cs

```

using System.Net;
using System.Net.Http.Headers;
using System.Net.Http.Json;
using System.Text.RegularExpressions;
using BridgeControl.Tests.Infrastructure;
using Xunit.Abstractions;

namespace BridgeControl.Tests.Integration;

/// <summary>
/// Verifies api identity acceptance behavior and regression scenarios.
/// </summary>
public sealed class ApiIdentityAcceptanceTests(ITestOutputHelper output) : InstallationStateTest
{
    private static readonly HashSet<string> CredentialEndpoints =
["/api/v1/platform/auth/login", "/api/v1/platform/auth/refresh", "/api/v1/auth/login",
"/api/v1/auth/refresh",
"/api/v1/auth/activate", "/api/v1/auth/password/forgot",
"/api/v1/auth/password/reset", "/api/v1/agent/register", "/api/v1/agent/auth/token"];
    /// <summary>
    /// Verifies that every protected route rejects anonymous and both wrong identity types
before binding or mutation.
    /// </summary>
    [Fact]
    public async Task
EveryProtectedRouteRejectsAnonymousAndBothWrongIdentityTypes_BeforeBindingOrMutation()
    {
        StateFixture state = await StateAsync(); string platform =
Client.DefaultRequestHeaders.Authorization!.Parameter!;

```

```

using HttpClient boundaryClient = Factory.CreateApiClient();
int checkedRoutes = 0, rejectedRequests = 0;
foreach (var operation in FrozenApiContract.Operations())
{
    if (CredentialEndpoints.Contains(operation.Path)) { continue; }
    string expected = operation.Path.StartsWith("/api/v1/platform/",
StringComparison.Ordinal) ? "platform" : operation.Path.StartsWith("/api/v1/agent/",
StringComparison.Ordinal) ? "agent" : "tenant";
    foreach (var identity in new[] { ("anonymous", ""), ("platform", platform),
("tenant", AdminToken), ("agent", state.Token) }.Where(x => x.Item1 != expected))
    {
        string path = Regex.Replace(operation.Path, @"\[^\]+\\"",
"11111111-1111-1111-1111-111111111111");
        using HttpRequestMessage request = new(new
HttpMethod(operation.Method), path);
        if (identity.Item2.Length != 0) { request.Headers.Authorization =
new AuthenticationHeaderValue("Bearer", identity.Item2); }
        if (operation.Method is "POST" or "PUT" or "PATCH") {
request.Content = JsonContent.Create(new { }); }
        if (operation.Method == "PUT" &&
(operation.Path.EndsWith("/installer", StringComparison.Ordinal) ||
operation.Path.EndsWith("/schema", StringComparison.Ordinal))) { request.Content?.Dispose();
request.Content = new MultipartFormDataContent(); }
        using HttpResponseMessage response = await
boundaryClient.SendAsync(request);
        Assert.True(response.StatusCode == HttpStatusCode.Unauthorized,
$"{{operation.Method}} {{operation.Path}} accepted {{identity.Item1}}: {{(int)response.StatusCode}}");
        rejectedRequests++;
    }
    checkedRoutes++;
}
Assert.Equal(105, checkedRoutes); output.WriteLine($"{{checkedRoutes}} protected
operations: {{rejectedRequests}} rejected anonymous/wrong-identity requests.");
}
}

```

Integration/ApiValidationTests.cs

```

using System.Net;
using System.Text;
using System.Text.Json;
using BridgeControl.Tests.Infrastructure;

namespace BridgeControl.Tests.Integration;

/// <summary>
/// Verifies api validation behavior and regression scenarios.
/// </summary>
[Trait("Category", "Integration")]
public class ApiValidationTests
{
    /// <summary>
    /// Verifies that login invalid public request returns validation details.
    /// </summary>
    [Theory]
    [InlineData("{}")]
    [InlineData("{\"email\":\"not-an-email\",\"password\":\"1234567890\"}")]
    [InlineData("{\"email\":\"user@example.test\"}")]
    public async Task Login_InvalidPublicRequestReturnsValidationDetails(string json)
    {

```

```

        // Validation should finish without connecting to a database.
        await using ApiFactory factory =
new("Server=(localdb)\\MSSQLLocalDB;Database=BridgeControl_Test_NoConnection;Integrated
Security=True");
        using HttpClient client = factory.CreateApiClient();
        using StringContent content = new(json, Encoding.UTF8, "application/json");
        using HttpResponseMessage response = await
client.PostAsync("/api/v1/platform/auth/login", content);

        Assert.Equal(HttpStatusCode.BadRequest, response.StatusCode);
        Assert.Equal("application/problem+json",
response.Content.Headers.ContentType?.MediaType);
        using JsonDocument document = JsonDocument.Parse(await
response.Content.ReadAsStringAsync());
        Assert.Equal("https://bridgecontrol/errors/validation",
document.RootElement.GetProperty("type").GetString());
        Assert.NotEmpty(document.RootElement.GetProperty("errors").EnumerateObject());

Assert.False(string.IsNullOrEmpty(document.RootElement.GetProperty("traceId").GetString()));
    }
}

```

Integration/ApplicationAssignmentTests.cs

```

using System.Net;
using BridgeControl.Tests.Infrastructure;

namespace BridgeControl.Tests.Integration;

/// <summary>
/// Verifies application assignment behavior and regression scenarios.
/// </summary>
public sealed class ApplicationAssignmentTests : AgentTest
{
    /// <summary>
    /// Verifies that published initialization assignment retries agent ownership and
    immutability.
    /// </summary>
    [Fact]
    public async Task
PublishedInitialization_AssignmentRetries_AgentOwnership_AndImmutability()
    {
        await PrepareAsync();
        var (product, version) = await CatalogAsync(installer: true);
        using var published = await PublishAsync(product, version);
        Assert.Equal(HttpStatusCode.OK, published.StatusCode);
        var settings = new { Kind = "Desktop", ExecutablePath = @"C:\Program
Files\Widget\Widget.exe", ConfigurationPath = @"C:\Program Files\Widget\defaults.json" };
        string path =
$/api/v1/products/{product}/versions/{version}/deployment-settings";
        using var initialized = await SendAsync(HttpMethod.Post, path + "/initialize",
settings);

        Assert.Equal(HttpStatusCode.OK, initialized.StatusCode);
        Assert.True((await ReadAsync(initialized)).GetProperty("frozen").GetBoolean());
        using var repeated = await SendAsync(HttpMethod.Post, path + "/initialize",
settings);

        Assert.Equal(HttpStatusCode.OK, repeated.StatusCode);
        using var changed = await SendAsync(HttpMethod.Post, path + "/initialize",
settings with { ConfigurationPath = @"C:\changed.json" });
        Assert.Equal(HttpStatusCode.Conflict, changed.StatusCode);
    }
}

```

```

        Guid customer = await CustomerAsync();
        var agent = await AgentAsync(customer);
        var otherAgent = await AgentAsync(customer);
        var body = new { AgentId = agent.AgentId, ProductId = product, Name = "Widget" };
        using var assigned = await SendAsync(HttpMethod.Post, "/api/v1/installations",
body);

        Assert.Equal(HttpStatusCode.Created, assigned.StatusCode);
        Guid installation = (await ReadAsync(assigned)).GetProperty("id").GetGuid();
        using var duplicate = await SendAsync(HttpMethod.Post, "/api/v1/installations",
body);

        Assert.Equal(installation, (await
ReadAsync(duplicate)).GetProperty("id").GetGuid());
        using var discovery = await SendAsync(HttpMethod.Get,
"/api/v1/agent/installations", token: await TokenAsync(agent.Credential));
        var discovered = await ReadAsync(discovery);
        Assert.Equal(1, discovered.GetProperty("totalCount").GetInt32());
        var item = Assert.Single(discovered.GetProperty("items").EnumerateArray());
        Assert.Equal(installation, item.GetProperty("id").GetGuid());
        Assert.Equal(customer, item.GetProperty("customerId").GetGuid());
        Assert.Equal("Desktop",
item.GetProperty("application").GetProperty("kind").GetString());
        Assert.Equal(version,
item.GetProperty("application").GetProperty("versions")[0].GetProperty("productVersionId").GetGuid());

        using var nextPage = await SendAsync(HttpMethod.Get,
"/api/v1/agent/installations?page=2&pageSize=1", token: await TokenAsync(agent.Credential));
        var emptyPage = await ReadAsync(nextPage);
        Assert.Empty(emptyPage.GetProperty("items").EnumerateArray());
        Assert.Equal(1, emptyPage.GetProperty("totalCount").GetInt32());
        Assert.Equal(2, emptyPage.GetProperty("page").GetInt32());
        Assert.Equal(1, emptyPage.GetProperty("pageSize").GetInt32());
        using var hidden = await SendAsync(HttpMethod.Get, "/api/v1/agent/installations",
token: await TokenAsync(otherAgent.Credential));
        Assert.Empty((await ReadAsync(hidden)).GetProperty("items").EnumerateArray());
        var outsiderCustomer = await CustomerAsync("Other");
        var outsider = await UserAsync("outside@example.com", "CustomerAdministrator",
outsiderCustomer);
        using var forbidden = await SendAsync(HttpMethod.Post, "/api/v1/installations",
body, outsider.Token);
        Assert.Contains(forbidden.StatusCode, new[] { HttpStatusCode.Forbidden,
HttpStatusCode.NotFound });
    }
}

```

Integration/CapacityBaselineTests.cs

```

using System.Diagnostics;
using System.Net;
using System.Net.Http.Json;
using System.Text.Json;
using BridgeControl.Models.Vendor;
using BridgeControl.Tests.Infrastructure;
using Microsoft.EntityFrameworkCore;
using Xunit.Abstractions;
using Agent = BridgeControl.Models.AgentModels.Agent;

namespace BridgeControl.Tests.Integration;

/// <summary>Reproducible correctness workload with measurements, deliberately without invented
timing acceptance thresholds.</summary>

```

```

public sealed class CapacityBaselineTests(ITestOutputHelper output) : InstallationStateTest
{
    /// <summary>
    /// Records observed capacity for bounded queries, log ingestion, artifact streaming, and
    configuration recalculation.
    /// </summary>
    [Fact, Trait("Category", "CapacityBaseline")]
    public async Task
BoundedQueriesIngestionStreamingAndRecalculation_RecordObservedCapacity()
    {
        StateFixture state = await StateAsync(); Guid secondCustomer = await
CustomerAsync("Second capacity customer");
        await using (var db = await OpenVendorAsync(VendorId))
        {
            for (int i = 1; i < 100; i++)
            {
                db.Installations.Add(new Installation
                {
                    CustomerId = i < 50 ? state.Customer : secondCustomer,
                    ProductId = state.Product,
                    AgentId = i < 50 ? state.Agent : null,
                    Name = "Capacity " + i,
                    RegistrationName = i < 50 ? "Capacity " + i : null,
                    RegisteredAt = DateTimeOffset.UtcNow
                });
            }
            await db.SaveChangesAsync();
        }
        await DrainAsync();
        using var large = await
SetAsync($"api/v1/installations/{state.Installation}/configuration", state.Version, new { label
= new string('x', 512 * 1024) }); Assert.Equal(HttpStatusCode.OK, large.StatusCode);
        Guid configuration = await GenerateAsync(state.Installation, state.Version);
        Guid artifact; long artifactBytes;
        await using (var db = await OpenVendorAsync(VendorId))
        { var revision = await db.EffectiveConfigurationRevisions.Include(x =>
x.FileArtifact).SingleAsync(x => x.Id == configuration); artifact = revision.FileArtifactId;
artifactBytes = revision.FileArtifact.SizeBytes; }
        using var desired = await SetDesiredAsync(state, configuration: configuration);
        Assert.Equal(HttpStatusCode.OK, desired.StatusCode);
        using var actual = await ReportAsync(state, "NotInstalled",
DateTimeOffset.UtcNow); Assert.Equal(HttpStatusCode.OK, actual.StatusCode); await
ReconcileAsync();
        await MeasureAsync("installation-page", 20, async i =>
        {
            using var response = await SendAsync(HttpMethod.Get,
$"api/v1/installations?page={i % 4 + 1}&pageSize=25"); Assert.Equal(HttpStatusCode.OK,
response.StatusCode);
            var body = await ReadAsync(response); Assert.Equal(100,
body.GetProperty("totalCount").GetInt32()); Assert.Equal(25,
body.GetProperty("items").GetArrayLength());
        });
        await MeasureAsync("monitoring-page", 20, async _ =>
        {
            using var response = await SendAsync(HttpMethod.Get,
"/api/v1/monitoring/overview?pageSize=25"); Assert.Equal(HttpStatusCode.OK, response.StatusCode);
            var body = await ReadAsync(response); Assert.Equal(100,
body.GetProperty("totalInstallations").GetInt32()); Assert.Equal(25,
body.GetProperty("installations").GetProperty("items").GetArrayLength());
        });
        await MeasureAsync("heartbeat-ingestion", 10, async _ =>

```

```

        { using var response = await SendAsync(HttpMethod.Post, "/api/v1/agent/heartbeat",
new { AgentVersion = "capacity-fixture", Status = "Online", ReportedAt = DateTimeOffset.UtcNow },
state.Token); Assert.Equal(HttpStatusCode.NoContent, response.StatusCode); });
        await MeasureAsync("log-batch-10-entries", 5, async batch =>
        {
            var entries = Enumerable.Range(0, 10).Select(i => new { InstallationId =
state.Installation, Severity = "Information", Source = "capacity-fixture", Message = $"Sample
{batch}/{i}", CreatedAt = DateTimeOffset.UtcNow }).ToArray();
            using var response = await SendAsync(HttpMethod.Post, "/api/v1/agent/logs",
new { Entries = entries }, state.Token); Assert.Equal(HttpStatusCode.NoContent,
response.StatusCode);
        });
        await MeasureAsync("configuration-download-512KiB", 10, async _ =>
        { using var response = await SendAsync(HttpMethod.Get,
$"/api/v1/agent/artifacts/{artifact}/content", token: state.Token);
Assert.Equal(HttpStatusCode.OK, response.StatusCode); Assert.Equal(artifactBytes, (await
response.Content.ReadAsByteArrayAsync()).LongLength); });
        await MeasureAsync("pending-deployment-poll", 20, async _ =>
        { using var response = await SendAsync(HttpMethod.Get,
"/api/v1/agent/deployments?pageSize=25", token: state.Token); Assert.Equal(HttpStatusCode.OK,
response.StatusCode); Assert.Single((await ReadAsync(response)).EnumerateArray()); });
        Guid otherVendor = await CreateVendorAsync("Capacity isolation vendor"); string
invitation = await InviteAsync(otherVendor);
        using var activation = await Client.PostAsJsonAsync("/api/v1/auth/activate", new {
Token = invitation, Password }); Assert.Equal(HttpStatusCode.NoContent, activation.StatusCode);
        using var login = await Client.PostAsJsonAsync("/api/v1/auth/login", new {
VendorName = "Capacity isolation vendor", Email = "invited@example.test", Password });
Assert.Equal(HttpStatusCode.OK, login.StatusCode);
        string otherToken = (await
ReadAsync(login)).GetProperty("accessToken").GetString();
        await MeasureAsync("four-concurrent-cross-vendor-queries", 5, async _ =>
        {
            await Task.WhenAll(Enumerable.Range(0, 4).Select(async n =>
            {
                using var response = await SendAsync(HttpMethod.Get,
"/api/v1/installations?pageSize=25", token: n % 2 == 0 ? AdminToken : otherToken);
Assert.Equal(HttpStatusCode.OK, response.StatusCode);
                Assert.Equal(n % 2 == 0 ? 100 : 0, (await
ReadAsync(response)).GetProperty("totalCount").GetInt32());
            }));
        });
        await DrainAsync(); Dictionary<Guid, Guid?> untouched, affected;
        await using (var db = await OpenVendorAsync(VendorId))
        {
            untouched = await db.Installations.Where(x => x.CustomerId ==
secondCustomer).ToDictionaryAsync(x => x.Id, x => x.CurrentEffectiveConfigurationRevisionId);
            affected = await db.Installations.Where(x => x.CustomerId ==
state.Customer).ToDictionaryAsync(x => x.Id, x => x.CurrentEffectiveConfigurationRevisionId);
        }
        Stopwatch recalculate = Stopwatch.StartNew();
        using var changed = await
SetAsync($"/api/v1/customers/{state.Customer}/products/{state.Product}/configuration",
state.Version, new { port = 8123 }); Assert.Equal(HttpStatusCode.OK, changed.StatusCode);
        await DrainAsync();
        await using (var db = await OpenVendorAsync(VendorId))
        {
            var after = await db.Installations.Where(x => x.CustomerId ==
secondCustomer).ToDictionaryAsync(x => x.Id, x => x.CurrentEffectiveConfigurationRevisionId);
            Assert.Equal(untouched.OrderBy(x => x.Key), after.OrderBy(x => x.Key));
            var changedRevisions = await db.Installations.Where(x => x.CustomerId ==
state.Customer).ToDictionaryAsync(x => x.Id, x => x.CurrentEffectiveConfigurationRevisionId);

```

```

        Assert.Equal(50, changedRevisions.Count(x => x.Value is not null && x.Value
!= affected[x.Key]));
        Assert.False(await db.EffectiveConfigurationWork.AnyAsync(x => x.Status ==
"Pending"));
    }
    output.WriteLine(JsonSerializer.Serialize(new { measurement =
"affected-only-recalculation", affectedInstallations = 50, unchangedInstallations = 50, elapsedMs
= recalculate.Elapsed.TotalMilliseconds }));
    output.WriteLine($"Sample topology: two vendors, two customers/100 installations
in the measured vendor, one Agent, 512 KiB configuration; archive bytes={artifactBytes}.
In-memory HTTP + LocalDB, not a production SLA or network benchmark.");
}
/// <summary>
/// Measures a test workload and records its observed execution time.
/// </summary>
private async Task MeasureAsync(string name, int count, Func<int, Task> action)
{
    List<double> samples = []; Stopwatch total = Stopwatch.StartNew();
    for (int i = 0; i < count; i++) { Stopwatch timer = Stopwatch.StartNew(); await
action(i); samples.Add(timer.Elapsed.TotalMilliseconds); }
    double elapsed = total.Elapsed.TotalSeconds; samples.Sort();
    output.WriteLine(JsonSerializer.Serialize(new { measurement = name, requests =
count, p50Ms = samples[(count - 1) / 2], p95Ms = samples[(int)Math.Ceiling(count * .95) - 1],
operationsPerSecond = count / elapsed }));
}
}

```

Integration/ConfigurationAdministrationTests.cs

```

using System.Net;
using System.Text;
using System.Text.Json;
using System.Security.Cryptography;
using BridgeControl.Models.Vendor;
using BridgeControl.Tests.Infrastructure;
using Microsoft.Data.SqlClient;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Infrastructure;
using Microsoft.EntityFrameworkCore.Migrations;

namespace BridgeControl.Tests.Integration;

/// <summary>
/// Verifies configuration administration behavior and regression scenarios.
/// </summary>
[Trait("Category", "Integration")]
public class ConfigurationAdministrationTests : ConfigurationTest
{
    /// <summary>
    /// Verifies that configuration levels preserve history and reuse canonical values.
    /// </summary>
    [Theory]
    [InlineData("product")]
    [InlineData("customer")]
    [InlineData("installation")]
    public async Task ConfigurationLevelsPreserveHistoryAndReuseCanonicalValues(string level)
    {
        await PrepareAsync(); var (product, version) = await CatalogAsync(); Guid customer
= await CustomerAsync(); Guid installation = await InstallationAsync(customer, product);
    }
}

```

```

        string path = level switch { "product" =>
$/api/v1/products/{product}/configuration", "customer" =>
$/api/v1/customers/{customer}/products/{product}/configuration", _ =>
$/api/v1/installations/{installation}/configuration" };
        using var missing = await SendAsync(HttpMethod.Get, path);
Assert.Equal(HttpStatusCode.NotFound, missing.StatusCode);
        using var emptyHistory = await SendAsync(HttpMethod.Get, path + "/revisions");
Assert.Equal(0, (await ReadAsync(emptyHistory)).GetProperty("totalCount").GetInt32());
        using var initial = await SetAsync(path, version, new { Port = 80, Label = "
exact ", Tags = new string?[] { "one", null } }); Assert.Equal(HttpStatusCode.OK,
initial.StatusCode);
        var first = await ReadAsync(initial); Guid id = first.GetProperty("id").GetGuid();
Assert.Equal(new[] { "createdAt", "createdByUserId", "id", "productVersionId",
"revisionNumber", "values" }, first.EnumerateObject().Select(x => x.Name).Order());
Assert.Equal(1, first.GetProperty("revisionNumber").GetInt32()); Assert.Equal("
exact ", first.GetProperty("values").GetProperty("label").GetString());
        using var repeated = await SetAsync(path, version,
JsonDocument.Parse("""{"tags":["one",null],"label":" exact
", "port":801, "unused":null}""").RootElement);
        Assert.Equal(HttpStatusCode.OK, repeated.StatusCode); Assert.Equal(id, (await
ReadAsync(repeated)).GetProperty("id").GetGuid());
        using var changed = await SetAsync(path, version, new { Port = 443 });
Assert.Equal(HttpStatusCode.OK, changed.StatusCode);
        Assert.Equal(2, (await
ReadAsync(changed)).GetProperty("revisionNumber").GetInt32());
        using var history = await SendAsync(HttpMethod.Get, path +
"/revisions?page=2&pageSize=1"); var page = await ReadAsync(history);
Assert.Equal(2, page.GetProperty("totalCount").GetInt32()); Assert.Equal(id,
page.GetProperty("items")[0].GetProperty("id").GetGuid());
        using var current = await SendAsync(HttpMethod.Get, path); Assert.Equal(443,
(await ReadAsync(current)).GetProperty("values").GetProperty("port").GetInt32());
        await using var db = await OpenVendorAsync(VendorId);
        Assert.Equal(2, await db.ConfigurationRevisions.CountAsync()); Assert.Equal(2,
await db.ConfigurationSourceChanges.CountAsync());
        Assert.True(await db.ConfigurationRevisions.AllAsync(x => x.ValidationStatus ==
"Valid" && x.SchemaArtifactId != null));
        Assert.Equal(2, await db.VendorAuditLogs.CountAsync(x => x.Action ==
"ConfigurationRevised"));
        var file = await db.FileArtifacts.SingleAsync(x => x.Id ==
db.ConfigurationRevisions.Where(r => r.Id == id).Select(r => r.FileArtifactId).First());
        byte[] bytes = await File.ReadAllBytesAsync(Path.Combine(ArchiveRoot, "blobs",
file.StorageIdentifier + ".blob"));
        Assert.Equal(file.Sha256, Convert.ToHexStringLower(SHA256.HashData(bytes)));
Assert.DoesNotContain("storageIdentifier", first.GetRawText());
        Assert.Equal("application/json", file.ContentType);
        using var invalidPage = await SendAsync(HttpMethod.Get, path +
"/revisions?page=0"); Assert.Equal(HttpStatusCode.BadRequest, invalidPage.StatusCode);
    }
    /// <summary>
    /// Verifies that version specific retries do not switch current or copy other versions.
    /// </summary>
    [Fact]
    public async Task VersionSpecificRetriesDoNotSwitchCurrentOrCopyOtherVersions()
    {
        await PrepareAsync(); var (product, firstVersion) = await CatalogAsync(); Guid
secondVersion = await VersionAsync(product, "2"); string path =
$/api/v1/products/{product}/configuration";
        using var first = await SetAsync(path, firstVersion, new { Port = 80 }); Guid
firstId = (await ReadAsync(first)).GetProperty("id").GetGuid();
        using var second = await SetAsync(path, secondVersion, new { Port = 443 }); Guid
secondId = (await ReadAsync(second)).GetProperty("id").GetGuid();

```

```

        using var retry = await SetAsync(path, firstVersion, new { Port = 80 });
Assert.Equal(firstId, (await ReadAsync(retry)).GetProperty("id").GetGuid());
        using var current = await SendAsync(HttpMethod.Get, path); Assert.Equal(secondId,
(await ReadAsync(current)).GetProperty("id").GetGuid());
        using var installer = await UploadAsync(product, firstVersion, "installer",
MsiFixture.Create(Factory.PrivateRoot)); Assert.Equal(HttpStatusCode.OK, installer.StatusCode);
        using var published = await PublishAsync(product, firstVersion);
Assert.Equal(HttpStatusCode.OK, published.StatusCode);
        await using var db = await OpenVendorAsync(VendorId); Assert.Equal(firstId, (await
db.ProductVersions.SingleAsync(x => x.Id == firstVersion)).PublishedDefaultRevisionId);
        using var frozen = await SetAsync(path, firstVersion, new { Port = 8080 });
Assert.Equal(HttpStatusCode.Conflict, frozen.StatusCode);
        using var sameFrozen = await SetAsync(path, firstVersion, new { Port = 80 });
Assert.Equal(HttpStatusCode.OK, sameFrozen.StatusCode);
        using var editable = await SetAsync(path, secondVersion, new { Port = 8080 });
Assert.Equal(HttpStatusCode.OK, editable.StatusCode);
    }
    /// <summary>
    /// Verifies that invalid values and wrong relationships cannot replace current revision.
    /// </summary>
    [Fact]
    public async Task InvalidValuesAndWrongRelationshipsCannotReplaceCurrentRevision()
    {
        await PrepareAsync(); var (product, version) = await CatalogAsync(); var (_,
wrongVersion) = await CatalogAsync(); string path = $"/api/v1/products/{product}/configuration";
        using var valid = await SetAsync(path, version, new { Port = 80 }); Guid original
= (await ReadAsync(valid)).GetProperty("id").GetGuid();
        foreach (object input in new object[] { new { Port = 0 }, new { Port = "80" }, new
{ Unknown = true }, Array.Empty<string>(),
JsonDocument.Parse("{\"port\":\"80\",\"port\":\"90\"").RootElement })
        { using var bad = await SetAsync(path, version, input);
Assert.Equal(HttpStatusCode.BadRequest, bad.StatusCode); }
        using var crossed = await SetAsync(path, wrongVersion, new { Port = 80 });
Assert.Equal(HttpStatusCode.NotFound, crossed.StatusCode);
        using var overposted = await SendAsync(HttpMethod.Put, path, new {
ProductVersionId = version, Values = new { Port = 80 }, RevisionNumber = 100 });
Assert.Equal(HttpStatusCode.BadRequest, overposted.StatusCode);
        using var missingValue = await SendAsync(HttpMethod.Put, path, new {
ProductVersionId = version }); Assert.Equal(HttpStatusCode.BadRequest, missingValue.StatusCode);
        using var noSchema = await SendAsync(HttpMethod.Delete,
$"/api/v1/products/{product}/versions/{version}/schema"); Assert.Equal(HttpStatusCode.NoContent,
noSchema.StatusCode);
        using var blocked = await SetAsync(path, version, new { Port = 443 });
Assert.Equal(HttpStatusCode.Conflict, blocked.StatusCode);
        using var current = await SendAsync(HttpMethod.Get, path); Assert.Equal(original,
(await ReadAsync(current)).GetProperty("id").GetGuid());
        await using var db = await OpenVendorAsync(VendorId); Assert.Equal(1, await
db.ConfigurationRevisions.CountAsync());
    }
    /// <summary>
    /// Verifies that required properties are not relaxed for overrides and empty put is
schema dependent.
    /// </summary>
    [Fact]
    public async Task
RequiredPropertiesAreNotRelaxedForOverridesAndEmptyPutIsSchemaDependent()
    {
        await PrepareAsync(); string required =
Schema.Replace($"\"additionalProperties\":false",
$"\"required\":[\"port\"],\"additionalProperties\":false");

```

```

        var (product, version) = await CatalogAsync(schema: required); Guid customer =
await CustomerAsync(); string path =
$"/api/v1/customers/{customer}/products/{product}/configuration";
        using var omitted = await SetAsync(path, version, new { Label = "partial" });
Assert.Equal(HttpStatusCode.BadRequest, omitted.StatusCode);
        using var nulled = await SetAsync(path, version, new { Port = (int?)null });
Assert.Equal(HttpStatusCode.BadRequest, nulled.StatusCode);
        using var valid = await SetAsync(path, version, new { Port = 80 });
Assert.Equal(HttpStatusCode.OK, valid.StatusCode);
        Guid optional = await VersionAsync(product, "optional"); using var empty = await
SetAsync(path, optional, new { }); Assert.Equal(HttpStatusCode.OK, empty.StatusCode);
    }
    /// <summary>
    /// Verifies that customer technician and installation assignments are enforced.
    /// </summary>
    [Fact]
    public async Task CustomerTechnicianAndInstallationAssignmentsAreEnforced()
    {
        await PrepareAsync(); var (product, version) = await CatalogAsync(); Guid first =
await CustomerAsync("First"), second = await CustomerAsync("Second");
        Guid installation = await InstallationAsync(first, product), otherInstallation =
await InstallationAsync(second, product);
        string own = $"/api/v1/customers/{first}/products/{product}/configuration", other
= $"/api/v1/customers/{second}/products/{product}/configuration";
        string installed = $"/api/v1/installations/{installation}/configuration";
        var admin = await UserAsync("customeradmin@example.test", "CustomerAdministrator",
first);
        using var allowed = await SetAsync(own, version, new { Port = 80 }, admin.Token);
Assert.Equal(HttpStatusCode.OK, allowed.StatusCode);
        using var forbidden = await SetAsync(other, version, new { Port = 80 },
admin.Token); Assert.Equal(HttpStatusCode.NotFound, forbidden.StatusCode);
        using var defaultDenied = await
SetAsync($"/api/v1/products/{product}/configuration", version, new { Port = 80 }, admin.Token);
Assert.Equal(HttpStatusCode.Forbidden, defaultDenied.StatusCode);
        using var installationAllowed = await SetAsync(installed, version, new { Port =
443 }, admin.Token); Assert.Equal(HttpStatusCode.OK, installationAllowed.StatusCode);
        using var installationDenied = await
SetAsync($"/api/v1/installations/{otherInstallation}/configuration", version, new { Port = 443 },
admin.Token); Assert.Equal(HttpStatusCode.NotFound, installationDenied.StatusCode);
        var tech = await UserAsync("tech@example.test", "VendorTechnician");
        using var noAssignment = await SetAsync(own, version, new { Port = 443 },
tech.Token); Assert.Equal(HttpStatusCode.NotFound, noAssignment.StatusCode);
        using var assign = await SendAsync(HttpMethod.Put,
$"/api/v1/users/{tech.Id}/customer-access", new { CustomerIds = new[] { first } });
Assert.Equal(HttpStatusCode.NoContent, assign.StatusCode);
        using var assigned = await SetAsync(own, version, new { Port = 443 }, tech.Token);
Assert.Equal(HttpStatusCode.OK, assigned.StatusCode);
        var ordinary = await UserAsync("ordinary@example.test", "CustomerUser", first);
        using var unassignedRead = await SendAsync(HttpMethod.Get, installed, token:
ordinary.Token); Assert.Equal(HttpStatusCode.NotFound, unassignedRead.StatusCode);
        using var grant = await SendAsync(HttpMethod.Put,
$"/api/v1/users/{ordinary.Id}/installation-access", new { InstallationIds = new[] { installation
} }); Assert.Equal(HttpStatusCode.NoContent, grant.StatusCode);
        using var read = await SendAsync(HttpMethod.Get, installed, token:
ordinary.Token); Assert.Equal(HttpStatusCode.OK, read.StatusCode);
        using var readOnly = await SetAsync(installed, version, new { Port = 90 },
ordinary.Token); Assert.Equal(HttpStatusCode.Forbidden, readOnly.StatusCode);
        using var revoke = await SendAsync(HttpMethod.Put,
$"/api/v1/users/{tech.Id}/customer-access", new { CustomerIds = Array.Empty<Guid>() });
Assert.Equal(HttpStatusCode.NoContent, revoke.StatusCode);
        using var deniedAgain = await SendAsync(HttpMethod.Get, own, token: tech.Token);
Assert.Equal(HttpStatusCode.NotFound, deniedAgain.StatusCode);
    }

```

```

        using var disabled = await SendAsync(HttpMethod.Post,
$"/api/v1/customers/{first}/deactivate"); Assert.Equal(HttpStatusCode.NoContent,
disabled.StatusCode);
        using var inactiveWrite = await SetAsync(installed, version, new { Port = 123 });
Assert.Equal(HttpStatusCode.NotFound, inactiveWrite.StatusCode);
        using var history = await SendAsync(HttpMethod.Get, installed + "/revisions");
Assert.Equal(HttpStatusCode.OK, history.StatusCode);
    }
    /// <summary>
    /// Verifies that audit and outbox failures leave prior current and history intact.
    /// </summary>
    [Fact]
    public async Task AuditAndOutboxFailuresLeavePriorCurrentAndHistoryIntact()
    {
        await PrepareAsync(); var (product, version) = await CatalogAsync(); string path =
$"/api/v1/products/{product}/configuration";
        using var original = await SetAsync(path, version, new { Port = 80 }); Guid id =
(await ReadAsync(original)).GetProperty("id").GetGuid();
        await using var db = await OpenVendorAsync(VendorId);
        foreach (string table in new[] { "VendorAuditLogs", "ConfigurationSourceChanges"
})
        {
            string sql = table == "VendorAuditLogs"
                ? "CREATE TRIGGER TestSourceFailure ON VendorAuditLogs AFTER INSERT
AS THROW 51021, 'Test failure.', 1;"
                : "CREATE TRIGGER TestSourceFailure ON ConfigurationSourceChanges
AFTER INSERT AS THROW 51021, 'Test failure.', 1;";
            await db.Database.ExecuteSqlRawAsync(sql);
            using var failed = await SetAsync(path, version, new { Port = 443 });
Assert.Equal(HttpStatusCode.InternalServerError, failed.StatusCode);
            using var current = await SendAsync(HttpMethod.Get, path); Assert.Equal(id,
(await ReadAsync(current)).GetProperty("id").GetGuid());
            Assert.Equal(1, await db.ConfigurationRevisions.CountAsync());
Assert.Equal(1, await db.ConfigurationSourceChanges.CountAsync());
            Assert.Equal(1, (await
db.ConfigurationProfiles.AsNoTracking().SingleAsync()).LastRevisionNumber);
            await db.Database.ExecuteSqlRawAsync("DROP TRIGGER TestSourceFailure;");
        }
        Assert.Empty(Directory.GetFiles(Path.Combine(ArchiveRoot, "staging")));
    }
    /// <summary>
    /// Verifies that concurrent identical puts allocate once and stale profiles cannot
    overwrite history.
    /// </summary>
    [Fact]
    public async Task
ConcurrentIdenticalPutsAllocateOnceAndStaleProfilesCannotOverwriteHistory()
    {
        await PrepareAsync(); var (product, version) = await CatalogAsync(); string path =
$"/api/v1/products/{product}/configuration";
        var results = await Task.WhenAll(Enumerable.Range(0, 3).Select(_ => SetAsync(path,
version, new { Port = 80 })));
        List<Guid> ids = [];
        foreach (var response in results) { Assert.Equal(HttpStatusCode.OK,
response.StatusCode); ids.Add((await ReadAsync(response)).GetProperty("id").GetGuid());
response.Dispose(); }
        Assert.Single(ids.Distinct());
        await using var stale = await OpenVendorAsync(VendorId); var profile = await
stale.ConfigurationProfiles.SingleAsync();
        using var newer = await SetAsync(path, version, new { Port = 443 });
Assert.Equal(HttpStatusCode.OK, newer.StatusCode);

```

```

        profile.LastRevisionNumber = 99; await
Assert.ThrowsAsync<DbUpdateConcurrencyException>(() => stale.SaveChangesAsync());
        await Assert.ThrowsAsync<SqlException>(() =>
stale.Database.ExecuteSqlRawAsync("UPDATE ConfigurationRevisions SET RevisionNumber = 100"));
        await Assert.ThrowsAsync<SqlException>(() =>
stale.Database.ExecuteSqlRawAsync("DELETE FROM ConfigurationRevisions"));
        Assert.Equal(2, await stale.ConfigurationRevisions.CountAsync());
    }
    /// <summary>
    /// Verifies that other vendor ids and platform credentials do not access configuration.
    /// </summary>
    [Fact]
    public async Task OtherVendorIdsAndPlatformCredentialsDoNotAccessConfiguration()
    {
        await PrepareAsync(); var (product, version) = await CatalogAsync();
        Guid other = await CreateVendorAsync("Other vendor"); await using var otherDb =
await OpenVendorAsync(other);
        Product secret = new() { Id = Guid.NewGuid(), Name = "Secret", Status = "Active"
}; otherDb.Products.Add(secret); await otherDb.SaveChangesAsync();
        using var missing = await SendAsync(HttpMethod.Get,
$/api/v1/products/{secret.Id}/configuration/revisions"); Assert.Equal(HttpStatusCode.NotFound,
missing.StatusCode);
        using var crossed = await SetAsync($"/api/v1/products/{secret.Id}/configuration",
version, new { Port = 80 }); Assert.Equal(HttpStatusCode.NotFound, crossed.StatusCode);
        using var platform = await
Client.GetAsync($"/api/v1/products/{product}/configuration");
        Assert.Equal(HttpStatusCode.Unauthorized, platform.StatusCode);
    }
    /// <summary>
    /// Verifies that migration retains legacy values and continues revision numbers.
    /// </summary>
    [Fact]
    public async Task MigrationRetainsLegacyValuesAndContinuesRevisionNumbers()
    {
        await PrepareAsync(); var (product, version) = await CatalogAsync(); await using
var db = await OpenVendorAsync(VendorId);
        Guid author = await db.UserAccounts.Select(x => x.Id).SingleAsync(); var migrator
= db.GetService<IMigrator>();
        await migrator.MigrateAsync(db.Database.GetMigrations().Single(x =>
x.EndsWith("_AddProductArtifactAdministration", StringComparison.Ordinal)));
        Guid profile = Guid.NewGuid(), revision = Guid.NewGuid(), file = Guid.NewGuid();
string storage = Guid.NewGuid().ToString("N");
        byte[] bytes = Encoding.UTF8.GetBytes($"{"\port\:80}"); string hash =
Convert.ToHexStringLower(SHA256.HashData(bytes));
        await File.WriteAllBytesAsync(Path.Combine(ArchiveRoot, "blobs", storage +
".blob"), bytes);
        await db.Database.ExecuteSqlInterpolatedAsync($"INSERT INTO ConfigurationProfiles
(Id,ProductId,CreatedAt) VALUES ({profile},{product},SYSDATETIMEOFFSET())");
        await db.Database.ExecuteSqlInterpolatedAsync($"INSERT INTO FileArtifacts
(Id,ArtifactType,FileName,StorageIdentifier,ContentType,Sha256,SizeBytes,CreatedAt) VALUES
({file},'Configuration','configuration.json',{storage},'application/json',{hash},{bytes.Length},S
YSDATETIMEOFFSET())");
        await db.Database.ExecuteSqlInterpolatedAsync($"INSERT INTO ConfigurationRevisions
(Id,ConfigurationProfileId,ProductVersionId,FileArtifactId,RevisionNumber,CreatedByUserId,Created
At) VALUES ({revision},{profile},{version},{file},7,{author},SYSDATETIMEOFFSET())");
        await migrator.MigrateAsync();
        var legacy = await db.ConfigurationRevisions.SingleAsync();
        Assert.Equal("LegacyUnverified", legacy.ValidationStatus); Assert.Null(legacy.SchemaArtifactId);
        Assert.Equal(7, (await
db.ConfigurationProfiles.SingleAsync()).LastRevisionNumber);
        using var next = await SetAsync($"/api/v1/products/{product}/configuration",
version, new { Port = 90 }); Assert.Equal(HttpStatusCode.OK, next.StatusCode);

```

```

        Assert.Equal(8, (await ReadAsync(next)).GetProperty("revisionNumber").GetInt32());
        Assert.Equal(2, await db.ConfigurationRevisions.CountAsync());
    }
}

```

Integration/ConfigurationMigrationBoundaryTests.cs

```

using System.Text;
using BridgeControl.Api.Exceptions;
using BridgeControl.DTOs.Vendor;
using BridgeControl.Services.Vendor;
using BridgeControl.Tests.Infrastructure;
using Microsoft.Data.SqlClient;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;

namespace BridgeControl.Tests.Integration;

/// <summary>
/// Verifies configuration migration boundary behavior and regression scenarios.
/// </summary>
public class ConfigurationMigrationBoundaryTests : MigrationTest
{
    /// <summary>
    /// Verifies that changed schema or rules invalidates approval and target validation
    blocks commit.
    /// </summary>
    [Fact]
    public async Task
        ChangedSchemaOrRules_InvalidatesApproval_AndTargetValidationBlocksCommit()
    {
        await PrepareAsync(); var catalog = await CatalogAsync(schema: OpenSchema); Guid
        target = await VersionAsync(catalog.Product, "2", OpenSchema);
        Guid installation = await InstallationAsync(await CustomerAsync(),
        catalog.Product);
        using var source = await
        SetAsync($" /api/v1/installations/{installation}/configuration", catalog.Version, new { port = 123
        }); await GenerateAsync(installation, catalog.Version);
        Guid definition = await DefinitionAsync(catalog.Product, catalog.Version, target);
        var review = await ReviewAsync(installation, target);
        using var replacement = await SendAsync(HttpMethod.Put,
        $" /api/v1/products/{catalog.Product}/configuration-migrations/{definition}", new {
        FromProductVersionId = catalog.Version, ToProductVersionId = target, Rules = new[] { new
        MigrationRuleRequest(1, "Preserve", "/port", null, null, false) } });
        Assert.True(replacement.IsSuccessStatusCode, await
        replacement.Content.ReadAsStringAsync());
        Assert.Equal("migration-preview-stale", (await
        Assert.ThrowsAsync<ApiConflictException>(() => ApplyAsync(review))).Code);
        review = await ReviewAsync(installation, target);
        const string strict =
        ""{"$schema":"https://json-schema.org/draft/2020-12/schema","type":"object","properties":{"port"
        :{"type":"integer","maximum":10}}"";
        using var schema = await UploadAsync(catalog.Product, target, "schema",
        Encoding.UTF8.GetBytes(strict));
        Assert.Equal("migration-preview-stale", (await
        Assert.ThrowsAsync<ApiConflictException>(() => ApplyAsync(review))).Code);
        var invalid = await ReviewAsync(installation, target);
        Assert.NotEmpty(Plan(invalid).Errors); Assert.Equal(123,
        Plan(invalid).Values.GetProperty("port").GetInt32());
        await Assert.ThrowsAsync<ApiConflictException>(() => ApplyAsync(invalid));
    }
}

```

```

        using var badSourceSchema = await UploadAsync(catalog.Product, catalog.Version,
"schema", Encoding.UTF8.GetBytes(strict));
        await Assert.ThrowsAsync<ApiConflictException>(() => ReviewAsync(installation,
target));
    }
    /// <summary>
    /// Verifies that review and rule evidence is immutable and expiry and foreign user are
rejected.
    /// </summary>
    [Fact]
    public async Task ReviewAndRuleEvidence_IsImmutable_AndExpiryAndForeignUserAreRejected()
    {
        await PrepareAsync(); var catalog = await CatalogAsync(schema: OpenSchema); Guid
target = await VersionAsync(catalog.Product, "2", OpenSchema);
        Guid customer = await CustomerAsync(), installation = await
InstallationAsync(customer, catalog.Product);
        await GenerateAsync(installation, catalog.Version); await
DefinitionAsync(catalog.Product, catalog.Version, target);
        var review = await ReviewAsync(installation, target);
        await using var db = await OpenVendorAsync(VendorId);
        await Assert.ThrowsAsync<SqlException>(() =>
db.Database.ExecuteSqlRawAsync("UPDATE MigrationDefinitionRevisions SET RulesJson = '[]'"));
        await Assert.ThrowsAsync<SqlException>(() =>
db.Database.ExecuteSqlRawAsync("UPDATE ConfigurationMigrationReviews SET PlanJson = '{}{}'"));
        await Assert.ThrowsAsync<SqlException>(() =>
db.Database.ExecuteSqlRawAsync("DELETE FROM ConfigurationMigrationReviews"));
        var customerUser = await UserAsync("migration-customer@example.test",
"CustomerAdministrator", customer);
        var session = await db.AuthenticationSessions.Where(x => x.AccountId ==
customerUser.Id && x.RevokedAt == null).SingleAsync();
        await using var scope = Factory.Services.CreateAsyncScope(); var service =
scope.ServiceProvider.GetRequiredService<ConfigurationMigrationService>();
        var denied = await Assert.ThrowsAsync<ApiConflictException>(() =>
service.ApplyReviewedAsync(new(VendorId, customerUser.Id, session.Id), installation, target,
review.Id, [], default));
        Assert.Equal("migration-preview-required", denied.Code);
        // An old snapshot can be inserted by the trusted test fixture, but never mutated
after review.
        db.ConfigurationMigrationReviews.Add(new() { UserId = review.UserId,
InstallationId = installation, TargetProductVersionId = target, CreatedAt =
DateTimeOffset.UtcNow.AddHours(-1), PlanJson = review.PlanJson, Fingerprint = review.Fingerprint
});
        await db.SaveChangesAsync(); var expired = await
db.ConfigurationMigrationReviews.SingleAsync(x => x.CreatedAt <
DateTimeOffset.UtcNow.AddMinutes(-30));
        Assert.Equal("migration-preview-expired", (await
Assert.ThrowsAsync<ApiConflictException>(() => ApplyAsync(expired)).Code);
        await ApplyAsync(review); await Assert.ThrowsAsync<SqlException>(() =>
db.Database.ExecuteSqlRawAsync("UPDATE ConfigurationMigrationReviews SET ApprovedRuleIdsJson =
'[]' WHERE AppliedAt IS NOT NULL"));
    }
    /// <summary>
    /// Verifies that existing target overrides are never silently overwritten and malformed
cycle is rejected.
    /// </summary>
    [Fact]
    public async Task
ExistingTargetOverrides_AreNeverSilentlyOverwritten_AndMalformedCycleIsRejected()
    {
        await PrepareAsync(); var catalog = await CatalogAsync(schema: OpenSchema); Guid
target = await VersionAsync(catalog.Product, "2", OpenSchema);

```

```

        Guid installation = await InstallationAsync(await CustomerAsync(),
catalog.Product);
        using var source = await
SetAsync($"api/v1/installations/{installation}/configuration", catalog.Version, new { port = 1
});
        using var destination = await
SetAsync($"api/v1/installations/{installation}/configuration", target, new { port = 2 });
        await GenerateAsync(installation, catalog.Version); await
DefinitionAsync(catalog.Product, catalog.Version, target);
        var review = await ReviewAsync(installation, target);
Assert.Contains(Plan(review).Errors, x => x.Contains("existing target overrides",
StringComparison.Ordinal)); await Assert.ThrowsAsync<ApiConflictException>(() =>
ApplyAsync(review));
        await using var db = await OpenVendorAsync(VendorId);
        db.ConfigurationMigrations.Add(new() { Id = Guid.NewGuid(), FromProductVersionId =
target, ToProductVersionId = catalog.Version, CreatedByUserId = (await ActorAsync()).UserId,
CreatedAt = DateTimeOffset.UtcNow }); await db.SaveChangesAsync();
        Assert.Equal("migration-cycle", (await Assert.ThrowsAsync<ApiConflictException>(()
=> ReviewAsync(installation, target))).Code);
    }
}

```

Integration/ConfigurationMigrationDefinitionTests.cs

```

using System.Net;
using BridgeControl.DTOs.Vendor;
using BridgeControl.Tests.Infrastructure;
using Microsoft.EntityFrameworkCore;

namespace BridgeControl.Tests.Integration;

/// <summary>
/// Verifies configuration migration definition behavior and regression scenarios.
/// </summary>
public class ConfigurationMigrationDefinitionTests : EffectiveConfigurationTest
{
    /// <summary>
    /// Verifies that definitions are scoped ordered replaceable and retain immutable
    snapshots.
    /// </summary>
    [Fact]
    public async Task Definitions_AreScopedOrderedReplaceableAndRetainImmutableSnapshots()
    {
        await PrepareAsync(); var catalog = await CatalogAsync(); Guid target = await
VersionAsync(catalog.Product, "2");
        string path = $"api/v1/products/{catalog.Product}/configuration-migrations";
        var rules = new[] { new MigrationRuleRequest(2, "Preserve", "/port", null, null,
false), new MigrationRuleRequest(1, "Delete", "/label", null, null, true) };
        using var created = await SendAsync(HttpMethod.Post, path, new {
FromProductVersionId = catalog.Version, ToProductVersionId = target, Rules = rules });
        Assert.Equal(HttpStatusCode.Created, created.StatusCode); var body = await
ReadAsync(created); Guid id = body.GetProperty("id").GetGuid();
        Assert.Equal(1, body.GetProperty("rules")[0].GetProperty("sequence").GetInt32());
        using var list = await SendAsync(HttpMethod.Get, path); Assert.Equal(1, (await
ReadAsync(list)).GetProperty("totalCount").GetInt32());
        using var detail = await SendAsync(HttpMethod.Get, path + "/" + id);
        Assert.Equal(HttpStatusCode.OK, detail.StatusCode);
        using var unchanged = await SendAsync(HttpMethod.Put, path + "/" + id, new {
FromProductVersionId = catalog.Version, ToProductVersionId = target, Rules = rules });
        Assert.Equal(HttpStatusCode.OK, unchanged.StatusCode);
    }
}

```

```

        using var replaced = await SendAsync(HttpMethod.Put, path + "/" + id, new {
            FromProductVersionId = catalog.Version, ToProductVersionId = target, Rules =
            Array.Empty<MigrationRuleRequest>() }); Assert.Equal(HttpStatusCode.OK, replaced.StatusCode);
        await using var db = await OpenVendorAsync(VendorId); Assert.Equal(2, await
            db.MigrationDefinitionRevisions.CountAsync()); Assert.Empty(await
            db.MigrationRules.ToListAsync());
        Assert.Contains("Delete", (await db.MigrationDefinitionRevisions.OrderBy(x =>
            x.RevisionNumber).FirstAsync()).RulesJson);
        using var foreign = await SendAsync(HttpMethod.Get,
            $"/api/v1/products/{Guid.NewGuid()}/configuration-migrations/{id}");
        Assert.Equal(HttpStatusCode.NotFound, foreign.StatusCode);
        var technician = await UserAsync("migration-tech@example.test",
            "VendorTechnician");
        using var denied = await SendAsync(HttpMethod.Put, path + "/" + id, new {
            FromProductVersionId = catalog.Version, ToProductVersionId = target, Rules = rules },
            technician.Token); Assert.Equal(HttpStatusCode.Forbidden, denied.StatusCode);
    }
    /// <summary>
    /// Verifies that definitions reject cross product duplicate cycles and invalid rules.
    /// </summary>
    [Fact]
    public async Task Definitions_RejectCrossProductDuplicateCyclesAndInvalidRules()
    {
        await PrepareAsync(); var catalog = await CatalogAsync(); Guid target = await
        VersionAsync(catalog.Product, "2"); var foreign = await CatalogAsync();
        string path = $"/api/v1/products/{catalog.Product}/configuration-migrations";
        async Task<HttpResponseMessage> Create(Guid from, Guid to, params
        MigrationRuleRequest[] rules) => await SendAsync(HttpMethod.Post, path, new {
            FromProductVersionId = from, ToProductVersionId = to, Rules = rules });
        using var wrong = await Create(catalog.Version, foreign.Version);
        Assert.Equal(HttpStatusCode.NotFound, wrong.StatusCode);
        using var invalid = await Create(catalog.Version, target, new
        MigrationRuleRequest(1, "Delete", "/label", null, null, false));
        Assert.Equal(HttpStatusCode.BadRequest, invalid.StatusCode);
        using var good = await Create(catalog.Version, target);
        Assert.Equal(HttpStatusCode.Created, good.StatusCode);
        using var duplicate = await Create(catalog.Version, target);
        Assert.Equal(HttpStatusCode.Conflict, duplicate.StatusCode);
        using var cycle = await Create(target, catalog.Version);
        Assert.Equal(HttpStatusCode.Conflict, cycle.StatusCode);
    }
}

```

Integration/ConfigurationMigrationExecutionTests.cs

```

using System.Net;
using System.Text.Json;
using BridgeControl.DTOS.Vendor;
using BridgeControl.Tests.Infrastructure;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using BridgeControl.Services.Vendor;
using BridgeControl.Api.Exceptions;

namespace BridgeControl.Tests.Integration;

/// <summary>
/// Verifies configuration migration execution behavior and regression scenarios.
/// </summary>
public class ConfigurationMigrationExecutionTests : MigrationTest

```

```

{
    /// <summary>
    /// Verifies that direct migration uses new defaults and independent layers with atomic
    history and retry.
    /// </summary>
    [Fact]
    public async Task
    DirectMigration_UsesNewDefaultsAndIndependentLayers_WithAtomicHistoryAndRetry()
    {
        await PrepareAsync(); var catalog = await CatalogAsync(schema: OpenSchema); Guid
        target = await VersionAsync(catalog.Product, "2", OpenSchema);
        Guid customer = await CustomerAsync(), installation = await
        InstallationAsync(customer, catalog.Product);
        using var defaults = await
        SetAsync($"api/v1/products/{catalog.Product}/configuration", catalog.Version, new { oldDefault =
        1 });
        using var newDefaults = await
        SetAsync($"api/v1/products/{catalog.Product}/configuration", target, new { newDefault = 2 });
        using var customerSource = await
        SetAsync($"api/v1/customers/{customer}/products/{catalog.Product}/configuration",
        catalog.Version, new { old = "customer", inherited = true });
        using var localSource = await
        SetAsync($"api/v1/installations/{installation}/configuration", catalog.Version, new { old =
        "local" });
        Guid original = await GenerateAsync(installation, catalog.Version);
        await DefinitionAsync(catalog.Product, catalog.Version, target, new
        MigrationRuleRequest(1, "Rename", "/old", "/new", null, false));
        var review = await ReviewAsync(installation, target); var plan = Plan(review);
        Assert.Empty(plan.Errors); Assert.Equal("local",
        plan.Values.GetProperty("new").GetString());
        Assert.True(plan.Values.GetProperty("inherited").GetBoolean());
        Assert.Equal(2, plan.Values.GetProperty("newDefault").GetInt32());
        Assert.False(plan.Values.TryGetProperty("oldDefault", out _));
        await using var db = await OpenVendorAsync(VendorId); Assert.Equal(original,
        (await db.Installations.AsNoTracking().SingleAsync()).CurrentEffectiveConfigurationRevisionId);
        var result = await ApplyAsync(review); var retry = await ApplyAsync(review);
        Assert.Equal(result.RevisionId, retry.RevisionId);
        Assert.Equal(target, result.ProductVersionId); Assert.Equal(3,
        result.SourceRevisionIds.Count);
        var migrated = await db.ConfigurationRevisions.Where(x => x.ProductVersionId ==
        target && x.ConfigurationProfile.InstallationId != null).SingleAsync();
        Assert.Equal(2, migrated.RevisionNumber); Assert.Equal(1, await
        db.VendorAuditLogs.CountAsync(x => x.Action == "ConfigurationMigrationApplied"));
        await DrainAsync(); Assert.Equal(result.RevisionId, (await
        db.Installations.AsNoTracking().SingleAsync()).CurrentEffectiveConfigurationRevisionId);
    }
    /// <summary>
    /// Verifies that chained migration completes every hop and direct path takes precedence.
    /// </summary>
    [Fact]
    public async Task ChainedMigration_CompletesEveryHop_AndDirectPathTakesPrecedence()
    {
        await PrepareAsync(); var catalog = await CatalogAsync(schema: OpenSchema); Guid
        middle = await VersionAsync(catalog.Product, "2", OpenSchema), target = await
        VersionAsync(catalog.Product, "3", OpenSchema);
        Guid installation = await InstallationAsync(await CustomerAsync(),
        catalog.Product);
        using var source = await
        SetAsync($"api/v1/installations/{installation}/configuration", catalog.Version, new { one = 1
        }); await GenerateAsync(installation, catalog.Version);
        var missing = await Assert.ThrowsAsync<ApiConflictException>(() =>
        ReviewAsync(installation, target)); Assert.Equal("migration-path-missing", missing.Code);
    }
}

```

```

        await DefinitionAsync(catalog.Product, catalog.Version, middle, new
MigrationRuleRequest(1, "Move", "/one", "/two", null, false));
        await DefinitionAsync(catalog.Product, middle, target, new MigrationRuleRequest(1,
"Move", "/two", "/three", null, false));
        var chained = await ReviewAsync(installation, target); Assert.Equal(2,
Plan(chained).DefinitionRevisionIds.Count); Assert.Equal(1,
Plan(chained).Values.GetProperty("three").GetInt32());
        await DefinitionAsync(catalog.Product, catalog.Version, target, new
MigrationRuleRequest(1, "Move", "/one", "/direct", null, false));
        var stale = await Assert.ThrowsAsync<ApiConflictException>(() =>
ApplyAsync(chained)); Assert.Equal("migration-preview-stale", stale.Code);
        var direct = await ReviewAsync(installation, target);
Assert.Single(Plan(direct).DefinitionRevisionIds); Assert.Equal(1, (await
ApplyAsync(direct)).Values.GetProperty("direct").GetInt32());
    }
    /// <summary>
    /// Verifies that destruction requires exact approval and audit failure rolls back all
target state.
    /// </summary>
    [Fact]
    public async Task
Destruction_RequiresExactApproval_AndAuditFailureRollsBackAllTargetState()
    {
        await PrepareAsync(); var catalog = await CatalogAsync(schema: OpenSchema); Guid
target = await VersionAsync(catalog.Product, "2", OpenSchema);
        Guid installation = await InstallationAsync(await CustomerAsync(),
catalog.Product);
        using var source = await
SetAsync($"api/v1/installations/{installation}/configuration", catalog.Version, new { obsolete =
"private-value", keep = 1 });
        Guid original = await GenerateAsync(installation, catalog.Version);
        await DefinitionAsync(catalog.Product, catalog.Version, target, new
MigrationRuleRequest(1, "Delete", "/obsolete", null, null, true));
        var review = await ReviewAsync(installation, target); Guid rule =
Assert.Single(Plan(review).Changes).RuleId;
        var rejected = await Assert.ThrowsAsync<ApiConflictException>(() =>
ApplyAsync(review)); Assert.Equal("migration-approval-required", rejected.Code);
        await using var db = await OpenVendorAsync(VendorId);
        await db.Database.ExecuteSqlRawAsync("CREATE TRIGGER FailMigrationAudit ON
VendorAuditLogs AFTER INSERT AS BEGIN IF EXISTS (SELECT 1 FROM inserted WHERE Action =
'ConfigurationMigrationApplied') THROW 51000, 'test failure', 1; END");
        await Assert.ThrowsAsync<DbUpdateException>(() => ApplyAsync(review, rule));
        Assert.Equal(original, (await
db.Installations.AsNoTracking().SingleAsync()).CurrentEffectiveConfigurationRevisionId);
        Assert.False(await db.ConfigurationRevisions.AnyAsync(x => x.ProductVersionId ==
target)); Assert.False(await db.EffectiveConfigurationRevisions.AnyAsync(x => x.ProductVersionId
== target));
        await db.Database.ExecuteSqlRawAsync("DROP TRIGGER FailMigrationAudit");
        var result = await ApplyAsync(review, rule);
        Assert.False(result.Values.TryGetProperty("obsolete", out _));
        var audit = await db.VendorAuditLogs.SingleAsync(x => x.Action ==
"ConfigurationMigrationDestructiveChangeApproved"); Assert.DoesNotContain("private-value",
audit.Context);
        using var context = JsonDocument.Parse(audit.Context); Assert.Equal(rule,
context.RootElement.GetProperty("ruleId").GetGuid());
        Assert.Equal("/obsolete",
Plan(review).Changes[context.RootElement.GetProperty("changeIndex").GetInt32()].SourcePath);
        await using var operatorScope = Factory.Services.CreateAsyncScope();
        var exported = await
operatorScope.ServiceProvider.GetRequiredService<AuditHistoryOperator>().VendorAsync(VendorId,
(await ActorAsync()).UserId,

```

```

        new(DateTimeOffset.UtcNow.AddDays(-1), DateTimeOffset.UtcNow.AddDays(1),
Action: "ConfigurationMigrationDestructiveChangeApproved"), new(), installationId: installation);
        var approval = Assert.Single(exported.Items); using var evidence =
JsonDocument.Parse(approval.Context);
        Assert.Equal(rule, evidence.RootElement.GetProperty("ruleId").GetGuid());
Assert.Equal(review.Id, evidence.RootElement.GetProperty("reviewId").GetGuid());
        Assert.DoesNotContain("private-value", approval.Context);
        Assert.Equal(result.RevisionId, (await ApplyAsync(review, rule)).RevisionId);
    }
    /// <summary>
    /// Verifies that source changes and rule replacement invalidate reviews while unresolved
values block commit.
    /// </summary>
    [Fact]
    public async Task
SourceChangesAndRuleReplacement_InvalidateReviews_WhileUnresolvedValuesBlockCommit()
    {
        await PrepareAsync(); var catalog = await CatalogAsync(schema: OpenSchema); Guid
target = await VersionAsync(catalog.Product, "2", OpenSchema);
        Guid installation = await InstallationAsync(await CustomerAsync(),
catalog.Product);
        using var source = await
SetAsync($"api/v1/installations/{installation}/configuration", catalog.Version, new { old = 1,
occupied = 2 }); await GenerateAsync(installation, catalog.Version);
        Guid definition = await DefinitionAsync(catalog.Product, catalog.Version, target,
new MigrationRuleRequest(1, "Move", "/old", "/occupied", null, false));
        var unresolved = await ReviewAsync(installation, target);
Assert.NotEmpty(Plan(unresolved).Errors); await Assert.ThrowsAsync<ApiConflictException>(() =>
ApplyAsync(unresolved));
        using var replaced = await SendAsync(HttpMethod.Put,
$"api/v1/products/{catalog.Product}/configuration-migrations/{definition}", new {
FromProductVersionId = catalog.Version, ToProductVersionId = target, Rules =
Array.Empty<MigrationRuleRequest>() }); Assert.Equal(HttpStatusCode.OK, replaced.StatusCode);
        var review = await ReviewAsync(installation, target);
        using var changed = await
SetAsync($"api/v1/installations/{installation}/configuration", catalog.Version, new { old = 2
});
        await Assert.ThrowsAsync<ApiConflictException>(() => ApplyAsync(review)); await
GenerateAsync(installation, catalog.Version);
        var stale = await Assert.ThrowsAsync<ApiConflictException>(() =>
ApplyAsync(review)); Assert.Equal("migration-preview-stale", stale.Code);
    }
}

```

Integration/ConfigurationMigrationHttpTests.cs

```

using System.Net;
using System.Net.Http.Json;
using BridgeControl.DTOs.Vendor;
using BridgeControl.Tests.Infrastructure;
using Microsoft.EntityFrameworkCore;

namespace BridgeControl.Tests.Integration;

/// <summary>
/// Verifies configuration migration http behavior and regression scenarios.
/// </summary>
public class ConfigurationMigrationHttpTests : MigrationTest
{
    /// <summary>

```

```

        /// Builds the migration HTTP route for the test installation and target.
        /// </summary>
        private static string Route(Guid installation, string action) =>
        $"api/v1/installations/{installation}/configuration/{action}";
        /// <summary>
        /// Verifies that latest preview is per user and target requires exact approval and
        retries return one result.
        /// </summary>
        [Fact]
        public async Task
        LatestPreview_IsPerUserAndTarget_RequiresExactApproval_AndRetriesReturnOneResult()
        {
            await PrepareAsync(); var catalog = await CatalogAsync(schema: OpenSchema); Guid
            target = await VersionAsync(catalog.Product, "2", OpenSchema);
            Guid customer = await CustomerAsync(), installation = await
            InstallationAsync(customer, catalog.Product);
            using var source = await
            SetAsync($"api/v1/installations/{installation}/configuration", catalog.Version, new { obsolete =
            1, other = 2 }); await GenerateAsync(installation, catalog.Version);
            Guid definition = await DefinitionAsync(catalog.Product, catalog.Version, target,
            new MigrationRuleRequest(1, "Delete", "/obsolete", null, null, true));
            using var preview = await SendAsync(HttpMethod.Post, Route(installation,
            "migration-preview"), new { TargetProductVersionId = target }); Assert.Equal(HttpStatusCode.OK,
            preview.StatusCode);
            var body = await ReadAsync(preview); Assert.Equal(new[] { "changes", "isValid",
            "targetProductVersionId", "validationErrors", "values" }, body.EnumerateObject().Select(x =>
            x.Name).Order());
            Assert.True(body.GetProperty("isValid").GetBoolean()); Guid oldRule =
            body.GetProperty("changes")[0].GetProperty("ruleId").GetGuid();
            var otherUser = await UserAsync("migration-owner@example.test",
            "CustomerAdministrator", customer);
            using var noReview = await SendAsync(HttpMethod.Post, Route(installation,
            "migrate"), new { TargetProductVersionId = target, ApprovedDestructiveRuleIds = new[] { oldRule }
            }, otherUser.Token);
            Assert.Equal(HttpStatusCode.Conflict, noReview.StatusCode);
            Assert.Contains("migration-preview-required", await noReview.Content.ReadAsStringAsync());
            using var replacement = await SendAsync(HttpMethod.Put,
            $"api/v1/products/{catalog.Product}/configuration-migrations/{definition}", new {
            FromProductVersionId = catalog.Version, ToProductVersionId = target, Rules = new[] { new
            MigrationRuleRequest(1, "Delete", "/other", null, null, true) } });
            Assert.Equal(HttpStatusCode.OK, replacement.StatusCode);
            using var latest = await SendAsync(HttpMethod.Post, Route(installation,
            "migration-preview"), new { TargetProductVersionId = target }); var next = await
            ReadAsync(latest); Guid newRule = next.GetProperty("changes")[0].GetProperty("ruleId").GetGuid();
            Assert.NotEqual(oldRule, newRule);
            using var oldApproval = await SendAsync(HttpMethod.Post, Route(installation,
            "migrate"), new { TargetProductVersionId = target, ApprovedDestructiveRuleIds = new[] { oldRule }
            }); Assert.Equal(HttpStatusCode.Conflict, oldApproval.StatusCode);
            using var missing = await SendAsync(HttpMethod.Post, Route(installation,
            "migrate"), new { TargetProductVersionId = target }); Assert.Equal(HttpStatusCode.BadRequest,
            missing.StatusCode);
            var simultaneous = await Task.WhenAll(Enumerable.Range(0, 2).Select(_ =>
            SendAsync(HttpMethod.Post, Route(installation, "migrate"), new { TargetProductVersionId = target,
            ApprovedDestructiveRuleIds = new[] { newRule } })));
            using var applied = simultaneous[0]; using var duplicate = simultaneous[1];
            Assert.Equal(HttpStatusCode.OK, applied.StatusCode); Assert.Equal(HttpStatusCode.OK,
            duplicate.StatusCode);
            var result = await ReadAsync(applied);
            Assert.False(result.GetProperty("values").TryGetProperty("other", out _)); Assert.Equal(1,
            result.GetProperty("values").GetProperty("obsolete").GetInt32());
            Assert.Equal(result.GetProperty("revisionId").GetGuid(), (await
            ReadAsync(duplicate)).GetProperty("revisionId").GetGuid());

```

```

        using var retry = await SendAsync(HttpMethod.Post, Route(installation, "migrate"),
new { TargetProductVersionId = target, ApprovedDestructiveRuleIds = new[] { newRule } });
Assert.Equal(result.GetProperty("revisionId").GetGuid(), (await
ReadAsync(retry)).GetProperty("revisionId").GetGuid());
        await using var db = await OpenVendorAsync(VendorId); Assert.Equal(1, await
db.VendorAuditLogs.CountAsync(x => x.Action == "ConfigurationMigrationApplied"));
        Assert.Equal(2, await db.ConfigurationMigrationReviews.Select(x =>
x.CreatedAt).Distinct().CountAsync());
    }
    /// <summary>
    /// Verifies that stale and unresolved latest reviews block apply without falling back.
    /// </summary>
    [Fact]
    public async Task StaleAndUnresolvedLatestReviews_BlockApplyWithoutFallingBack()
    {
        await PrepareAsync(); var catalog = await CatalogAsync(schema: OpenSchema); Guid
target = await VersionAsync(catalog.Product, "2", OpenSchema);
        Guid installation = await InstallationAsync(await CustomerAsync(),
catalog.Product);
        using var source = await
SetAsync($"api/v1/installations/{installation}/configuration", catalog.Version, new { value = 1
}); Guid original = await GenerateAsync(installation, catalog.Version);
        Guid definition = await DefinitionAsync(catalog.Product, catalog.Version, target);
        using var preview = await SendAsync(HttpMethod.Post, Route(installation,
"migration-preview"), new { TargetProductVersionId = target }); Assert.Equal(HttpStatusCode.OK,
preview.StatusCode);
        using var changed = await
SetAsync($"api/v1/installations/{installation}/configuration", catalog.Version, new { value = 2
});
        using var stale = await SendAsync(HttpMethod.Post, Route(installation, "migrate"),
new { TargetProductVersionId = target, ApprovedDestructiveRuleIds = Array.Empty<Guid>() });
Assert.Equal(HttpStatusCode.Conflict, stale.StatusCode);
        await GenerateAsync(installation, catalog.Version);
        using var good = await SendAsync(HttpMethod.Post, Route(installation,
"migration-preview"), new { TargetProductVersionId = target }); Assert.Equal(HttpStatusCode.OK,
good.StatusCode);
        using var replacement = await SendAsync(HttpMethod.Put,
$"api/v1/products/{catalog.Product}/configuration-migrations/{definition}", new {
FromProductVersionId = catalog.Version, ToProductVersionId = target, Rules = new[] { new
MigrationRuleRequest(1, "Introduce", null, "/value", "3", false) } });
Assert.Equal(HttpStatusCode.OK, replacement.StatusCode);
        using var invalid = await SendAsync(HttpMethod.Post, Route(installation,
"migration-preview"), new { TargetProductVersionId = target }); Assert.Equal(HttpStatusCode.OK,
invalid.StatusCode); Assert.False(await
ReadAsync(invalid).GetProperty("isValid").GetBoolean());
        using var blocked = await SendAsync(HttpMethod.Post, Route(installation,
"migrate"), new { TargetProductVersionId = target, ApprovedDestructiveRuleIds =
Array.Empty<Guid>() }); Assert.Equal(HttpStatusCode.Conflict, blocked.StatusCode);
Assert.Contains("migration-unresolved", await blocked.Content.ReadAsStringAsync());
        await using var db = await OpenVendorAsync(VendorId); Assert.False(await
db.EffectiveConfigurationRevisions.AnyAsync(x => x.ProductVersionId == target));
    }
    /// <summary>
    /// Verifies that preview and apply enforce role scope and recheck revoked access.
    /// </summary>
    [Fact]
    public async Task PreviewAndApply_EnforceRoleScopeAndRecheckRevokedAccess()
    {
        await PrepareAsync(); var catalog = await CatalogAsync(schema: OpenSchema); Guid
target = await VersionAsync(catalog.Product, "2", OpenSchema);

```

```

        Guid customer = await CustomerAsync(), installation = await
InstallationAsync(customer, catalog.Product), foreign = await InstallationAsync(await
CustomerAsync("Other"), catalog.Product);
        await GenerateAsync(installation, catalog.Version); await GenerateAsync(foreign,
catalog.Version); await DefinitionAsync(catalog.Product, catalog.Version, target);
        var user = await UserAsync("migration-readonly@example.test", "CustomerUser",
customer); var admin = await UserAsync("migration-admin@example.test", "CustomerAdministrator",
customer); var tech = await UserAsync("migration-tech@example.test", "VendorTechnician");
        foreach (string action in new[] { "migration-preview", "migrate" })
        {
            var request = new { TargetProductVersionId = target,
ApprovedDestructiveRuleIds = Array.Empty<Guid>() };
            using var forbidden = await SendAsync(HttpMethod.Post, Route(installation,
action), request, user.Token); Assert.Equal(HttpStatusCode.Forbidden, forbidden.StatusCode);
            using var hidden = await SendAsync(HttpMethod.Post, Route(foreign, action),
request, admin.Token); Assert.Equal(HttpStatusCode.NotFound, hidden.StatusCode);
            using var unassigned = await SendAsync(HttpMethod.Post, Route(installation,
action), request, tech.Token); Assert.Equal(HttpStatusCode.NotFound, unassigned.StatusCode);
        }
        using var grant = await SendAsync(HttpMethod.Put,
$"/api/v1/users/{tech.Id}/customer-access", new { CustomerIds = new[] { customer } });
        using var reviewed = await SendAsync(HttpMethod.Post, Route(installation,
"migration-preview"), new { TargetProductVersionId = target }, tech.Token);
Assert.Equal(HttpStatusCode.OK, reviewed.StatusCode);
        using var revoke = await SendAsync(HttpMethod.Put,
$"/api/v1/users/{tech.Id}/customer-access", new { CustomerIds = Array.Empty<Guid>() });
        using var denied = await SendAsync(HttpMethod.Post, Route(installation,
"migrate"), new { TargetProductVersionId = target, ApprovedDestructiveRuleIds =
Array.Empty<Guid>() }, tech.Token); Assert.Equal(HttpStatusCode.NotFound, denied.StatusCode);
    }
    /// <summary>
    /// Verifies that every migration route conceals other vendors resources.
    /// </summary>
    [Fact]
    public async Task EveryMigrationRoute_ConcealsOtherVendorsResources()
    {
        await PrepareAsync(); var catalog = await CatalogAsync(schema: OpenSchema); Guid
target = await VersionAsync(catalog.Product, "2", OpenSchema);
        Guid installation = await InstallationAsync(await CustomerAsync(),
catalog.Product); await GenerateAsync(installation, catalog.Version);
        Guid definition = await DefinitionAsync(catalog.Product, catalog.Version, target);
        Guid vendor = await CreateVendorAsync("Other vendor"); string invitation = await
InviteAsync(vendor);
        using var activated = await Client.PostAsJsonAsync("/api/v1/auth/activate", new {
Token = invitation, Password }); Assert.Equal(HttpStatusCode.NoContent, activated.StatusCode);
        using var login = await Client.PostAsJsonAsync("/api/v1/auth/login", new {
VendorName = "Other vendor", Email = "invited@example.test", Password }); string token = (await
ReadAsync(login)).GetProperty("accessToken").GetString();
        string collection =
$"/api/v1/products/{catalog.Product}/configuration-migrations";
        var definitionBody = new { FromProductVersionId = catalog.Version,
ToProductVersionId = target, Rules = Array.Empty<MigrationRuleRequest>() };
        using var list = await SendAsync(HttpMethod.Get, collection, token: token);
Assert.Equal(HttpStatusCode.NotFound, list.StatusCode);
        using var detail = await SendAsync(HttpMethod.Get, collection + "/" + definition,
token: token); Assert.Equal(HttpStatusCode.NotFound, detail.StatusCode);
        using var create = await SendAsync(HttpMethod.Post, collection, definitionBody,
token); Assert.Equal(HttpStatusCode.NotFound, create.StatusCode);
        using var replace = await SendAsync(HttpMethod.Put, collection + "/" + definition,
definitionBody, token); Assert.Equal(HttpStatusCode.NotFound, replace.StatusCode);
    }

```

```

        using var preview = await SendAsync(HttpMethod.Post, Route(installation,
"migration-preview"), new { TargetProductVersionId = target }, token);
Assert.Equal(HttpStatusCode.NotFound, preview.StatusCode);
        using var apply = await SendAsync(HttpMethod.Post, Route(installation, "migrate"),
new { TargetProductVersionId = target, ApprovedDestructiveRuleIds = Array.Empty<Guid>() },
token); Assert.Equal(HttpStatusCode.NotFound, apply.StatusCode);
    }
}

```

Integration/DatabaseOperationsTests.cs

```

using System.Diagnostics;
using System.Security.Cryptography;
using BridgeControl.Data;
using BridgeControl.Models.Platform;
using BridgeControl.Models.Vendor;
using BridgeControl.Options;
using BridgeControl.Services.Platform;
using BridgeControl.Services.Shared;
using BridgeControl.Services.Vendor;
using BridgeControl.Tests.Infrastructure;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Infrastructure;
using Microsoft.EntityFrameworkCore.Migrations;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Options;
using Xunit.Abstractions;

namespace BridgeControl.Tests.Integration;

/// <summary>
/// Verifies database operations behavior and regression scenarios.
/// </summary>
public sealed class DatabaseOperationsTests(ITestOutputHelper output) : VendorTest
{
    /// <summary>
    /// Verifies that upgrade resumes after committed migration and vendors remain
    independent.
    /// </summary>
    [Fact]
    public async Task UpgradeResumesAfterCommittedMigration_AndVendorsRemainIndependent()
    {
        await SignInAsync(); Guid first = await CreateVendorAsync("first"), second = await
CreateVendorAsync("second");
        foreach (Guid id in new[] { first, second })
        {
            await using VendorDbContext db = await OpenVendorAsync(id);
            await
db.GetService<IMigrator>().MigrateAsync(db.Database.GetMigrations().SkipLast(1).Last());
        }
        Guid successful = Guid.NewGuid(), interrupted = Guid.NewGuid();
        await Assert.ThrowsAsync<ArgumentException>(() => ExecuteAsync(first,
Guid.NewGuid(), "Backup"));
        Assert.Equal("Ready", await StatusAsync(first));
        await ExecuteAsync(first, successful, "Upgrade");
        await Assert.ThrowsAsync<IOException>(() => ExecuteAsync(second, interrupted,
"Upgrade", fault: true));
        Assert.Equal("Ready", await StatusAsync(first)); Assert.Equal("Maintenance", await
StatusAsync(second));
    }
}

```

```

        await Assert.ThrowsAsync<ArgumentException>(() => ExecuteAsync(second,
Guid.NewGuid(), "Upgrade"));
        // A fresh scope represents a restarted operator process; the actual migration
already committed.
        await ExecuteAsync(second, interrupted, "Upgrade");
        Assert.Equal("Ready", await StatusAsync(second));
        Assert.Equal(2, await Database.VendorMaintenanceOperations.CountAsync(x =>
x.Status == "Succeeded"));
        int count = await Database.PlatformAuditLogs.CountAsync();
        await ExecuteAsync(first, successful, "Upgrade"); Assert.Equal(count, await
Database.PlatformAuditLogs.CountAsync());
        Assert.True(await Database.PlatformAuditLogs.AnyAsync(x => x.EntityId ==
interrupted.ToString() && x.Action == "VendorMaintenanceFailed"));
    }

    /// <summary>
    /// Verifies that daily backups resume interrupted native backup and run again on the next
day.
    /// </summary>
    [Fact]
    public async Task DailyBackupsResumeInterruptedNativeBackup_AndRunAgainOnTheNextDay()
    {
        await SignInAsync(); Guid first = await CreateVendorAsync("first"), second = await
CreateVendorAsync("second");
        Guid interrupted = Guid.NewGuid(); await Assert.ThrowsAsync<IOException>(() =>
ExecuteAsync(first, interrupted, "Backup", fault: true));
        Guid admin = await Database.PlatformAdmins.Select(x => x.Id).SingleAsync();
        var operations = Services.GetRequiredService<VendorDatabaseOperations>();
        Assert.Equal(0, await operations.BackupDueAsync(admin));
        Assert.Equal(2, await Database.VendorMaintenanceOperations.CountAsync());
        Assert.Equal("Succeeded", (await
Database.VendorMaintenanceOperations.AsNoTracking().SingleAsync(x => x.Id ==
interrupted)).Status);
        Assert.Equal(0, await operations.BackupDueAsync(admin)); Assert.Equal(2, await
Database.VendorMaintenanceOperations.CountAsync());
        Factory.Clock.Offset = TimeSpan.FromDays(1);
        Assert.Equal(0, await operations.BackupDueAsync(admin)); Assert.Equal(4, await
Database.VendorMaintenanceOperations.CountAsync());
        Assert.Equal("Ready", await StatusAsync(first)); Assert.Equal("Ready", await
StatusAsync(second));
    }

    /// <summary>
    /// Verifies that native restore recovers database and blobs after interruption without
changing another vendor.
    /// </summary>
    [Fact]
    public async Task
NativeRestoreRecoversDatabaseAndBlobs_AfterInterruption_WithoutChangingAnotherVendor()
    {
        await SignInAsync(); Guid first = await CreateVendorAsync("restore"), other =
await CreateVendorAsync("untouched");
        (Guid customer, FileArtifact artifact, string path) = await SeedAsync(first,
"before");
        (_, FileArtifact otherArtifact, string otherPath) = await SeedAsync(other,
"other");
        Guid backup = Guid.NewGuid(), restore = Guid.NewGuid();
        Stopwatch watch = Stopwatch.StartNew(); await ExecuteAsync(first, backup,
"Backup");
        output.WriteLine($"Native backup plus encrypted archive:
{watch.Elapsed.TotalSeconds:F2}s.");
        await using (VendorDbContext db = await OpenVendorAsync(first))

```

```

        { (await db.Customers.SingleAsync(x => x.Id == customer)).Name = "after"; await
db.SaveChangesAsync(); }
        await File.WriteAllTextAsync(path, "damaged");
        watch.Restart(); await Assert.ThrowsAsync<IOException>(() => ExecuteAsync(first,
restore, "Restore", backup, fault: true));
        Assert.Equal("Maintenance", await StatusAsync(first)); Assert.Equal("Ready", await
StatusAsync(other));
        await ExecuteAsync(first, restore, "Restore", backup);
        output.WriteLine($"Interrupted restore plus verified retry:
{watch.Elapsed.TotalSeconds:F2}s; one database and one blob.");
        await using (VendorDbContext db = await OpenVendorAsync(first)) {
Assert.Equal("before", (await db.Customers.SingleAsync(x => x.Id == customer)).Name); }
        Assert.Equal(artifact.Sha256, await VendorBackupStore.HashAsync(path, default));
        Assert.Equal(otherArtifact.Sha256, await VendorBackupStore.HashAsync(otherPath,
default));
        await using (VendorDbContext db = await OpenVendorAsync(other)) {
Assert.Equal("other", (await db.Customers.SingleAsync()).Name); }
        Assert.Equal("Ready", await StatusAsync(first));
        Assert.NotNull(await Database.SoftwareVendors.Where(x => x.Id == first).Select(x
=> x.Sessions.RevokedBefore).SingleAsync());
        Assert.True(await Database.PlatformAuditLogs.AnyAsync(x => x.EntityId ==
restore.ToString() && x.Action == "VendorMaintenanceFailed"));
        Assert.Equal("Succeeded", (await
Database.VendorMaintenanceOperations.AsNoTracking().SingleAsync(x => x.Id == backup)).Status);
    }

    /// <summary>
    /// Verifies that tampered or wrong vendor backup cannot replace data and retry uses
original identity.
    /// </summary>
    [Fact]
    public async Task
TamperedOrWrongVendorBackupCannotReplaceData_AndRetryUsesOriginalIdentity()
    {
        await SignInAsync(); Guid first = await CreateVendorAsync("first"), second = await
CreateVendorAsync("second");
        await SeedAsync(first, "first"); await SeedAsync(second, "second");
        Guid backup = Guid.NewGuid(); await ExecuteAsync(first, backup, "Backup");
        Guid wrong = Guid.NewGuid();
        await Assert.ThrowsAsync<InvalidDataException>(() => ExecuteAsync(second, wrong,
"Restore", backup));
        await using (VendorDbContext db = await OpenVendorAsync(second)) {
Assert.Equal("second", (await db.Customers.SingleAsync()).Name); }
        string path = Services.GetRequiredService<VendorBackupStore>().Bundle(backup);
        byte[] original = await File.ReadAllBytesAsync(path); byte[] altered =
(byte[])original.Clone(); altered[100] ^= 1; await File.WriteAllBytesAsync(path, altered);
        Guid restore = Guid.NewGuid(); await Assert.ThrowsAsync<InvalidDataException>(()
=> ExecuteAsync(first, restore, "Restore", backup));
        await File.WriteAllBytesAsync(path, original); await ExecuteAsync(first, restore,
"Restore", backup);
        Assert.Equal("Ready", await StatusAsync(first));
    }

    /// <summary>
    /// Verifies that host lease excludes offline operations and retention preserves active
restore backup.
    /// </summary>
    [Fact]
    public async Task
HostLeaseExcludesOfflineOperations_AndRetentionPreservesActiveRestoreBackup()
    {
        await SignInAsync(); Guid vendor = await CreateVendorAsync();

```

```

        await using (DatabaseHostLease host = await
DatabaseHostLease.AcquireAsync(Database, default))
    { await Assert.ThrowsAsync<InvalidOperationException>(() => ExecuteAsync(vendor,
Guid.NewGuid(), "Backup")); }
    Guid backup = Guid.NewGuid(); await ExecuteAsync(vendor, backup, "Backup");
    Guid restore = Guid.NewGuid(); await Assert.ThrowsAsync<IOException>(() =>
ExecuteAsync(vendor, restore, "Restore", backup, fault: true));
    Factory.Clock.Offset = TimeSpan.FromDays(31);
    Guid admin = await Database.PlatformAdmins.Select(x => x.Id).SingleAsync();
    await
Services.GetRequiredService<VendorDatabaseOperations>().ExpireBackupsAsync(admin);

Assert.True(File.Exists(Services.GetRequiredService<VendorBackupStore>().Bundle(backup)));
    await ExecuteAsync(vendor, restore, "Restore", backup);
    Database.ChangeTracker.Clear(); await
Services.GetRequiredService<VendorDatabaseOperations>().ExpireBackupsAsync(admin);

Assert.False(File.Exists(Services.GetRequiredService<VendorBackupStore>().Bundle(backup)));
    Assert.NotNull(await
Database.VendorMaintenanceOperations.AsNoTracking().SingleAsync(x => x.Id == backup).ExpiredAt);
}

/// <summary>
/// Runs the requested vendor database operation for the test scenario.
/// </summary>
private async Task<VendorMaintenanceOperation> ExecuteAsync(Guid vendor, Guid operation,
string kind, Guid? backup = null, bool fault = false)
{
    await using AsyncServiceScope scope = Factory.Services.CreateAsyncScope();
    IServiceProvider services = scope.ServiceProvider;
    PlatformDbContext platform = services.GetRequiredService<PlatformDbContext>();
    Guid admin = await platform.PlatformAdmins.Select(x => x.Id).SingleAsync();
    IVendorDatabaseMaintenance sql =
services.GetRequiredService<IVendorDatabaseMaintenance>();
    if (fault) { sql = new InterruptAfterCommit(sql); }
    var infrastructure = services.GetRequiredService<IVendorInfrastructure>(); var
options = services.GetRequiredService<IOptions<DatabaseOperationsOptions>>();
    VendorBackupStore store = new(options, infrastructure, sql,
services.GetRequiredService<VendorArtifactArchive>());
    VendorDatabaseOperations operations = new(platform, infrastructure, sql, store,
options, Factory.Clock);
    return await operations.ExecuteAsync(admin, vendor, operation, kind, backup);
}

/// <summary>
/// Reads the recorded maintenance-operation status for the test.
/// </summary>
private Task<string> StatusAsync(Guid vendor) =>
Database.VendorDatabases.AsNoTracking().Where(x => x.VendorId == vendor).Select(x =>
x.Status).SingleAsync();

/// <summary>
/// Writes database and artifact fixtures used to verify backup recovery.
/// </summary>
private async Task<(Guid Customer, FileArtifact Artifact, string Path)> SeedAsync(Guid
vendor, string name)
{
    await using VendorDbContext db = await OpenVendorAsync(vendor);
    Customer customer = new() { Id = Guid.NewGuid(), Name = name, Status = "Active",
CreatedAt = DateTimeOffset.UtcNow };
    byte[] content = $"{\"restored\":true}\"u8.ToArray();
    FileArtifact artifact = new()
    {
        Id = Guid.NewGuid(),

```

```

        ArtifactType = "Configuration",
        FileName = "config.json",
        StorageIdentifier = Guid.NewGuid().ToString("N"),
        ContentType = "application/json",
        SizeBytes = content.Length,
        Sha256 = Convert.ToHexStringLower(SHA256.HashData(content)),
        CreatedAt = DateTimeOffset.UtcNow
    };
    db.Customers.Add(customer); db.FileArtifacts.Add(artifact); await
db.SaveChangesAsync();
    string root = PrivateDirectory.Ensure(Path.Combine(Factory.ArchiveRoot,
vendor.ToString("N"), "blobs")); string path = Path.Combine(root, artifact.StorageIdentifier +
".blob");
    await File.WriteAllBytesAsync(path, content); return (customer.Id, artifact,
path);
    }
    /// <summary>
    /// Injects an interruption after a committed maintenance step to test recovery.
    /// </summary>
    private sealed class InterruptAfterCommit(IVendorDatabaseMaintenance inner) :
IVendorDatabaseMaintenance
    {
        /// <summary>
        /// Delegates the maintenance step and injects the configured interruption to test
recovery.
        /// </summary>
        public async Task MigrateAsync(VendorDbContext database, string migration,
CancellationToken ct) { await inner.MigrateAsync(database, migration, ct); throw new
IOException("Simulated operator interruption."); }
        /// <summary>
        /// Delegates the maintenance step and injects the configured interruption to test
recovery.
        /// </summary>
        public async Task BackupAsync(VendorDbContext database, string path,
CancellationToken ct) { await inner.BackupAsync(database, path, ct); throw new
IOException("Simulated backup interruption."); }
        /// <summary>
        /// Delegates the maintenance step and injects the configured interruption to test
recovery.
        /// </summary>
        public async Task RestoreAsync(VendorDbContext database, string path,
CancellationToken ct) { await inner.RestoreAsync(database, path, ct); throw new
IOException("Simulated operator interruption."); }
        /// <summary>
        /// Delegates the maintenance step and injects the configured interruption to test
recovery.
        /// </summary>
        public Task VerifyAsync(VendorDbContext database, CancellationToken ct) =>
inner.VerifyAsync(database, ct);
    }
}

```

Integration/DatabaseRecoveryWorkflowTests.cs

```

using System.Diagnostics;
using System.Net;
using BridgeControl.Data;
using BridgeControl.Services.Platform;
using BridgeControl.Services.Vendor;
using BridgeControl.Tests.Infrastructure;

```

```

using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using Xunit.Abstractions;

namespace BridgeControl.Tests.Integration;

/// <summary>
/// Verifies database recovery workflow behavior and regression scenarios.
/// </summary>
public sealed class DatabaseRecoveryWorkflowTests(ITestOutputHelper output) :
    InstallationStateTest
{
    /// <summary>
    /// Verifies that restores published artifacts and queued work and rejects restored
    credentials.
    /// </summary>
    [Fact]
    public async Task RestoresPublishedArtifactsAndQueuedWork_AndRejectsRestoredCredentials()
    {
        StateFixture state = await StateAsync();
        using HttpResponseMessage unusedEnrollment = await SendAsync(HttpMethod.Post,
            "/api/v1/agent-enrollments", new { CustomerId = state.CustomerId });
        Assert.Equal(HttpStatusCode.Created, unusedEnrollment.StatusCode);
        using HttpResponseMessage reported = await ReportAsync(state, "NotInstalled",
            DateTimeOffset.UtcNow); Assert.Equal(HttpStatusCode.OK, reported.StatusCode);
        using HttpResponseMessage desired = await SetDesiredAsync(state);
        Assert.Equal(HttpStatusCode.OK, desired.StatusCode);
        Guid admin = await Database.PlatformAdmins.Select(x => x.Id).SingleAsync(), backup
            = Guid.NewGuid();
        Stopwatch timer = Stopwatch.StartNew();
        await ExecuteAsync(admin, backup, "Backup");
        await ReconcileAsync();
        await using (VendorDbContext db = await OpenVendorAsync(VendorId)) {
            Assert.Single(await db.Deployments.ToListAsync());
            await ExecuteAsync(admin, Guid.NewGuid(), "Restore", backup);
            await using (VendorDbContext db = await OpenVendorAsync(VendorId))
            {
                Assert.Empty(await db.Deployments.ToListAsync());
                Assert.False(await db.AgentEnrollments.AnyAsync(x => x.UsedAt == null &&
                    x.RevokedAt == null));
                var revision = await db.EffectiveConfigurationRevisions.Include(x =>
                    x.FileArtifact).SingleAsync(x => x.Id == state.ConfigurationId);
                Assert.Equal(state.Hash, revision.FileArtifact.Sha256);
                var vendor = await Database.SoftwareVendors.AsNoTracking().Include(x =>
                    x.FileArchive).SingleAsync(x => x.Id == VendorId);
                string archive =
                    Services.GetRequiredService<IVendorInfrastructure>().GetArchivePath(vendor);
                foreach (var artifact in await
                    db.FileArtifacts.AsNoTracking().ToArrayAsync())
                {
                    await using Stream stream = await
                        Services.GetRequiredService<VendorArtifactArchive>().OpenAsync(archive, artifact, default);
                    Assert.Equal(artifact.SizeBytes, stream.Length);
                }
                using HttpResponseMessage oldAgent = await SendAsync(HttpMethod.Get,
                    $"/api/v1/agent/installations/{state.InstallationId}/desired-state", token: state.Token);
                Assert.Equal(HttpStatusCode.Unauthorized, oldAgent.StatusCode);
                using HttpResponseMessage oldHuman = await SendAsync(HttpMethod.Get,
                    "/api/v1/products"); Assert.Equal(HttpStatusCode.Unauthorized, oldHuman.StatusCode);
                // New service scopes after restore process durable work without any in-memory
                recovery state.
                await ReconcileAsync();
            }
        }
    }
}

```

```

        await using (VendorDbContext db = await OpenVendorAsync(VendorId)) {
Assert.Single(await db.Deployments.ToListAsync()); }
        output.WriteLine($"Published MSI/schema/configuration, SQL, durable work and
credential revocation verified in {timer.Elapsed.TotalSeconds:F2}s (local fixture, not a
production RTO).");
    }
    /// <summary>
    /// Runs the requested vendor database operation for the test scenario.
    /// </summary>
    private async Task ExecuteAsync(Guid admin, Guid operation, string kind, Guid? backup =
null)
    {
        await using AsyncServiceScope scope = Factory.Services.CreateAsyncScope();
        await
scope.ServiceProvider.GetRequiredService<VendorDatabaseOperations>().ExecuteAsync(admin,
VendorId, operation, kind, backup);
    }
}

```

Integration/DeliveryAcceptanceTests.cs

```

using BridgeControl.DTOS.AgentDTOS.OperationalHistoryContracts;
using System.Net;
using System.Security.Cryptography;
using BridgeControl.DTOS.Vendor;
using BridgeControl.DTOS.Vendor.OperationalHistoryContracts;
using BridgeControl.Tests.Infrastructure;
using Microsoft.EntityFrameworkCore;

namespace BridgeControl.Tests.Integration;

/// <summary>
/// Verifies delivery acceptance behavior and regression scenarios.
/// </summary>
public sealed class DeliveryAcceptanceTests : InstallationStateTest
{
    /// <summary>
    /// Verifies that provisioning through migration and deployment preserves history and
reports actual success.
    /// </summary>
    [Fact]
    public async Task
ProvisioningThroughMigrationAndDeployment_PreservesHistoryAndReportsActualSuccess()
    {
        // StateAsync provisions a vendor, invitation/login, catalog, Agent, installation
and assigned license over HTTP.
        var state = await StateAsync();
        using var vendor = await Client.GetAsync($"api/v1/platform/vendors/{VendorId}");
        Assert.Equal(HttpStatusCode.OK, vendor.StatusCode); Assert.Equal(VendorId, await
ReadAsync(vendor)).GetProperty("id").GetGuid());
        using var license = await SendAsync(HttpMethod.Get,
$"api/v1/licenses/{state.License}");
        Assert.Equal(HttpStatusCode.OK, license.StatusCode); Assert.Equal(state.Customer,
(await ReadAsync(license)).GetProperty("customerId").GetGuid());
        Guid user;
        await using (var database = await OpenVendorAsync(VendorId)) { user = await
database.UserAccounts.Select(x => x.Id).SingleAsync(); }
        using var detail = await SendAsync(HttpMethod.Get, $"api/v1/users/{user}");
        Assert.Equal(HttpStatusCode.OK, detail.StatusCode);
        Assert.Equal(user, (await ReadAsync(detail)).GetProperty("id").GetGuid());
    }
}

```

```

        foreach (string kind in new[] { "installer", "schema" })
        {
            using var metadata = await SendAsync(HttpMethod.Get,
$"/api/v1/products/{state.Product}/versions/{state.Version}/{kind}");
            Assert.Equal(HttpStatusCode.OK, metadata.StatusCode);
            Assert.Equal(new[] { "artifactType", "contentType", "createdAt",
"fileName", "id", "sha256", "sizeBytes" }, (await ReadAsync(metadata)).EnumerateObject().Select(x
=> x.Name).Order());
        }
        using var configured = await
SetAsync($"/api/v1/installations/{state.Installation}/configuration", state.Version, new { port =
8123 }); Assert.Equal(HttpStatusCode.OK, configured.StatusCode);
        Guid original = await GenerateAsync(state.Installation, state.Version);
        using var desired = await SetDesiredAsync(state, configuration: original);
        Assert.Equal(HttpStatusCode.OK, desired.StatusCode);
        using var missing = await ReportAsync(state, "NotInstalled",
DateTimeOffset.UtcNow); Assert.Equal(HttpStatusCode.OK, missing.StatusCode);
        await DeployAndReportAsync(state, state.Version, original, "Install");

        Guid target = await VersionAsync(state.Product, "2", Schema);
        using var upload = await UploadAsync(state.Product, target, "installer",
MsiFixture.Create(Factory.PrivateRoot)); Assert.Equal(HttpStatusCode.OK, upload.StatusCode);
        using var published = await PublishAsync(state.Product, target);
        Assert.Equal(HttpStatusCode.OK, published.StatusCode);
        using var definition = await SendAsync(HttpMethod.Post,
$"/api/v1/products/{state.Product}/configuration-migrations", new
        {
            FromProductVersionId = state.Version,
            ToProductVersionId = target,
            Rules = new[] { new MigrationRuleRequest(1, "Preserve", "/port", null,
null, false) }
        }); Assert.Equal(HttpStatusCode.Created, definition.StatusCode);
        string route = $"/api/v1/installations/{state.Installation}/configuration";
        using var preview = await SendAsync(HttpMethod.Post, route + "/migration-preview",
new { TargetProductVersionId = target });
        Assert.Equal(HttpStatusCode.OK, preview.StatusCode); Assert.True(await
ReadAsync(preview)).GetProperty("isValid").GetBoolean());
        using var applied = await SendAsync(HttpMethod.Post, route + "/migrate", new {
TargetProductVersionId = target, ApprovedDestructiveRuleIds = Array.Empty() });
        Assert.Equal(HttpStatusCode.OK, applied.StatusCode); var migrated = await
ReadAsync(applied); Assert.Equal(8123,
migrated.GetProperty("values").GetProperty("port").GetInt32());
        Guid revision = migrated.GetProperty("revisionId").GetGuid();
        using var next = await SetDesiredAsync(state, target, revision);
        Assert.Equal(HttpStatusCode.OK, next.StatusCode);
        await DeployAndReportAsync(state, target, revision, "Update");
        using var log = await SendAsync(HttpMethod.Post, "/api/v1/agent/logs", new
SubmitAgentLogsRequest([new(state.Installation, "Information", "acceptance", "Deployment
completed", DateTimeOffset.UtcNow)]), state.Token);
        Assert.Equal(HttpStatusCode.NoContent, log.StatusCode);
        using var history = await SendAsync(HttpMethod.Get,
$"/api/v1/installations/{state.Installation}/logs"); Assert.Equal(HttpStatusCode.OK,
history.StatusCode);
        Assert.Equal(1, (await ReadAsync(history)).GetProperty("totalCount").GetInt32());
        using var overview = await SendAsync(HttpMethod.Get,
"/api/v1/monitoring/overview"); Assert.Equal(HttpStatusCode.OK, overview.StatusCode);
        Assert.Equal(1, (await
ReadAsync(overview)).GetProperty("totalInstallations").GetInt32());
        await using var db = await OpenVendorAsync(VendorId);
        Assert.Equal(2, await db.Deployments.CountAsync(x => x.Status == "Reconciled"));
        Assert.Equal(2, await db.DeploymentAttempts.CountAsync(x => x.Status ==
"Succeeded"));

```

```

        Assert.True(await db.EffectiveConfigurationRevisions.AnyAsync(x => x.Id ==
original && x.ProductVersionId == state.Version));
        Assert.True(await db.VendorAuditLogs.AnyAsync(x => x.Action ==
"ConfigurationMigrationApplied" && x.ActorId == user));
    }

    /// <summary>
    /// Executes the test deployment flow and reports matching installation evidence.
    /// </summary>
    private async Task DeployAndReportAsync(StateFixture state, Guid version, Guid revision,
string operation)
    {
        await ReconcileAsync();
        using var pending = await SendAsync(HttpMethod.Get,
"/api/v1/agent/deployments?status=pending", token: state.Token);
        Assert.Equal(HttpStatusCode.OK, pending.StatusCode); var work = (await
ReadAsync(pending)).EnumerateArray().Single();
        Assert.Equal(operation, work.GetProperty("operationType").GetString()); Guid
deployment = work.GetProperty("id").GetGuid();
        Guid artifact;
        await using (var db = await OpenVendorAsync(VendorId)) { artifact = (await
db.EffectiveConfigurationRevisions.SingleAsync(x => x.Id == revision)).FileArtifactId; }
        using var download = await SendAsync(HttpMethod.Get,
$"/api/v1/agent/artifacts/{artifact}/content", token: state.Token);
        Assert.Equal(HttpStatusCode.OK, download.StatusCode); string hash =
Convert.ToHexStringLower(SHA256.HashData(await download.Content.ReadAsByteArrayAsync()));
        Assert.Equal($"\"{hash}\"", download.Headers.ETag!.Tag);
        using var started = await SendAsync(HttpMethod.Post,
$"/api/v1/agent/deployments/{deployment}/attempts", token: state.Token);
        Assert.Equal(HttpStatusCode.OK, started.StatusCode); Guid attempt = (await
ReadAsync(started)).GetProperty("attemptId").GetGuid();
        foreach (string phase in new[] { "Installing", "Configuring", "Succeeded" })
        {
            using var updated = await SendAsync(HttpMethod.Patch,
"/api/v1/agent/deployments/{deployment}/attempts/{attempt}", new { Status = phase, CompletedAt =
phase == "Succeeded" ? (DateTimeOffset?)DateTimeOffset.UtcNow : null }, state.Token);
            Assert.Equal(HttpStatusCode.OK, updated.StatusCode);
        }
        using var reported = await ReportAsync(state, "Running",
DateTimeOffset.UtcNow.AddSeconds(1), version, revision, hash); Assert.Equal(HttpStatusCode.OK,
reported.StatusCode);
        await ReconcileAsync();
        using var completed = await SendAsync(HttpMethod.Get,
"/api/v1/deployments/{deployment}"); Assert.Equal(HttpStatusCode.OK, completed.StatusCode);
        Assert.Equal("Reconciled", (await
ReadAsync(completed)).GetProperty("status").GetString());
    }
}

```

Integration/DeploymentMetadataTests.cs

```

using System.ComponentModel.DataAnnotations;
using BridgeControl.DTOs.Vendor.DeploymentSettings;
using BridgeControl.Services.Vendor;
using BridgeControl.Tests.Infrastructure;

namespace BridgeControl.Tests.Integration;

/// <summary>
/// Verifies deployment metadata behavior and regression scenarios.

```

```

/// </summary>
public sealed class DeploymentMetadataTests
{
    /// <summary>
    /// Verifies that settings reject ambiguous remote and per user paths.
    /// </summary>
    [Theory]
    [InlineData(@"C:\Program Files\Widget\defaults.json", true)]
    [InlineData(@"%APPDATA%\Widget\settings.json", false)]
    [InlineData(@"\\server\share\settings.json", false)]
    [InlineData(@"C:\Widget\..\settings.json", false)]
    [InlineData(@"C:\Widget\settings.json:stream", false)]
    public void Settings_RejectAmbiguousRemoteAndPerUserPaths(string path, bool valid)
    {
        var settings = new DeploymentSettings("Desktop", path, @"C:\Program
Files\Widget\Widget.exe");
        List<ValidationResult> errors = [];
        Assert.Equal(valid, Validator.TryValidateObject(settings, new(settings), errors,
true));
    }
    /// <summary>
    /// Verifies that installer identity is read from MSI without executing it.
    /// </summary>
    [Fact]
    public void InstallerIdentity_IsReadFromMsiWithoutExecutingIt()
    {
        string path = Path.Combine(Path.GetTempPath(), "BridgeControlMetadata_" +
Guid.NewGuid().ToString("N") + ".msi");
        try
        {
            File.WriteAllBytes(path, MsiFixture.Create(Path.GetTempPath()));
            MsiArtifactValidator.Validate(path);
            InstallerIdentity identity = MsiArtifactValidator.ReadIdentity(path);
            Assert.NotEqual(Guid.Empty, identity.ProductCode);
            Assert.NotNull(identity.UpgradeCode);
            Assert.NotEqual(Guid.Empty, identity.UpgradeCode);
            Assert.Equal("1.0.0", identity.Version);
            Assert.Equal(identity, MsiArtifactValidator.ReadIdentity(path));
        }
        finally { File.Delete(path); }
    }
}

```

Integration/DeploymentTests.cs

```

using System.Net;
using BridgeControl.DTOS.Vendor;
using BridgeControl.DTOS.Vendor.DeploymentContracts;
using BridgeControl.Repositories.Vendor;
using BridgeControl.Security.AgentSecurity;
using BridgeControl.Tests.Infrastructure;
using Microsoft.Data.SqlClient;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;

namespace BridgeControl.Tests.Integration;

/// <summary>
/// Verifies deployment behavior and regression scenarios.
/// </summary>

```

```

public class DeploymentTests : InstallationStateTest
{
    /// <summary>
    /// Verifies that desired change and start race never authorizes overlapping targets.
    /// </summary>
    [Fact]
    public async Task DesiredChangeAndStartRace_NeverAuthorizesOverlappingTargets()
    {
        var (s, d) = await PendingAsync();
        using var changed = await
SetAsync($"api/v1/installations/{s.Installation}/configuration", s.Version, new { port = 9000
});
        Guid nextConfiguration = await GenerateAsync(s.Installation, s.Version);
        var startTask = StartAsync(s, d); var desiredTask = SetDesiredAsync(s,
configuration: nextConfiguration);
        using var start = await startTask; using var desired = await desiredTask;
        Assert.Equal(HttpStatusCode.OK, desired.StatusCode);
        Assert.Contains(start.StatusCode, new[] { HttpStatusCode.OK, HttpStatusCode.Conflict });
        using var late = await StartAsync(s, d); Assert.Equal(HttpStatusCode.Conflict,
late.StatusCode);
        await ReconcileAsync(); await using var db = await OpenVendorAsync(VendorId);
        Assert.Equal("Superseded", (await db.Deployments.SingleAsync(x => x.Id ==
d)).Status);
        Guid next = (await db.Deployments.SingleAsync(x => x.IsActive)).Id;
        using var nextStart = await StartAsync(s, next);
        Assert.Equal(start.IsSuccessStatusCode ? HttpStatusCode.Conflict :
HttpStatusCode.OK, nextStart.StatusCode);
        Assert.Equal(1, await db.DeploymentAttempts.CountAsync(x => x.CompletedAt ==
null));
    }

    /// <summary>
    /// Verifies that cancellation audit failure preserves pending work.
    /// </summary>
    [Fact]
    public async Task CancellationAuditFailure_PreservesPendingWork()
    {
        var (s, d) = await PendingAsync(); await using var db = await
OpenVendorAsync(VendorId);
        await db.Database.ExecuteSqlRawAsync("CREATE TRIGGER FailCancelAudit ON
VendorAuditLogs AFTER INSERT AS BEGIN IF EXISTS (SELECT 1 FROM inserted WHERE Action =
'DeploymentCancelledByUser') THROW 51000, 'test failure', 1; END");
        using var failed = await SendAsync(HttpMethod.Post,
$/api/v1/deployments/{d}/cancel"); Assert.Equal(HttpStatusCode.InternalServerError,
failed.StatusCode);
        Assert.Equal("Pending", (await
db.Deployments.AsNoTracking().SingleAsync()).Status);
        await db.Database.ExecuteSqlRawAsync("DROP TRIGGER FailCancelAudit");
        using var pending = await SendAsync(HttpMethod.Get, "/api/v1/agent/deployments",
token: s.Token);
        using var replay = await SendAsync(HttpMethod.Get, "/api/v1/agent/deployments",
token: s.Token);
        Assert.Equal(d, (await ReadAsync(pending))[0].GetProperty("id").GetGuid());
        Assert.Equal(d, (await ReadAsync(replay))[0].GetProperty("id").GetGuid());
        Assert.False(await db.DeploymentAttempts.AnyAsync());
    }

    /// <summary>
    /// Verifies that fresh host recovers lost start and terminal responses.
    /// </summary>
    [Fact]
    public async Task FreshHost_RecoversLostStartAndTerminalResponses()
    {

```

```

        var (s, d) = await PendingAsync(); using var started = await StartAsync(s, d);
        Guid a = (await ReadAsync(started)).GetProperty("attemptId").GetGuid();
        await using var db = await OpenVendorAsync(VendorId); Guid credential = (await
db.AgentCredentials.SingleAsync(x => x.AgentId == s.Agent)).Id;
        AgentActor actor = new(VendorId, s.Agent, credential);
        await using (var restarted = new
ApiFactory(Database.Database.GetConnectionString()!, archiveRoot: Factory.ArchiveRoot))
        {
            using var client = restarted.CreateApiClient(); await using var scope =
restarted.Services.CreateAsyncScope();
            var repository =
scope.ServiceProvider.GetRequiredService<IDeploymentRepository>();
            var recovered = await repository.StartAsync(actor, d, default);
Assert.Equal(a, recovered.AttemptId); Assert.Equal(1, recovered.AttemptNumber);
            var request = new UpdateDeploymentAttemptRequest("Failed", "INTERRUPTED",
"local details", DateTimeOffset.UtcNow);
            var result = await repository.UpdateAsync(actor, d, a, request, default);
            await using var secondScope = Factory.Services.CreateAsyncScope();
            var duplicate = await
secondScope.ServiceProvider.GetRequiredService<IDeploymentRepository>().UpdateAsync(actor, d, a,
request, default);
            Assert.Equal(result, duplicate);
        }
        Assert.Equal(1, await db.DeploymentAttempts.CountAsync());
    }

    /// <summary>
    /// Verifies that existing installation dispatches appropriate operation.
    /// </summary>
    [Theory]
    [InlineData("Configure")]
    [InlineData("Update")]
    public async Task ExistingInstallation_DispatchesAppropriateOperation(string operation)
    {
        var s = await StateAsync(); using var desired = await SetDesiredAsync(s);
        Guid installed = operation == "Update" ? await VersionAsync(s.Product, "older") :
s.Version;
        using var actual = await ReportAsync(s, "Stopped", DateTimeOffset.UtcNow,
installed); Assert.Equal(HttpStatusCode.OK, actual.StatusCode); await ReconcileAsync();
        using var pending = await SendAsync(HttpMethod.Get, "/api/v1/agent/deployments",
token: s.Token);
        var item = (await ReadAsync(pending))[0]; Assert.Equal(operation,
item.GetProperty("operationType").GetString()); Guid d = item.GetProperty("id").GetGuid();
        using var started = await StartAsync(s, d); Assert.Equal(HttpStatusCode.OK,
started.StatusCode); Guid a = (await ReadAsync(started)).GetProperty("attemptId").GetGuid();
        using var installing = await UpdateAsync(s, d, a, "Installing");
Assert.Equal(operation == "Configure" ? HttpStatusCode.Conflict : HttpStatusCode.OK,
installing.StatusCode);
        using var configuring = await UpdateAsync(s, d, a, "Configuring");
Assert.Equal(HttpStatusCode.OK, configuring.StatusCode);
        using var restart = await UpdateAsync(s, d, a, "RestartRequired", completed:
DateTimeOffset.UtcNow); Assert.Equal(HttpStatusCode.OK, restart.StatusCode);
        await using var db = await OpenVendorAsync(VendorId);
Assert.Equal("AwaitingState", (await db.Deployments.SingleAsync()).Status);
    }

    /// <summary>
    /// Verifies that newer actual evidence is checked before dispatch even before worker
runs.
    /// </summary>
    [Fact]
    public async Task NewerActualEvidence_IsCheckedBeforeDispatch_EvenBeforeWorkerRuns()

```

```

    {
        var (s, d) = await PendingAsync();
        using var unknown = await ReportAsync(s, "Unknown",
DateTimeOffset.UtcNow.AddSeconds(1));
        using var blocked = await StartAsync(s, d); Assert.Equal(HttpStatusCode.Conflict,
blocked.StatusCode);
        using var matching = await ReportAsync(s, "Running",
DateTimeOffset.UtcNow.AddSeconds(2), s.Version, s.Configuration, s.Hash);
        using var unnecessary = await StartAsync(s, d);
Assert.Equal(HttpStatusCode.Conflict, unnecessary.StatusCode);
        using var pending = await SendAsync(HttpMethod.Get, "/api/v1/agent/deployments",
token: s.Token); Assert.Empty((await ReadAsync(pending)).EnumerateArray());
        await using var db = await OpenVendorAsync(VendorId); Assert.False(await
db.DeploymentAttempts.AnyAsync());
        await ReconcileAsync(); Assert.Equal("Reconciled", (await
db.Deployments.SingleAsync()).Status);
    }
    /// <summary>
    /// Reads pending deployment work for the test agent.
    /// </summary>
    private async Task<(StateFixture State, Guid Deployment)> PendingAsync()
    {
        var s = await StateAsync(); using var desired = await SetDesiredAsync(s);
Assert.Equal(HttpStatusCode.OK, desired.StatusCode);
        using var actual = await ReportAsync(s, "NotInstalled", DateTimeOffset.UtcNow);
Assert.Equal(HttpStatusCode.OK, actual.StatusCode);
        await ReconcileAsync(); await using var db = await OpenVendorAsync(VendorId);
        return (s, (await db.Deployments.SingleAsync()).Id);
    }
    /// <summary>
    /// Starts a deployment attempt through the test API.
    /// </summary>
    private Task<HttpResponseMessage> StartAsync(StateFixture s, Guid d) =>
SendAsync(HttpMethod.Post, $"/api/v1/agent/deployments/{d}/attempts", token: s.Token);
    /// <summary>
    /// Reports a deployment-attempt update through the test API.
    /// </summary>
    private Task<HttpResponseMessage> UpdateAsync(StateFixture s, Guid d, Guid a, string
status, string? code = null, DateTimeOffset? completed = null, string? message = null)
        => SendAsync(HttpMethod.Patch, $"/api/v1/agent/deployments/{d}/attempts/{a}", new
{ Status = status, ErrorCode = code, ErrorMessage = message, CompletedAt = completed }, s.Token);

    /// <summary>
    /// Verifies that concurrent starts reuse identity and only matching actual state
reconciles.
    /// </summary>
    [Fact]
    public async Task ConcurrentStarts_ReuseIdentity_AndOnlyMatchingActualStateReconciles()
    {
        var (s, d) = await PendingAsync();
        var starts = await Task.WhenAll(StartAsync(s, d), StartAsync(s, d));
        foreach (var response in starts) { Assert.Equal(HttpStatusCode.OK,
response.StatusCode); }
        Guid a = (await ReadAsync(starts[0])).GetProperty("attemptId").GetGuid();
        Assert.Equal(a, (await ReadAsync(starts[1])).GetProperty("attemptId").GetGuid());
        foreach (var response in starts) { response.Dispose(); }
        using var installing = await UpdateAsync(s, d, a, "Installing");
Assert.Equal(HttpStatusCode.OK, installing.StatusCode);
        using var configuring = await UpdateAsync(s, d, a, "Configuring");
Assert.Equal(HttpStatusCode.OK, configuring.StatusCode);
        using var backwards = await UpdateAsync(s, d, a, "Installing");
Assert.Equal(HttpStatusCode.Conflict, backwards.StatusCode);
    }

```

```

        DateTimeOffset completed = DateTimeOffset.UtcNow;
        using var success = await UpdateAsync(s, d, a, "Succeeded", completed: completed);
Assert.Equal(HttpStatusCode.OK, success.StatusCode);
        using var duplicate = await UpdateAsync(s, d, a, "Succeeded", completed:
completed); Assert.Equal(HttpStatusCode.OK, duplicate.StatusCode);
        using var rewrite = await UpdateAsync(s, d, a, "Failed", completed: completed);
Assert.Equal(HttpStatusCode.Conflict, rewrite.StatusCode);
        using var restart = await StartAsync(s, d); Assert.Equal(HttpStatusCode.Conflict,
restart.StatusCode);
        await using var db = await OpenVendorAsync(VendorId);
        Assert.Equal("AwaitingState", (await
db.Deployments.AsNoTracking().SingleAsync()).Status);
        using var actual = await ReportAsync(s, "Running",
DateTimeOffset.UtcNow.AddSeconds(1), s.Version, s.Configuration, s.Hash);
Assert.Equal(HttpStatusCode.OK, actual.StatusCode);
        await ReconcileAsync(); Assert.Equal("Reconciled", (await
db.Deployments.AsNoTracking().SingleAsync()).Status);
        Assert.Equal(1, await db.DeploymentAttempts.CountAsync());
        await Assert.ThrowsAsync<SqlException>(() =>
db.Database.ExecuteSqlRawAsync("UPDATE DeploymentAttempts SET ErrorMessage = 'changed'"));
        await Assert.ThrowsAsync<SqlException>(() =>
db.Database.ExecuteSqlRawAsync("DELETE FROM DeploymentAttempts"));
        using var detail = await SendAsync(HttpMethod.Get, $"/api/v1/deployments/{d}");
var body = await ReadAsync(detail);
        Assert.Equal("Succeeded",
body.GetProperty("attempts")[0].GetProperty("status").GetString());
    }

    /// <summary>
    /// Verifies that failures are redacted retries are bounded and completed history
survives.
    /// </summary>
    [Theory]
    [InlineData("MSI_INSTALL_FAILED", "The installer failed.")]
    [InlineData("CONFIGURATION_APPLY_FAILED", "Configuration application failed.")]
    [InlineData("DOWNLOAD_FAILED", "Artifact transfer or integrity verification failed.")]
    [InlineData("secret-code", "The Agent reported a deployment failure.")]
    public async Task
Failures_AreRedacted_RetriesAreBounded_AndCompletedHistorySurvives(string code, string
safeMessage)
    {
        var (s, d) = await PendingAsync();
        for (int number = 1; number <= 3; number++)
        {
            using var started = await StartAsync(s, d); Assert.Equal(HttpStatusCode.OK,
started.StatusCode);
            var start = await ReadAsync(started); Assert.Equal(number,
start.GetProperty("attemptNumber").GetInt32());
            Guid a = start.GetProperty("attemptId").GetGuid(); DateTimeOffset at =
Factory.Clock.UtcNow();
            using var failed = await UpdateAsync(s, d, a, "Failed", code, at,
"password=secret; C:\\private\\machine");
            Assert.Equal(HttpStatusCode.OK, failed.StatusCode);
            Assert.Equal(safeMessage, (await ReadAsync(failed)).GetProperty("errorMessage").GetString());
            using var duplicate = await UpdateAsync(s, d, a, "Failed", code, at,
"password=secret; C:\\private\\machine"); Assert.Equal(HttpStatusCode.OK, duplicate.StatusCode);
            using var immediateRetry = await StartAsync(s, d);
            Assert.Equal(HttpStatusCode.Conflict, immediateRetry.StatusCode);
            Factory.Clock.Offset += TimeSpan.FromSeconds(30 * number + 1);
        }
        using var exhausted = await StartAsync(s, d);
        Assert.Equal(HttpStatusCode.Conflict, exhausted.StatusCode);
    }

```

```

        using var actual = await ReportAsync(s, "NotInstalled",
Factory.Clock.UtcNow()); await ReconcileAsync();
        await using var db = await OpenVendorAsync(VendorId); Assert.Equal(1, await
db.Deployments.CountAsync());
        Assert.Equal("Failed", (await db.Deployments.SingleAsync()).Status);
Assert.Equal(3, await db.DeploymentAttempts.CountAsync());
        using var detail = await SendAsync(HttpMethod.Get, $"/api/v1/deployments/{d}");
Assert.DoesNotContain("password=", await detail.Content.ReadAsStringAsync());
    }

    /// <summary>
    /// Verifies that cancellation is idempotent and does not recreate on another report.
    /// </summary>
    [Fact]
    public async Task Cancellation_IsIdempotent_AndDoesNotRecreateOnAnotherReport()
    {
        var (s, d) = await PendingAsync();
        for (int n = 0; n < 2; n++) { using var cancel = await SendAsync(HttpMethod.Post,
$"/api/v1/deployments/{d}/cancel"); Assert.Equal(HttpStatusCode.OK, cancel.StatusCode); }
        using var start = await StartAsync(s, d); Assert.Equal(HttpStatusCode.Conflict,
start.StatusCode);
        using var report = await ReportAsync(s, "NotInstalled",
DateTimeOffset.UtcNow.AddSeconds(1)); await ReconcileAsync();
        using var pending = await SendAsync(HttpMethod.Get,
"/api/v1/agent/deployments?status=pending", token: s.Token); Assert.Empty((await
ReadAsync(pending)).EnumerateArray());
        await using var db = await OpenVendorAsync(VendorId); Assert.Equal(1, await
db.Deployments.CountAsync());
        Assert.Equal(1, await db.VendorAuditLogs.CountAsync(x => x.Action ==
"DeploymentCancelledByUser" && x.ActorId != Guid.Empty));
    }

    /// <summary>
    /// Verifies that cancel start race has one winner.
    /// </summary>
    [Fact]
    public async Task CancelStartRace_HasOneWinner()
    {
        var (s, d) = await PendingAsync();
        Task<HttpResponseBody> startTask = StartAsync(s, d);
        Task<HttpResponseBody> cancelTask = SendAsync(HttpMethod.Post,
$"/api/v1/deployments/{d}/cancel");
        using var start = await startTask; using var cancel = await cancelTask;
        Assert.Equal(new[] { HttpStatusCode.OK, HttpStatusCode.Conflict }, new[] {
start.StatusCode, cancel.StatusCode }.Order());
        await using var db = await OpenVendorAsync(VendorId);
        Assert.Equal(start.IsSuccessStatusCode ? 1 : 0, await
db.DeploymentAttempts.CountAsync());
    }

    /// <summary>
    /// Verifies that supersession blocks overlap but accepts old attempts final evidence.
    /// </summary>
    [Fact]
    public async Task Supersession_BlocksOverlap_ButAcceptsOldAttemptsFinalEvidence()
    {
        var (s, d) = await PendingAsync(); using var started = await StartAsync(s, d);
        Guid a = (await ReadAsync(started)).GetProperty("attemptId").GetGuid();
        using var changed = await
SetAsync($"/api/v1/installations/{s.Installation}/configuration", s.Version, new { port = 8080
});
    }

```

```

        Guid config = await GenerateAsync(s.Installation, s.Version); using var desired =
await SetDesiredAsync(s, configuration: config); Assert.Equal(HttpStatusCode.OK,
desired.StatusCode); await ReconcileAsync();
        await using var db = await OpenVendorAsync(VendorId); Guid next = (await
db.Deployments.SingleAsync(x => x.IsActive)).Id;
        using var stale = await StartAsync(s, d); Assert.Equal(HttpStatusCode.Conflict,
stale.StatusCode);
        using var overlap = await StartAsync(s, next);
Assert.Equal(HttpStatusCode.Conflict, overlap.StatusCode);
        using var pending = await SendAsync(HttpMethod.Get, "/api/v1/agent/deployments",
token: s.Token); Assert.Empty((await ReadAsync(pending)).EnumerateArray());
        using var stopped = await UpdateAsync(s, d, a, "Failed", "INTERRUPTED",
DateTimeOffset.UtcNow); Assert.Equal(HttpStatusCode.OK, stopped.StatusCode);
        using var allowed = await StartAsync(s, next); Assert.Equal(HttpStatusCode.OK,
allowed.StatusCode);
        Assert.Equal("Superseded", (await db.Deployments.AsNoTracking().SingleAsync(x =>
x.Id == d)).Status);
    }

    /// <summary>
    /// Verifies that queries and mutations enforce current ownership and scope.
    /// </summary>
    [Fact]
    public async Task QueriesAndMutations_EnforceCurrentOwnershipAndScope()
    {
        var (s, d) = await PendingAsync(); var otherAgent = await AgentAsync(s.Customer);
        using var foreignStart = await SendAsync(HttpMethod.Post,
$/api/v1/agent/deployments/{d}/attempts", token: otherAgent.Token);
Assert.Equal(HttpStatusCode.NotFound, foreignStart.StatusCode);
        using var foreignList = await SendAsync(HttpMethod.Get,
"/api/v1/agent/deployments", token: otherAgent.Token); Assert.Empty((await
ReadAsync(foreignList)).EnumerateArray());
        var user = await UserAsync("deploy-reader@example.test", "CustomerUser",
s.Customer);
        using var hidden = await SendAsync(HttpMethod.Get, "/api/v1/deployments", token:
user.Token); Assert.Equal(0, (await ReadAsync(hidden)).GetProperty("totalCount").GetInt32());
        using var hiddenDetail = await SendAsync(HttpMethod.Get,
$/api/v1/deployments/{d}", token: user.Token); Assert.Equal(HttpStatusCode.NotFound,
hiddenDetail.StatusCode);
        using var grant = await SendAsync(HttpMethod.Put,
$/api/v1/users/{user.Id}/installation-access", new { InstallationIds = new[] { s.Installation }
}); Assert.Equal(HttpStatusCode.NoContent, grant.StatusCode);
        using var list = await SendAsync(HttpMethod.Get,
$/api/v1/deployments?installationId={s.Installation}&status=Pending&pageSize=1", token:
user.Token); Assert.Equal(1, (await ReadAsync(list)).GetProperty("totalCount").GetInt32());
        using var history = await SendAsync(HttpMethod.Get,
$/api/v1/installations/{s.Installation}/deployments", token: user.Token); Assert.Equal(1, (await
ReadAsync(history)).GetProperty("items").GetArrayLength());
        using var forbidden = await SendAsync(HttpMethod.Post,
$/api/v1/deployments/{d}/cancel", token: user.Token); Assert.Equal(HttpStatusCode.Forbidden,
forbidden.StatusCode);
        using var started = await StartAsync(s, d); Guid a = (await
ReadAsync(started)).GetProperty("attemptId").GetGuid();
        using var foreignUpdate = await UpdateAsync(s with { Token = otherAgent.Token },
d, a, "Failed", completed: DateTimeOffset.UtcNow); Assert.Equal(HttpStatusCode.NotFound,
foreignUpdate.StatusCode);
        using var badPage = await SendAsync(HttpMethod.Get,
"/api/v1/agent/deployments?pageSize=101", token: s.Token);
Assert.Equal(HttpStatusCode.BadRequest, badPage.StatusCode);
    }

    /// <summary>

```

```

    /// Verifies that license revocation prevents dispatch but permits final attempt evidence.
    /// </summary>
    [Fact]
    public async Task LicenseRevocation_PreventsDispatch_ButPermitsFinalAttemptEvidence()
    {
        var (s, d) = await PendingAsync();
        using var pending = await SendAsync(HttpMethod.Get, "/api/v1/agent/deployments",
            token: s.Token); Assert.Single((await ReadAsync(pending)).EnumerateArray());
        using var started = await StartAsync(s, d); Guid a = (await
            ReadAsync(started)).GetProperty("attemptId").GetGuid();
        using var revoked = await SendAsync(HttpMethod.Post,
            $"/api/v1/licenses/{s.License}/deactivate"); Assert.Equal(HttpStatusCode.NoContent,
            revoked.StatusCode);
        using var retry = await StartAsync(s, d); Assert.Equal(HttpStatusCode.Conflict,
            retry.StatusCode);
        using var result = await UpdateAsync(s, d, a, "Failed", "INTERRUPTED",
            DateTimeOffset.UtcNow); Assert.Equal(HttpStatusCode.OK, result.StatusCode);
        Factory.Clock.Offset += TimeSpan.FromMinutes(1);
        using var hidden = await SendAsync(HttpMethod.Get, "/api/v1/agent/deployments",
            token: s.Token); Assert.Empty((await ReadAsync(hidden)).EnumerateArray());
    }

    /// <summary>
    /// Verifies that corrupt installer is not dispatched and audit failure rolls back start.
    /// </summary>
    [Fact]
    public async Task CorruptInstaller_IsNotDispatched_AndAuditFailureRollsBackStart()
    {
        var (s, d) = await PendingAsync(); await using var db = await
            OpenVendorAsync(VendorId);
        var artifact = await db.FileArtifacts.SingleAsync(x => x.Id == s.Installer);
        string path = Path.Combine(ArchiveRoot, "blobs", artifact.StorageIdentifier +
            ".blob"); byte[] original = await File.ReadAllBytesAsync(path); await
            File.WriteAllBytesAsync(path, [0]);
        using var hidden = await SendAsync(HttpMethod.Get, "/api/v1/agent/deployments",
            token: s.Token); Assert.Empty((await ReadAsync(hidden)).EnumerateArray());
        using var denied = await StartAsync(s, d); Assert.Equal(HttpStatusCode.Conflict,
            denied.StatusCode); Assert.False(await db.DeploymentAttempts.AnyAsync());
        await File.WriteAllBytesAsync(path, original);
        await db.Database.ExecuteSqlRawAsync("CREATE TRIGGER FailAttemptAudit ON
            VendorAuditLogs AFTER INSERT AS BEGIN IF EXISTS (SELECT 1 FROM inserted WHERE Action =
            'DeploymentAttemptStarted') THROW 51000, 'test failure', 1; END");
        using var failed = await StartAsync(s, d);
        Assert.Equal(HttpStatusCode.InternalServerError, failed.StatusCode); Assert.False(await
            db.DeploymentAttempts.AnyAsync());
        await db.Database.ExecuteSqlRawAsync("DROP TRIGGER FailAttemptAudit");
        using var retried = await StartAsync(s, d); Assert.Equal(HttpStatusCode.OK,
            retried.StatusCode); Assert.Equal(1, (await
            ReadAsync(retried)).GetProperty("attemptNumber").GetInt32());
    }
}

```

Integration/EffectiveConfigurationAccessTests.cs

```

using System.Net;
using System.Text;
using BridgeControl.Tests.Infrastructure;
using Microsoft.EntityFrameworkCore;

namespace BridgeControl.Tests.Integration;

```

```

/// <summary>
/// Verifies effective configuration access behavior and regression scenarios.
/// </summary>
[Trait("Category", "Integration")]
public class EffectiveConfigurationAccessTests : EffectiveConfigurationTest
{
    /// <summary>
    /// Verifies that every read enforces installation scope including specific revision
    lookups.
    /// </summary>
    [Fact]
    public async Task EveryRead_EnforcesInstallationScopeIncludingSpecificRevisionLookups()
    {
        await PrepareAsync(); var catalog = await CatalogAsync(); Guid customer = await
CustomerAsync(); Guid other = await CustomerAsync("Other");
        Guid installation = await InstallationAsync(customer, catalog.Product), foreign =
await InstallationAsync(other, catalog.Product);
        Guid revision = await GenerateAsync(installation, catalog.Version),
foreignRevision = await GenerateAsync(foreign, catalog.Version);
        var user = await UserAsync("reader@example.test", "CustomerUser", customer); var
admin = await UserAsync("customeradmin@example.test", "CustomerAdministrator", customer);
        var technician = await UserAsync("technician@example.test", "VendorTechnician");
        string[] ownPaths =
["$"/api/v1/installations/{installation}/effective-configuration",
$"/api/v1/installations/{installation}/effective-configuration/revisions",
$"/api/v1/installations/{installation}/effective-configuration/revisions/{revision}"];
        foreach (string path in ownPaths)
        {
            using var denied = await SendAsync(HttpMethod.Get, path, token:
user.Token); Assert.Equal(HttpStatusCode.NotFound, denied.StatusCode);
            using var allowed = await SendAsync(HttpMethod.Get, path, token:
admin.Token); Assert.Equal(HttpStatusCode.OK, allowed.StatusCode);
        }
        using (var grant = await SendAsync(HttpMethod.Put,
$"/api/v1/users/{user.Id}/installation-access", new { InstallationIds = new[] { installation }
})) { Assert.Equal(HttpStatusCode.NoContent, grant.StatusCode); }
        using (var grant = await SendAsync(HttpMethod.Put,
$"/api/v1/users/{technician.Id}/customer-access", new { CustomerIds = new[] { customer } })) {
Assert.Equal(HttpStatusCode.NoContent, grant.StatusCode); }
        foreach (string token in new[] { user.Token, technician.Token, admin.Token })
        {
            foreach (string path in ownPaths) { using var allowed = await
SendAsync(HttpMethod.Get, path, token: token); Assert.Equal(HttpStatusCode.OK,
allowed.StatusCode); }
            using var hidden = await SendAsync(HttpMethod.Get,
$"/api/v1/installations/{installation}/effective-configuration/revisions/{foreignRevision}",
token: token); Assert.Equal(HttpStatusCode.NotFound, hidden.StatusCode);
            using var foreignList = await SendAsync(HttpMethod.Get,
$"/api/v1/installations/{foreign}/effective-configuration/revisions", token: token);
Assert.Equal(HttpStatusCode.NotFound, foreignList.StatusCode);
        }
        using (var revoke = await SendAsync(HttpMethod.Put,
$"/api/v1/users/{user.Id}/installation-access", new { InstallationIds = Array.Empty<Guid>() })) {
Assert.Equal(HttpStatusCode.NoContent, revoke.StatusCode); }
        using (var hidden = await SendAsync(HttpMethod.Get, ownPaths[2], token:
user.Token)) { Assert.Equal(HttpStatusCode.NotFound, hidden.StatusCode); }
        using var invalid = await SendAsync(HttpMethod.Get, ownPaths[1] +
"?pageSize=101"); Assert.Equal(HttpStatusCode.BadRequest, invalid.StatusCode);
    }

    /// <summary>

```

```

        /// Verifies that schema replacement revalidates exact target and retains historical
values.
        /// </summary>
        [Fact]
        public async Task SchemaReplacement_RevalidatesExactTargetAndRetainsHistoricalValues()
        {
            await PrepareAsync(); var catalog = await CatalogAsync(); Guid installation =
await InstallationAsync(await CustomerAsync(), catalog.Product);
            using (var source = await
SetAsync($"api/v1/products/{catalog.Product}/configuration", catalog.Version, new { port = 500
})) { Assert.Equal(HttpStatusCode.OK, source.StatusCode); }
            Guid original = await GenerateAsync(installation, catalog.Version);
            const string stricter =
""{"$schema":"https://json-schema.org/draft/2020-12/schema","type":"object","properties":{"port"
:{"type":"integer","maximum":10}}"";
            using (var changed = await UploadAsync(catalog.Product, catalog.Version, "schema",
Encoding.UTF8.GetBytes(stricter))) { Assert.Equal(HttpStatusCode.OK, changed.StatusCode); }
            await DrainAsync();
            using (var current = await SendAsync(HttpMethod.Get,
$"api/v1/installations/{installation}/effective-configuration")) { Assert.Equal(original, (await
ReadAsync(current)).GetProperty("revisionId").GetGuid()); }
            await using var db = await OpenVendorAsync(VendorId); var failure = await
db.EffectiveConfigurationWork.SingleAsync(); Assert.Equal("Failed", failure.Status);
Assert.Contains("/port", failure.LastError);
            using (var fixedSource = await
SetAsync($"api/v1/products/{catalog.Product}/configuration", catalog.Version, new { port = 5 }))
{ Assert.Equal(HttpStatusCode.OK, fixedSource.StatusCode); }
            await DrainAsync();
            using var history = await SendAsync(HttpMethod.Get,
$"api/v1/installations/{installation}/effective-configuration/revisions/{original}");
Assert.Equal(500, (await
ReadAsync(history)).GetProperty("values").GetProperty("port").GetInt32());
        }

        /// <summary>
        /// Verifies that new registration after source event was processed still gets effective
configuration.
        /// </summary>
        [Fact]
        public async Task
NewRegistration_AfterSourceEventWasProcessed_StillGetsEffectiveConfiguration()
        {
            await PrepareAsync(); var catalog = await CatalogAsync(); Guid customer = await
CustomerAsync();
            using (var source = await
SetAsync($"api/v1/products/{catalog.Product}/configuration", catalog.Version, new { port = 88
})) { Assert.Equal(HttpStatusCode.OK, source.StatusCode); }
            await DrainAsync();
            using var enrollment = await SendAsync(HttpMethod.Post,
"/api/v1/agent-enrollments", new { CustomerId = customer });
            using var registered = await SendAsync(HttpMethod.Post, "/api/v1/agent/register",
new { EnrollmentCode = (await ReadAsync(enrollment)).GetProperty("enrollmentCode").GetString(),
MachineName = "Machine", AgentVersion = "1" });
            using var authenticated = await SendAsync(HttpMethod.Post,
"/api/v1/agent/auth/token", new { Credential = (await
ReadAsync(registered)).GetProperty("credential").GetString() });
            string token = (await
ReadAsync(authenticated)).GetProperty("accessToken").GetString();
            using var response = await SendAsync(HttpMethod.Post,
"/api/v1/agent/installations", new { ProductId = catalog.Product, Name = "Service" }, token);
            Assert.Equal(HttpStatusCode.Created, response.StatusCode); Guid installation =
(await ReadAsync(response)).GetProperty("id").GetGuid();

```

```

        await DrainAsync(); using var current = await SendAsync(HttpMethod.Get,
$"/api/v1/installations/{installation}/effective-configuration"); Assert.Equal(88, (await
ReadAsync(current)).GetProperty("values").GetProperty("port").GetInt32());
        using var machineRead = await SendAsync(HttpMethod.Get,
$"/api/v1/installations/{installation}/effective-configuration", token: token);
Assert.Equal(HttpStatusCode.Unauthorized, machineRead.StatusCode);
    }
}

```

Integration/EffectiveConfigurationReliabilityTests.cs

```

using System.Net;
using System.Text;
using BridgeControl.Api.Exceptions;
using BridgeControl.Models.Vendor;
using BridgeControl.Services.Vendor;
using BridgeControl.Tests.Infrastructure;
using Microsoft.Data.SqlClient;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Infrastructure;
using Microsoft.EntityFrameworkCore.Migrations;
using Microsoft.Extensions.DependencyInjection;

namespace BridgeControl.Tests.Integration;

/// <summary>
/// Verifies effective configuration reliability behavior and regression scenarios.
/// </summary>
[Trait("Category", "Integration")]
public class EffectiveConfigurationReliabilityTests : EffectiveConfigurationTest
{
    /// <summary>
    /// Verifies that concurrent changes and generation always converge to newest sources.
    /// </summary>
    [Fact]
    public async Task ConcurrentChangesAndGeneration_AlwaysConvergeToNewestSources()
    {
        await PrepareAsync(); var catalog = await CatalogAsync(); Guid installation =
await InstallationAsync(await CustomerAsync(), catalog.Product);
        async Task GenerateInScope()
        {
            await using var scope = Factory.Services.CreateAsyncScope();
            await
scope.ServiceProvider.GetRequiredService<EffectiveConfigurationGenerator>().GenerateAsync(VendorId,
installation, catalog.Version);
        }
        var update = SetAsync($"/api/v1/installations/{installation}/configuration",
catalog.Version, new { port = 2 });
        await Task.WhenAll(GenerateInScope(), GenerateInScope(), update);
        using (var response = await update) { Assert.Equal(HttpStatusCode.OK,
response.StatusCode); }
        using (var newest = await
SetAsync($"/api/v1/installations/{installation}/configuration", catalog.Version, new { port = 3
})) { Assert.Equal(HttpStatusCode.OK, newest.StatusCode); }
        await DrainAsync();
        using var current = await SendAsync(HttpMethod.Get,
$"/api/v1/installations/{installation}/effective-configuration");
        Assert.Equal(3, (await
ReadAsync(current)).GetProperty("values").GetProperty("port").GetInt32());
    }
}

```

```

        await using var db = await OpenVendorAsync(VendorId); int count = await
db.EffectiveConfigurationRevisions.CountAsync();
        await Task.WhenAll(GenerateInScope(), GenerateInScope()); Assert.Equal(count,
await db.EffectiveConfigurationRevisions.CountAsync());
    }

    /// <summary>
    /// Verifies that hosted worker resumes durable work after fresh host starts.
    /// </summary>
    [Fact]
    public async Task HostedWorker_ResumesDurableWorkAfterFreshHostStarts()
    {
        await PrepareAsync(); var catalog = await CatalogAsync(); Guid installation =
await InstallationAsync(await CustomerAsync(), catalog.Product);
        using (var source = await
SetAsync($"api/v1/installations/{installation}/configuration", catalog.Version, new { port = 123
})) { Assert.Equal(HttpStatusCode.OK, source.StatusCode); }
        await using var restarted = new
ApiFactory(Database.Database.GetConnectionString()!, configurationWorker: true, archiveRoot:
Factory.ArchiveRoot);
        using var client = restarted.CreateApiClient();
        using var deadline = new CancellationTokenSource(TimeSpan.FromSeconds(30));
        await using var db = await OpenVendorAsync(VendorId);
        while (!await db.Installations.AsNoTracking().AnyAsync(x => x.Id == installation
&& x.CurrentEffectiveConfigurationRevisionId != null, deadline.Token))
        { await Task.Delay(100, deadline.Token); }
        using var current = await SendAsync(HttpMethod.Get,
$"api/v1/installations/{installation}/effective-configuration"); Assert.Equal(HttpStatusCode.OK,
current.StatusCode);
        Assert.Equal(123, (await
ReadAsync(current)).GetProperty("values").GetProperty("port").GetInt32());
    }

    /// <summary>
    /// Verifies that fanout resumes after one hundred targets without skipping installations.
    /// </summary>
    [Fact]
    public async Task Fanout_ResumesAfterOneHundredTargets_WithoutSkippingInstallations()
    {
        await PrepareAsync(); var catalog = await CatalogAsync(); Guid customer = await
CustomerAsync();
        await using var db = await OpenVendorAsync(VendorId);
        db.Installations.AddRange(Enumerable.Range(0, 101).Select(i => new Installation {
CustomerId = customer, ProductId = catalog.Product, Name = "Machine " + i }));
        await db.SaveChangesAsync();
        using (var source = await
SetAsync($"api/v1/products/{catalog.Product}/configuration", catalog.Version, new { port = 1 }))
        { Assert.Equal(HttpStatusCode.OK, source.StatusCode); }
        await
Services.GetRequiredService<EffectiveConfigurationProcessor>().ProcessVendorAsync(VendorId);
        var notification = await
db.ConfigurationSourceChanges.AsNoTracking().SingleAsync();
        Assert.Null(notification.ProcessedAt); Assert.NotNull(notification.LastInstallationId);
        await DrainAsync(); Assert.Equal(101, await
db.EffectiveConfigurationRevisions.CountAsync());
        Assert.Equal(101, await db.EffectiveConfigurationWork.CountAsync(x => x.Status ==
"Succeeded"));
        Assert.Equal(101, await db.Installations.CountAsync(x =>
x.CurrentEffectiveConfigurationRevisionId != null));
    }

    /// <summary>

```

```

    /// Verifies that retry budget is bounded and new source change resets failed work.
    /// </summary>
    [Fact]
    public async Task RetryBudget_IsBounded_AndNewSourceChangeResetsFailedWork()
    {
        await PrepareAsync(); var catalog = await CatalogAsync(); Guid installation =
        await InstallationAsync(await CustomerAsync(), catalog.Product); await DrainAsync();
        await using var db = await OpenVendorAsync(VendorId); Guid original = (await
        db.Installations.AsNoTracking().SingleAsync()).CurrentEffectiveConfigurationRevisionId!.Value;
        using (var source = await
        SetAsync($"api/v1/installations/{installation}/configuration", catalog.Version, new { port = 1
        })) { Assert.Equal(HttpStatusCode.OK, source.StatusCode); }
        await db.Database.ExecuteSqlRawAsync("CREATE TRIGGER FailEffectiveGeneration ON
        VendorAuditLogs AFTER INSERT AS BEGIN IF EXISTS (SELECT 1 FROM inserted WHERE Action =
        'EffectiveConfigurationGenerated') THROW 51000, 'test failure', 1; END");
        for (int attempt = 1; attempt <= 3; attempt++)
        {
            await
            Services.GetRequiredService<EffectiveConfigurationProcessor>().ProcessVendorAsync(VendorId);
            var work = await
            db.EffectiveConfigurationWork.AsNoTracking().SingleAsync(); Assert.Equal(attempt, work.Attempts);
            Assert.Equal(attempt == 3 ? "Failed" : "Pending", work.Status);
            await db.EffectiveConfigurationWork.ExecuteUpdateAsync(s => s.SetProperty(x
            => x.NextAttemptAt, DateTimeOffset.UtcNow.AddSeconds(-1)));
        }
        Assert.Equal(original, (await
        db.Installations.AsNoTracking().SingleAsync()).CurrentEffectiveConfigurationRevisionId);
        await db.Database.ExecuteSqlRawAsync("DROP TRIGGER FailEffectiveGeneration");
        using (var source = await
        SetAsync($"api/v1/installations/{installation}/configuration", catalog.Version, new { port = 2
        })) { Assert.Equal(HttpStatusCode.OK, source.StatusCode); }
        await DrainAsync(); Assert.Equal("Succeeded", (await
        db.EffectiveConfigurationWork.AsNoTracking().SingleAsync()).Status);
    }

    /// <summary>
    /// Verifies that history is immutable and migration preserves legacy revisions and
    counters.
    /// </summary>
    [Fact]
    public async Task History_IsImmutable_AndMigrationPreservesLegacyRevisionsAndCounters()
    {
        await PrepareAsync(); var catalog = await CatalogAsync(); Guid installation =
        await InstallationAsync(await CustomerAsync(), catalog.Product);
        using (var source = await
        SetAsync($"api/v1/products/{catalog.Product}/configuration", catalog.Version, new { port = 1 }))
        { Assert.Equal(HttpStatusCode.OK, source.StatusCode); }
        Guid revision = await GenerateAsync(installation, catalog.Version);
        await using var db = await OpenVendorAsync(VendorId);
        await Assert.ThrowsAsync<SqlException>(() =>
        db.Database.ExecuteSqlRawAsync("UPDATE EffectiveConfigurationRevisions SET RevisionNumber = 2"));
        await Assert.ThrowsAsync<SqlException>(() =>
        db.Database.ExecuteSqlRawAsync("DELETE FROM EffectiveConfigurationSourceRevisions"));
        var migrator = db.GetService<IMigrator>();
        await migrator.MigrateAsync(db.Database.GetMigrations().Single(x =>
        x.EndsWith("_AddInstallationLicensing", StringComparison.Ordinal)));
        await migrator.MigrateAsync(); db.ChangeTracker.Clear();
        Assert.Null((await
        db.EffectiveConfigurationRevisions.SingleAsync()).SchemaArtifactId);
        Assert.Equal(1, (await
        db.Installations.SingleAsync()).LastEffectiveRevisionNumber);
    }

```

```

        using (var old = await SendAsync(HttpMethod.Get,
$"/api/v1/installations/{installation}/effective-configuration/revisions/{revision}")) {
Assert.Equal(HttpStatusCode.OK, old.StatusCode); }

        await GenerateAsync(installation, catalog.Version); Assert.Equal(2, await
db.EffectiveConfigurationRevisions.MaxAsync(x => x.RevisionNumber));
Assert.False(db.Database.HasPendingModelChanges());
    }

    /// <summary>
    /// Verifies that published default source cannot be taken from another customer profile.
    /// </summary>
    [Fact]
    public async Task PublishedDefaultSource_CannotBeTakenFromAnotherCustomerProfile()
    {
        await PrepareAsync(); var catalog = await CatalogAsync(installer: true); Guid
customer = await CustomerAsync(); Guid installation = await InstallationAsync(customer,
catalog.Product);

        using var source = await
SetAsync($"/api/v1/customers/{customer}/products/{catalog.Product}/configuration",
catalog.Version, new { port = 1 });
        Guid sourceId = (await ReadAsync(source)).GetProperty("id").GetGuid();
        using (var published = await PublishAsync(catalog.Product, catalog.Version)) {
Assert.Equal(HttpStatusCode.OK, published.StatusCode); }
        await using var db = await OpenVendorAsync(VendorId);
        await db.ProductVersions.Where(x => x.Id == catalog.Version).ExecuteUpdateAsync(s
=> s.SetProperty(x => x.PublishedDefaultRevisionId, sourceId));
        await Assert.ThrowsAsync<ApiConflictException>(() => GenerateAsync(installation,
catalog.Version)); Assert.Empty(await db.EffectiveConfigurationRevisions.ToArrayAsync());
    }
}

```

Integration/EffectiveConfigurationTests.cs

```

using System.Net;
using System.Text;
using System.Text.Json;
using BridgeControl.Api.Exceptions;
using BridgeControl.Tests.Infrastructure;
using Microsoft.EntityFrameworkCore;

namespace BridgeControl.Tests.Integration;

/// <summary>
/// Verifies effective configuration behavior and regression scenarios.
/// </summary>
[Trait("Category", "Integration")]
public class EffectiveConfigurationTests : EffectiveConfigurationTest
{
    private const string MergeSchema =
    """{"$schema":"https://json-schema.org/draft/2020-12/schema","type":"object","properties":{"netwo
rk":{"type":"object","properties":{"port":{"type":"integer"},"host":{"type":"string"}}},"tags":{"
type":"array"},"empty":{"type":"object"},"additionalProperties":false}}""";

    /// <summary>
    /// Verifies that generation merges three layers preserves lineage and reuses same input.
    /// </summary>
    [Fact]
    public async Task Generation_MergesThreeLayers_PreservesLineageAndReusesSameInput()
    {

```

```

        await PrepareAsync(); var catalog = await CatalogAsync(schema: MergeSchema); Guid
customer = await CustomerAsync(); Guid installation = await InstallationAsync(customer,
catalog.Product);
        using var defaults = await
SetAsync($"api/v1/products/{catalog.Product}/configuration", catalog.Version, new { network =
new { port = 80, host = "base" }, tags = new[] { "base", "old" }, empty = new { } });
        using var client = await
SetAsync($"api/v1/customers/{customer}/products/{catalog.Product}/configuration",
catalog.Version, new { network = new { port = 443 }, tags = new object?[] { "customer", null }
});
        using var local = await
SetAsync($"api/v1/installations/{installation}/configuration", catalog.Version, new { network =
new { host = "machine" } });
        Guid id = await GenerateAsync(installation, catalog.Version); Assert.Equal(id,
await GenerateAsync(installation, catalog.Version));
        using var current = await SendAsync(HttpMethod.Get,
$"api/v1/installations/{installation}/effective-configuration"); Assert.Equal(HttpStatusCode.OK,
current.StatusCode); var body = await ReadAsync(current);
        Assert.Equal(new[] { "createdAt", "installationId", "productVersionId",
"revisionId", "revisionNumber", "sourceRevisionIds", "values" }, body.EnumerateObject().Select(x
=> x.Name).Order());
        Assert.Equal(443,
body.GetProperty("values").GetProperty("network").GetProperty("port").GetInt32());
        Assert.Equal("machine",
body.GetProperty("values").GetProperty("network").GetProperty("host").GetString());
        Assert.Equal(2, body.GetProperty("values").GetProperty("tags").GetArrayLength());
        Assert.Equal(JsonValueKind.Null, body.GetProperty("values").GetProperty("tags")[1].ValueKind);
        Assert.True(body.GetProperty("values").TryGetProperty("empty", out _));
        Assert.Equal(3, body.GetProperty("sourceRevisionIds").GetArrayLength());
        await using var db = await OpenVendorAsync(VendorId); Assert.Equal(1, await
db.EffectiveConfigurationRevisions.CountAsync()); Assert.Equal(3, await
db.EffectiveConfigurationSourceRevisions.CountAsync());
        Assert.NotNull(await
db.EffectiveConfigurationRevisions.SingleAsync()).SchemaArtifactId);
        using var historical = await SendAsync(HttpMethod.Get,
$"api/v1/installations/{installation}/effective-configuration/revisions/{id}");
        Assert.Equal(HttpStatusCode.OK, historical.StatusCode);
    }

    /// <summary>
    /// Verifies that invalid merged values preserve previous pointer and record property
level failure.
    /// </summary>
    [Fact]
    public async Task
InvalidMergedValues_PreservePreviousPointer_AndRecordPropertyLevelFailure()
    {
        const string schema =
""{"$schema":"https://json-schema.org/draft/2020-12/schema","type":"object","maxProperties":1,"p
roperties":{"left":{"type":"integer"},"right":{"type":"integer"}}}"";
        await PrepareAsync(); var catalog = await CatalogAsync(schema: schema); Guid
customer = await CustomerAsync(); Guid installation = await InstallationAsync(customer,
catalog.Product);
        using (var source = await
SetAsync($"api/v1/products/{catalog.Product}/configuration", catalog.Version, new { left = 1 }))
        { Assert.Equal(HttpStatusCode.OK, source.StatusCode); }
        Guid original = await GenerateAsync(installation, catalog.Version);
        using (var source = await
SetAsync($"api/v1/installations/{installation}/configuration", catalog.Version, new { right = 2
})) { Assert.Equal(HttpStatusCode.OK, source.StatusCode); }
        await DrainAsync(); await using var db = await OpenVendorAsync(VendorId);

```

```

        Assert.Equal(original, (await
db.Installations.SingleAsync()).CurrentEffectiveConfigurationRevisionId); Assert.Equal(1, await
db.EffectiveConfigurationRevisions.CountAsync());
        var failed = await db.EffectiveConfigurationWork.SingleAsync();
Assert.Equal("Failed", failed.Status); Assert.Contains("maxProperties", failed.LastError);
        using (var fixedInput = await
SetAsync($"/api/v1/installations/{installation}/configuration", catalog.Version, new { left = 2
})) { Assert.Equal(HttpStatusCode.OK, fixedInput.StatusCode); }
        await DrainAsync(); db.ChangeTracker.Clear(); Assert.Equal("Succeeded", (await
db.EffectiveConfigurationWork.SingleAsync()).Status);
        Assert.Equal(2, await db.EffectiveConfigurationRevisions.CountAsync());
    }

    /// <summary>
    /// Verifies that version specific history does not switch existing target and publication
pins absence.
    /// </summary>
    [Fact]
    public async Task
VersionSpecificHistory_DoesNotSwitchExistingTarget_AndPublicationPinsAbsence()
    {
        await PrepareAsync(); var catalog = await CatalogAsync(installer: true); Guid
customer = await CustomerAsync(); Guid installation = await InstallationAsync(customer,
catalog.Product);
        using (var published = await PublishAsync(catalog.Product, catalog.Version)) {
Assert.Equal(HttpStatusCode.OK, published.StatusCode); }
        Guid first = await GenerateAsync(installation, catalog.Version);
        Guid next = await VersionAsync(catalog.Product, "2");
        using (var value = await
SetAsync($"/api/v1/products/{catalog.Product}/configuration", next, new { port = 9000 })) {
Assert.Equal(HttpStatusCode.OK, value.StatusCode); }
        await DrainAsync();
        using (var current = await SendAsync(HttpMethod.Get,
$"/api/v1/installations/{installation}/effective-configuration"))
        { var body = await ReadAsync(current); Assert.Equal(first,
body.GetProperty("revisionId").GetGuid());
Assert.Empty(body.GetProperty("values").EnumerateObject()); }
        await GenerateAsync(installation, next, selectTarget: true);
        using var selected = await SendAsync(HttpMethod.Get,
$"/api/v1/installations/{installation}/effective-configuration"); Assert.Equal(next, (await
ReadAsync(selected)).GetProperty("productVersionId").GetGuid());
        using var history = await SendAsync(HttpMethod.Get,
$"/api/v1/installations/{installation}/effective-configuration/revisions?pageSize=1");
        Assert.Equal(2, (await ReadAsync(history)).GetProperty("totalCount").GetInt32());
    }

    /// <summary>
    /// Verifies that recalculation is scoped to affected customer and installation.
    /// </summary>
    [Fact]
    public async Task Recalculation_IsScopedToAffectedCustomerAndInstallation()
    {
        await PrepareAsync(); var catalog = await CatalogAsync(); Guid customer = await
CustomerAsync(); Guid other = await CustomerAsync("Other");
        Guid a = await InstallationAsync(customer, catalog.Product), b = await
InstallationAsync(other, catalog.Product);
        using (var source = await
SetAsync($"/api/v1/products/{catalog.Product}/configuration", catalog.Version, new { port = 1 }))
        { Assert.Equal(HttpStatusCode.OK, source.StatusCode); }
        await DrainAsync(); await using var db = await OpenVendorAsync(VendorId); Guid
oldB = (await db.Installations.AsNoTracking().SingleAsync(x => x.Id ==
b)).CurrentEffectiveConfigurationRevisionId!.Value;

```

```

        using (var source = await
SetAsync($"api/v1/customers/{customer}/products/{catalog.Product}/configuration",
catalog.Version, new { port = 2 })) { Assert.Equal(HttpStatusCode.OK, source.StatusCode); }
        await DrainAsync(); Assert.Equal(oldB, await
db.Installations.AsNoTracking().SingleAsync(x => x.Id ==
b)).CurrentEffectiveConfigurationRevisionId);
        Assert.Equal(2, await db.EffectiveConfigurationRevisions.CountAsync(x =>
x.InstallationId == a)); Assert.Equal(1, await db.EffectiveConfigurationRevisions.CountAsync(x =>
x.InstallationId == b));
    }

    /// <summary>
    /// Verifies that audit failure and corrupt artifacts cannot replace valid state.
    /// </summary>
    [Fact]
    public async Task AuditFailureAndCorruptArtifacts_CannotReplaceValidState()
    {
        await PrepareAsync(); var catalog = await CatalogAsync(); Guid installation =
await InstallationAsync(await CustomerAsync(), catalog.Product);
        using (var source = await
SetAsync($"api/v1/products/{catalog.Product}/configuration", catalog.Version, new { port = 1 }))
        { Assert.Equal(HttpStatusCode.OK, source.StatusCode); }
        Guid original = await GenerateAsync(installation, catalog.Version);
        using (var source = await
SetAsync($"api/v1/products/{catalog.Product}/configuration", catalog.Version, new { port = 2 }))
        { Assert.Equal(HttpStatusCode.OK, source.StatusCode); }
        await using var db = await OpenVendorAsync(VendorId);
        await db.Database.ExecuteSqlRawAsync("CREATE TRIGGER FailEffectiveAudit ON
VendorAuditLogs AFTER INSERT AS BEGIN IF EXISTS (SELECT 1 FROM inserted WHERE Action =
'EffectiveConfigurationGenerated') THROW 51000, 'test failure', 1; END");
        await Assert.ThrowsAsync<DbUpdateException>(() => GenerateAsync(installation,
catalog.Version));
        Assert.Equal(original, await
db.Installations.AsNoTracking().SingleAsync()).CurrentEffectiveConfigurationRevisionId);
        Assert.Equal(1, await db.EffectiveConfigurationRevisions.CountAsync());
        await db.Database.ExecuteSqlRawAsync("DROP TRIGGER FailEffectiveAudit");
        var artifact = await db.ConfigurationRevisions.OrderByDescending(x =>
x.RevisionNumber).Select(x => x.FileArtifact).FirstAsync();
        await File.WriteAllTextAsync(Path.Combine(ArchiveRoot, "blobs",
artifact.StorageIdentifier + ".blob"), "{}");
        await Assert.ThrowsAsync<ApiConflictException>(() => GenerateAsync(installation,
catalog.Version));
        Assert.Equal(1, await db.EffectiveConfigurationRevisions.CountAsync());
    }
}

```

Integration/HostingProfileTests.cs

```

using System.Net;
using BridgeControl.Options;
using BridgeControl.Tests.Infrastructure;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace BridgeControl.Tests.Integration;

/// <summary>
/// Verifies hosting profile behavior and regression scenarios.
/// </summary>
public sealed class HostingProfileTests

```

```

{
    private const string Connection =
"Server=(localdb)\\MSSQLLocalDB;Database=BridgeControl_Unused;Integrated
Security=True;TrustServerCertificate=True";
    /// <summary>
    /// Verifies that only configured proxy can assert https.
    /// </summary>
    [Theory]
    [InlineData("192.0.2.10", true, 200)]
    [InlineData("192.0.2.11", true, 400)]
    [InlineData("192.0.2.11", false, 400)]
    public async Task OnlyConfiguredProxyCanAssertHttps(string peer, bool proxyRequired, int
expected)
    {
        await using ApiFactory factory = new(Connection, configure: services =>
services.PostConfigure<HostingProfileOptions>(options =>
        { options.RequireProxy = proxyRequired; options.TrustedProxies = ["192.0.2.10"];
    }));
        using HttpClient client = factory.CreateApiClient();
        HttpContext context = await factory.Server.SendAsync(request =>
        {
            request.Connection.RemoteIpAddress = IPAddress.Parse(peer);
            request.Request.Method = "GET"; request.Request.Scheme = "http";
request.Request.Host = new("localhost"); request.Request.Path = "/api/v1/unimplemented";
            request.Request.Headers["X-Forwarded-For"] = "203.0.113.4";
request.Request.Headers["X-Forwarded-Proto"] = "https";
        });
        // HTTPS succeeds through the trusted proxy and reaches the protected API routing
boundary.
        Assert.Equal(expected == 200 ? 401 : expected, context.Response.StatusCode);
        if (expected == 200) { Assert.Equal("203.0.113.4",
context.Connection.RemoteIpAddress!.ToString()); }
    }
    /// <summary>
    /// Verifies that browser writes reject foreign origin or host.
    /// </summary>
    [Theory]
    [InlineData("https://attacker.test", "localhost")]
    [InlineData("null", "localhost")]
    [InlineData("https://localhost", "attacker.test")]
    public async Task BrowserWritesRejectForeignOriginOrHost(string origin, string host)
    {
        await using ApiFactory factory = new(Connection); using HttpClient client =
factory.CreateApiClient();
        using HttpRequestMessage request = new(HttpMethod.Post, "https://" + host +
"/api/v1/auth/refresh"); request.Headers.Add("Origin", origin);
        using HttpResponseMessage response = await client.SendAsync(request);
        Assert.Equal(HttpStatusCode.BadRequest, response.StatusCode);
    }
}

```

Integration/InstallationAdministrationTests.cs

```

using System.Net;
using BridgeControl.Models.Vendor;
using BridgeControl.Tests.Infrastructure;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Infrastructure;
using Microsoft.EntityFrameworkCore.Migrations;

```

```

namespace BridgeControl.Tests.Integration;

/// <summary>
/// Verifies installation administration behavior and regression scenarios.
/// </summary>
[Trait("Category", "Integration")]
public class InstallationAdministrationTests : LicensingTest
{
    /// <summary>
    /// Verifies that rename and lifecycle preserve agent retry identity and registration
    metadata.
    /// </summary>
    [Fact]
    public async Task RenameAndLifecycle_PreserveAgentRetryIdentityAndRegistrationMetadata()
    {
        await PrepareAsync(); Guid customer = await CustomerAsync(); Guid product = await
ProductAsync(); var managed = await ManagedAsync(customer, product);
        using var before = await SendAsync(HttpMethod.Get,
$"/api/v1/installations/{managed.Installation}"); var original = await ReadAsync(before);
        using (var renamed = await SendAsync(HttpMethod.Patch,
$"/api/v1/installations/{managed.Installation}", new { Name = " Friendly name " })) {
Assert.Equal(HttpStatusCode.OK, renamed.StatusCode); }
        using (var empty = await SendAsync(HttpMethod.Patch,
$"/api/v1/installations/{managed.Installation}", new { })) { Assert.Equal("Friendly name", (await
ReadAsync(empty)).GetProperty("name").GetString()); }
        using (var retry = await SendAsync(HttpMethod.Post, "/api/v1/agent/installations",
new { ProductId = product, Name = "Service" }, managed.AgentToken))
        { var body = await ReadAsync(retry); Assert.Equal(managed.Installation,
body.GetProperty("id").GetGuid()); Assert.Equal("Friendly name",
body.GetProperty("name").GetString());
Assert.Equal(original.GetProperty("registeredAt").GetString(),
body.GetProperty("registeredAt").GetString()); }
        foreach (object invalid in new object[] { new { AgentId = Guid.NewGuid() }, new {
CustomerId = Guid.NewGuid() }, new { ProductId = Guid.NewGuid() }, new { Status = "Disabled" },
new { Name = (string?)null }, new { Name = " " } })
        { using var denied = await SendAsync(HttpMethod.Patch,
$"/api/v1/installations/{managed.Installation}", invalid);
Assert.Equal(HttpStatusCode.BadRequest, denied.StatusCode); }
        using (var disabled = await SendAsync(HttpMethod.Post,
$"/api/v1/installations/{managed.Installation}/deactivate")) {
Assert.Equal(HttpStatusCode.NoContent, disabled.StatusCode); }
        using (var list = await SendAsync(HttpMethod.Get, "/api/v1/agent/installations",
token: managed.AgentToken)) { Assert.Equal(0, (await
ReadAsync(list)).GetProperty("totalCount").GetInt32()); }
        using (var retry = await SendAsync(HttpMethod.Post, "/api/v1/agent/installations",
new { ProductId = product, Name = "Service" }, managed.AgentToken)) {
Assert.Equal(HttpStatusCode.Conflict, retry.StatusCode); }
        using (var enabled = await SendAsync(HttpMethod.Post,
$"/api/v1/installations/{managed.Installation}/reactivate")) {
Assert.Equal(HttpStatusCode.NoContent, enabled.StatusCode); }
        using (var list = await SendAsync(HttpMethod.Get, "/api/v1/agent/installations",
token: managed.AgentToken)) { Assert.Equal(1, (await
ReadAsync(list)).GetProperty("totalCount").GetInt32()); }
        await using var db = await OpenVendorAsync(VendorId); Assert.Equal(1, await
db.Installations.CountAsync()); Assert.Equal("Service", (await
db.Installations.SingleAsync()).RegistrationName);
    }

    /// <summary>
    /// Verifies that filtered reads and writes enforce all four roles and current
    assignments.
    /// </summary>

```

```

[Fact]
public async Task FilteredReadsAndWrites_EnforceAllFourRolesAndCurrentAssignments()
{
    await PrepareAsync(); Guid customer = await CustomerAsync(); Guid other = await
CustomerAsync("Other"); Guid product = await ProductAsync();
    Guid first = await InstallationAsync(customer, product), second = await
InstallationAsync(customer, product), foreign = await InstallationAsync(other, product);
    Guid firstLicense = await LicenseAsync(customer, product), secondLicense = await
LicenseAsync(customer, product), foreignLicense = await LicenseAsync(other, product);
    using (var assigned = await AssignAsync(first, firstLicense)) {
Assert.Equal(HttpStatusCode.NoContent, assigned.StatusCode); }
    using (var disabled = await SendAsync(HttpMethod.Post,
$/api/v1/installations/{second}/deactivate")) { Assert.Equal(HttpStatusCode.NoContent,
disabled.StatusCode); }
    var admin = await UserAsync("admin@example.test", "CustomerAdministrator",
customer);
    var user = await UserAsync("user@example.test", "CustomerUser", customer);
    var technician = await UserAsync("tech@example.test", "VendorTechnician");
    using (var grant = await SendAsync(HttpMethod.Put,
$/api/v1/users/{user.Id}/installation-access", new { InstallationIds = new[] { first } })) {
Assert.Equal(HttpStatusCode.NoContent, grant.StatusCode); }
    using (var grant = await SendAsync(HttpMethod.Put,
$/api/v1/users/{technician.Id}/customer-access", new { CustomerIds = new[] { customer } })) {
Assert.Equal(HttpStatusCode.NoContent, grant.StatusCode); }
    foreach (var (token, installations, licenses) in new[] { (AdminToken, 3, 3),
(admin.Token, 2, 2), (technician.Token, 2, 2), (user.Token, 1, 1) })
    {
        using var list = await SendAsync(HttpMethod.Get,
"/api/v1/installations?pageSize=1", token: token); var body = await ReadAsync(list);
Assert.Equal(installations, body.GetProperty("totalCount").GetInt32());
Assert.Single(body.GetProperty("items").EnumerateArray());
        using var licenseList = await SendAsync(HttpMethod.Get,
"/api/v1/licenses?pageSize=1", token: token); Assert.Equal(licenses, (await
ReadAsync(licenseList)).GetProperty("totalCount").GetInt32());
    }
    using (var filter = await SendAsync(HttpMethod.Get,
$/api/v1/installations?customerId={customer}&productId={product}&status=Disabled", token:
admin.Token)) { Assert.Equal(second, (await
ReadAsync(filter)).GetProperty("items")[0].GetProperty("id").GetGuid()); }
    using (var filter = await SendAsync(HttpMethod.Get,
$/api/v1/licenses?customerId={other}&productId={product}", token: admin.Token)) {
Assert.Equal(0, (await ReadAsync(filter)).GetProperty("totalCount").GetInt32()); }
    foreach (string path in new[] { $"/api/v1/installations/{foreign}",
$/api/v1/licenses/{foreignLicense} })
    { using var hidden = await SendAsync(HttpMethod.Get, path, token: admin.Token);
Assert.Equal(HttpStatusCode.NotFound, hidden.StatusCode); }
    foreach (string token in new[] { admin.Token, technician.Token, user.Token })
    { using var denied = await AssignAsync(first, firstLicense, token);
Assert.Equal(HttpStatusCode.Forbidden, denied.StatusCode); }
    using (var denied = await SendAsync(HttpMethod.Patch,
$/api/v1/installations/{first}", new { Name = "Unauthorized" }, user.Token)) {
Assert.Equal(HttpStatusCode.Forbidden, denied.StatusCode); }
    using (var changed = await SendAsync(HttpMethod.Patch,
$/api/v1/installations/{first}", new { Name = "Technician edit" }, technician.Token)) {
Assert.Equal(HttpStatusCode.OK, changed.StatusCode); }
    using (var revoke = await SendAsync(HttpMethod.Put,
$/api/v1/users/{user.Id}/installation-access", new { InstallationIds = Array.Empty<Guid>() })) {
Assert.Equal(HttpStatusCode.NoContent, revoke.StatusCode); }
    using (var hidden = await SendAsync(HttpMethod.Get,
$/api/v1/licenses/{firstLicense}", token: user.Token)) { Assert.Equal(HttpStatusCode.NotFound,
hidden.StatusCode); }

```

```

        using (var invalid = await SendAsync(HttpMethod.Get,
"/api/v1/installations?pageSize=101")) { Assert.Equal(HttpStatusCode.BadRequest,
invalid.StatusCode); }
    }

    /// <summary>
    /// Verifies that migration backfills stable registration names and rowversions reject
stale edits.
    /// </summary>
    [Fact]
    public async Task
Migration_BackfillsStableRegistrationNames_AndRowversionsRejectStaleEdits()
    {
        await PrepareAsync(); Guid customer = await CustomerAsync(); Guid product = await
ProductAsync(); var managed = await ManagedAsync(customer, product);
        await using var db = await OpenVendorAsync(VendorId); var migrator =
db.GetService<IMigrator>();
        await migrator.MigrateAsync(db.Database.GetMigrations().Single(x =>
x.EndsWith("_AddAgentIdentity", StringComparison.Ordinal)));
        await migrator.MigrateAsync(); db.ChangeTracker.Clear();
        Installation original = await db.Installations.SingleAsync(x => x.Id ==
managed.Installation); Assert.Equal("Service", original.RegistrationName);
Assert.NotEmpty(original.RowVersion);
        await using var stale = await OpenVendorAsync(VendorId); Installation outdated =
await stale.Installations.SingleAsync(x => x.Id == managed.Installation);
        original.Name = "New"; await db.SaveChangesAsync(); outdated.Name = "Stale"; await
Assert.ThrowsAsync<DbUpdateConcurrencyException>(() => stale.SaveChangesAsync());
        Guid licenseId = await LicenseAsync(customer, product); var license = await
db.Licenses.SingleAsync(x => x.Id == licenseId); var staleLicense = await
stale.Licenses.SingleAsync(x => x.Id == licenseId);
        license.LicenseReference = "Updated"; await db.SaveChangesAsync();
stale.ChangeTracker.Clear();
        stale.Attach(staleLicense); staleLicense.LicenseReference = "Outdated"; await
Assert.ThrowsAsync<DbUpdateConcurrencyException>(() => stale.SaveChangesAsync());
        Assert.False(db.Database.HasPendingModelChanges());
    }

    /// <summary>
    /// Verifies that database rejects cross customer assignments and duplicate current
assignments.
    /// </summary>
    [Fact]
    public async Task Database_RejectsCrossCustomerAssignmentsAndDuplicateCurrentAssignments()
    {
        await PrepareAsync(); Guid customer = await CustomerAsync(); Guid other = await
CustomerAsync("Other"); Guid product = await ProductAsync();
        Guid installation = await InstallationAsync(customer, product), wrong = await
LicenseAsync(other, product), matching = await LicenseAsync(customer, product);
        await using var db = await OpenVendorAsync(VendorId);
        db.InstallationLicenseAssignments.Add(new() { InstallationId = installation,
LicenseId = wrong, CustomerId = customer, ProductId = product, AssignedAt = DateTimeOffset.UtcNow
});
        await Assert.ThrowsAsync<DbUpdateException>(() => db.SaveChangesAsync());
db.ChangeTracker.Clear();
        using (var assigned = await AssignAsync(installation, matching)) {
Assert.Equal(HttpStatusCode.NoContent, assigned.StatusCode); }
        db.InstallationLicenseAssignments.Add(new() { InstallationId = installation,
LicenseId = matching, CustomerId = customer, ProductId = product, AssignedAt =
DateTimeOffset.UtcNow });
        await Assert.ThrowsAsync<DbUpdateException>(() => db.SaveChangesAsync());
    }
}

```

Integration/InstallationStateBoundaryTests.cs

```
using System.Net;
using System.Net.Http.Json;
using BridgeControl.Tests.Infrastructure;
using Microsoft.EntityFrameworkCore;

namespace BridgeControl.Tests.Integration;

/// <summary>
/// Verifies installation state boundary behavior and regression scenarios.
/// </summary>
public class InstallationStateBoundaryTests : InstallationStateTest
{
    /// <summary>
    /// Verifies that version mismatch creates update and license recovery resumes blocked
    work.
    /// </summary>
    [Fact]
    public async Task VersionMismatchCreatesUpdate_AndLicenseRecoveryResumesBlockedWork()
    {
        var state = await StateAsync(); Guid otherVersion = await
        VersionAsync(state.Product, "2");
        using var desired = await SetDesiredAsync(state); Assert.Equal(HttpStatusCode.OK,
        desired.StatusCode);
        using var older = await ReportAsync(state, "Running",
        DateTimeOffset.UtcNow.AddHours(-1), otherVersion); await ReconcileAsync();
        await using var db = await OpenVendorAsync(VendorId); Assert.Empty(await
        db.Deployments.ToListAsync()); Assert.Equal("Unknown", (await
        db.StateReconciliationWork.AsNoTracking().SingleAsync()).Reason);
        using var actual = await ReportAsync(state, "Running", DateTimeOffset.UtcNow,
        otherVersion);
        using var expired = await SendAsync(HttpMethod.Post,
        $"/api/v1/licenses/{state.License}/deactivate"); await ReconcileAsync(); Assert.Empty(await
        db.Deployments.ToListAsync()); Assert.Equal("Blocked", (await
        db.StateReconciliationWork.AsNoTracking().SingleAsync()).Status);
        using var restored = await SendAsync(HttpMethod.Post,
        $"/api/v1/licenses/{state.License}/reactivate"); Assert.Equal(HttpStatusCode.NoContent,
        restored.StatusCode);
        await db.StateReconciliationWork.ExecuteUpdateAsync(s => s.SetProperty(x =>
        x.NextAttemptAt, DateTimeOffset.UtcNow.AddSeconds(-1))); await ReconcileAsync();
        Assert.Equal("Update", (await db.Deployments.SingleAsync()).OperationType);
    }
    /// <summary>
    /// Verifies that cross installation and cross vendor references are concealed.
    /// </summary>
    [Fact]
    public async Task CrossInstallationAndCrossVendorReferences_AreConcealed()
    {
        var state = await StateAsync(); Guid anotherInstallation = await
        InstallationAsync(state.Customer, state.Product); Guid foreignConfig = await
        GenerateAsync(anotherInstallation, state.Version);
        using var wrongConfig = await SetDesiredAsync(state, configuration:
        foreignConfig); Assert.Equal(HttpStatusCode.NotFound, wrongConfig.StatusCode);
        using var wrongReport = await ReportAsync(state, "Running", DateTimeOffset.UtcNow,
        state.Version, foreignConfig, state.Hash); Assert.Equal(HttpStatusCode.NotFound,
        wrongReport.StatusCode);
    }
}
```

```

        var otherProduct = await CatalogAsync(); using var wrongProduct = await
ReportAsync(state, "Running", DateTimeOffset.UtcNow, otherProduct.Version);
Assert.Equal(HttpStatusCode.NotFound, wrongProduct.StatusCode);
        using var desired = await SetDesiredAsync(state); Assert.Equal(HttpStatusCode.OK,
desired.StatusCode);
        Guid otherVendor = await CreateVendorAsync("State other vendor"); string
invitation = await InviteAsync(otherVendor);
        using var activate = await Client.PostAsJsonAsync("/api/v1/auth/activate", new {
Token = invitation, Password }); Assert.Equal(HttpStatusCode.NoContent, activate.StatusCode);
        using var login = await Client.PostAsJsonAsync("/api/v1/auth/login", new {
VendorName = "State other vendor", Email = "invited@example.test", Password }); string otherToken
= (await ReadAsync(login)).GetProperty("accessToken").GetString();
        foreach (string kind in new[] { "desired-state", "reported-state" }) { using var
hidden = await SendAsync(HttpMethod.Get, $"/api/v1/installations/{state.Installation}/{kind}",
token: otherToken); Assert.Equal(HttpStatusCode.NotFound, hidden.StatusCode); }
        using var setForeign = await SetDesiredAsync(state, token: otherToken);
Assert.Equal(HttpStatusCode.NotFound, setForeign.StatusCode);
        using var customer = await SendAsync(HttpMethod.Post, "/api/v1/customers", new {
Name = "Other" }, otherToken); Guid otherCustomer = (await
ReadAsync(customer)).GetProperty("id").GetGuid();
        using var enroll = await SendAsync(HttpMethod.Post, "/api/v1/agent-enrollments",
new { CustomerId = otherCustomer }, otherToken);
        using var registered = await SendAsync(HttpMethod.Post, "/api/v1/agent/register",
new { EnrollmentCode = (await ReadAsync(enroll)).GetProperty("enrollmentCode").GetString(),
MachineName = "Other", AgentVersion = "1" });
        using var authenticated = await SendAsync(HttpMethod.Post,
"/api/v1/agent/auth/token", new { Credential = (await
ReadAsync(registered)).GetProperty("credential").GetString() }); string agentToken = (await
ReadAsync(authenticated)).GetProperty("accessToken").GetString();
        using var agentDesired = await SendAsync(HttpMethod.Get,
$"/api/v1/agent/installations/{state.Installation}/desired-state", token: agentToken);
Assert.Equal(HttpStatusCode.NotFound, agentDesired.StatusCode);
        using var content = await SendAsync(HttpMethod.Get,
$"/api/v1/agent/artifacts/{state.Installer}/content", token: agentToken);
Assert.Equal(HttpStatusCode.NotFound, content.StatusCode);
        using var report = await SendAsync(HttpMethod.Put,
$"/api/v1/agent/installations/{state.Installation}/reported-state", new { Status = "Unknown",
ReportedAt = DateTimeOffset.UtcNow }, agentToken); Assert.Equal(HttpStatusCode.NotFound,
report.StatusCode);
    }
}

```

Integration/InstallationStateDeliveryTests.cs

```

using System.Net;
using System.Net.Http.Headers;
using System.Security.Cryptography;
using BridgeControl.Tests.Infrastructure;
using Microsoft.EntityFrameworkCore;

namespace BridgeControl.Tests.Integration;

/// <summary>
/// Verifies installation state delivery behavior and regression scenarios.
/// </summary>
public class InstallationStateDeliveryTests : InstallationStateTest
{
    /// <summary>
    /// Verifies that agent downloads are pinned complete and verified and recheck
entitlement.

```

```

    /// </summary>
    [Fact]
    public async Task AgentDownloads_ArePinnedCompleteAndVerified_AndRecheckEntitlement()
    {
        var state = await StateAsync(); using var set = await SetDesiredAsync(state);
        Assert.Equal(HttpStatusCode.OK, set.StatusCode);
        using var desired = await SendAsync(HttpMethod.Get,
            $"/api/v1/agent/installations/{state.Installation}/desired-state", token: state.Token);
        Assert.Equal(HttpStatusCode.OK, desired.StatusCode);
        Assert.Contains(state.Installer.ToString(), string.Join(',',
            desired.Headers.GetValues("Link"))); Assert.Contains(state.ConfigurationArtifact.ToString(),
            string.Join(',', desired.Headers.GetValues("Link")));
        foreach (Guid artifact in new[] { state.Installer, state.ConfigurationArtifact })
        {
            using var request = new HttpRequestMessage(HttpMethod.Get,
                $"/api/v1/agent/artifacts/{artifact}/content"); request.Headers.Authorization = new
                AuthenticationHeaderValue("Bearer", state.Token); request.Headers.Range = new RangeHeaderValue(0,
                1);

            using var download = await Client.SendAsync(request,
                HttpCompletionOption.ResponseHeadersRead); Assert.Equal(HttpStatusCode.OK, download.StatusCode);
            byte[] content = await download.Content.ReadAsByteArrayAsync(); string hash
            = Convert.ToHexStringLower(SHA256.HashData(content)); Assert.Equal($"\"{hash}\"",
                download.Headers.ETag!.Tag); Assert.Equal(content.LongLength,
                download.Content.Headers.ContentLength);
            Assert.Equal(state.Version.ToString(),
                download.Headers.GetValues("X-Product-Version-Id").Single());
            Assert.True(download.Headers.CacheControl!.NoStore);
        }
        // Abandoning a client read cannot publish partial metadata; a subsequent full
        download is independently verified.
        using (var request = new HttpRequestMessage(HttpMethod.Get,
            $"/api/v1/agent/artifacts/{state.Installer}/content"))
        {
            request.Headers.Authorization = new AuthenticationHeaderValue("Bearer",
                state.Token);

            using var partial = await Client.SendAsync(request,
                HttpCompletionOption.ResponseHeadersRead);
            await using var stream = await partial.Content.ReadAsStreamAsync();
            Assert.NotEqual(-1, stream.ReadByte());
        }
        using var retry = await SendAsync(HttpMethod.Get,
            $"/api/v1/agent/artifacts/{state.Installer}/content", token: state.Token);
        Assert.Equal(HttpStatusCode.OK, retry.StatusCode);
        await using var db = await OpenVendorAsync(VendorId); Guid schema = (await
            db.ProductVersions.SingleAsync(x => x.Id == state.Version)).PublishedSchemaArtifactId!.Value;
        using var unrelated = await SendAsync(HttpMethod.Get,
            $"/api/v1/agent/artifacts/{schema}/content", token: state.Token);
        Assert.Equal(HttpStatusCode.NotFound, unrelated.StatusCode);
        using var disabled = await SendAsync(HttpMethod.Post,
            $"/api/v1/licenses/{state.License}/deactivate");
        using var denied = await SendAsync(HttpMethod.Get,
            $"/api/v1/agent/artifacts/{state.Installer}/content", token: state.Token);
        Assert.Equal(HttpStatusCode.NotFound, denied.StatusCode);
        using var targetDenied = await SendAsync(HttpMethod.Get,
            $"/api/v1/agent/installations/{state.Installation}/desired-state", token: state.Token);
        Assert.Equal(HttpStatusCode.Conflict, targetDenied.StatusCode);
    }
    /// </summary>
    /// Verifies that corrupt or unsafe archives fail before streaming any content.
    /// </summary>
    [Fact]
    public async Task CorruptOrUnsafeArchives_FailBeforeStreamingAnyContent()

```

```

    {
        var state = await StateAsync(); using var set = await SetDesiredAsync(state);
Assert.Equal(HttpStatusCode.OK, set.StatusCode);
        await using var db = await OpenVendorAsync(VendorId); var artifact = await
db.FileArtifacts.SingleAsync(x => x.Id == state.ConfigurationArtifact);
        string path = Path.Combine(ArchiveRoot, "blobs", artifact.StorageIdentifier +
".blob"); byte[] original = await File.ReadAllBytesAsync(path); await
File.WriteAllBytesAsync(path, [0]);
        using var corrupt = await SendAsync(HttpMethod.Get,
$"/api/v1/agent/artifacts/{state.ConfigurationArtifact}/content", token: state.Token);
Assert.Equal(HttpStatusCode.Conflict, corrupt.StatusCode);
Assert.Equal("application/problem+json", corrupt.Content.Headers.ContentType!.MediaType);
        await File.WriteAllBytesAsync(path, original);
        artifact.StorageIdentifier = "../outside"; await db.SaveChangesAsync();
        using var unsafePath = await SendAsync(HttpMethod.Get,
$"/api/v1/agent/artifacts/{state.ConfigurationArtifact}/content", token: state.Token);
Assert.Equal(HttpStatusCode.Conflict, unsafePath.StatusCode);
    }
    /// <summary>
    /// Verifies that human and agent boundaries are live and independent.
    /// </summary>
    [Fact]
    public async Task HumanAndAgentBoundaries_AreLiveAndIndependent()
    {
        var state = await StateAsync(); var secondAgent = await
AgentAsync(state.Customer);
        using var set = await SetDesiredAsync(state); Assert.Equal(HttpStatusCode.OK,
set.StatusCode);
        using var reported = await ReportAsync(state, "NotInstalled",
DateTimeOffset.UtcNow);
        foreach (string path in new[] {
$"/api/v1/agent/installations/{state.Installation}/desired-state",
$"/api/v1/agent/artifacts/{state.Installer}/content" })
        { using var denied = await SendAsync(HttpMethod.Get, path, token:
secondAgent.Token); Assert.Equal(HttpStatusCode.NotFound, denied.StatusCode); }
        using var foreignReport = await SendAsync(HttpMethod.Put,
$"/api/v1/agent/installations/{state.Installation}/reported-state", new { Status = "Unknown",
ReportedAt = DateTimeOffset.UtcNow }, secondAgent.Token); Assert.Equal(HttpStatusCode.NotFound,
foreignReport.StatusCode);
        var other = await UserAsync("other-state@example.test", "CustomerAdministrator",
await CustomerAsync("Other"));
        var user = await UserAsync("state-reader@example.test", "CustomerUser",
state.Customer);
        foreach (string kind in new[] { "desired-state", "reported-state" })
        {
            using var hidden = await SendAsync(HttpMethod.Get,
$"/api/v1/installations/{state.Installation}/{kind}", token: other.Token);
Assert.Equal(HttpStatusCode.NotFound, hidden.StatusCode);
            using var unassigned = await SendAsync(HttpMethod.Get,
$"/api/v1/installations/{state.Installation}/{kind}", token: user.Token);
Assert.Equal(HttpStatusCode.NotFound, unassigned.StatusCode);
        }
        using var grant = await SendAsync(HttpMethod.Put,
$"/api/v1/users/{user.Id}/installation-access", new { InstallationIds = new[] {
state.Installation } });
        using var readable = await SendAsync(HttpMethod.Get,
$"/api/v1/installations/{state.Installation}/desired-state", token: user.Token);
Assert.Equal(HttpStatusCode.OK, readable.StatusCode);
        using var forbidden = await SetDesiredAsync(state, token: user.Token);
Assert.Equal(HttpStatusCode.Forbidden, forbidden.StatusCode);
    }

```

```

        await using var db = await OpenVendorAsync(VendorId); await
db.AgentCredentials.Where(x => x.AgentId == state.Agent).ExecuteUpdateAsync(s => s.SetProperty(x
=> x.RevokedAt, DateTimeOffset.UtcNow));
        using var revoked = await SendAsync(HttpMethod.Get,
$"/api/v1/agent/installations/{state.Installation}/desired-state", token: state.Token);
Assert.Equal(HttpStatusCode.Unauthorized, revoked.StatusCode);
    }
}

```

Integration/InstallationStateRecoveryTests.cs

```

using System.Net;
using BridgeControl.Tests.Infrastructure;
using Microsoft.Data.SqlClient;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Infrastructure;
using Microsoft.EntityFrameworkCore.Migrations;

namespace BridgeControl.Tests.Integration;

/// <summary>
/// Verifies installation state recovery behavior and regression scenarios.
/// </summary>
public class InstallationStateRecoveryTests : InstallationStateTest
{
    /// <summary>
    /// Verifies that new desired revision supersedes old work and downloads without duplicate
active work.
    /// </summary>
    [Fact]
    public async Task
NewDesiredRevision_SupersedesOldWorkAndDownloads_WithoutDuplicateActiveWork()
    {
        var state = await StateAsync(); using var first = await SetDesiredAsync(state);
Assert.Equal(HttpStatusCode.OK, first.StatusCode);
        using var actual = await ReportAsync(state, "NotInstalled",
DateTimeOffset.UtcNow); await ReconcileAsync();
        using var changed = await
SetAsync($"/api/v1/installations/{state.Installation}/configuration", state.Version, new { port =
80 }); Guid next = await GenerateAsync(state.Installation, state.Version);
        var responses = await Task.WhenAll(SetDesiredAsync(state, configuration: next),
SetDesiredAsync(state, configuration: next)); foreach (var response in responses) { using
(response) { Assert.Equal(HttpStatusCode.OK, response.StatusCode); } }
        await ReconcileAsync(); await ReconcileAsync();
        await using var db = await OpenVendorAsync(VendorId); Assert.Equal(2, await
db.DesiredStateRevisions.CountAsync()); Assert.Equal(0, await db.Deployments.CountAsync());
        Assert.Equal(1, await db.Deployments.CountAsync(x => x.IsActive)); Assert.Equal(1,
await db.Deployments.CountAsync(x => x.Status == "Superseded"));
        using var superseded = await SendAsync(HttpMethod.Get,
$"/api/v1/agent/artifacts/{state.ConfigurationArtifact}/content", token: state.Token);
Assert.Equal(HttpStatusCode.NotFound, superseded.StatusCode);
        using var current = await SendAsync(HttpMethod.Get,
$"/api/v1/agent/installations/{state.Installation}/desired-state", token: state.Token);
Assert.Equal(next, (await
ReadAsync(current)).GetProperty("effectiveConfigurationRevisionId").GetGuid());
    }
    /// <summary>
    /// Verifies that fresh host resumes durable reconciliation.
    /// </summary>
    [Fact]

```

```

    public async Task FreshHost_ResumesDurableReconciliation()
    {
        var state = await StateAsync(); using var desired = await SetDesiredAsync(state);
        Assert.Equal(HttpStatusCode.OK, desired.StatusCode);
        using var actual = await ReportAsync(state, "NotInstalled",
        DateTimeOffset.UtcNow); Assert.Equal(HttpStatusCode.OK, actual.StatusCode);
        await using var restarted = new
        ApiFactory(Database.Database.GetConnectionString()!, archiveRoot: Factory.ArchiveRoot,
        stateWorker: true); using var client = restarted.CreateApiClient();
        await using var db = await OpenVendorAsync(VendorId); using var deadline = new
        CancellationTokenSource(TimeSpan.FromSeconds(30));
        while (!await db.Deployments.AnyAsync(x => x.InstallationId == state.Installation,
        deadline.Token)) { await Task.Delay(100, deadline.Token); }
        Assert.Equal("Pending", (await db.Deployments.SingleAsync()).Status);
    }
    /// <summary>
    /// Verifies that worker failure rolls back work and stops after three attempts new report
    resets it.
    /// </summary>
    [Fact]
    public async Task
    WorkerFailure_RollsBackWorkAndStopsAfterThreeAttempts_NewReportResetsIt()
    {
        var state = await StateAsync(); using var desired = await SetDesiredAsync(state);
        using var actual = await ReportAsync(state, "NotInstalled", DateTimeOffset.UtcNow);
        await using var db = await OpenVendorAsync(VendorId);
        await db.Database.ExecuteSqlRawAsync("CREATE TRIGGER FailDeploymentAudit ON
        VendorAuditLogs AFTER INSERT AS BEGIN IF EXISTS (SELECT 1 FROM inserted WHERE Action =
        'DeploymentPlanned') THROW 51000, 'test failure', 1; END");
        for (int i = 1; i <= 3; i++)
        {
            await ReconcileAsync(); Assert.Equal(i, (await
            db.StateReconciliationWork.AsNoTracking().SingleAsync()).Attempts); Assert.False(await
            db.Deployments.AnyAsync());
            await db.StateReconciliationWork.ExecuteUpdateAsync(s => s.SetProperty(x =>
            x.NextAttemptAt, DateTimeOffset.UtcNow.AddSeconds(-1)));
        }
        Assert.Equal("Failed", (await
        db.StateReconciliationWork.AsNoTracking().SingleAsync()).Status);
        await db.Database.ExecuteSqlRawAsync("DROP TRIGGER FailDeploymentAudit");
        using var newer = await ReportAsync(state, "NotInstalled",
        DateTimeOffset.UtcNow.AddSeconds(1)); await ReconcileAsync(); Assert.Equal(1, await
        db.Deployments.CountAsync());
    }
    /// <summary>
    /// Verifies that upgrade preserves existing configuration and state history is immutable.
    /// </summary>
    [Fact]
    public async Task Upgrade_PreservesExistingConfiguration_AndStateHistoryIsImmutable()
    {
        var state = await StateAsync(); await using var db = await
        OpenVendorAsync(VendorId);
        var migrator = db.GetService<IMigrator>(); await
        migrator.MigrateAsync(db.Database.GetMigrations().Single(x =>
        x.EndsWith("_AddConfigurationMigrationReviews", StringComparison.Ordinal))); await
        migrator.MigrateAsync();
        Assert.Equal(state.Configuration, (await
        db.Installations.SingleAsync()).CurrentEffectiveConfigurationRevisionId); Assert.Null(await
        db.Installations.SingleAsync()).DesiredStateRevisionId); db.ChangeTracker.Clear();
        using var desired = await SetDesiredAsync(state); using var actual = await
        ReportAsync(state, "NotInstalled", DateTimeOffset.UtcNow);

```

```

        Assert.Equal(HttpStatusCode.OK, desired.StatusCode);
Assert.Equal(HttpStatusCode.OK, actual.StatusCode);
        await Assert.ThrowsAsync<SqlException>(() =>
db.Database.ExecuteSqlRawAsync("DELETE FROM InstallationStateReports"));
        await Assert.ThrowsAsync<SqlException>(() =>
db.Database.ExecuteSqlRawAsync("UPDATE DesiredStateRevisions SET RevisionNumber = 2"));
        Assert.False(db.Database.HasPendingModelChanges());
    }
}

```

Integration/InstallationStateTests.cs

```

using System.Net;
using BridgeControl.Tests.Infrastructure;
using Microsoft.EntityFrameworkCore;

namespace BridgeControl.Tests.Integration;

/// <summary>
/// Verifies installation state behavior and regression scenarios.
/// </summary>
public class InstallationStateTests : InstallationStateTest
{
    /// <summary>
    /// Verifies that desired state is licensed published scoped and idempotent with atomic
    scheduling.
    /// </summary>
    [Fact]
    public async Task
DesiredState_IsLicensedPublishedScopedAndIdempotent_WithAtomicScheduling()
    {
        var state = await StateAsync();
        using var absent = await SendAsync(HttpMethod.Get,
$"/api/v1/installations/{state.Installation}/desired-state");
Assert.Equal(HttpStatusCode.NotFound, absent.StatusCode);
        using var desired = await SetDesiredAsync(state); Assert.Equal(HttpStatusCode.OK,
desired.StatusCode); var first = await ReadAsync(desired);
        Assert.Equal(new[] { "effectiveConfigurationRevisionId", "installationId",
"productVersionId", "updatedAt" }, first.EnumerateObject().Select(x => x.Name).Order());
        using var retry = await SetDesiredAsync(state);
Assert.Equal(first.GetProperty("updatedAt").GetDateTimeOffset(), (await
ReadAsync(retry)).GetProperty("updatedAt").GetDateTimeOffset());
        await using var db = await OpenVendorAsync(VendorId); Assert.Equal(1, await
db.DesiredStateRevisions.CountAsync()); Assert.Equal(1, (await
db.StateReconciliationWork.SingleAsync()).RequestedGeneration);
        Guid draft = await VersionAsync(state.Product, "2"); Guid draftConfig = await
GenerateAsync(state.Installation, draft);
        using var refused = await SetDesiredAsync(state, draft, draftConfig);
Assert.Equal(HttpStatusCode.Conflict, refused.StatusCode);
        using var disabled = await SendAsync(HttpMethod.Post,
$"/api/v1/licenses/{state.License}/deactivate"); Assert.Equal(HttpStatusCode.NoContent,
disabled.StatusCode);
        using var unlicensed = await SetDesiredAsync(state);
Assert.Equal(HttpStatusCode.Conflict, unlicensed.StatusCode);
        using var history = await SendAsync(HttpMethod.Get,
$"/api/v1/installations/{state.Installation}/desired-state"); Assert.Equal(HttpStatusCode.OK,
history.StatusCode);
    }
    /// <summary>

```

```

    /// Verifies that reports are ordered deduplicated and independent of license and desired
state.
    /// </summary>
    [Fact]
    public async Task Reports_AreOrderedDeduplicatedAndIndependentOfLicenseAndDesiredState()
    {
        var state = await StateAsync(); DateTimeOffset now = DateTimeOffset.UtcNow;
        using var first = await ReportAsync(state, "NotInstalled", now);
        Assert.Equal(HttpStatusCode.OK, first.StatusCode);
        using var newer = await ReportAsync(state, "Running", now.AddSeconds(1),
state.Version, state.Configuration, state.Hash); Assert.Equal(HttpStatusCode.OK,
newer.StatusCode);
        using var duplicate = await ReportAsync(state, "NotInstalled", now);
        Assert.Equal(HttpStatusCode.OK, duplicate.StatusCode);
        using var conflict = await ReportAsync(state, "Stopped", now);
        Assert.Equal(HttpStatusCode.Conflict, conflict.StatusCode);
        using var stale = await ReportAsync(state, "NotInstalled", now.AddSeconds(-1));
        Assert.Equal(HttpStatusCode.Conflict, stale.StatusCode);
        using var future = await ReportAsync(state, "Unknown", now.AddHours(1));
        Assert.Equal(HttpStatusCode.BadRequest, future.StatusCode);
        using var invalid = await ReportAsync(state, "NotInstalled", now.AddSeconds(2),
state.Version); Assert.Equal(HttpStatusCode.BadRequest, invalid.StatusCode);
        using var disabled = await SendAsync(HttpMethod.Post,
$"/api/v1/licenses/{state.License}/deactivate");
        using var expiredReport = await ReportAsync(state, "Stopped", now.AddSeconds(2),
state.Version, state.Configuration, state.Hash); Assert.Equal(HttpStatusCode.OK,
expiredReport.StatusCode);
        using var actual = await SendAsync(HttpMethod.Get,
$"/api/v1/installations/{state.Installation}/reported-state"); var body = await
ReadAsync(actual);
        Assert.Equal("Stopped", body.GetProperty("status").GetString());
        Assert.Equal(new[] { "configurationHash", "effectiveConfigurationRevisionId", "installationId",
"installedProductVersionId", "reportedAt", "status" }, body.EnumerateObject().Select(x =>
x.Name).Order());
        await using var db = await OpenVendorAsync(VendorId); Assert.Equal(3, await
db.InstallationStateReports.CountAsync()); Assert.Empty(await
db.DesiredStateRevisions.ToListAsync());
    }
    /// </summary>
    /// Verifies that reconciliation waits for evidence and completes only exact running
state.
    /// </summary>
    [Fact]
    public async Task Reconciliation_WaitsForEvidence_AndCompletesOnlyExactRunningState()
    {
        var state = await StateAsync(); using var desired = await SetDesiredAsync(state);
        Assert.Equal(HttpStatusCode.OK, desired.StatusCode);
        await ReconcileAsync(); await using var db = await OpenVendorAsync(VendorId);
        Assert.Empty(await db.Deployments.ToListAsync()); Assert.Equal("Unknown", (await
db.StateReconciliationWork.AsNoTracking().SingleAsync()).Reason);
        DateTimeOffset now = DateTimeOffset.UtcNow; using var fresh = await
ReportAsync(state, "NotInstalled", now); Assert.Equal(HttpStatusCode.OK, fresh.StatusCode);
        await ReconcileAsync(); var deployment = await
db.Deployments.AsNoTracking().SingleAsync(); Assert.Equal("Install", deployment.OperationType);
        Assert.Equal("Pending", deployment.Status);
        await ReconcileAsync(); Assert.Equal(1, await db.Deployments.CountAsync());
        using var wrongHash = await ReportAsync(state, "Running", now.AddSeconds(1),
state.Version, state.Configuration, new string('0', 64)); Assert.Equal(HttpStatusCode.OK,
wrongHash.StatusCode);
        await ReconcileAsync(); Assert.Equal("Configure", (await
db.Deployments.AsNoTracking().SingleAsync()).OperationType); Assert.True(await
db.Deployments.AsNoTracking().SingleAsync().IsActive);
    }

```

```

        using var restarting = await ReportAsync(state, "RestartRequired",
now.AddSeconds(2), state.Version, state.Configuration, state.Hash); await ReconcileAsync();
Assert.True((await db.Deployments.AsNoTracking().SingleAsync()).IsActive);
        using var matching = await ReportAsync(state, "Running", now.AddSeconds(3),
state.Version, state.Configuration, state.Hash); await ReconcileAsync();
        var completed = await db.Deployments.AsNoTracking().SingleAsync();
Assert.False(completed.IsActive); Assert.Equal("Reconciled", completed.Status);
Assert.NotNull(completed.ReconciledReportId);
        Assert.Empty(await db.DeploymentAttempts.ToListAsync());
    }
    /// <summary>
    /// Verifies that stale configuration and audit failures preserve prior state.
    /// </summary>
    [Fact]
    public async Task StaleConfigurationAndAuditFailures_PreservePriorState()
    {
        var state = await StateAsync();
        using var changed = await
SetAsync($"api/v1/installations/{state.Installation}/configuration", state.Version, new { port =
123 }); Assert.Equal(HttpStatusCode.OK, changed.StatusCode);
        using var stale = await SetDesiredAsync(state);
Assert.Equal(HttpStatusCode.Conflict, stale.StatusCode);
        Guid config = await GenerateAsync(state.Installation, state.Version);
        await using var db = await OpenVendorAsync(VendorId);
        await db.Database.ExecuteSqlRawAsync("CREATE TRIGGER FailStateAudit ON
VendorAuditLogs AFTER INSERT AS BEGIN IF EXISTS (SELECT 1 FROM inserted WHERE Action IN
('InstallationDesiredStateSet','InstallationStateReported')) THROW 51000, 'test failure', 1;
END");
        using var desired = await SetDesiredAsync(state, configuration: config);
Assert.Equal(HttpStatusCode.InternalServerError, desired.StatusCode);
        using var reported = await ReportAsync(state, "NotInstalled",
DateTimeOffset.UtcNow); Assert.Equal(HttpStatusCode.InternalServerError, reported.StatusCode);
        Assert.Empty(await db.DesiredStateRevisions.ToListAsync()); Assert.Empty(await
db.InstallationStateReports.ToListAsync()); Assert.Empty(await
db.StateReconciliationWork.ToListAsync());
        await db.Database.ExecuteSqlRawAsync("DROP TRIGGER FailStateAudit");
    }
}

```

Integration/LicensingTests.cs

```

using System.Net;
using BridgeControl.Services.Vendor;
using BridgeControl.Tests.Infrastructure;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;

namespace BridgeControl.Tests.Integration;

/// <summary>
/// Verifies licensing behavior and regression scenarios.
/// </summary>
[Trait("Category", "Integration")]
public class LicensingTests : LicensingTest
{
    /// <summary>
    /// Verifies that creation validates terms and conceals resources from another vendor.
    /// </summary>
    [Fact]
    public async Task Creation_ValidatesTermsAndConcealsResourcesFromAnotherVendor()

```

```

    {
        await PrepareAsync(); Guid customer = await CustomerAsync(); Guid product = await
ProductAsync();
        foreach (var change in new Dictionary<string, object?>[]
        {
            new() { ["validFrom"] = default(DateTimeOffset) }, new() {
["maxInstallations"] = -1 },
            new() { ["validUntil"] = DateTimeOffset.UtcNow.AddYears(-1) }, new() {
["licenseReference"] = " " },
            new() { ["customerId"] = Guid.Empty }, new() { ["status"] = "Active" }
        })
        {
            Dictionary<string, object?> body = new()
            {
                ["customerId"] = customer,
                ["productId"] = product,
                ["licenseReference"] = "Reference",
                ["validFrom"] = DateTimeOffset.UtcNow.AddDays(-1)
            };
            foreach (var pair in change) { body[pair.Key] = pair.Value; }
            using var invalid = await SendAsync(HttpMethod.Post, "/api/v1/licenses",
body); Assert.Equal(HttpStatusCode.BadRequest, invalid.StatusCode);
        }
        Guid vendor = await CreateVendorAsync("Other vendor"); await InviteAsync(vendor,
"other@example.test");
        await using var other = await OpenVendorAsync(vendor);
        var foreignCustomer = new BridgeControl.Models.Vendor.Customer { Id =
Guid.NewGuid(), Name = "Other", Status = "Active" };
        var foreignProduct = new BridgeControl.Models.Vendor.Product { Id =
Guid.NewGuid(), Name = "Other", Status = "Active" };
        other.Customers.Add(foreignCustomer); other.Products.Add(foreignProduct); await
other.SaveChangesAsync();
        var license = new BridgeControl.Models.Vendor.License { CustomerId =
foreignCustomer.Id, ProductId = foreignProduct.Id, LicenseReference = "Foreign", ValidFrom =
DateTimeOffset.UtcNow.AddDays(-1) };
        other.Licenses.Add(license); await other.SaveChangesAsync();
        Guid installation = await InstallationAsync(customer, product);
        using (var hidden = await SendAsync(HttpMethod.Get,
$/api/v1/licenses/{license.Id}")) { Assert.Equal(HttpStatusCode.NotFound, hidden.StatusCode); }
        using (var denied = await AssignAsync(installation, license.Id)) {
Assert.Equal(HttpStatusCode.NotFound, denied.StatusCode); }
        using var create = await SendAsync(HttpMethod.Post, "/api/v1/licenses", new {
CustomerId = customer, ProductId = foreignProduct.Id, LicenseReference = "Wrong", ValidFrom =
DateTimeOffset.UtcNow.AddDays(-1) });
        Assert.Equal(HttpStatusCode.NotFound, create.StatusCode);
    }

    /// <summary>
    /// Verifies that parent disablement blocks eligibility and reactivation without
discarding reservations.
    /// </summary>
    [Theory]
    [InlineData("customer")]
    [InlineData("product")]
    public async Task
ParentDisablement_BlocksEligibilityAndReactivationWithoutDiscardingReservations(string parent)
    {
        await PrepareAsync(); Guid customer = await CustomerAsync(); Guid product = await
ProductAsync(); var managed = await ManagedAsync(customer, product);
        Guid license = await LicenseAsync(customer, product, 1);
        using (var assigned = await AssignAsync(managed.Installation, license)) {
Assert.Equal(HttpStatusCode.NoContent, assigned.StatusCode); }
    }

```

```

        string path = parent == "customer" ? $"/api/v1/customers/{customer}/deactivate" :
$"/api/v1/products/{product}/deactivate";
        using (var disabled = await SendAsync(HttpMethod.Post, path)) {
Assert.Equal(HttpStatusCode.NoContent, disabled.StatusCode); }
        await using var db = await OpenVendorAsync(VendorId); var eligibility =
Services.GetRequiredService<LicenseEligibility>();
        Assert.False((await eligibility.EvaluateAsync(db, managed.Installation)).Allowed);
        Assert.Equal(1, await eligibility.ActiveCountAsync(db, license, null, default));
        using (var disabled = await SendAsync(HttpMethod.Post,
$"/api/v1/installations/{managed.Installation}/deactivate")) {
Assert.Equal(HttpStatusCode.NoContent, disabled.StatusCode); }
        using (var denied = await SendAsync(HttpMethod.Post,
$"/api/v1/installations/{managed.Installation}/reactivate")) {
Assert.Equal(HttpStatusCode.Conflict, denied.StatusCode); }
        using (var disabled = await SendAsync(HttpMethod.Post,
$"/api/v1/licenses/{license}/deactivate")) { Assert.Equal(HttpStatusCode.NoContent,
disabled.StatusCode); }
        using (var denied = await SendAsync(HttpMethod.Post,
$"/api/v1/licenses/{license}/reactivate")) { Assert.Equal(HttpStatusCode.Conflict,
denied.StatusCode); }
        Assert.Equal(1, await db.InstallationLicenseAssignments.CountAsync(x =>
x.RemovedAt == null));
    }

    /// <summary>
    /// Verifies that capacity is atomic and disabled installation cannot reactivate into
    occupied slot.
    /// </summary>
    [Fact]
    public async Task
Capacity_IsAtomic_AndDisabledInstallationCannotReactivateIntoOccupiedSlot()
    {
        await PrepareAsync(); Guid customer = await CustomerAsync(); Guid product = await
ProductAsync(); Guid license = await LicenseAsync(customer, product, 1);
        Guid first = await InstallationAsync(customer, product), second = await
InstallationAsync(customer, product);
        HttpResponseMessage[] race = await Task.WhenAll(AssignAsync(first, license),
AssignAsync(second, license));
        Assert.Single(race, x => x.StatusCode == HttpStatusCode.NoContent);
        Assert.Single(race, x => x.StatusCode == HttpStatusCode.Conflict);
        Guid winner = race[0].IsSuccessStatusCode ? first : second, loser = winner ==
first ? second : first;
        foreach (var response in race) { response.Dispose(); }
        using (var retry = await AssignAsync(winner, license)) {
Assert.Equal(HttpStatusCode.NoContent, retry.StatusCode); }
        using (var disabled = await SendAsync(HttpMethod.Post,
$"/api/v1/installations/{winner}/deactivate")) { Assert.Equal(HttpStatusCode.NoContent,
disabled.StatusCode); }
        using (var assign = await AssignAsync(loser, license)) {
Assert.Equal(HttpStatusCode.NoContent, assign.StatusCode); }
        using (var denied = await SendAsync(HttpMethod.Post,
$"/api/v1/installations/{winner}/reactivate")) { Assert.Equal(HttpStatusCode.Conflict,
denied.StatusCode); }
        for (int i = 0; i < 2; i++)
        { using var remove = await SendAsync(HttpMethod.Delete,
$"/api/v1/installations/{loser}/license"); Assert.Equal(HttpStatusCode.NoContent,
remove.StatusCode); }
        using (var restored = await SendAsync(HttpMethod.Post,
$"/api/v1/installations/{winner}/reactivate")) { Assert.Equal(HttpStatusCode.NoContent,
restored.StatusCode); }
        await using var db = await OpenVendorAsync(VendorId); Assert.Equal(2, await
db.InstallationLicenseAssignments.CountAsync());
    }

```

```

        Assert.Equal(1, await db.InstallationLicenseAssignments.CountAsync(x =>
x.RemovedAt == null));
        Assert.Equal(1, await db.VendorAuditLogs.CountAsync(x => x.Action ==
"InstallationLicenseRemoved"));
    }

    /// <summary>
    /// Verifies that assignment rejects wrong customer product and invalid license periods.
    /// </summary>
    [Fact]
    public async Task Assignment_RejectsWrongCustomerProductAndInvalidLicensePeriods()
    {
        await PrepareAsync(); Guid customer = await CustomerAsync(); Guid otherCustomer =
await CustomerAsync("Other"); Guid product = await ProductAsync();
        Guid installation = await InstallationAsync(customer, product);
        using var otherProductResponse = await SendAsync(HttpMethod.Post,
"/api/v1/products", new { Name = "Other" });
        Guid otherProduct = (await
ReadAsync(otherProductResponse)).GetProperty("id").GetGuid();
        foreach (Guid license in new[] { await LicenseAsync(otherCustomer, product), await
LicenseAsync(customer, otherProduct),
            await LicenseAsync(customer, product, from:
DateTimeOffset.UtcNow.AddDays(1)),
            await LicenseAsync(customer, product, from:
DateTimeOffset.UtcNow.AddDays(-2), until: DateTimeOffset.UtcNow.AddDays(-1)) })
        { using var denied = await AssignAsync(installation, license);
Assert.Equal(HttpStatusCode.Conflict, denied.StatusCode); }
        Guid inactive = await LicenseAsync(customer, product);
        using (var disabled = await SendAsync(HttpMethod.Post,
$"/api/v1/licenses/{inactive}/deactivate")) { Assert.Equal(HttpStatusCode.NoContent,
disabled.StatusCode); }
        using (var denied = await AssignAsync(installation, inactive)) {
Assert.Equal(HttpStatusCode.Conflict, denied.StatusCode); }
        using (var missing = await AssignAsync(installation, Guid.NewGuid())) {
Assert.Equal(HttpStatusCode.NotFound, missing.StatusCode); }
        await using var db = await OpenVendorAsync(VendorId); Assert.Empty(await
db.InstallationLicenseAssignments.ToArrayAsync());
    }

    /// <summary>
    /// Verifies that patch preserves omitted fields clears optional limits and rejects
ownership or invalid capacity.
    /// </summary>
    [Fact]
    public async Task
Patch_PreservesOmittedFields_ClearsOptionalLimits_AndRejectsOwnershipOrInvalidCapacity()
    {
        await PrepareAsync(); Guid customer = await CustomerAsync(); Guid product = await
ProductAsync(); Guid license = await LicenseAsync(customer, product, 2, until:
DateTimeOffset.UtcNow.AddDays(2));
        Guid first = await InstallationAsync(customer, product), second = await
InstallationAsync(customer, product);
        using (var assign = await AssignAsync(first, license)) {
Assert.Equal(HttpStatusCode.NoContent, assign.StatusCode); }
        using (var assign = await AssignAsync(second, license)) {
Assert.Equal(HttpStatusCode.NoContent, assign.StatusCode); }
        using (var conflict = await SendAsync(HttpMethod.Patch,
$"/api/v1/licenses/{license}", new { MaxInstallations = 1 })) {
Assert.Equal(HttpStatusCode.Conflict, conflict.StatusCode); }
        using (var rename = await SendAsync(HttpMethod.Patch,
$"/api/v1/licenses/{license}", new { LicenseReference = " Renewed " }))

```

```

        { var body = await ReadAsync(rename); Assert.Equal(2,
body.GetProperty("maxInstallations").GetInt32()); Assert.Equal("Renewed",
body.GetProperty("licenseReference").GetString()); }
        using (var cleared = await SendAsync(HttpMethod.Patch,
$/api/v1/licenses/{license}", new { MaxInstallations = (int?)null, ValidUntil =
(DateTimeOffset?)null }))
        { Assert.Equal(HttpStatusCode.OK, cleared.StatusCode); var body = await
ReadAsync(cleared); Assert.Equal(System.Text.Json.JsonValueKind.Null,
body.GetProperty("maxInstallations").ValueKind);
Assert.Equal(System.Text.Json.JsonValueKind.Null, body.GetProperty("validUntil").ValueKind); }
        foreach (object invalid in new object[] { new { CustomerId = Guid.NewGuid() }, new
{ ProductId = Guid.NewGuid() }, new { Status = "Active" }, new { ValidFrom =
(DateTimeOffset?)null }, new { MaxInstallations = 0 }, new { LicenseReference = " " }, new {
ValidUntil = DateTimeOffset.UtcNow.AddYears(-1) } })
        { using var response = await SendAsync(HttpMethod.Patch,
$/api/v1/licenses/{license}", invalid); Assert.Equal(HttpStatusCode.BadRequest,
response.StatusCode); }
    }

    /// <summary>
    /// Verifies that lifecycle and expiry gate new work without removing assignments or
    monitoring.
    /// </summary>
    [Fact]
    public async Task LifecycleAndExpiry_GateNewWorkWithoutRemovingAssignmentsOrMonitoring()
    {
        await PrepareAsync(); Guid customer = await CustomerAsync(); Guid product = await
ProductAsync(); var managed = await ManagedAsync(customer, product);
        Guid license = await LicenseAsync(customer, product);
        await using var db = await OpenVendorAsync(VendorId); var eligibility =
Services.GetRequiredService<LicenseEligibility>();
        Assert.False((await eligibility.EvaluateAsync(db, managed.Installation)).Allowed);
        using (var assigned = await AssignAsync(managed.Installation, license)) {
Assert.Equal(HttpStatusCode.NoContent, assigned.StatusCode); }
        Assert.True((await eligibility.EvaluateAsync(db, managed.Installation)).Allowed);
        using (var disabled = await SendAsync(HttpMethod.Post,
$/api/v1/licenses/{license}/deactivate")) { Assert.Equal(HttpStatusCode.NoContent,
disabled.StatusCode); }
        Assert.False((await eligibility.EvaluateAsync(db, managed.Installation)).Allowed);
        using (var monitoring = await SendAsync(HttpMethod.Get,
$/api/v1/installations/{managed.Installation}")) { Assert.Equal(HttpStatusCode.OK,
monitoring.StatusCode); }
        using (var heartbeat = await SendAsync(HttpMethod.Post, "/api/v1/agent/heartbeat",
new { AgentVersion = "1", Status = "Healthy", ReportedAt = DateTimeOffset.UtcNow },
managed.AgentToken)) { Assert.Equal(HttpStatusCode.NoContent, heartbeat.StatusCode); }
        Assert.Equal(1, await db.InstallationLicenseAssignments.CountAsync(x =>
x.RemovedAt == null));
        using (var enabled = await SendAsync(HttpMethod.Post,
$/api/v1/licenses/{license}/reactivate")) { Assert.Equal(HttpStatusCode.NoContent,
enabled.StatusCode); }
        Assert.True((await eligibility.EvaluateAsync(db, managed.Installation)).Allowed);
        using (var expired = await SendAsync(HttpMethod.Patch,
$/api/v1/licenses/{license}", new { ValidUntil = DateTimeOffset.UtcNow.AddMinutes(-1) })) {
Assert.Equal(HttpStatusCode.OK, expired.StatusCode); }
        Assert.False((await eligibility.EvaluateAsync(db, managed.Installation)).Allowed);
        using (var disabled = await SendAsync(HttpMethod.Post,
$/api/v1/installations/{managed.Installation}/deactivate")) {
Assert.Equal(HttpStatusCode.NoContent, disabled.StatusCode); }
        using (var denied = await SendAsync(HttpMethod.Post,
$/api/v1/installations/{managed.Installation}/reactivate")) {
Assert.Equal(HttpStatusCode.Conflict, denied.StatusCode); }
    }

```

```

        using (var removed = await SendAsync(HttpMethod.Delete,
$"/api/v1/installations/{managed.Installation}/license")) {
Assert.Equal(HttpStatusCode.NoContent, removed.StatusCode); }
        using (var enabled = await SendAsync(HttpMethod.Post,
$"/api/v1/installations/{managed.Installation}/reactivate")) {
Assert.Equal(HttpStatusCode.NoContent, enabled.StatusCode); }
        Assert.False((await eligibility.EvaluateAsync(db, managed.Installation)).Allowed);
    }

    /// <summary>
    /// Verifies that replacement audit failure rolls back and retains old assignment.
    /// </summary>
    [Fact]
    public async Task Replacement_AuditFailureRollsBackAndRetainsOldAssignment()
    {
        await PrepareAsync(); Guid customer = await CustomerAsync(); Guid product = await
ProductAsync(); Guid installation = await InstallationAsync(customer, product);
        Guid old = await LicenseAsync(customer, product), replacement = await
LicenseAsync(customer, product);
        using (var assigned = await AssignAsync(installation, old)) {
Assert.Equal(HttpStatusCode.NoContent, assigned.StatusCode); }
        await using var db = await OpenVendorAsync(VendorId);
        await db.Database.ExecuteSqlRawAsync("CREATE TRIGGER RejectLicenseAudit ON
VendorAuditLogs AFTER INSERT AS BEGIN IF EXISTS (SELECT 1 FROM inserted WHERE Action =
'InstallationLicenseAssigned') THROW 51000, 'test audit failure', 1; END");
        using (var failed = await AssignAsync(installation, replacement)) {
Assert.Equal(HttpStatusCode.InternalServerError, failed.StatusCode); }
        var original = await
db.InstallationLicenseAssignments.AsNoTracking().SingleAsync(); Assert.Equal(old,
original.LicenseId); Assert.Null(original.RemovedAt);
        await db.Database.ExecuteSqlRawAsync("DROP TRIGGER RejectLicenseAudit");
        using (var replaced = await AssignAsync(installation, replacement)) {
Assert.Equal(HttpStatusCode.NoContent, replaced.StatusCode); }
        Assert.Equal(2, await db.InstallationLicenseAssignments.CountAsync());
Assert.Equal(replacement, (await db.InstallationLicenseAssignments.SingleAsync(x => x.RemovedAt
== null)).LicenseId);
    }
}

```

Integration/OperationalHistoryTests.cs

```

using BridgeControl.DTOS.AgentDTOS.OperationalHistoryContracts;
using System.Net;
using System.Text.Json;
using BridgeControl.Api.Exceptions;
using BridgeControl.DTOS.Shared;
using BridgeControl.DTOS.Vendor.OperationalHistoryContracts;
using BridgeControl.Models.Vendor;
using BridgeControl.Options;
using BridgeControl.Services.Vendor;
using BridgeControl.Tests.Infrastructure;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Options;
using Agent = BridgeControl.Models.AgentModels.Agent;

namespace BridgeControl.Tests.Integration;

/// <summary>
/// Verifies operational history behavior and regression scenarios.

```

```

/// </summary>
public class OperationalHistoryTests : InstallationStateTest
{
    /// <summary>
    /// Verifies that malformed batches consume the request budget before model binding.
    /// </summary>
    [Fact]
    public async Task MalformedBatches_ConsumeTheRequestBudgetBeforeModelBinding()
    {
        var s = await StateAsync();
        for (int n = 0; n < 10; n++) { using var invalid = await SubmitAsync(s);
Assert.Equal(HttpStatusCode.BadRequest, invalid.StatusCode); }
        using var limited = await SubmitAsync(s);
Assert.Equal(HttpStatusCode.TooManyRequests, limited.StatusCode);
        Assert.Equal(TimeSpan.FromSeconds(60), limited.Headers.RetryAfter!.Delta);
        await using var db = await OpenVendorAsync(VendorId); Assert.False(await
db.OperationalLogs.AnyAsync());
    }
    /// <summary>
    /// Verifies that refresh cleanup preserves consumed tokens for live sessions in both
databases.
    /// </summary>
    [Fact]
    public async Task RefreshCleanup_PreservesConsumedTokensForLiveSessions_InBothDatabases()
    {
        var s = await StateAsync(); await using var db = await OpenVendorAsync(VendorId);
        var old = DateTimeOffset.UtcNow.AddDays(-100); Guid user = (await
db.UserAccounts.SingleAsync()).Id;
        var expired = new BridgeControl.Models.Shared.AuthenticationSession { Id =
Guid.NewGuid(), AccountId = user, CreatedAt = old, ExpiresAt = old.AddDays(1) };
        var live = new BridgeControl.Models.Shared.AuthenticationSession { Id =
Guid.NewGuid(), AccountId = user, CreatedAt = old, ExpiresAt = DateTimeOffset.UtcNow.AddDays(1)
};

        db.AuthenticationSessions.AddRange(expired, live);
        var expiredToken = new BridgeControl.Models.Shared.RefreshToken { Id =
Guid.NewGuid(), AuthenticationSessionId = expired.Id, TokenHash =
System.Security.Cryptography.RandomNumberGenerator.GetBytes(32), CreatedAt = old, ExpiresAt =
old.AddDays(1), UsedAt = old.AddHours(1) };
        var consumed = new BridgeControl.Models.Shared.RefreshToken { Id = Guid.NewGuid(),
AuthenticationSessionId = live.Id, TokenHash =
System.Security.Cryptography.RandomNumberGenerator.GetBytes(32), CreatedAt = old, ExpiresAt =
old.AddDays(1), UsedAt = old.AddHours(1) };
        db.RefreshTokens.AddRange(expiredToken, consumed); await db.SaveChangesAsync();
        Guid admin = (await Database.PlatformAdmins.SingleAsync()).Id;
        var platformSession = new BridgeControl.Models.Shared.AuthenticationSession { Id =
Guid.NewGuid(), AccountId = admin, CreatedAt = old, ExpiresAt = old.AddDays(1) };
        Database.AuthenticationSessions.Add(platformSession);
        Database.RefreshTokens.Add(new() { Id = Guid.NewGuid(), AuthenticationSessionId =
platformSession.Id, TokenHash = System.Security.Cryptography.RandomNumberGenerator.GetBytes(32),
CreatedAt = old, ExpiresAt = old.AddDays(1) }); await Database.SaveChangesAsync();
        await using var scope = Factory.Services.CreateAsyncScope(); var cleanup =
scope.ServiceProvider.GetRequiredService<HistoryRetentionService>();
        await cleanup.VendorAsync(VendorId); var platform = await cleanup.PlatformAsync();
Assert.Equal(1, platform.Sessions); Assert.Equal(1, platform.RefreshTokens);
        Assert.False(await db.RefreshTokens.AnyAsync(x => x.Id == expiredToken.Id));
Assert.False(await db.AuthenticationSessions.AnyAsync(x => x.Id == expired.Id));
        Assert.True(await db.RefreshTokens.AnyAsync(x => x.Id == consumed.Id));
Assert.True(await db.AuthenticationSessions.AnyAsync(x => x.Id == live.Id));
        Assert.False(await Database.AuthenticationSessions.AnyAsync(x => x.Id ==
platformSession.Id)); Assert.True(await Database.PlatformAuditLogs.AnyAsync());
    }
}

```

```

    /// <summary>
    /// Verifies that byte limit rejects large unicode batch and revoked credentials cannot
write.
    /// </summary>
    [Fact]
    public async Task ByteLimit_RejectsLargeUnicodeBatch_AndRevokedCredentialsCannotWrite()
    {
        var s = await StateAsync(); var entries = Enumerable.Range(0, 100).Select(n =>
Entry(s.Installation, new string('\u4e00', 1500) + n)).ToArray();
        using var oversized = await SubmitAsync(s, entries);
        Assert.Equal(HttpStatusCode.RequestEntityTooLarge, oversized.StatusCode);
        await using var db = await OpenVendorAsync(VendorId); Assert.False(await
db.OperationalLogs.AnyAsync());
        await db.AgentCredentials.ExecuteUpdateAsync(x => x.SetProperty(c => c.RevokedAt,
DateTimeOffset.UtcNow));
        using var denied = await SubmitAsync(s, Entry(s.Installation));
        Assert.Equal(HttpStatusCode.Unauthorized, denied.StatusCode);
    }

    /// <summary>
    /// Verifies that retention is vendor scoped including inactive vendor history.
    /// </summary>
    [Fact]
    public async Task Retention_IsVendorScoped_IncludingInactiveVendorHistory()
    {
        var s = await StateAsync(); using var post = await SubmitAsync(s,
Entry(s.Installation)); await using var db = await OpenVendorAsync(VendorId);
        await db.OperationalLogs.ExecuteUpdateAsync(x => x.SetProperty(l => l.ReceivedAt,
DateTimeOffset.UtcNow.AddDays(-100)));
        Guid other = await CreateVendorAsync("Other vendor");
        await using var scope = Factory.Services.CreateAsyncScope(); var cleanup =
scope.ServiceProvider.GetRequiredService<HistoryRetentionService>();
        await cleanup.VendorAsync(other); Assert.Equal(1, await
db.OperationalLogs.CountAsync());
        await Database.SoftwareVendors.Where(x => x.Id == VendorId).ExecuteUpdateAsync(x
=> x.SetProperty(v => v.Status, "Inactive"));
        var cleaned = await cleanup.VendorAsync(VendorId); Assert.Equal(1, cleaned.Logs);
        Assert.True(await db.VendorAuditLogs.AnyAsync());
    }

    /// <summary>
    /// Submits a batch of operational log entries through the test API.
    /// </summary>
    private Task<HttpResponseMessage> SubmitAsync(StateFixture s, params
AgentLogEntryRequest[] entries)
    => SendAsync(HttpMethod.Post, "/api/v1/agent/logs", new
SubmitAgentLogsRequest(entries), s.Token);

    /// <summary>
    /// Builds an operational log entry for the test scenario.
    /// </summary>
    private static AgentLogEntryRequest Entry(Guid? installation, string message = "Service
started", string severity = "Information")
    => new(installation, severity, "Runtime", message, DateTimeOffset.UtcNow);

    /// <summary>
    /// Verifies that log batches are atomic deduplicated and redacted with exact frozen
fields.
    /// </summary>
    [Fact]
    public async Task LogBatches_AreAtomicDeduplicatedAndRedacted_WithExactFrozenFields()
    {
        var s = await StateAsync(); var entry = Entry(s.Installation); var secret =
Entry(s.Installation, "Authorization: Bearer private-value", "Error");

```

```

        var posts = await Task.WhenAll(SubmitAsync(s, entry, secret), SubmitAsync(s,
entry, secret));
        foreach (var post in posts) { using (post) {
Assert.Equal(HttpStatusCode.NoContent, post.StatusCode); } }
        using var logs = await SendAsync(HttpMethod.Get, "/api/v1/logs?pageSize=1");
Assert.Equal(HttpStatusCode.OK, logs.StatusCode);
        var body = await ReadAsync(logs); Assert.Equal(2,
body.GetProperty("totalCount").GetInt32()); Assert.Equal(1,
body.GetProperty("items").GetArrayLength());
        using var filtered = await SendAsync(HttpMethod.Get,
$/api/v1/installations/{s.Installation}/logs?severity=Error");
        Assert.Equal("[Sensitive log content removed]", (await
ReadAsync(filtered)).GetProperty("items")[0].GetProperty("message").GetString());
        await using var db = await OpenVendorAsync(VendorId); Assert.Equal(2, await
db.OperationalLogs.CountAsync());
        Assert.DoesNotContain("private-value", JsonSerializer.Serialize(await
db.OperationalLogs.ToArrayAsync()));
        using var foreign = await SubmitAsync(s, Entry(s.Installation, "valid but rolled
back"), Entry(Guid.NewGuid())); Assert.Equal(HttpStatusCode.NotFound, foreign.StatusCode);
        Assert.Equal(2, await db.OperationalLogs.CountAsync());
        using var malformed = await SendAsync(HttpMethod.Get,
"/api/v1/logs?severity=MadeUp"); Assert.Equal(HttpStatusCode.BadRequest, malformed.StatusCode);
    }

    /// <summary>
    /// Verifies that invalid logs are rejected without writes.
    /// </summary>
    [Theory]
    [InlineData("severity")]
    [InlineData("source")]
    [InlineData("message")]
    [InlineData("future")]
    [InlineData("old")]
    [InlineData("batch")]
    [InlineData("null")]
    public async Task InvalidLogs_AreRejectedWithoutWrites(string kind)
    {
        var s = await StateAsync(); var e = Entry(s.Installation);
        e = kind switch
        {
            "severity" => e with { Severity = "Invalid" },
            "source" => e with { Source = "bad\\nsource" },
            "message" => e with { Message = new string('x', 2001) },
            "future" => e with { CreatedAt = DateTimeOffset.UtcNow.AddHours(1) },
            "old" => e with { CreatedAt = DateTimeOffset.UtcNow.AddYears(-1) },
            _ => e
        };
        using var response = await SubmitAsync(s, kind == "batch" ? Enumerable.Repeat(e,
101).ToArray() : kind == "null" ? [null!] : [e]);
        Assert.Equal(HttpStatusCode.BadRequest, response.StatusCode); await using var db =
await OpenVendorAsync(VendorId); Assert.False(await db.OperationalLogs.AnyAsync());
    }

    /// <summary>
    /// Verifies that installation user cannot see shared agent or other installation logs or
counts.
    /// </summary>
    [Fact]
    public async Task InstallationUser_CannotSeeSharedAgentOrOtherInstallationLogsOrCounts()
    {

```

```

        var s = await StateAsync(); using var registered = await
SendAsync(HttpMethod.Post, "/api/v1/agent/installations", new { ProductId = s.Product, Name =
"Second" }, s.Token);
        Guid second = (await ReadAsync(registered)).GetProperty("id").GetGuid();
        using var logs = await SubmitAsync(s, Entry(s.Installation), Entry(second),
Entry(null), Entry(s.Installation, "password=private")); Assert.Equal(HttpStatusCode.NoContent,
logs.StatusCode);
        var reader = await UserAsync("log-reader@example.test", "CustomerUser",
s.Customer);
        using var grant = await SendAsync(HttpMethod.Put,
$"/api/v1/users/{reader.Id}/installation-access", new { InstallationIds = new[] { s.Installation
} });
        foreach (string path in new[] { "/api/v1/logs", $"/api/v1/agents/{s.Agent}/logs",
$"/api/v1/installations/{s.Installation}/logs" })
        {
            using var result = await SendAsync(HttpMethod.Get, path, token:
reader.Token); Assert.Equal(HttpStatusCode.OK, result.StatusCode);
            Assert.Equal(1, (await
ReadAsync(result)).GetProperty("totalCount").GetInt32());
        }
        using var hidden = await SendAsync(HttpMethod.Get,
$"/api/v1/installations/{second}/logs", token: reader.Token);
Assert.Equal(HttpStatusCode.NotFound, hidden.StatusCode);
        var technician = await UserAsync("log-tech@example.test", "VendorTechnician");
        using var empty = await SendAsync(HttpMethod.Get, "/api/v1/logs", token:
technician.Token); Assert.Equal(0, (await
ReadAsync(empty)).GetProperty("totalCount").GetInt32());
        using var customerGrant = await SendAsync(HttpMethod.Put,
$"/api/v1/users/{technician.Id}/customer-access", new { CustomerIds = new[] { s.Customer } });
Assert.Equal(HttpStatusCode.NoContent, customerGrant.StatusCode);
        using var assigned = await SendAsync(HttpMethod.Get, "/api/v1/logs", token:
technician.Token); Assert.Equal(3, (await
ReadAsync(assigned)).GetProperty("totalCount").GetInt32());
        var otherAgent = await AgentAsync(s.Customer);
        using var spoofed = await SubmitAsync(s with { Token = otherAgent.Token },
Entry(s.Installation)); Assert.Equal(HttpStatusCode.NotFound, spoofed.StatusCode);
    }

    /// <summary>
    /// Verifies that log quota and durable rate limit reject new writes but allow stored
duplicates.
    /// </summary>
    [Fact]
    public async Task LogQuotaAndDurableRateLimit_RejectNewWrites_ButAllowStoredDuplicates()
    {
        var s = await StateAsync(); var settings =
Factory.Services.GetRequiredService<IOptions<OperationalHistoryOptions>>().Value;
settings.MaxLogsPerVendor = 100;
        var entries = Enumerable.Range(0, 100).Select(n => Entry(s.Installation, $"Event
{n}")).ToArray();
        using var first = await SubmitAsync(s, entries);
Assert.Equal(HttpStatusCode.NoContent, first.StatusCode);
        using var overflow = await SubmitAsync(s, Entry(s.Installation, "over quota"));
Assert.Equal(HttpStatusCode.Conflict, overflow.StatusCode);
        for (int n = 0; n < 4; n++) { using var replay = await SubmitAsync(s, entries);
Assert.Equal(HttpStatusCode.NoContent, replay.StatusCode); }
        using var limited = await SubmitAsync(s, entries);
Assert.Equal(HttpStatusCode.TooManyRequests, limited.StatusCode);
        await using var db = await OpenVendorAsync(VendorId); Assert.Equal(100, await
db.OperationalLogs.CountAsync());
        Assert.Equal(500, (await db.LogIngestionWindows.SingleAsync()).Entries);
    }

```

```

        Guid credential = (await db.AgentCredentials.SingleAsync(x => x.AgentId ==
s.Agent)).Id;
        await using var restarted = new
ApiFactory(Database.Database.GetConnectionString(), archiveRoot: Factory.ArchiveRoot); using var
client = restarted.CreateApiClient();
        await using var restartedScope = restarted.Services.CreateAsyncScope();
        await Assert.ThrowsAsync<AgentRateLimitException>(() =>
restartedScope.ServiceProvider.GetRequiredService<BridgeControl.Repositories.Vendor.IOperationalH
istoryRepository>().
            .SubmitAsync(new BridgeControl.Security.AgentSecurity.AgentActor(VendorId,
s.Agent, credential), new(entries), default));
    }

    /// <summary>
    /// Verifies that monitoring distinguishes unknown drift failures and expired license then
matching.
    /// </summary>
    [Fact]
    public async Task
Monitoring_DistinguishesUnknownDriftFailuresAndExpiredLicense_ThenMatching()
    {
        var s = await StateAsync();
        using var first = await SendAsync(HttpMethod.Get, "/api/v1/monitoring/overview");
Assert.Equal(HttpStatusCode.OK, first.StatusCode);
        var initial = await ReadAsync(first); Assert.Equal(1,
initial.GetProperty("totalInstallations").GetInt32());
        Assert.Equal("Unknown",
initial.GetProperty("installations").GetProperty("items")[0].GetProperty("drift").GetString());
        using var desired = await SetDesiredAsync(s); using var actual = await
ReportAsync(s, "NotInstalled", DateTimeOffset.UtcNow); await ReconcileAsync();
        using var pending = await SendAsync(HttpMethod.Get,
"/api/v1/monitoring/overview"); Assert.Equal(1, (await
ReadAsync(pending)).GetProperty("pendingDeployments").GetInt32());
        await using var db = await OpenVendorAsync(VendorId);
        await db.Agents.ExecuteUpdateAsync(x => x.SetProperty(a => a.LastSeenAt,
DateTimeOffset.UtcNow.AddMinutes(-6)));
        await db.Licenses.ExecuteUpdateAsync(x => x.SetProperty(l => l.ValidUntil,
DateTimeOffset.UtcNow.AddMinutes(-1)));
        db.EffectiveConfigurationWork.Add(new() { InstallationId = s.Installation,
ProductVersionId = s.Version, Status = "Failed", LastError = "password=hidden", UpdatedAt =
DateTimeOffset.UtcNow, NextAttemptAt = DateTimeOffset.UtcNow }); await db.SaveChangesAsync();
        using var unhealthy = await SendAsync(HttpMethod.Get,
"/api/v1/monitoring/overview"); Assert.Equal(HttpStatusCode.OK, unhealthy.StatusCode);
        var bad = await ReadAsync(unhealthy); Assert.Equal(1,
bad.GetProperty("staleInstallations").GetInt32()); Assert.Equal(1,
bad.GetProperty("driftedInstallations").GetInt32());
        Assert.Equal(1, bad.GetProperty("ineligibleInstallations").GetInt32());
Assert.Equal(1, bad.GetProperty("configurationFailures").GetInt32());
Assert.DoesNotContain("hidden", bad.ToString());
        await db.Licenses.ExecuteUpdateAsync(x => x.SetProperty(l => l.ValidUntil,
(DateTimeOffset?)null));
        await db.EffectiveConfigurationWork.ExecuteUpdateAsync(x => x.SetProperty(w =>
w.Status, "Succeeded"));
        using var match = await ReportAsync(s, "Running",
DateTimeOffset.UtcNow.AddSeconds(1), s.Version, s.Configuration, s.Hash); await ReconcileAsync();
        await db.Agents.ExecuteUpdateAsync(x => x.SetProperty(a => a.LastSeenAt,
DateTimeOffset.UtcNow));
        using var healthy = await SendAsync(HttpMethod.Get,
"/api/v1/monitoring/overview"); Assert.Equal(0, (await
ReadAsync(healthy)).GetProperty("attentionCount").GetInt32());
        await db.Agents.ExecuteUpdateAsync(x => x.SetProperty(a => a.LastSeenAt,
DateTimeOffset.UtcNow.AddMinutes(-16)));
    }

```

```

        using var offline = await SendAsync(HttpMethod.Get,
"/api/v1/monitoring/overview"); Assert.Equal(1, (await
ReadAsync(offline)).GetProperty("offlineInstallations").GetInt32());
        await db.Deployments.ExecuteUpdateAsync(x => x.SetProperty(d => d.Status,
"Failed").SetProperty(d => d.IsActive, false));
        Factory.Clock.Offset = TimeSpan.FromMinutes(16);
        using var stale = await SendAsync(HttpMethod.Get, "/api/v1/monitoring/overview");
var staleBody = await ReadAsync(stale);
        Assert.Equal(1, staleBody.GetProperty("failedDeployments").GetInt32());
        Assert.Equal("Stale",
staleBody.GetProperty("installations").GetProperty("items")[0].GetProperty("freshness").GetString
());
    }

    /// <summary>
    /// Verifies that overview scopes agree with installation assignments.
    /// </summary>
    [Fact]
    public async Task OverviewScopes_AgreeWithInstallationAssignments()
    {
        var s = await StateAsync(); Guid other = await InstallationAsync(await
CustomerAsync("Other"), s.Product);
        var reader = await UserAsync("monitor-reader@example.test", "CustomerUser",
s.Customer);
        using var empty = await SendAsync(HttpMethod.Get, "/api/v1/monitoring/overview",
token: reader.Token); Assert.Equal(0, (await
ReadAsync(empty)).GetProperty("totalInstallations").GetInt32());
        using var grant = await SendAsync(HttpMethod.Put,
$"/api/v1/users/{reader.Id}/installation-access", new { InstallationIds = new[] { s.Installation
} });
        using var mine = await SendAsync(HttpMethod.Get,
"/api/v1/monitoring/overview?pageSize=1", token: reader.Token); var body = await ReadAsync(mine);
        Assert.Equal(1, body.GetProperty("totalInstallations").GetInt32());
        Assert.Equal(s.Installation,
body.GetProperty("installations").GetProperty("items")[0].GetProperty("installationId").GetGuid()
); Assert.DoesNotContain(other.ToString(), body.ToString());
        using var all = await SendAsync(HttpMethod.Get,
"/api/v1/monitoring/overview?pageSize=1"); Assert.Equal(2, (await
ReadAsync(all)).GetProperty("totalInstallations").GetInt32());
    }

    /// <summary>
    /// Verifies that cleanup is bounded atomic restart safe and preserves required evidence.
    /// </summary>
    [Fact]
    public async Task Cleanup_IsBoundedAtomicRestartSafe_AndPreservesRequiredEvidence()
    {
        var s = await StateAsync(); using var submitted = await SubmitAsync(s,
Entry(s.Installation, "Old one"), Entry(s.Installation, "Old two"), Entry(s.Installation,
"Current"));
        await using var db = await OpenVendorAsync(VendorId); var old =
DateTimeOffset.UtcNow.AddDays(-100);
        await db.OperationalLogs.Where(x =>
x.Message.StartsWith("Old")).ExecuteUpdateAsync(x => x.SetProperty(l => l.ReceivedAt, old));
        db.AgentHeartbeats.Add(new() { AgentId = s.Agent, AgentVersion = "old", Status =
"Online", ReceivedAt = old, ReportedAt = old }); await db.SaveChangesAsync();
        using var desired = await SetDesiredAsync(s); int auditCount = await
db.VendorAuditLogs.CountAsync(); int artifacts = await db.FileArtifacts.CountAsync();

        Factory.Services.GetRequiredService<IOptions<OperationalHistoryOptions>>().Value.CleanupBatchSize
= 1;

```

```

        await db.Database.ExecuteSqlRawAsync("CREATE TRIGGER FailHeartbeatCleanup ON
AgentHeartbeats AFTER DELETE AS BEGIN THROW 51000, 'test interruption', 1; END");
        await using var scope = Factory.Services.CreateAsyncScope(); var retention =
scope.ServiceProvider.GetRequiredService<HistoryRetentionService>();
        await Assert.ThrowsAsync<Microsoft.Data.SqlClient.SqlException>(() =>
retention.VendorAsync(VendorId)); Assert.Equal(3, await db.OperationallLogs.CountAsync());
        await db.Database.ExecuteSqlRawAsync("DROP TRIGGER FailHeartbeatCleanup");
        var first = await retention.VendorAsync(VendorId); Assert.Equal(1, first.Logs);
Assert.Equal(1, first.Heartbeats);
        await using var restarted = new
ApiFactory(Database.Database.GetConnectionString()!, archiveRoot: Factory.ArchiveRoot); using var
client = restarted.CreateApiClient();
        await using var restartedScope = restarted.Services.CreateAsyncScope(); var second
= await
restartedScope.ServiceProvider.GetRequiredService<HistoryRetentionService>().VendorAsync(VendorId
); Assert.Equal(1, second.Logs);
        Assert.Equal(1, await db.OperationallLogs.CountAsync()); Assert.Equal(auditCount,
await db.VendorAuditLogs.CountAsync()); Assert.Equal(artifacts, await
db.FileArtifacts.CountAsync()); Assert.True(await db.DesiredStateRevisions.AnyAsync());
    }

    /// <summary>
    /// Verifies that operator audit is read only bounded scoped and sanitizes context.
    /// </summary>
    [Fact]
    public async Task OperatorAudit_IsReadOnlyBoundedScopedAndSanitizesContext()
    {
        var s = await StateAsync(); var reader = await
UserAsync("audit-reader@example.test", "CustomerUser", s.Customer);
        using var grant = await SendAsync(HttpMethod.Put,
$"/api/v1/users/{reader.Id}/installation-access", new { InstallationIds = new[] { s.Installation
} });
        using var desired = await SetDesiredAsync(s); Guid foreign = await
InstallationAsync(await CustomerAsync("Other"), s.Product);
        await using var db = await OpenVendorAsync(VendorId);
        Guid rule = Guid.NewGuid(), review = Guid.NewGuid();
        db.VendorAuditLogs.Add(new()
        {
            ActorId = reader.Id,
            ActorType = "User",
            EntityType = "Installation",
            EntityId = s.Installation,
            Action = "ConfigurationMigrationDestructiveChangeApproved",
            CreatedAt = DateTimeOffset.UtcNow,
            Context = JsonSerializer.Serialize(new { reviewId = review, ruleId = rule,
changeIndex = 2, previousValueHash = new string('a', 64), password = "hidden" })
        });
        db.VendorAuditLogs.Add(new() { ActorId = reader.Id, ActorType = "User", EntityType
= "Installation", EntityId = foreign, Action = "ForeignEvent", CreatedAt = DateTimeOffset.UtcNow
}); await db.SaveChangesAsync();
        int count = await db.VendorAuditLogs.CountAsync();
        await using var scope = Factory.Services.CreateAsyncScope(); var audit =
scope.ServiceProvider.GetRequiredService<AuditHistoryOperator>();
        var filter = new AuditHistoryQuery(DateTimeOffset.UtcNow.AddDays(-1),
DateTimeOffset.UtcNow.AddDays(1));
        var rows = await audit.VendorAsync(VendorId, reader.Id, filter, new());
Assert.All(rows.Items, x => Assert.Equal(s.Installation, Guid.Parse(x.EntityId)));
        var evidence = Assert.Single(rows.Items, x => x.Action ==
"ConfigurationMigrationDestructiveChangeApproved");
        using var context = JsonDocument.Parse(evidence.Context); Assert.Equal(rule,
context.RootElement.GetProperty("ruleId").GetGuid()); Assert.DoesNotContain("hidden",
evidence.Context);

```

```

        await Assert.ThrowsAsync<ApiAccessException>(() => audit.VendorAsync(VendorId,
reader.Id, filter, new(), installationId: foreign));
        await Assert.ThrowsAsync<ApiAccessException>(() => audit.PlatformAsync(reader.Id,
filter, new()));
        Guid platformAdmin = (await Database.PlatformAdmins.SingleAsync()).Id;
        var platformRows = await audit.PlatformAsync(platformAdmin, filter, new());
Assert.NotEmpty(platformRows.Items);
        Assert.All(platformRows.Items, x => Assert.Equal("PlatformAdministrator",
x.ActorType));
        Guid vendorAdmin = (await db.UserAccounts.SingleAsync(x => x.Email ==
"invited@example.test")).Id;
        var grants = await audit.VendorAsync(VendorId, vendorAdmin, filter with { Action =
"InstallationAccessGranted", ResourceId = reader.Id }, new());
        Assert.Single(grants.Items);
        await Assert.ThrowsAsync<ArgumentException>(() => audit.VendorAsync(VendorId,
reader.Id, filter with { From = DateTimeOffset.UtcNow.AddYears(-1) }, new()));
        Assert.Equal(count, await db.VendorAuditLogs.CountAsync());
        using var noEndpoint = await SendAsync(HttpMethod.Get, "/api/v1/audit");
Assert.Equal(HttpStatusCode.Unauthorized, noEndpoint.StatusCode); // Unmapped routes use the
existing fallback policy.
    }
}

```

Integration/PlatformAdministratorTests.cs

```

using BridgeControl.Models.Platform;
using BridgeControl.Services.Platform;
using BridgeControl.Services.Shared;
using BridgeControl.Tests.Infrastructure;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;

namespace BridgeControl.Tests.Integration;

/// <summary>
/// Verifies platform administrator behavior and regression scenarios.
/// </summary>
[Trait("Category", "Integration")]
public class PlatformAdministratorTests : SqlServerTest
{
    private const string Email = "admin@example.test";
    private const string Password = "1234567890";

    /// <summary>
    /// Verifies that create async concurrent setup creates only one administrator and repeat
    is refused.
    /// </summary>
    [Fact]
    public async Task
CreateAsync_ConcurrentSetupCreatesOnlyOneAdministratorAndRepeatIsRefused()
    {
        async Task<bool> Create()
        {
            await using AsyncServiceScope scope = Factory.Services.CreateAsyncScope();
            return await
scope.ServiceProvider.GetRequiredService<PlatformAdminSetup>().CreateAsync(" ADMIN@EXAMPLE.TEST
", Password);
        }
        bool[] results = await Task.WhenAll(Create(), Create());
    }
}

```

```

        Assert.Single(results, created => created);
        PlatformAdmin admin = await Database.PlatformAdmins.AsNoTracking().SingleAsync();
        Assert.Equal(Email, admin.Email);
        Assert.NotEqual>Password, admin.PasswordHash);
        Assert.Equal(1, await Database.PlatformAuditLogs.CountAsync(log => log.Action ==
"FirstAdministratorCreated"));
        await Database.PlatformAdmins.ExecuteUpdateAsync(setters =>
setters.SetProperty(account => account.Status, "Disabled"));
        Assert.False(await Create());
    }

    /// <summary>
    /// Verifies that reset async changes only an existing password and revokes sessions
without changing status.
    /// </summary>
    [Theory]
    [InlineData("Active")]
    [InlineData("Disabled")]
    public async Task
ResetAsync_ChangesOnlyAnExistingPasswordAndRevokesSessionsWithoutChangingStatus(string status)
    {
        await CreateAdministratorAndSession();
        await Database.PlatformAdmins.ExecuteUpdateAsync(setters =>
setters.SetProperty(account => account.Status, status));
        Database.ChangeTracker.Clear();
        const string replacement = " new-password ";
        PlatformAdminRecovery recovery =
Services.GetRequiredService<PlatformAdminRecovery>();

        Assert.False(await recovery.ResetAsync("absent@example.test", replacement));
        Assert.True(await recovery.ResetAsync(" ADMIN@EXAMPLE.TEST ", replacement));

        PlatformAdmin admin = await Database.PlatformAdmins.AsNoTracking().SingleAsync();
        PasswordService<PlatformAdmin> passwords =
Services.GetRequiredService<PasswordService<PlatformAdmin>>();
        Assert.Equal(status, admin.Status);
        Assert.Equal>PasswordVerificationResult.Success, passwords.Verify(admin,
admin.PasswordHash, replacement));
        Assert.Equal>PasswordVerificationResult.Failed, passwords.Verify(admin,
admin.PasswordHash, Password));
        Assert.Equal>PasswordVerificationResult.Failed, passwords.Verify(admin,
admin.PasswordHash, replacement.Trim()));
        Assert.False(await Database.AuthenticationSessions.AnyAsync(session =>
session.RevokedAt == null));
        Assert.Equal(1, await Database.PlatformAuditLogs.CountAsync(log => log.Action ==
"PlatformAdministratorPasswordRecovered"));
    }

    /// <summary>
    /// Verifies that reset async rolls back password and revocation if its audit cannot be
saved.
    /// </summary>
    [Fact]
    public async Task ResetAsync_RollsBackPasswordAndRevocationIfItsAuditCannotBeSaved()
    {
        await CreateAdministratorAndSession();
        string hash = (await
Database.PlatformAdmins.AsNoTracking().SingleAsync()).PasswordHash;
        await Database.Database.ExecuteSqlRawAsync("""
            CREATE TRIGGER TestRecoveryAuditFailure ON PlatformAuditLogs AFTER INSERT
AS

```

```

        BEGIN
            IF EXISTS (SELECT 1 FROM inserted WHERE Action =
'PlatformAdministratorPasswordRecovered')
                THROW 51008, 'Test recovery audit failure.', 1;
            END
        """);

        await using AsyncServiceScope scope = Factory.Services.CreateAsyncScope();
        await Assert.ThrowsAsync<DbUpdateException>(() =>
scope.ServiceProvider.GetRequiredService<PlatformAdminRecovery>()
    .ResetAsync(Email, "new-password"));

        Assert.Equal(hash, (await
Database.PlatformAdmins.AsNoTracking().SingleAsync()).PasswordHash);
        Assert.Null((await
Database.AuthenticationSessions.AsNoTracking().SingleAsync()).RevokedAt);
    }

    /// <summary>
    /// Creates a platform administrator and an active session for recovery tests.
    /// </summary>
    private async Task CreateAdministratorAndSession()
    {
        Assert.True(await
Services.GetRequiredService<PlatformAdminSetup>().CreateAsync(Email, Password));
        Assert.NotNull(await
Services.GetRequiredService<PlatformAuthenticationService>().LoginAsync(Email, Password,
CancellationToken.None));
    }
}

```

Integration/PlatformAuthenticationTests.cs

```

using System.IdentityModel.Tokens.Jwt;
using System.Net;
using System.Net.Http.Headers;
using System.Net.Http.Json;
using System.Security.Claims;
using System.Security.Cryptography;
using System.Text.Json;
using BridgeControl.Security;
using BridgeControl.Services.Platform;
using BridgeControl.Tests.Infrastructure;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.IdentityModel.Tokens;

namespace BridgeControl.Tests.Integration;

/// <summary>
/// Verifies platform authentication behavior and regression scenarios.
/// </summary>
[Trait("Category", "Integration")]
public class PlatformAuthenticationTests : SqlServerTest
{
    private const string Email = "admin@example.test";
    private const string Password = " 12345678 ";
    private const string LoginPath = "/api/v1/platform/auth/login";
    private const string RefreshPath = "/api/v1/platform/auth/refresh";
    private const string LogoutPath = "/api/v1/platform/auth/logout";
}

```

```

private const string MePath = "/api/v1/platform/account/me";

/// <summary>
/// Verifies that login returns a secure cookie and access to only the current account.
/// </summary>
[Fact]
public async Task Login_ReturnsASecureCookieAndAccessToOnlyTheCurrentAccount()
{
    Session session = await StartSession();
    using HttpResponseMessage response = await SendAuthorized(HttpMethod.Get, MePath,
session.Token);
    Assert.Equal(HttpStatusCode.OK, response.StatusCode);
    JsonElement body = await ReadJson(response);
    Assert.Equal(Email, body.GetProperty("email").GetString());
    Assert.Equal(new[] { "createdAt", "email", "id", "lastLoginAt", "status" },
body.EnumerateObject().Select(property => property.Name).Order());
    Assert.Equal((await Database.PlatformAdmins.AsNoTracking().SingleAsync()).Id,
body.GetProperty("id").GetGuid());
}

/// <summary>
/// Verifies that login uses the same public error for unknown accounts and wrong
passwords.
/// </summary>
[Fact]
public async Task Login_UsesTheSamePublicErrorForUnknownAccountsAndWrongPasswords()
{
    await CreateAdministrator();
    using HttpResponseMessage wrong = await Client.PostAsJsonAsync(LoginPath, new {
Email, Password = Password.Trim() });
    using HttpResponseMessage unknown = await Client.PostAsJsonAsync(LoginPath, new {
Email = "unknown@example.test", Password });
    Assert.Equal(HttpStatusCode.Unauthorized, wrong.StatusCode);
    Assert.Equal(wrong.StatusCode, unknown.StatusCode);
    Assert.Equal((await ReadJson(wrong)).GetProperty("title").GetString(), (await
ReadJson(unknown)).GetProperty("title").GetString());
}

/// <summary>
/// Verifies that me requires https and authentication.
/// </summary>
[Theory]
[InlineData("http://localhost/api/v1/platform/account/me", HttpStatusCode.BadRequest)]
[InlineData("https://localhost/api/v1/platform/account/me", HttpStatusCode.Unauthorized)]
public async Task Me_RequiresHttpsAndAuthentication(string url, HttpStatusCode status)
{
    using HttpResponseMessage response = await Client.GetAsync(url);
    Assert.Equal(status, response.StatusCode);
}

/// <summary>
/// Verifies that me rejects invalid tokens or insufficient roles.
/// </summary>
[Theory]
[InlineData("audience", HttpStatusCode.Unauthorized)]
[InlineData("signature", HttpStatusCode.Unauthorized)]
[InlineData("identity", HttpStatusCode.Unauthorized)]
[InlineData("expired", HttpStatusCode.Unauthorized)]
[InlineData("account", HttpStatusCode.Unauthorized)]
[InlineData("token-type", HttpStatusCode.Unauthorized)]
[InlineData("role", HttpStatusCode.Forbidden)]

```

```

        public async Task Me_RejectsInvalidTokensOrInsufficientRoles(string change, HttpStatusCode
expected)
        {
            Session session = await StartSession();
            JwtSecurityToken original = new
JwtSecurityTokenHandler().ReadJwtToken(session.Token);
            List<Claim> claims = original.Claims.Where(claim => claim.Type is not "aud" and
not "iss" and not "exp" and not "nbf" and not "identity_type" and not "role" and not
"sub").ToList();
            claims.Add(new Claim("sub", change == "account" ? Guid.NewGuid().ToString() :
original.Subject));
            claims.Add(new Claim("identity_type", change == "identity" ? "vendor_user" :
PlatformAuthentication.IdentityType));
            if (change != "role") { claims.Add(new Claim("role",
PlatformAuthentication.Role)); }
            string key = change == "signature" ?
Convert.ToBase64String(RandomNumberGenerator.GetBytes(32)) : Factory.SigningKey;
            JwtSecurityToken token = new("BridgeControl", change == "audience" ?
"BridgeControl.Vendor" : "BridgeControl.Platform", claims,
                DateTime.UtcNow.AddMinutes(-20), change == "expired" ?
DateTime.UtcNow.AddMinutes(-1) : DateTime.UtcNow.AddMinutes(5),
                new SigningCredentials(new
SymmetricSecurityKey(Convert.FromBase64String(key)), SecurityAlgorithms.HmacSha256));
            token.Header["typ"] = change == "token-type" ? "refresh+jwt" : "at+jwt";

            using HttpResponseMessage response = await SendAuthorized(HttpMethod.Get, MePath,
new JwtSecurityTokenHandler().WriteToken(token));

            Assert.Equal(expected, response.StatusCode);
            Assert.Equal("application/problem+json",
response.Content.Headers.ContentType?.MediaType);
        }

        /// <summary>
        /// Verifies that refresh requires the header and revokes the session when a credential is
reused.
        /// </summary>
        [Fact]
        public async Task Refresh_RequiresTheHeaderAndRevokesTheSessionWhenACredentialIsReused()
        {
            Session original = await StartSession();
            using HttpResponseMessage noHeader = await Refresh(original.Cookie, includeHeader:
false);

            Assert.Equal(HttpStatusCode.BadRequest, noHeader.StatusCode);
            using HttpResponseMessage rotatedResponse = await Refresh(original.Cookie);
            Session rotated = await ReadSession(rotatedResponse);
            Assert.NotEqual(original.Cookie, rotated.Cookie);
            using HttpResponseMessage reused = await Refresh(original.Cookie);
            Assert.Equal(HttpStatusCode.Unauthorized, reused.StatusCode);
            using HttpResponseMessage revoked = await SendAuthorized(HttpMethod.Get, MePath,
rotated.Token);
            Assert.Equal(HttpStatusCode.Unauthorized, revoked.StatusCode);
        }

        /// <summary>
        /// Verifies that refresh concurrent requests cannot both rotate the same credential.
        /// </summary>
        [Fact]
        public async Task Refresh_ConcurrentRequestsCannotBothRotateTheSameCredential()
        {
            Session session = await StartSession();

```

```

        HttpResponseMessage[] responses = await Task.WhenAll(Refresh(session.Cookie),
Refresh(session.Cookie));
        try
        {
            Assert.Single(responses, response => response.StatusCode ==
HttpStatusCode.OK);
            Assert.Single(responses, response => response.StatusCode ==
HttpStatusCode.Unauthorized);
            Session winner = await ReadSession(responses.Single(response =>
response.IsSuccessStatusCode));
            using HttpResponseMessage revoked = await SendAuthorized(HttpMethod.Get,
MePath, winner.Token);
            Assert.Equal(HttpStatusCode.Unauthorized, revoked.StatusCode);
        }
        finally { foreach (HttpResponseMessage response in responses) {
response.Dispose(); } }
    }

    /// <summary>
    /// Verifies that logout revokes its session but preserves an independent session.
    /// </summary>
    [Fact]
    public async Task Logout_RevokesItsSessionButPreservesAnIndependentSession()
    {
        Session first = await StartSession();
        Session independent = await Login();
        using HttpResponseMessage logout = await SendAuthorized(HttpMethod.Post,
LogoutPath, first.Token);
        Assert.Equal(HttpStatusCode.NoContent, logout.StatusCode);
        using HttpResponseMessage revokedAccess = await SendAuthorized(HttpMethod.Get,
MePath, first.Token);
        using HttpResponseMessage revokedRefresh = await Refresh(first.Cookie);
        using HttpResponseMessage active = await SendAuthorized(HttpMethod.Get, MePath,
independent.Token);
        Assert.Equal(HttpStatusCode.Unauthorized, revokedAccess.StatusCode);
        Assert.Equal(HttpStatusCode.Unauthorized, revokedRefresh.StatusCode);
        Assert.Equal(HttpStatusCode.OK, active.StatusCode);
    }

    /// <summary>
    /// Verifies that me disabling an account revokes its existing session permanently.
    /// </summary>
    [Fact]
    public async Task Me_DisablingAnAccountRevokesItsExistingSessionPermanently()
    {
        Session session = await StartSession();
        await Database.PlatformAdmins.ExecuteUpdateAsync(setters =>
setters.SetProperty(account => account.Status, "Disabled"));
        using HttpResponseMessage disabled = await SendAuthorized(HttpMethod.Get, MePath,
session.Token);
        Assert.Equal(HttpStatusCode.Unauthorized, disabled.StatusCode);
        await Database.PlatformAdmins.ExecuteUpdateAsync(setters =>
setters.SetProperty(account => account.Status, "Active"));
        using HttpResponseMessage reactivated = await SendAuthorized(HttpMethod.Get,
MePath, session.Token);
        Assert.Equal(HttpStatusCode.Unauthorized, reactivated.StatusCode);
    }

    /// <summary>
    /// Verifies that refresh rejects expired sessions or refresh credentials.
    /// </summary>
    [Theory]

```

```

[InlineData(false)]
[InlineData(true)]
public async Task Refresh_RejectsExpiredSessionsOrRefreshCredentials(bool expireSession)
{
    Session session = await StartSession();
    DateTimeOffset yesterday = DateTimeOffset.UtcNow.AddDays(-1);
    if (expireSession)
    {
        await Database.AuthenticationSessions.ExecuteUpdateAsync(setters => setters
            .SetProperty(item => item.CreatedAt,
                yesterday.AddDays(-1)).SetProperty(item => item.ExpiresAt, yesterday));
    }
    else
    {
        await Database.RefreshTokens.ExecuteUpdateAsync(setters => setters
            .SetProperty(item => item.CreatedAt,
                yesterday.AddDays(-1)).SetProperty(item => item.ExpiresAt, yesterday));
    }
    using HttpResponseMessage response = await Refresh(session.Cookie);
    Assert.Equal(HttpStatusCode.Unauthorized, response.StatusCode);
}

/// <summary>
/// Verifies that login rate limit applies across route variants.
/// </summary>
[Fact]
public async Task Login_RateLimitAppliesAcrossRouteVariants()
{
    await using ApiFactory limited = new(Database.Database.GetConnectionString()!,
        loginLimit: 2);
    using HttpClient client = limited.CreateApiClient();
    foreach (string path in new[] { LoginPath, LoginPath.ToUpperInvariant() + "/" })
    {
        using HttpResponseMessage response = await client.PostAsJsonAsync(path, new
            { Email, Password });
        Assert.Equal(HttpStatusCode.Unauthorized, response.StatusCode);
    }
    using HttpResponseMessage rejected = await client.PostAsJsonAsync(LoginPath, new {
        Email, Password });
    Assert.Equal(HttpStatusCode.TooManyRequests, rejected.StatusCode);
    Assert.NotNull(rejected.Headers.RetryAfter);
}

/// <summary>
/// Creates the platform administrator account used by the test.
/// </summary>
private Task<bool> CreateAdministrator() =>
    Services.GetRequiredService<PlatformAdminSetup>().CreateAsync(Email, Password);
/// <summary>
/// Starts a platform authentication session for the test administrator.
/// </summary>
private async Task<Session> StartSession()
{
    Assert.True(await CreateAdministrator());
    return await Login();
}

/// <summary>
/// Sends a platform login request with the test credentials.
/// </summary>
private async Task<Session> Login()
{

```

```

        using HttpResponseMessage response = await Client.PostAsJsonAsync(LoginPath, new {
Email, Password });
        return await ReadSession(response);
    }
    /// <summary>
    /// Reads the session response and refresh cookie from a test login.
    /// </summary>
    private static async Task<Session> ReadSession(HttpResponseMessage response)
    {
        Assert.Equal(HttpStatusCode.OK, response.StatusCode);
        JsonElement body = await ReadJson(response);
        Assert.False(body.TryGetProperty("refreshToken", out _));
        Assert.False(body.TryGetProperty("refreshCredential", out _));
        string cookie = Assert.Single(response.Headers.GetValues("Set-Cookie"), value =>
value.StartsWith(PlatformRefreshCookie.Name + "="));
        Assert.Contains("secure", cookie, StringComparison.OrdinalIgnoreCase);
        Assert.Contains("httponly", cookie, StringComparison.OrdinalIgnoreCase);
        Assert.Contains("samesite=strict", cookie, StringComparison.OrdinalIgnoreCase);
        Assert.Contains("path=/api/v1/platform/auth", cookie);
        Assert.True(response.Headers.CacheControl?.NoStore);
        return new Session(body.GetProperty("accessToken").GetString()!,
cookie.Split(';')[0]);
    }
    /// <summary>
    /// Sends a refresh request using the test session's cookie and required header.
    /// </summary>
    private async Task<HttpResponseMessage> Refresh(string cookie, bool includeHeader = true)
    {
        using HttpRequestMessage request = new(HttpMethod.Post, RefreshPath);
        request.Headers.Add("Cookie", cookie);
        if (includeHeader) { request.Headers.Add(PlatformRefreshCookie.RequestHeader,
"1"); }
        return await Client.SendAsync(request);
    }
    /// <summary>
    /// Sends an HTTP request with the test access token.
    /// </summary>
    private async Task<HttpResponseMessage> SendAuthorized(HttpMethod method, string path,
string token)
    {
        using HttpRequestMessage request = new(method, path);
        request.Headers.Authorization = new AuthenticationHeaderValue("Bearer", token);
        return await Client.SendAsync(request);
    }
    /// <summary>
    /// Deserializes JSON from a test HTTP response.
    /// </summary>
    private static async Task<JsonElement> ReadJson(HttpResponseMessage response)
    => JsonSerializer.Deserialize<JsonElement>(await
response.Content.ReadAsStringAsync());
    /// <summary>
    /// Represents session data.
    /// </summary>
    private sealed record Session(string Token, string Cookie);
}

```

Integration/ProductAdministrationTests.cs

```

using System.Net;
using System.Net.Http.Headers;

```

```

using System.Net.Http.Json;
using System.Security.Cryptography;
using System.Text;
using System.Text.Json;
using BridgeControl.Models.Vendor;
using BridgeControl.Options;
using BridgeControl.Tests.Infrastructure;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Infrastructure;
using Microsoft.EntityFrameworkCore.Migrations;
using BridgeControl.Services.Vendor;
using BridgeControl.Repositories.Vendor;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Options;
using BridgeControl.Api.Exceptions;

namespace BridgeControl.Tests.Integration;

/// <summary>
/// Verifies product administration behavior and regression scenarios.
/// </summary>
[Trait("Category", "Integration")]
public class ProductAdministrationTests : VendorTest
{
    private Guid vendorId;
    private string token = "";
    private const string Password = "sprint-seven-password";
    internal const string Schema =
        ""{"$schema":"https://json-schema.org/draft/2020-12/schema","type":"object","properties":{"port":
        :{"type":"integer","minimum":1,"maximum":65535}},"additionalProperties":false}"";
    private string ArchiveRoot => Path.Combine(Factory.ArchiveRoot, vendorId.ToString("N"));
    /// <summary>
    /// Creates the accounts and resources required by the test scenario.
    /// </summary>
    private async Task PrepareAsync()
    {
        await SignInAsync(); vendorId = await CreateVendorAsync();
        string invitation = await InviteAsync(vendorId);
        using var activate = await Client.PostAsJsonAsync("/api/v1/auth/activate", new {
Token = invitation, Password });
        Assert.Equal(HttpStatusCode.NoContent, activate.StatusCode);
        using var login = await Client.PostAsJsonAsync("/api/v1/auth/login", new {
VendorName = "Test vendor", Email = "invited@example.test", Password });
        Assert.Equal(HttpStatusCode.OK, login.StatusCode); token = await
ReadAsync(login).GetProperty("accessToken").GetString()!;
    }
    /// <summary>
    /// Sends the test HTTP request with the supplied method, credentials, and payload.
    /// </summary>
    private async Task<HttpResponseMessage> SendAsync(HttpMethod method, string path, object?
body = null)
    {
        using HttpRequestMessage request = new(method, path);
        request.Headers.Authorization = new AuthenticationHeaderValue("Bearer", token);
        if (body is not null) { request.Content = JsonContent.Create(body); }
        return await Client.SendAsync(request);
    }
    /// <summary>
    /// Creates a product for the test scenario.
    /// </summary>
    private async Task<Guid> ProductAsync(string name = "Bridge", string? description =
"Description")

```

```

        {
            using var response = await SendAsync(HttpMethod.Post, "/api/v1/products", new {
                Name = name, Description = description });
            Assert.Equal(HttpStatusCode.Created, response.StatusCode);
            var json = await ReadAsync(response);
            Assert.Equal(new[] { "createdAt", "description", "id", "name", "status" },
                json.EnumerateObject().Select(x => x.Name).Order());
            Assert.Equal("Active", json.GetProperty("status").GetString()); return
            json.GetProperty("id").GetGuid();
        }
        /// <summary>
        /// Creates a product version for the test scenario.
        /// </summary>
        private async Task<Guid> VersionAsync(Guid productId, string version = "1.0.0")
        {
            using var response = await SendAsync(HttpMethod.Post,
                $"/api/v1/products/{productId}/versions", new { Version = version });
            Assert.Equal(HttpStatusCode.Created, response.StatusCode);
            var json = await ReadAsync(response);
            Assert.Equal(new[] { "createdAt", "id", "productId", "publishedAt", "status",
                "version" }, json.EnumerateObject().Select(x => x.Name).Order());
            Assert.Equal("Draft", json.GetProperty("status").GetString()); return
            json.GetProperty("id").GetGuid();
        }
        /// <summary>
        /// Builds the API path for the selected product-version artifact.
        /// </summary>
        private static string PathFor(Guid product, Guid version, string kind = "schema") =>
            $"/api/v1/products/{product}/versions/{version}/{kind}";
        /// <summary>
        /// Uploads the supplied artifact through the test API.
        /// </summary>
        private async Task<HttpResponseMessage> UploadAsync(string path, byte[] bytes, string
            fileName = "schema.json", bool extraField = false)
        {
            using var request = new HttpRequestMessage(HttpMethod.Put, path);
            request.Headers.Authorization = new AuthenticationHeaderValue("Bearer", token);
            var body = new MultipartFormDataContent();
            var file = new ByteArrayContent(bytes); file.Headers.ContentType = new
            MediaTypeHeaderValue("application/octet-stream");
            body.Add(file, "file", fileName);
            if (extraField) { body.Add(new StringContent("ignored?"), "sha256"); }
            request.Content = body; return await Client.SendAsync(request);
        }
        /// <summary>
        /// Verifies that products patch pagination lifecycle and history follow contract.
        /// </summary>
        [Fact]
        public async Task Products_PatchPaginationLifecycleAndHistoryFollowContract()
        {
            await PrepareAsync(); Guid product = await ProductAsync(); Guid version = await
            VersionAsync(product);
            await ProductAsync("Second", null);
            using var page = await SendAsync(HttpMethod.Get,
                $"/api/v1/products?pageSize=1&page=2");
            var json = await ReadAsync(page); Assert.Equal(2,
                json.GetProperty("totalCount").GetInt32()); Assert.Equal(1,
                json.GetProperty("items").GetArrayLength());
            using var rename = await SendAsync(HttpMethod.Patch,
                $"/api/v1/products/{product}", new { Name = "Renamed " });

```

```

        Assert.Equal("Description", (await
ReadAsync(rename)).GetProperty("description").GetString());
        using var clear = await SendAsync(HttpMethod.Patch, $"/api/v1/products/{product}",
new { Description = (string?)null });
        Assert.Equal(JsonValueKind.Null, (await
ReadAsync(clear)).GetProperty("description").ValueKind);
        Assert.Equal("Renamed", (await ReadAsync(clear)).GetProperty("name").GetString());
        await using var db = await OpenVendorAsync(vendorId);
        var customer = new Customer { Id = Guid.NewGuid(), Name = "History", Status =
"Active" };
        db.AddRange(customer, new Installation { Name = "Installed", CustomerId =
customer.Id, ProductId = product }); await db.SaveChangesAsync();
        foreach (string action in new[] { "deactivate", "deactivate", "reactivate" })
        { using var response = await SendAsync(HttpMethod.Post,
$"/api/v1/products/{product}/{action}"); Assert.Equal(HttpStatusCode.NoContent,
response.StatusCode); }
        Assert.Equal(1, await db.Installations.CountAsync()); Assert.Equal(1, await
db.ProductVersions.CountAsync());
        Assert.Equal(1, await db.VendorAuditLogs.CountAsync(x => x.Action ==
"ProductDeactivated"));
        using var versions = await SendAsync(HttpMethod.Get,
$"/api/v1/products/{product}/versions"); Assert.Equal(1, (await
ReadAsync(versions)).GetProperty("totalCount").GetInt32());
        using var invalidPage = await SendAsync(HttpMethod.Get,
"/api/v1/products?pageSize=101"); Assert.Equal(HttpStatusCode.BadRequest,
invalidPage.StatusCode);
    }
    /// <summary>
    /// Verifies that drafts enforce nested ownership uniqueness and published immutability.
    /// </summary>
    [Fact]
    public async Task Drafts_EnforceNestedOwnershipUniquenessAndPublishedImmutability()
    {
        await PrepareAsync(); Guid product = await ProductAsync(); Guid other = await
ProductAsync("Other"); Guid version = await VersionAsync(product);
        using var duplicate = await SendAsync(HttpMethod.Post,
$"/api/v1/products/{product}/versions", new { Version = " 1.0.0 " });
        Assert.Equal(HttpStatusCode.Conflict, duplicate.StatusCode);
        await VersionAsync(other); // The same label is legal in a different product.
        using var wrongParent = await SendAsync(HttpMethod.Get,
$"/api/v1/products/{other}/versions/{version}"); Assert.Equal(HttpStatusCode.NotFound,
wrongParent.StatusCode);
        using var changed = await SendAsync(HttpMethod.Patch,
$"/api/v1/products/{product}/versions/{version}", new { Version = "2.0.0" });
        Assert.Equal(HttpStatusCode.OK, changed.StatusCode);
        await using var db = await OpenVendorAsync(vendorId);
        await db.ProductVersions.Where(x => x.Id == version).ExecuteUpdateAsync(s =>
s.SetProperty(x => x.Status, "Published").SetProperty(x => x.PublishedAt,
DateTimeOffset.UtcNow));
        using var frozen = await SendAsync(HttpMethod.Patch,
$"/api/v1/products/{product}/versions/{version}", new { Version = "3.0.0" });
        Assert.Equal(HttpStatusCode.Conflict, frozen.StatusCode);
        using var read = await SendAsync(HttpMethod.Get,
$"/api/v1/products/{product}/versions/{version}"); Assert.Equal("2.0.0", (await
ReadAsync(read)).GetProperty("version").GetString());
    }
    /// <summary>
    /// Verifies that artifacts round trip idempotent replacement removal and frozen release.
    /// </summary>
    [Theory]
    [InlineData("schema")]
    [InlineData("installer")]

```

```

        public async Task Artifacts_RoundTripIdempotentReplacementRemovalAndFrozenRelease(string
kind)
    {
        await PrepareAsync(); Guid product = await ProductAsync(); Guid version = await
VersionAsync(product);
        string path = PathFor(product, version, kind); string fileName = kind == "schema"
? "schema.json" : "setup.msi";
        byte[] bytes = kind == "schema" ? Encoding.UTF8.GetBytes(Schema) :
MsiFixture.Create(Factory.PrivateRoot);
        string otherPath = PathFor(product, version, kind == "schema" ? "installer" :
"schema");
        foreach (string missingPath in new[] { path, otherPath })
        {
            using var missingArtifact = await SendAsync(HttpMethod.Get, missingPath);
            Assert.Equal(HttpStatusCode.NotFound, missingArtifact.StatusCode);
            Assert.Equal("Resource not found.", (await
ReadAsync(missingArtifact)).GetProperty("title").GetString());
        }
        using var uploaded = await UploadAsync(path, bytes, fileName);
        Assert.Equal(HttpStatusCode.OK, uploaded.StatusCode);
        using var otherArtifact = await SendAsync(HttpMethod.Get, otherPath);
        Assert.Equal(HttpStatusCode.NotFound, otherArtifact.StatusCode);
        var metadata = await ReadAsync(uploaded); Guid original =
metadata.GetProperty("id").GetGuid();
        Assert.Equal(new[] { "artifactType", "contentType", "createdAt", "fileName", "id",
"sha256", "sizeBytes" }, metadata.EnumerateObject().Select(x => x.Name).Order());
        Assert.Equal(Convert.ToHexStringLower(SHA256.HashData(bytes)),
metadata.GetProperty("sha256").GetString()); Assert.Equal(bytes.Length,
metadata.GetProperty("sizeBytes").GetInt64());
        using var download = await SendAsync(HttpMethod.Get, path + "/content");
        Assert.Equal(bytes, await download.Content.ReadAsByteArrayAsync());
        Assert.NotNull(download.Content.Headers.ContentDisposition);
        Assert.True(download.Headers.CacheControl!.NoStore);
        using var retry = await UploadAsync(path, bytes, fileName); Assert.Equal(original,
(await ReadAsync(retry)).GetProperty("id").GetGuid());
        await using var db = await OpenVendorAsync(vendorId);
        Assert.Equal(1, await db.FileArtifacts.CountAsync()); Assert.Equal(1, await
db.VendorAuditLogs.CountAsync(x => x.Action == (kind == "schema" ? "SchemaUploaded" :
"InstallerUploaded")));
        using var replaced = await UploadAsync(path, bytes, "new-" + fileName);
        Assert.Equal(HttpStatusCode.OK, replaced.StatusCode);
        Assert.NotEqual(original, (await
ReadAsync(replaced)).GetProperty("id").GetGuid());
        Assert.Equal(2, await db.FileArtifacts.CountAsync()); Assert.Equal(1, await
db.ProductVersionArtifacts.CountAsync());
        Assert.Equal(2, Directory.GetFiles(Path.Combine(ArchiveRoot, "blobs")).Length);
        Assert.Empty(Directory.GetFiles(Path.Combine(ArchiveRoot, "staging")));
        using var deleted = await SendAsync(HttpMethod.Delete, path);
        Assert.Equal(HttpStatusCode.NoContent, deleted.StatusCode);
        using var missing = await SendAsync(HttpMethod.Get, path);
        Assert.Equal(HttpStatusCode.NotFound, missing.StatusCode);
        using var again = await SendAsync(HttpMethod.Delete, path);
        Assert.Equal(HttpStatusCode.NoContent, again.StatusCode);
        using var attached = await UploadAsync(path, bytes, fileName);
        Assert.Equal(HttpStatusCode.OK, attached.StatusCode);
        await db.ProductVersions.Where(x => x.Id == version).ExecuteUpdateAsync(s =>
s.SetProperty(x => x.Status, "Published").SetProperty(x => x.PublishedAt,
DateTimeOffset.UtcNow));
        using var blockedPut = await UploadAsync(path, bytes, fileName);
        Assert.Equal(HttpStatusCode.Conflict, blockedPut.StatusCode);
        using var blockedDelete = await SendAsync(HttpMethod.Delete, path);
        Assert.Equal(HttpStatusCode.Conflict, blockedDelete.StatusCode);
    }

```

```

        using var frozenRead = await SendAsync(HttpMethod.Get, path + "/content");
Assert.Equal(bytes, await frozenRead.Content.ReadAsByteArrayAsync());
    }
    /// <summary>
    /// Verifies that invalid payloads cannot attach or replace artifacts.
    /// </summary>
    [Fact]
    public async Task InvalidPayloadsCannotAttachOrReplaceArtifacts()
    {
        await PrepareAsync(); Guid product = await ProductAsync(); Guid version = await
VersionAsync(product); string path = PathFor(product, version);
        using var valid = await UploadAsync(path, Encoding.UTF8.GetBytes(Schema));
Assert.Equal(HttpStatusCode.OK, valid.StatusCode);
        Guid original = (await ReadAsync(valid)).GetProperty("id").GetGuid();
        foreach (var (bytes, name, extra) in new[]
        {
            (Array.Empty<byte>(), "schema.json", false),
            (Encoding.UTF8.GetBytes("{\"type\":\"oops\"}"), "schema.json", false),
            (Encoding.UTF8.GetBytes(Schema), "../outside.json", false),
            (Encoding.UTF8.GetBytes(Schema), "schema.json", true)
        }) { using var invalid = await UploadAsync(path, bytes, name, extra);
Assert.Equal(HttpStatusCode.BadRequest, invalid.StatusCode); }
        using var fakeMsi = await UploadAsync(PathFor(product, version, "installer"),
[0xd0, 0xcf, 0x11, 0xe0, 0xa1, 0xb1, 0x1a, 0xe1], "setup.msi");
Assert.Equal(HttpStatusCode.BadRequest, fakeMsi.StatusCode);
        using var request = new HttpRequestMessage(HttpMethod.Put, path);
        request.Headers.Authorization = new AuthenticationHeaderValue("Bearer", token);
        request.Content = new StringContent("--bad\r\nContent-Disposition: form-data;
name=\"file\"; filename=\"schema.json\"\\r\\n\\r\\n\" + Schema);
        request.Content.Headers.ContentType =
MediaTypeHeaderValue.Parse("multipart/form-data; boundary=bad");
        using var truncated = await Client.SendAsync(request);
Assert.Equal(HttpStatusCode.BadRequest, truncated.StatusCode);

        Services.GetRequiredService<IOptions<ArtifactArchiveOptions>>().Value.SchemaMaxBytes = 10;
        using var oversized = await UploadAsync(path, Encoding.UTF8.GetBytes(Schema));
Assert.Equal(HttpStatusCode.BadRequest, oversized.StatusCode);
        await using var db = await OpenVendorAsync(vendorId);
        Assert.Equal(original, (await
db.ProductVersionArtifacts.SingleAsync()).FileArtifactId); Assert.Equal(1, await
db.FileArtifacts.CountAsync());
        Assert.Empty(Directory.GetFiles(Path.Combine(ArchiveRoot, "staging")));
    }
    /// <summary>
    /// Verifies that wrong tenant parent and missing permission cannot read or write catalog.
    /// </summary>
    [Fact]
    public async Task WrongTenantParentAndMissingPermissionCannotReadOrWriteCatalog()
    {
        await PrepareAsync(); Guid product = await ProductAsync(); Guid version = await
VersionAsync(product); Guid otherProduct = await ProductAsync("Other");
        using var wrong = await UploadAsync(PathFor(otherProduct, version),
Encoding.UTF8.GetBytes(Schema)); Assert.Equal(HttpStatusCode.NotFound, wrong.StatusCode);
        Guid otherVendor = await CreateVendorAsync("Other vendor");
        await using (var other = await OpenVendorAsync(otherVendor))
        {
            other.Products.Add(new() { Id = Guid.NewGuid(), Name = "Secret", Status =
"Active" }); await other.SaveChangesAsync();
            Guid secret = await other.Products.Select(x => x.Id).SingleAsync();
            using var hidden = await SendAsync(HttpMethod.Get,
$"/api/v1/products/{secret}"); Assert.Equal(HttpStatusCode.NotFound, hidden.StatusCode);
        }
    }

```

```

        using var list = await SendAsync(HttpMethod.Get, "/api/v1/products");
Assert.Equal(2, (await ReadAsync(list)).GetProperty("totalCount").GetInt32());
        using var platformToken = await Client.GetAsync("/api/v1/products");
Assert.Equal(HttpStatusCode.Unauthorized, platformToken.StatusCode);
        await using var db = await OpenVendorAsync(vendorId);
        await db.RolePermissions.Where(x => x.Permission.Code ==
"products.manage").ExecuteDeleteAsync();
        using var forbidden = await SendAsync(HttpMethod.Post, "/api/v1/products", new {
Name = "Denied" }); Assert.Equal(HttpStatusCode.Forbidden, forbidden.StatusCode);
        using var forbiddenUpload = await UploadAsync(PathFor(product, version),
Encoding.UTF8.GetBytes(Schema)); Assert.Equal(HttpStatusCode.Forbidden,
forbiddenUpload.StatusCode);
        using var readable = await SendAsync(HttpMethod.Get,
$/api/v1/products/{product}"); Assert.Equal(HttpStatusCode.OK, readable.StatusCode);
        await db.RolePermissions.Where(x => x.Permission.Code ==
"products.read").ExecuteDeleteAsync();
        using var forbiddenRead = await SendAsync(HttpMethod.Get, "/api/v1/products");
Assert.Equal(HttpStatusCode.Forbidden, forbiddenRead.StatusCode);
    }
    /// <summary>
    /// Verifies that audit failure rolls back replacement and corrupt blobs are not served.
    /// </summary>
    [Fact]
    public async Task AuditFailureRollsBackReplacementAndCorruptBlobsAreNotServed()
    {
        await PrepareAsync(); Guid product = await ProductAsync(); Guid version = await
VersionAsync(product); string path = PathFor(product, version);
        using var original = await UploadAsync(path, Encoding.UTF8.GetBytes(Schema));
Assert.Equal(HttpStatusCode.OK, original.StatusCode);
        Guid id = (await ReadAsync(original)).GetProperty("id").GetGuid();
        await using var db = await OpenVendorAsync(vendorId);
        await db.Database.ExecuteSqlRawAsync("CREATE TRIGGER TestArtifactAuditFailure ON
VendorAuditLogs AFTER INSERT AS THROW 51014, 'Test audit failure.', 1;");
        using var failed = await UploadAsync(path, Encoding.UTF8.GetBytes(Schema),
"replacement.json"); Assert.Equal(HttpStatusCode.InternalServerError, failed.StatusCode);
        Assert.Equal(id, (await db.ProductVersionArtifacts.SingleAsync()).FileArtifactId);
Assert.Equal(1, await db.FileArtifacts.CountAsync());
        using var preserved = await SendAsync(HttpMethod.Get, path + "/content");
Assert.Equal(Encoding.UTF8.GetBytes(Schema), await preserved.Content.ReadAsByteArrayAsync());
        Assert.Empty(Directory.GetFiles(Path.Combine(ArchiveRoot, "staging")));
        var artifact = await db.FileArtifacts.SingleAsync();
        await File.WriteAllTextAsync(Path.Combine(ArchiveRoot, "blobs",
artifact.StorageIdentifier + ".blob"), "Corrupted");
        using var corrupt = await SendAsync(HttpMethod.Get, path + "/content");
Assert.Equal(HttpStatusCode.InternalServerError, corrupt.StatusCode);
        Assert.DoesNotContain(ArchiveRoot, await corrupt.Content.ReadAsStringAsync());
    }
    /// <summary>
    /// Verifies that concurrent labels and stale tracked edits are rejected.
    /// </summary>
    [Fact]
    public async Task ConcurrentLabelsAndStaleTrackedEditsAreRejected()
    {
        await PrepareAsync(); Guid product = await ProductAsync(); Guid version = await
VersionAsync(product);
        var tasks = Enumerable.Range(0, 2).Select(_ => SendAsync(HttpMethod.Post,
$/api/v1/products/{product}/versions", new { Version = "2.0.0" })).ToArray();
        var responses = await Task.WhenAll(tasks);
        Assert.Equal(new[] { HttpStatusCode.Created, HttpStatusCode.Conflict },
responses.Select(x => x.StatusCode).Order()); foreach (var response in responses) {
response.Dispose(); }
        await using var stale = await OpenVendorAsync(vendorId);

```

```

        var oldProduct = await stale.Products.SingleAsync(); var oldVersion = await
stale.ProductVersions.SingleAsync(x => x.Id == version);
        using var updated = await SendAsync(HttpMethod.Patch,
$"/api/v1/products/{product}", new { Name = "Winner" }); Assert.Equal(HttpStatusCode.OK,
updated.StatusCode);
        oldProduct.Name = "Stale"; await
Assert.ThrowsAsync<DbUpdateConcurrencyException>(() => stale.SaveChangesAsync());
stale.Entry(oldProduct).State = EntityState.Detached;
        using var artifact = await UploadAsync(PathFor(product, version),
Encoding.UTF8.GetBytes(Schema)); Assert.Equal(HttpStatusCode.OK, artifact.StatusCode);
        oldVersion.Version = "stale"; await
Assert.ThrowsAsync<DbUpdateConcurrencyException>(() => stale.SaveChangesAsync());
    }
    /// <summary>
    /// Verifies that dto validation rejects unknown state ownership and invalid values.
    /// </summary>
    [Fact]
    public async Task DtoValidationRejectsUnknownStateOwnershipAndInvalidValues()
    {
        await PrepareAsync(); Guid product = await ProductAsync(); Guid version = await
VersionAsync(product);
        foreach (object input in new object[] { new { Name = " " }, new { Name = new
string('a', 201) }, new { Status = "Published" }, new { Id = Guid.NewGuid() } })
        { using var bad = await SendAsync(HttpMethod.Patch, $"/api/v1/products/{product}",
input); Assert.Equal(HttpStatusCode.BadRequest, bad.StatusCode); }
        foreach (object input in new object[] { new { Version = " " }, new { Version = new
string('a', 101) }, new { ProductId = product }, new { PublishedAt = DateTimeOffset.UtcNow } })
        { using var bad = await SendAsync(HttpMethod.Patch,
$"/api/v1/products/{product}/versions/{version}", input); Assert.Equal(HttpStatusCode.BadRequest,
bad.StatusCode); }
        using var create = await SendAsync(HttpMethod.Post,
$"/api/v1/products/{product}/versions", new { Version = "3", ProductId = product });
Assert.Equal(HttpStatusCode.BadRequest, create.StatusCode);
    }

    /// <summary>
    /// Verifies that upload rechecks publication and permissions after staging.
    /// </summary>
    [Fact]
    public async Task UploadRechecksPublicationAndPermissionsAfterStaging()
    {
        await PrepareAsync(); Guid product = await ProductAsync(); Guid version = await
VersionAsync(product);
        await using var db = await OpenVendorAsync(vendorId);
        var session = await db.AuthenticationSessions.SingleAsync();
        TenantActor actor = new(vendorId, session.AccountId, session.Id);
        var repository = Services.GetRequiredService<ProductArtifactRepository>();
        using MemoryStream first = new(Encoding.UTF8.GetBytes(Schema));
        await Assert.ThrowsAsync<ApiConflictException>(() => repository.PutAsync(actor,
product, version, "Schema", "schema.json", first,
        async ct => { await db.ProductVersions.Where(x => x.Id ==
version).ExecuteUpdateAsync(s => s.SetProperty(x => x.Status, "Published"), ct); }, default));
        await db.ProductVersions.Where(x => x.Id == version).ExecuteUpdateAsync(s =>
s.SetProperty(x => x.Status, "Draft"));
        using MemoryStream second = new(Encoding.UTF8.GetBytes(Schema));
        var denied = await Assert.ThrowsAsync<ApiAccessException>(() =>
repository.PutAsync(actor, product, version, "Schema", "schema.json", second,
        async ct => { await db.RolePermissions.Where(x => x.Permission.Code ==
"products.manage").ExecuteDeleteAsync(ct); }, default));
        Assert.Equal(403, denied.StatusCode);
        Assert.Empty(await db.FileArtifacts.ToListAsync());
        Assert.Empty(Directory.GetFiles(Path.Combine(ArchiveRoot, "staging")));
    }

```

```

        Assert.Empty(Directory.GetFiles(Path.Combine(ArchiveRoot, "blobs")));
    }

    /// <summary>
    /// Verifies that migration preserves existing artifact links and enforces unique typed
slots.
    /// </summary>
    [Fact]
    public async Task MigrationPreservesExistingArtifactLinksAndEnforcesUniqueTypedSlots()
    {
        await PrepareAsync(); await using var db = await OpenVendorAsync(vendorId);
        IMigrator migrator = db.GetService<IMigrator>();
        string previous = db.Database.GetMigrations().Single(x =>
x.EndsWith("_AddUserCustomerConcurrency", StringComparison.Ordinal));
        await migrator.MigrateAsync(previous);
        Guid product = Guid.NewGuid(), version = Guid.NewGuid(), file = Guid.NewGuid();
        string hash = new('a', 64);
        await db.Database.ExecuteSqlInterpolatedAsync($"INSERT INTO Products (Id, Name,
Description, Status, CreatedAt) VALUES ({product}, 'Legacy', '', 'Active',
SYSDATETIMEOFFSET())");
        await db.Database.ExecuteSqlInterpolatedAsync($"INSERT INTO ProductVersions (Id,
ProductId, Version, Status, CreatedAt) VALUES ({version}, {product}, '1', 'Draft',
SYSDATETIMEOFFSET())");
        await db.Database.ExecuteSqlInterpolatedAsync($"INSERT INTO
FileArtifacts (Id, ArtifactType, FileName, StorageIdentifier, ContentType, Sha256, SizeBytes, CreatedAt) VALUES
({file}, 'Schema', 'legacy.json', 'legacy-private-id', 'application/schema+json', {hash}, 123,
SYSDATETIMEOFFSET())");
        await db.Database.ExecuteSqlInterpolatedAsync($"INSERT INTO
ProductVersionArtifacts (ProductVersionId, FileArtifactId) VALUES ({version}, {file})");
        await migrator.MigrateAsync();
        var link = await db.ProductVersionArtifacts.SingleAsync(); Assert.Equal("Schema",
link.ArtifactType); Assert.Equal(file, link.FileArtifactId);
        FileArtifact another = new() { Id = Guid.NewGuid(), ArtifactType = "Schema",
FileName = "second.json", Sha256 = hash, StorageIdentifier = "other", ContentType =
"application/schema+json" };
        db.FileArtifacts.Add(another); await db.SaveChangesAsync();
        db.ProductVersionArtifacts.Add(new() { ProductVersionId = version, FileArtifactId
= another.Id, ArtifactType = "Schema" });
        await Assert.ThrowsAsync<DbUpdateException>(() => db.SaveChangesAsync());
        db.ChangeTracker.Clear();
        db.ProductVersionArtifacts.Add(new() { ProductVersionId = version, FileArtifactId
= another.Id, ArtifactType = "Installer" });
        await Assert.ThrowsAsync<DbUpdateException>(() => db.SaveChangesAsync());
    }
}

```

Integration/RepositoryTests.cs

```

using System.ComponentModel.DataAnnotations;
using BridgeControl.DTOS.Platform;
using BridgeControl.DTOS.Shared;
using BridgeControl.Models.Platform;
using BridgeControl.Repositories.Shared;
using BridgeControl.Tests.Infrastructure;
using Microsoft.EntityFrameworkCore;

namespace BridgeControl.Tests.Integration;

/// <summary>
/// Verifies repository behavior and regression scenarios.

```

```

/// </summary>
[Trait("Category", "Integration")]
public class RepositoryTests : SqlServerTest
{
    /// <summary>
    /// Verifies that page async keeps access filters for both count and rows.
    /// </summary>
    [Fact]
    public async Task PageAsync_KeepsAccessFiltersForBothCountAndRows()
    {
        Database.PlatformAdmins.AddRange(Account("a@allowed.test"),
Account("b@allowed.test"), Account("c@outside.test"));
        await Database.SaveChangesAsync();
        Database.ChangeTracker.Clear();
        IRepository<PlatformAdmin, PlatformAdminDto> repository = CreateRepository();

        PagedResponse<PlatformAdminDto> page = await repository.PageAsync(new() { Page =
2, PageSize = 1 }, account => account.Email);

        Assert.Equal(2, page.TotalCount);
        Assert.Equal("b@allowed.test", Assert.Single(page.Items).Email);
        Assert.Empty(Database.ChangeTracker.Entries());
        Assert.Empty((await repository.PageAsync(new() { Page = 3, PageSize = 1 }, account
=> account.Email)).Items);
    }

    /// <summary>
    /// Verifies that single or default async cannot return an excluded account.
    /// </summary>
    [Fact]
    public async Task SingleOrDefaultAsync_CannotReturnAnExcludedAccount()
    {
        Database.PlatformAdmins.Add(Account("c@outside.test"));
        await Database.SaveChangesAsync();
        Assert.Null(await CreateRepository().SingleOrDefaultAsync());
    }

    /// <summary>
    /// Verifies that page async rejects invalid bounds before querying.
    /// </summary>
    [Theory]
    [InlineData(0, 25)]
    [InlineData(1, 101)]
    [InlineData(int.MaxValue, 100)]
    public async Task PageAsync_RejectsInvalidBoundsBeforeQuerying(int page, int pageSize)
    {
        await Assert.ThrowsAsync<ValidationException>(() => CreateRepository()
        .PageAsync(new() { Page = page, PageSize = pageSize }, account =>
account.Id));
    }

    /// <summary>
    /// Verifies that page async observes cancellation.
    /// </summary>
    [Fact]
    public async Task PageAsync_ObservesCancellation()
    {
        using CancellationTokenSource cancellation = new();
        cancellation.Cancel();
        await Assert.ThrowsAnyAsync<OperationCanceledException>(() =>
CreateRepository().PageAsync(new(), account => account.Id, cancellation.Token));
    }
}

```

```

    /// <summary>
    /// Creates a mapped repository over the test database query.
    /// </summary>
    private Repository<PlatformAdmin, PlatformAdminDto> CreateRepository() => new(
        Database.PlatformAdmins.Where(account => account.Email.EndsWith("@allowed.test")),
        account => new PlatformAdminDto { Id = account.Id, Email = account.Email });

    /// <summary>
    /// Builds an account entity for repository tests.
    /// </summary>
    private static PlatformAdmin Account(string email) => new()
    {
        Id = Guid.NewGuid(), Email = email, Status = "Active", PasswordHash = "test-only",
        CreatedAt = DateTimeOffset.UtcNow
    };
}

```

Integration/SwaggerTests.cs

```

using System.Net;
using System.Text.Json;
using BridgeControl.Tests.Infrastructure;
using Jint;

namespace BridgeControl.Tests.Integration;

/// <summary>
/// Verifies swagger behavior and regression scenarios.
/// </summary>
[Trait("Category", "Integration")]
public class SwaggerTests
{
    private const string UnusedConnection =
        "Server=(localdb)\\MSSQLLocalDB;Database=BridgeControl_Test_NoConnection;Integrated Security=True";

    /// <summary>
    /// Verifies that open API describes only the existing public operations and their authentication.
    /// </summary>
    [Fact]
    public async Task OpenApi_DescribesOnlyTheExistingPublicOperationsAndTheirAuthentication()
    {
        await using ApiFactory factory = new(UnusedConnection);
        using HttpClient client = factory.CreateApiClient();
        using JsonDocument document = JsonDocument.Parse(await
            client.GetStringAsync("/openapi/v1.json"));
        JsonElement paths = document.RootElement.GetProperty("paths");

        Assert.Equal(new[] { "/api/v1/products/{productId}/deployment-settings",
            "/api/v1/products/{productId}/versions/{versionId}/deployment-settings",
            "/api/v1/products/{productId}/versions/{versionId}/deployment-settings/initialize",
            "/api/v1/auth/vendors", "/api/v1/agent/logs", "/api/v1/logs",
            "/api/v1/installations/{installationId}/logs", "/api/v1/agents/{agentId}/logs",
            "/api/v1/monitoring/overview", "/api/v1/deployments", "/api/v1/deployments/{deploymentId}",
            "/api/v1/deployments/{deploymentId}/cancel",
            "/api/v1/installations/{installationId}/deployments", "/api/v1/agent/deployments",
            "/api/v1/agent/deployments/{deploymentId}/attempts",
            "/api/v1/agent/deployments/{deploymentId}/attempts/{attemptId}",

```

```

"/api/v1/installations/{installationId}/desired-state",
"/api/v1/installations/{installationId}/reported-state",
"/api/v1/agent/installations/{installationId}/desired-state",
"/api/v1/agent/installations/{installationId}/reported-state",
"/api/v1/agent/artifacts/{artifactId}/content",
"/api/v1/installations/{installationId}/configuration/migration-preview",
"/api/v1/installations/{installationId}/configuration/migrate",
"/api/v1/products/{productId}/configuration-migrations",
"/api/v1/products/{productId}/configuration-migrations/{migrationId}",
"/api/v1/installations/{installationId}/effective-configuration",
"/api/v1/installations/{installationId}/effective-configuration/revisions",
"/api/v1/installations/{installationId}/effective-configuration/revisions/{revisionId}",
"/api/v1/installations", "/api/v1/installations/{installationId}",
"/api/v1/installations/{installationId}/deactivate",
"/api/v1/installations/{installationId}/reactivate",
"/api/v1/installations/{installationId}/license", "/api/v1/licenses",
"/api/v1/licenses/{licenseId}", "/api/v1/licenses/{licenseId}/deactivate",
"/api/v1/licenses/{licenseId}/reactivate", "/api/v1/agent-enrollments", "/api/v1/agent/register",
"/api/v1/agent/auth/token", "/api/v1/agent/heartbeat", "/api/v1/agent/installations",
"/api/v1/agents", "/api/v1/agents/{agentId}", "/api/v1/agents/{agentId}/heartbeats",
"/api/v1/auth/activate", "/api/v1/auth/login", "/api/v1/auth/logout",
"/api/v1/auth/password/change", "/api/v1/auth/password/forgot", "/api/v1/auth/password/reset",
"/api/v1/auth/refresh", "/api/v1/customers", "/api/v1/customers/{customerId}",
"/api/v1/customers/{customerId}/deactivate", "/api/v1/customers/{customerId}/reactivate",
"/api/v1/customers/{customerId}/products/{productId}/configuration",
"/api/v1/customers/{customerId}/products/{productId}/configuration/revisions",
"/api/v1/installations/{installationId}/configuration",
"/api/v1/installations/{installationId}/configuration/revisions", "/api/v1/permissions",
"/api/v1/platform/account/me", "/api/v1/platform/auth/login", "/api/v1/platform/auth/logout",
"/api/v1/platform/auth/refresh", "/api/v1/platform/vendors",
"/api/v1/platform/vendors/{vendorId}", "/api/v1/platform/vendors/{vendorId}/administrators",
"/api/v1/platform/vendors/{vendorId}/deactivate",
"/api/v1/platform/vendors/{vendorId}/reactivate", "/api/v1/products",
"/api/v1/products/{productId}", "/api/v1/products/{productId}/configuration",
"/api/v1/products/{productId}/configuration/revisions",
"/api/v1/products/{productId}/deactivate", "/api/v1/products/{productId}/reactivate",
"/api/v1/products/{productId}/versions", "/api/v1/products/{productId}/versions/{versionId}",
"/api/v1/products/{productId}/versions/{versionId}/installer",
"/api/v1/products/{productId}/versions/{versionId}/installer/content",
"/api/v1/products/{productId}/versions/{versionId}/publish",
"/api/v1/products/{productId}/versions/{versionId}/schema",
"/api/v1/products/{productId}/versions/{versionId}/schema/content", "/api/v1/roles",
"/api/v1/users", "/api/v1/users/{userId}", "/api/v1/users/{userId}/customer-access",
"/api/v1/users/{userId}/deactivate", "/api/v1/users/{userId}/installation-access",
"/api/v1/users/{userId}/reactivate", "/api/v1/users/{userId}/roles" }.Order(),
    paths.EnumerateObject().Select(property => property.Name).Order());
    JsonElement discovery =
paths.GetProperty("/api/v1/auth/vendors").GetProperty("get");
    Assert.False(discovery.TryGetProperty("security", out var discoverySecurity) &&
discoverySecurity.GetArrayLength() > 0);
    Assert.Contains("Authorization: Anonymous",
discovery.GetProperty("description").GetString());
    Assert.True(discovery.GetProperty("responses").TryGetProperty("429", out _));
    var nameFields =
document.RootElement.GetProperty("components").GetProperty("schemas").GetProperty("LoginVendorRes
ponse").GetProperty("properties");
    Assert.Equal("name", Assert.Single(nameFields.EnumerateObject()).Name);
    JsonElement create =
paths.GetProperty("/api/v1/platform/vendors").GetProperty("post");
    Assert.True(create.GetProperty("security")[0].TryGetProperty("PlatformBearer", out
_));
    Assert.True(create.GetProperty("responses").TryGetProperty("201", out _));

```

```

Assert.True(paths.GetProperty("/api/v1/agent/installations").GetProperty("get").GetProperty("security")[0].TryGetProperty("AgentBearer", out _));

Assert.True(paths.GetProperty("/api/v1/agent-enrollments").GetProperty("post").GetProperty("security")[0].TryGetProperty("TenantBearer", out _));

Assert.True(paths.GetProperty("/api/v1/agent/register").GetProperty("post").GetProperty("responses").TryGetProperty("201", out _));

Assert.True(paths.GetProperty("/api/v1/auth/password/change").GetProperty("post").GetProperty("security")[0].TryGetProperty("TenantBearer", out _));
    JsonElement refresh =
paths.GetProperty("/api/v1/platform/auth/refresh").GetProperty("post");
    Assert.Contains(refresh.GetProperty("parameters").EnumerateArray(), parameter =>
parameter.GetProperty("name").GetString() == "X-BridgeControl-Refresh"
    && parameter.GetProperty("required").GetBoolean() &&
parameter.GetProperty("schema").GetProperty("default").GetString() == "1");

Assert.True(refresh.GetProperty("security")[0].TryGetProperty("PlatformRefreshCookie", out _));
    JsonElement password =
document.RootElement.GetProperty("components").GetProperty("schemas")

.GetProperty("PlatformLoginRequest").GetProperty("properties").GetProperty("password");
    Assert.Equal("password", password.GetProperty("format").GetString());
    Assert.True(password.GetProperty("writeOnly").GetBoolean());
    Assert.DoesNotContain("passwordHash", document.RootElement.ToString(),
StringComparison.OrdinalIgnoreCase);
}

/// <summary>
/// Verifies that swagger served initialization can prepare its API definition request.
/// </summary>
[Fact]
public async Task Swagger_ServedInitializationCanPrepareItsApiDefinitionRequest()
{
    await using ApiFactory factory = new(UnusedConnection);
    using HttpClient client = factory.CreateApiClient();
    string script = await client.GetStringAsync("/swagger/index.js");
    using Engine engine = new(options =>
options.TimeoutInterval(TimeSpan.FromSeconds(5)));
    engine.SetValue("resolveUrl", (string path, string origin) => new Uri(new
Uri(origin), path).AbsoluteUri);
    // Supply the browser location and UI renderer, then execute the public script as
a page would.
    engine.Execute("""
        var config;
        var window = { location: { href: 'https://localhost/swagger/index.html' }
    };

        var URL = function (path, origin) { this.href = resolveUrl(path, origin);
    };

        var SwaggerUIBundle = function (options) { config = options; return {
initOAuth: function () {} }; };
        SwaggerUIBundle.presets = { apis: {} };
        var SwaggerUIStandalonePreset = {};
        """);
    engine.Execute(script);
    engine.Execute("""
        window.onload();
        var request = { url: config.urls[0].url, headers: {}, body: 'unchanged' };
        var result = config.requestInterceptor(request);
        """);
}

```

```

        Assert.True(engine.Evaluate("result === request").AsBoolean());
        Assert.Equal("same-origin", engine.Evaluate("result.credentials").AsString());
        Assert.Equal("unchanged", engine.Evaluate("result.body").AsString());
        Assert.Equal("/openapi/v1.json", new
Uri(engine.Evaluate("result.url").AsString()).AbsolutePath);
        Assert.False(engine.Evaluate("config.persistAuthorization").AsBoolean());
        Assert.True(engine.Evaluate("config.validatorUrl === null").AsBoolean());
    }

    /// <summary>
    /// Verifies that production does not expose development documentation.
    /// </summary>
    [Theory]
    [InlineData("/swagger/index.html")]
    [InlineData("/swagger/swagger-ui-bundle.js")]
    [InlineData("/openapi/v1.json")]
    public async Task Production_DoesNotExposeDevelopmentDocumentation(string path)
    {
        await using ApiFactory factory = new(UnusedConnection, environment: "Production");
        using HttpClient client = factory.CreateApiClient();
        using HttpResponseMessage response = await client.GetAsync(path);
        Assert.Equal(HttpStatusCode.Unauthorized, response.StatusCode);
    }
}

```

Integration/TenantAuthenticationTests.cs

```

using System.Net;
using System.Net.Http.Headers;
using System.Net.Http.Json;
using System.Text.Json;
using BridgeControl.Data;
using BridgeControl.Models.Vendor;
using BridgeControl.Security;
using BridgeControl.Services.Shared;
using BridgeControl.Services.Vendor;
using BridgeControl.Tests.Infrastructure;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;

namespace BridgeControl.Tests.Integration;

/// <summary>
/// Verifies tenant authentication behavior and regression scenarios.
/// </summary>
[Trait("Category", "Integration")]
public class TenantAuthenticationTests : VendorTest
{
    private const string Email = "invited@example.test";
    private const string Password = " 12345678 ";

    /// <summary>
    /// Creates the accounts and resources required by the test scenario.
    /// </summary>
    private async Task<Guid> PrepareAsync(string name = "Test vendor")
    {
        await SignInAsync();
        Guid id = await CreateVendorAsync(name);
        string token = await InviteAsync(id);
    }
}

```

```

        using var activated = await Client.PostAsJsonAsync("/api/v1/auth/activate", new {
Token = token, Password });
        Assert.Equal(HttpStatusCode.NoContent, activated.StatusCode);
        return id;
    }
    /// <summary>
    /// Authenticates the test user and returns the resulting session data.
    /// </summary>
    private async Task<JsonElement Body, string Cookie> LoginAsync(string name = "Test
vendor", string email = Email, string password = Password)
    {
        using var response = await Client.PostAsJsonAsync("/api/v1/auth/login", new {
VendorName = name, Email = email, Password = password });
        Assert.Equal(HttpStatusCode.OK, response.StatusCode);
        string cookie = response.Headers.GetValues("Set-Cookie").Single(x =>
x.StartsWith(TenantRefreshCookie.Name));
        Assert.Contains("secure", cookie); Assert.Contains("httponly", cookie);
Assert.Contains("samesite=strict", cookie);
        return (await ReadAsync(response), cookie.Split(';')[0]);
    }
    /// <summary>
    /// Sends a JSON POST request through the test client.
    /// </summary>
    private async Task<HttpResponseMessage> PostAsync(string route, object body, string? token
= null, string? cookie = null)
    {
        using var request = new HttpRequestMessage(HttpMethod.Post, route) { Content =
JsonContent.Create(body) };
        if (token is not null) { request.Headers.Authorization = new
AuthenticationHeaderValue("Bearer", token); }
        if (cookie is not null) { request.Headers.Add("Cookie", cookie); }
        request.Headers.Add(TenantRefreshCookie.RequestHeader, "1"); }
        return await Client.SendAsync(request);
    }
    /// <summary>
    /// Reads the access token from the test authentication response.
    /// </summary>
    private static string Token(JsonElement body) =>
body.GetProperty("accessToken").GetString()!;

    /// <summary>
    /// Verifies that login returns contract and refresh reuse revokes the whole session.
    /// </summary>
    [Fact]
    public async Task Login_ReturnsContractAndRefreshReuseRevokesTheWholeSession()
    {
        Guid id = await PrepareAsync();
        var login = await LoginAsync(" TEST VENDOR ", " INVITED@EXAMPLE.TEST ");
        Assert.Equal(id, login.Body.GetProperty("vendorId").GetGuid());
        Assert.Equal(new[] { "accessToken", "accessTokenExpiresAt", "customerId", "email",
"permissions", "roles", "userId", "vendorId" }, login.Body.EnumerateObject().Select(x =>
x.Name).Order());
        Assert.Contains(login.Body.GetProperty("permissions").EnumerateArray(), x =>
x.GetString() == "users.manage");
        using var refreshed = await PostAsync("/api/v1/auth/refresh", new { }, cookie:
login.Cookie);
        Assert.Equal(HttpStatusCode.OK, refreshed.StatusCode);
        string latest = Token(await ReadAsync(refreshed));
        using var reused = await PostAsync("/api/v1/auth/refresh", new { }, cookie:
login.Cookie);
        Assert.Equal(HttpStatusCode.Unauthorized, reused.StatusCode);
        using var denied = await PostAsync("/api/v1/auth/logout", new { }, latest);

```

```

        Assert.Equal(HttpStatusCode.Unauthorized, denied.StatusCode);
        await using var db = await OpenVendorAsync(id);
        Assert.Equal(1, await db.VendorAuditLogs.CountAsync(x => x.Action ==
"RefreshCredentialReused"));
    }

    /// <summary>
    /// Verifies that login unknown vendor user or wrong password uses generic failure.
    /// </summary>
    [Theory]
    [InlineData("missing", Email, Password)]
    [InlineData("Test vendor", "missing@example.test", Password)]
    [InlineData("Test vendor", Email, "12345678")]
    public async Task Login_UnknownVendorUserOrWrongPasswordUsesGenericFailure(string vendor,
string email, string password)
    {
        await PrepareAsync();
        using var response = await Client.PostAsJsonAsync("/api/v1/auth/login", new {
VendorName = vendor, Email = email, Password = password });
        Assert.Equal(HttpStatusCode.Unauthorized, response.StatusCode);
        Assert.Equal("Authentication failed.", (await
ReadAsync(response)).GetProperty("title").GetString());
        Assert.DoesNotContain(email, await response.Content.ReadAsStringAsync());
    }

    /// <summary>
    /// Verifies that logout and expiry deny access and refresh.
    /// </summary>
    [Fact]
    public async Task LogoutAndExpiry_DenyAccessAndRefresh()
    {
        Guid id = await PrepareAsync();
        var login = await LoginAsync();
        using var logout = await PostAsync("/api/v1/auth/logout", new { },
Token(login.Body));
        Assert.Equal(HttpStatusCode.NoContent, logout.StatusCode);
        using var refresh = await PostAsync("/api/v1/auth/refresh", new { }, cookie:
login.Cookie);
        Assert.Equal(HttpStatusCode.Unauthorized, refresh.StatusCode);
        var next = await LoginAsync();
        await using var db = await OpenVendorAsync(id);
        await db.AuthenticationSessions.Where(x => x.RevokedAt ==
null).ExecuteUpdateAsync(s => s
            .SetProperty(x => x.CreatedAt,
DateTimeOffset.UtcNow.AddDays(-2)).SetProperty(x => x.ExpiresAt,
DateTimeOffset.UtcNow.AddDays(-1)));
        using var expired = await PostAsync("/api/v1/auth/logout", new { },
Token(next.Body));
        Assert.Equal(HttpStatusCode.Unauthorized, expired.StatusCode);
        using var expiredRefresh = await PostAsync("/api/v1/auth/refresh", new { },
cookie: next.Cookie);
        Assert.Equal(HttpStatusCode.Unauthorized, expiredRefresh.StatusCode);
    }

    /// <summary>
    /// Verifies that vendor disable reactivate cannot revive old session without an
intervening request.
    /// </summary>
    [Fact]
    public async Task
VendorDisableReactivate_CannotReviveOldSessionWithoutAnInterveningRequest()
    {

```

```

        Guid id = await PrepareAsync();
        var login = await LoginAsync();
        using var disabled = await PostAsync($"/api/v1/platform/vendors/{id}/deactivate",
new { });
        Assert.Equal(HttpStatusCode.NoContent, disabled.StatusCode);
        using var enabled = await PostAsync($"/api/v1/platform/vendors/{id}/reactivate",
new { });
        Assert.Equal(HttpStatusCode.NoContent, enabled.StatusCode);
        using var old = await PostAsync("/api/v1/auth/logout", new { },
Token(login.Body));
        Assert.Equal(HttpStatusCode.Unauthorized, old.StatusCode);
        using var oldRefresh = await PostAsync("/api/v1/auth/refresh", new { }, cookie:
login.Cookie);
        Assert.Equal(HttpStatusCode.Unauthorized, oldRefresh.StatusCode);
        await LoginAsync();
    }

    /// <summary>
    /// Verifies that password change requires current password and revokes every session.
    /// </summary>
    [Fact]
    public async Task PasswordChange_RequiresCurrentPasswordAndRevokesEverySession()
    {
        Guid id = await PrepareAsync();
        var first = await LoginAsync(); var second = await LoginAsync();
        using var wrong = await PostAsync("/api/v1/auth/password/change", new {
CurrentPassword = "wrong", NewPassword = "new-password" }, Token(first.Body));
        Assert.Equal(HttpStatusCode.Unauthorized, wrong.StatusCode);
        using var changed = await PostAsync("/api/v1/auth/password/change", new {
CurrentPassword = Password, NewPassword = " new-password " }, Token(first.Body));
        Assert.Equal(HttpStatusCode.NoContent, changed.StatusCode);
        using var old = await PostAsync("/api/v1/auth/logout", new { },
Token(second.Body));
        Assert.Equal(HttpStatusCode.Unauthorized, old.StatusCode);
        await LoginAsync(password: " new-password ");
        await using var db = await OpenVendorAsync(id);
        Assert.Equal(1, await db.VendorAuditLogs.CountAsync(x => x.Action ==
"PasswordChanged"));
    }

    /// <summary>
    /// Reads the password-reset credential delivered to the local test outbox.
    /// </summary>
    private async Task<string> ResetTokenAsync(Guid id)
    {
        using var response = await PostAsync("/api/v1/auth/password/forgot", new {
VendorName = "Test vendor", Email });
        Assert.Equal(HttpStatusCode.NoContent, response.StatusCode);
        await using var db = await OpenVendorAsync(id);
        var reset = await db.PasswordResets.AsNoTracking().SingleAsync(x => x.UsedAt ==
null && x.RevokedAt == null);
        Assert.Null(reset.ProtectedToken);
        Assert.InRange((reset.ExpiresAt - reset.CreatedAt).TotalSeconds, 1799, 1801);
        string email = await
File.ReadAllTextAsync(Path.Combine(Factory.InvitationDirectory, $"{reset.Id:N}.eml"));
        return email.Split('\n').Single(x => x.StartsWith("Reset token: ")) [13..].Trim();
    }

    /// <summary>
    /// Verifies that recovery is generic purpose bound single use and revokes old sessions.
    /// </summary>
    [Fact]

```

```

public async Task Recovery_IsGenericPurposeBoundSingleUseAndRevokesOldSessions()
{
    Guid id = await PrepareAsync(); var login = await LoginAsync();
    foreach (string email in new[] { Email, "missing@example.test" })
    {
        using var response = await PostAsync("/api/v1/auth/password/forgot", new {
VendorName = "Test vendor", Email = email });
        Assert.Equal(HttpStatusCode.NoContent, response.StatusCode);
        Assert.Empty(await response.Content.ReadAsStringAsync());
    }
    string token = await ResetTokenAsync(id);
    using var wrongPurpose = await PostAsync("/api/v1/auth/activate", new { Token =
token, Password = "new-password" });
    Assert.Equal(HttpStatusCode.Unauthorized, wrongPurpose.StatusCode);
    using var reset = await PostAsync("/api/v1/auth/password/reset", new { Token =
token, Password = " new-password " });
    Assert.Equal(HttpStatusCode.NoContent, reset.StatusCode);
    using var reused = await PostAsync("/api/v1/auth/password/reset", new { Token =
token, Password = "other-password" });
    Assert.Equal(HttpStatusCode.Unauthorized, reused.StatusCode);
    using var old = await PostAsync("/api/v1/auth/logout", new { },
Token(login.Body));
    Assert.Equal(HttpStatusCode.Unauthorized, old.StatusCode);
    await LoginAsync(password: " new-password ");
}

/// <summary>
/// Verifies that recovery rejects unavailable tokens.
/// </summary>
[Theory]
[InlineData("expired")]
[InlineData("revoked")]
public async Task Recovery_RejectsUnavailableTokens(string reason)
{
    Guid id = await PrepareAsync(); string token = await ResetTokenAsync(id);
    await using var db = await OpenVendorAsync(id);
    if (reason == "expired") { await db.PasswordResets.ExecuteUpdateAsync(s =>
s.SetProperty(x => x.CreatedAt, DateTimeOffset.UtcNow.AddDays(-2)).SetProperty(x => x.ExpiresAt,
DateTimeOffset.UtcNow.AddDays(-1))); }
    else { await db.PasswordResets.ExecuteUpdateAsync(s => s.SetProperty(x =>
x.RevokedAt, DateTimeOffset.UtcNow)); }
    using var response = await PostAsync("/api/v1/auth/password/reset", new { Token =
token, Password = "new-password" });
    Assert.Equal(HttpStatusCode.Unauthorized, response.StatusCode);
    await LoginAsync();
}

/// <summary>
/// Verifies that password reset audit failure rolls back password credential and
sessions.
/// </summary>
[Fact]
public async Task PasswordReset_AuditFailureRollsBackPasswordCredentialAndSessions()
{
    Guid id = await PrepareAsync(); var login = await LoginAsync(); string token =
await ResetTokenAsync(id);
    await using var db = await OpenVendorAsync(id);
    await db.Database.ExecuteSqlRawAsync("CREATE TRIGGER TestResetAuditFailure ON
VendorAuditLogs AFTER INSERT AS THROW 51009, 'Test audit failure.', 1;");
    using var reset = await PostAsync("/api/v1/auth/password/reset", new { Token =
token, Password = "new-password" });
    Assert.Equal(HttpStatusCode.InternalServerError, reset.StatusCode);
}

```

```

        Assert.Null((await db.PasswordResets.SingleAsync()).UsedAt);
        Assert.Null((await db.AuthenticationSessions.SingleAsync()).RevokedAt);
        var user = await db.UserAccounts.SingleAsync();
        Assert.NotEqual(Microsoft.AspNetCore.Identity.PasswordVerificationResult.Failed,
Services.GetRequiredService<PasswordService<UserAccount>>().Verify(user, user.PasswordHash,
Password));
    }

    /// <summary>
    /// Verifies that concurrent refresh revokes the winning replacement after reuse.
    /// </summary>
    [Fact]
    public async Task ConcurrentRefresh_RevokesTheWinningReplacementAfterReuse()
    {
        await PrepareAsync(); var login = await LoginAsync();
        var responses = await Task.WhenAll(PostAsync("/api/v1/auth/refresh", new { },
cookie: login.Cookie), PostAsync("/api/v1/auth/refresh", new { }, cookie: login.Cookie));
        try
        {
            Assert.Single(responses, x => x.StatusCode == HttpStatusCode.OK);
            Assert.Single(responses, x => x.StatusCode == HttpStatusCode.Unauthorized);
            using var invalid = await PostAsync("/api/v1/auth/logout", new { },
Token(await ReadAsync(responses.Single(x => x.IsSuccessStatusCode))));
            Assert.Equal(HttpStatusCode.Unauthorized, invalid.StatusCode);
        }
        finally { foreach (var response in responses) { response.Dispose(); } }
    }

    /// <summary>
    /// Verifies that identities and vendors cannot cross authentication boundaries.
    /// </summary>
    [Fact]
    public async Task IdentitiesAndVendors_CannotCrossAuthenticationBoundaries()
    {
        Guid first = await PrepareAsync(); Guid second = await CreateVendorAsync("Second
vendor");
        string invitation = await InviteAsync(second);
        using var activated = await PostAsync("/api/v1/auth/activate", new { Token =
invitation, Password });
        var a = await LoginAsync(); var b = await LoginAsync("Second vendor");
        using var platformOnTenant = await PostAsync("/api/v1/auth/logout", new { });
        Assert.Equal(HttpStatusCode.Unauthorized, platformOnTenant.StatusCode);
        using var tenantOnPlatform = await
PostAsync($"/api/v1/platform/vendors/{first}/deactivate", new { }, Token(a.Body));
        Assert.Equal(HttpStatusCode.Unauthorized, tenantOnPlatform.StatusCode);
        using var crossUser = await
PostAsync($"/api/v1/users/{b.Body.GetProperty("userId").GetGuid()}/deactivate", new { },
Token(a.Body));
        Assert.Equal(HttpStatusCode.NotFound, crossUser.StatusCode);
        string tamperedCookie = a.Cookie.Replace(first.ToString("N"),
second.ToString("N"));
        using var crossRefresh = await PostAsync("/api/v1/auth/refresh", new { }, cookie:
tamperedCookie);
        Assert.Equal(HttpStatusCode.Unauthorized, crossRefresh.StatusCode);
    }

    /// <summary>
    /// Adds a user account to the test vendor database.
    /// </summary>
    private async Task<UserAccount> AddUserAsync(VendorDbContext db, string email, string
roleName, Guid? customerId = null)
    {

```

```

        var role = await db.Roles.SingleAsync(x => x.Name == roleName);
        var user = new UserAccount { Id = Guid.NewGuid(), CustomerId = customerId, Email =
email, Status = "Active", CreatedAt = DateTimeOffset.UtcNow };
        user.PasswordHash =
Services.GetRequiredService<PasswordService<UserAccount>>().Hash(user, Password);
        user.UserRoles.Add(new UserRole { Id = Guid.NewGuid(), UserId = user.Id, RoleId =
role.Id });
        db.UserAccounts.Add(user); await db.SaveChangesAsync(); return user;
    }

    /// <summary>
    /// Verifies that user disable reactivate and role changes are enforced on existing
tokens.
    /// </summary>
    [Fact]
    public async Task UserDisableReactivateAndRoleChanges_AreEnforcedOnExistingTokens()
    {
        Guid id = await PrepareAsync(); var admin = await LoginAsync();
        await using var db = await OpenVendorAsync(id);
        var target = await AddUserAsync(db, "second@example.test", "VendorAdministrator");
        var login = await LoginAsync(email: target.Email);
        var disabled = await
Task.WhenAll(PostAsync($"api/v1/users/{target.Id}/deactivate", new { }, Token(admin.Body)),
PostAsync($"api/v1/users/{target.Id}/deactivate", new { },
Token(admin.Body)));
        foreach (var response in disabled) { Assert.Equal(HttpStatusCode.NoContent,
response.StatusCode); response.Dispose(); }
        using var enabled = await PostAsync($"api/v1/users/{target.Id}/reactivate", new {
}, Token(admin.Body));
        Assert.Equal(HttpStatusCode.NoContent, enabled.StatusCode);
        using var old = await PostAsync("api/v1/auth/logout", new { },
Token(login.Body));
        Assert.Equal(HttpStatusCode.Unauthorized, old.StatusCode);
        using var oldRefresh = await PostAsync("api/v1/auth/refresh", new { }, cookie:
login.Cookie);
        Assert.Equal(HttpStatusCode.Unauthorized, oldRefresh.StatusCode);
        var next = await LoginAsync(email: target.Email);
        await db.UserRoles.Where(x => x.UserId == target.Id).ExecuteDeleteAsync();
        using var noPermission = await
PostAsync($"api/v1/users/{admin.Body.GetProperty("userId").GetGuid()}/deactivate", new { },
Token(next.Body));
        Assert.Equal(HttpStatusCode.Forbidden, noPermission.StatusCode);
    }

    /// <summary>
    /// Verifies that customer and technician scopes filter counts rows and current
assignments.
    /// </summary>
    [Fact]
    public async Task CustomerAndTechnicianScopes_FilterCountsRowsAndCurrentAssignments()
    {
        Guid id = await PrepareAsync();
        await using var db = await OpenVendorAsync(id);
        var first = new Customer { Id = Guid.NewGuid(), Name = "First", Status = "Active",
CreatedAt = DateTimeOffset.UtcNow };
        var second = new Customer { Id = Guid.NewGuid(), Name = "Second", Status =
"Active", CreatedAt = DateTimeOffset.UtcNow };
        var product = new Product { Id = Guid.NewGuid(), Name = "Product", Status =
"Active", CreatedAt = DateTimeOffset.UtcNow };
        db.AddRange(first, second, product); await db.SaveChangesAsync();
        var customerAdmin = await AddUserAsync(db, "customer-admin@example.test",
"CustomerAdministrator", first.Id);

```

```

        var customerUser = await AddUserAsync(db, "customer-user@example.test",
"CustomerUser", first.Id);
        var outsider = await AddUserAsync(db, "outside@example.test", "CustomerUser",
second.Id);
        var technician = await AddUserAsync(db, "technician@example.test",
"VendorTechnician");
        var one = new Installation { Name = "One", CustomerId = first.Id, ProductId =
product.Id };
        var two = new Installation { Name = "Two", CustomerId = first.Id, ProductId =
product.Id };
        var three = new Installation { Name = "Three", CustomerId = second.Id, ProductId =
product.Id };
        db.Installations.AddRange(one, two, three); await db.SaveChangesAsync();
        db.CustomerUserInstallationAccessGrants.Add(new CustomerUserInstallationAccess
        {
            Id = Guid.NewGuid(),
            CustomerId = first.Id,
            UserId = customerUser.Id,
            InstallationId = one.Id,
            GrantedByUserId = customerAdmin.Id,
            GrantedAt = DateTimeOffset.UtcNow
        });
        db.TechnicianCustomerAccessGrants.Add(new TechnicianCustomerAccess
        {
            Id = Guid.NewGuid(),
            CustomerId = first.Id,
            UserId = technician.Id,
            GrantedByUserId = customerAdmin.Id,
            GrantedAt = DateTimeOffset.UtcNow
        });
        await db.SaveChangesAsync();
        var userAccess = (await TenantAccess.LoadAsync(db, customerUser.Id, default))!;
        Assert.Equal(1, await userAccess.Installations(db).CountAsync());
        Assert.Equal(one.Id, (await userAccess.Installations(db).SingleAsync()).Id);
        Assert.False(await userAccess.Installations(db,
"installations.manage").AnyAsync());
        var techAccess = (await TenantAccess.LoadAsync(db, technician.Id, default))!;
        Assert.Equal(first.Id, (await techAccess.Customers(db).SingleAsync()).Id);
        Assert.Equal(2, await techAccess.Installations(db).CountAsync());
        await db.TechnicianCustomerAccessGrants.ExecuteDeleteAsync();
        Assert.Empty(await techAccess.Installations(db).ToListAsync());
        var adminLogin = await LoginAsync(email: customerAdmin.Email);
        Assert.Equal(first.Id, adminLogin.Body.GetProperty("customerId").GetGuid());
        using var denied = await PostAsync($"/api/v1/users/{outsider.Id}/deactivate", new
{ }, Token(adminLogin.Body));
        Assert.Equal(HttpStatusCode.NotFound, denied.StatusCode);
        using var allowed = await PostAsync($"/api/v1/users/{customerUser.Id}/deactivate",
new { }, Token(adminLogin.Body));
        Assert.Equal(HttpStatusCode.NoContent, allowed.StatusCode);
    }

    /// <summary>
    /// Verifies that interrupted vendor disable stays closed until cleanup succeeds.
    /// </summary>
    [Fact]
    public async Task InterruptedVendorDisable_StaysClosedUntilCleanupSucceeds()
    {
        Guid id = await PrepareAsync(); var login = await LoginAsync();
        await using var db = await OpenVendorAsync(id);
        await db.Database.ExecuteSqlRawAsync("CREATE TRIGGER TestDisableAuditFailure ON
VendorAuditLogs AFTER INSERT AS THROW 51010, 'Test audit failure.', 1;");

```

```

        using var failed = await PostAsync($"/api/v1/platform/vendors/{id}/deactivate",
new { });
        Assert.Equal(HttpStatusCode.InternalServerError, failed.StatusCode);
        Assert.Equal("Disabled", (await
Database.SoftwareVendors.AsNoTracking().SingleAsync()).Status);
        Assert.Null((await
db.AuthenticationSessions.AsNoTracking().SingleAsync()).RevokedAt);
        using var old = await PostAsync("/api/v1/auth/logout", new { },
Token(login.Body));
        Assert.Equal(HttpStatusCode.Unauthorized, old.StatusCode);
        using var failedEnable = await
PostAsync($"/api/v1/platform/vendors/{id}/reactivate", new { });
        Assert.Equal(HttpStatusCode.InternalServerError, failedEnable.StatusCode);
        Assert.Equal("Disabled", (await
Database.SoftwareVendors.AsNoTracking().SingleAsync()).Status);
        await db.Database.ExecuteSqlRawAsync("DROP TRIGGER TestDisableAuditFailure;");
        using var enabled = await PostAsync($"/api/v1/platform/vendors/{id}/reactivate",
new { });
        Assert.Equal(HttpStatusCode.NoContent, enabled.StatusCode);
        using var refresh = await PostAsync("/api/v1/auth/refresh", new { }, cookie:
login.Cookie);
        Assert.Equal(HttpStatusCode.Unauthorized, refresh.StatusCode);
        Assert.NotNull((await
db.AuthenticationSessions.AsNoTracking().SingleAsync()).RevokedAt);
    }

    /// <summary>
    /// Verifies that recovery retries delivery without leaking account existence or
duplicating tokens.
    /// </summary>
    [Fact]
    public async Task
Recovery_RetrievesDeliveryWithoutLeakingAccountExistenceOrDuplicatingTokens()
    {
        Guid id = await PrepareAsync();
        var blocked = Path.Combine(Factory.PrivateRoot, "blocked-reset");
        await File.WriteAllTextAsync(blocked, "Force directory failure");
        var options =
Services.GetRequiredService<Microsoft.Extensions.Options.IOptions<BridgeControl.Options.VendorPro
visioningOptions>>().Value;
        string original = options.InvitationDirectory; options.InvitationDirectory =
blocked;
        using var failedDelivery = await PostAsync("/api/v1/auth/password/forgot", new {
VendorName = "Test vendor", Email });
        Assert.Equal(HttpStatusCode.NoContent, failedDelivery.StatusCode);
        await using var db = await OpenVendorAsync(id);
        var pending = await db.PasswordResets.AsNoTracking().SingleAsync();
        Assert.NotNull(pending.ProtectedToken); Assert.Null(pending.DeliveredAt);
        options.InvitationDirectory = original;
        string token = await ResetTokenAsync(id);
        Assert.Equal(pending.Id, (await
db.PasswordResets.AsNoTracking().SingleAsync()).Id);
        Assert.DoesNotContain(token, pending.ProtectedToken!);
        Assert.Equal(1, await db.VendorAuditLogs.CountAsync(x => x.Action ==
"PasswordResetDeliveryFailed"));
    }

    /// <summary>
    /// Verifies that concurrent password resets consume the credential only once.
    /// </summary>
    [Fact]
    public async Task ConcurrentPasswordResets_ConsumeTheCredentialOnlyOnce()

```

```

    {
        Guid id = await PrepareAsync(); string token = await ResetTokenAsync(id);
        var responses = await Task.WhenAll(PostAsync("/api/v1/auth/password/reset", new {
Token = token, Password = "first-password" })),
        PostAsync("/api/v1/auth/password/reset", new { Token = token, Password =
"other-password" }));
        try
        {
            Assert.Single(responses, x => x.StatusCode == HttpStatusCode.NoContent);
            Assert.Single(responses, x => x.StatusCode == HttpStatusCode.Unauthorized);
            await using var db = await OpenVendorAsync(id);
            Assert.Equal(1, await db.VendorAuditLogs.CountAsync(x => x.Action ==
"PasswordReset"));
        }
        finally { foreach (var response in responses) { response.Dispose(); } }
    }

    /// <summary>
    /// Verifies that recovery rate limits unknown credentials.
    /// </summary>
    [Theory]
    [InlineData("/api/v1/auth/password/forgot", 5)]
    [InlineData("/api/v1/auth/password/reset", 10)]
    public async Task Recovery_RateLimitsUnknownCredentials(string route, int limit)
    {
        await PrepareAsync();
        object body = route.EndsWith("forgot") ? new { VendorName = "missing", Email } :
new { Token = "invalid", Password };
        for (int i = 0; i < limit; i++)
        { using var response = await PostAsync(route, body);
Assert.NotEqual(HttpStatusCode.TooManyRequests, response.StatusCode); }
        using var limited = await PostAsync(route, body);
        Assert.Equal(HttpStatusCode.TooManyRequests, limited.StatusCode);
        Assert.NotNull(limited.Headers.RetryAfter);
    }

    /// <summary>
    /// Verifies that bearer rejects wrong identity and expired access tokens.
    /// </summary>
    [Theory]
    [InlineData("agent", false)]
    [InlineData("vendor_user", true)]
    public async Task Bearer_RejectsWrongIdentityAndExpiredAccessTokens(string identityType,
bool expired)
    {
        await PrepareAsync(); var login = await LoginAsync();
        var original = new
System.IdentityModel.Tokens.Jwt.JwtSecurityTokenHandler().ReadJwtToken(Token(login.Body));
        var claims = original.Claims.Where(x => x.Type is "sub" or "sid" or
"vendor_id").ToList();
        claims.Add(new System.Security.Claims.Claim("identity_type", identityType));
        var token = new System.IdentityModel.Tokens.Jwt.JwtSecurityToken("BridgeControl",
"BridgeControl.Tenant", claims,
            DateTime.UtcNow.AddMinutes(-5), expired ? DateTime.UtcNow.AddMinutes(-1) :
DateTime.UtcNow.AddMinutes(5),
            new Microsoft.IdentityModel.Tokens.SigningCredentials(new
Microsoft.IdentityModel.Tokens.SymmetricSecurityKey(Convert.FromBase64String(Factory.TenantSignin
gKey))), Microsoft.IdentityModel.Tokens.SecurityAlgorithms.HmacSha256));
        token.Header["typ"] = "at+jwt";
        using var response = await PostAsync("/api/v1/auth/logout", new { }, new
System.IdentityModel.Tokens.Jwt.JwtSecurityTokenHandler().WriteToken(token));
        Assert.Equal(HttpStatusCode.Unauthorized, response.StatusCode);
    }

```

```

    }

    /// <summary>
    /// Verifies that user disable audit failure rolls back status and revocation.
    /// </summary>
    [Fact]
    public async Task UserDisable_AuditFailureRollsBackStatusAndRevocation()
    {
        Guid id = await PrepareAsync(); var admin = await LoginAsync();
        await using var db = await OpenVendorAsync(id);
        var target = await AddUserAsync(db, "target@example.test", "VendorTechnician");
        await LoginAsync(email: target.Email);
        await db.Database.ExecuteSqlRawAsync("CREATE TRIGGER TestUserAuditFailure ON
VendorAuditLogs AFTER INSERT AS THROW 51011, 'Test audit failure.', 1;");
        using var response = await PostAsync($"/api/v1/users/{target.Id}/deactivate", new
{ }, Token(admin.Body));
        Assert.Equal(HttpStatusCode.InternalServerError, response.StatusCode);
        Assert.Equal("Active", (await db.UserAccounts.AsNoTracking().SingleAsync(x => x.Id
== target.Id)).Status);
        Assert.Null((await db.AuthenticationSessions.AsNoTracking().SingleAsync(x =>
x.AccountId == target.Id)).RevokedAt);
    }

    /// <summary>
    /// Verifies that login rate limit and refresh header apply without exposing credentials
in audit.
    /// </summary>
    [Fact]
    public async Task LoginRateLimitAndRefreshHeader_ApplyWithoutExposingCredentialsInAudit()
    {
        Guid id = await PrepareAsync(); var login = await LoginAsync();
        using var missingHeader = await PostAsync("/api/v1/auth/refresh", new { });
        Assert.Equal(HttpStatusCode.BadRequest, missingHeader.StatusCode);
        for (int i = 0; i < 19; i++)
        {
            using var attempt = await PostAsync("/api/v1/auth/login", new { VendorName
= "Test vendor", Email, Password = "incorrect-secret" });
            Assert.Equal(HttpStatusCode.Unauthorized, attempt.StatusCode);
        }
        using var limited = await PostAsync("/API/v1/AUTH/login/", new { VendorName =
"Test vendor", Email, Password });
        Assert.Equal(HttpStatusCode.TooManyRequests, limited.StatusCode);
        await using var db = await OpenVendorAsync(id);
        string audit = JsonSerializer.Serialize(await
db.VendorAuditLogs.AsNoTracking().ToListAsync());
        Assert.DoesNotContain("incorrect-secret", audit); Assert.DoesNotContain(Email,
audit);

        Assert.DoesNotContain(Token(login.Body), audit);
        Assert.All(await db.VendorAuditLogs.ToListAsync(), entry => {
Assert.NotEmpty(entry.EntityType); Assert.NotEmpty(entry.Outcome); Assert.Equal("{",
entry.Context); });
    }
}

```

Integration/TenantResolutionTests.cs

```

using System.Security.Claims;
using BridgeControl.Services;
using BridgeControl.Tests.Infrastructure;
using Microsoft.EntityFrameworkCore;

```

```

using Microsoft.Extensions.DependencyInjection;

namespace BridgeControl.Tests.Integration;

/// <summary>
/// Verifies tenant resolution behavior and regression scenarios.
/// </summary>
[Trait("Category", "Integration")]
public class TenantResolutionTests : VendorTest
{
    /// <summary>
    /// Verifies that resolver uses current names and keeps two vendors in separate contexts.
    /// </summary>
    [Fact]
    public async Task Resolver_UsesCurrentNamesAndKeepsTwoVendorsInSeparateContexts()
    {
        await SignInAsync();
        Guid first = await CreateVendorAsync("First vendor");
        Guid second = await CreateVendorAsync("Second vendor");
        await InviteAsync(first, "first@example.test");
        await InviteAsync(second, "second@example.test");
        ITenantResolver resolver = Services.GetRequiredService<ITenantResolver>();
        await using TenantContext? alpha = await resolver.ResolveForLoginAsync(" FIRST
VENDOR ");
        await using TenantContext? beta = await
resolver.ResolveAuthenticatedAsync(Identity(second));
        Assert.NotNull(alpha);
        Assert.NotNull(beta);
        Assert.Equal(first, alpha.VendorId);
        Assert.Equal(second, beta.VendorId);
        Assert.Equal("first@example.test", (await
alpha.Database.UserAccounts.SingleAsync()).Email);
        Assert.Equal("second@example.test", (await
beta.Database.UserAccounts.SingleAsync()).Email);
        Assert.NotEqual(alpha.ArchivePath, beta.ArchivePath);
        await Database.SoftwareVendors.Where(vendor => vendor.Id ==
first).ExecuteUpdateAsync(setters => setters.SetProperty(vendor => vendor.Name, "Renamed
vendor"));
        Assert.Null(await resolver.ResolveForLoginAsync("First vendor"));
        await using TenantContext? renamed = await resolver.ResolveForLoginAsync("RENAMED
VENDOR");
        Assert.Equal(first, renamed!.VendorId);
        Assert.Null(await resolver.ResolveForLoginAsync("missing"));
        Assert.Null(await resolver.ResolveAuthenticatedAsync(Identity(first,
"platform_admin")));
        Assert.Null(await resolver.ResolveAuthenticatedAsync(new ClaimsPrincipal(new
ClaimsIdentity())));
        await Database.SoftwareVendors.Where(vendor => vendor.Id ==
first).ExecuteUpdateAsync(setters => setters.SetProperty(vendor => vendor.Status, "Disabled"));
        Assert.Null(await resolver.ResolveAuthenticatedAsync(Identity(first)));
        Assert.Null(await resolver.ResolveForLoginAsync("Renamed vendor"));
    }

    /// <summary>
    /// Verifies that resolver rejects tampered infrastructure references.
    /// </summary>
    [Fact]
    public async Task Resolver_RejectsTamperedInfrastructureReferences()
    {
        await SignInAsync();
        Guid id = await CreateVendorAsync();
        await InviteAsync(id);
    }
}

```

```

        ITenantResolver resolver = Services.GetRequiredService<ITenantResolver>();
        await Database VendorFileArchives.ExecuteUpdateAsync(setters =>
setters.SetProperty(archive => archive.StorageIdentifier, "../other-vendor"));
        Assert.Null(await resolver.ResolveForLoginAsync("Test vendor"));
    }

    /// <summary>
    /// Builds a claims principal with the requested tenant identity.
    /// </summary>
    private static ClaimsPrincipal Identity(Guid vendorId, string type = "vendor_user") =>
new(new ClaimsIdentity(
    [new Claim("identity_type", type), new Claim("vendor_id", vendorId.ToString())],
    "validated-test-identity"));
}

```

Integration/UserAdministrationTests.cs

```

using System.Net;
using System.Net.Http.Headers;
using System.Net.Http.Json;
using System.Text.Json;
using BridgeControl.Data;
using BridgeControl.Models.Vendor;
using BridgeControl.Services.Vendor;
using BridgeControl.Tests.Infrastructure;
using Microsoft.EntityFrameworkCore;

namespace BridgeControl.Tests.Integration;

/// <summary>
/// Verifies user administration behavior and regression scenarios.
/// </summary>
[Trait("Category", "Integration")]
public class UserAdministrationTests : VendorTest
{
    private Guid vendorId;
    private string adminToken = string.Empty;
    private Dictionary<string, Guid> roles = [];
    private const string Password = "sprint-six-password";

    /// <summary>
    /// Creates the accounts and resources required by the test scenario.
    /// </summary>
    private async Task PrepareAsync()
    {
        await SignInAsync(); vendorId = await CreateVendorAsync();
        string token = await InviteAsync(vendorId);
        using var response = await Client.PostAsJsonAsync("/api/v1/auth/activate", new {
Token = token, Password });
        Assert.Equal(HttpStatusCode.NoContent, response.StatusCode);
        adminToken = await LoginAsync("invited@example.test");
        using var roleResponse = await SendAsync(HttpMethod.Get, "/api/v1/roles");
        Assert.Equal(HttpStatusCode.OK, roleResponse.StatusCode);
        roles = (await
ReadAsync(roleResponse)).GetProperty("items").EnumerateArray().ToDictionary(x =>
x.GetProperty("name").GetString()!, x => x.GetProperty("id").GetGuid());
    }

    /// <summary>
    /// Authenticates the test user and returns the resulting session data.
    /// </summary>

```

```

private async Task<string> LoginAsync(string email)
{
    using var response = await Client.PostAsJsonAsync("/api/v1/auth/login", new {
VendorName = "Test vendor", Email = email, Password });
    Assert.Equal(HttpStatusCode.OK, response.StatusCode);
    return (await ReadAsync(response)).GetProperty("accessToken").GetString()!;
}
/// <summary>
/// Sends the test HTTP request with the supplied method, credentials, and payload.
/// </summary>
private async Task<HttpResponseMessage> SendAsync(HttpMethod method, string path, object?
body = null, string? token = null)
{
    using var request = new HttpRequestMessage(method, path);
    request.Headers.Authorization = new AuthenticationHeaderValue("Bearer", token ??
adminToken);
    if (body is not null) { request.Content = JsonContent.Create(body); }
    return await Client.SendAsync(request);
}
/// <summary>
/// Creates a customer for the test scenario.
/// </summary>
private async Task<Guid> CustomerAsync(string name)
{
    using var response = await SendAsync(HttpMethod.Post, "/api/v1/customers", new {
Name = name });
    Assert.Equal(HttpStatusCode.Created, response.StatusCode);
    JsonElement customer = await ReadAsync(response);
    Assert.Equal(new[] { "createdAt", "id", "name", "status" },
customer.EnumerateObject().Select(x => x.Name).Order());
    return customer.GetProperty("id").GetGuid();
}
/// <summary>
/// Creates a user for the test scenario.
/// </summary>
private async Task<Guid> UserAsync(string email, string role, Guid? customer = null,
string? actor = null, bool activate = true)
{
    using var response = await SendAsync(HttpMethod.Post, "/api/v1/users", new { Email
= email, CustomerId = customer, RoleIds = new[] { roles[role] } }, actor);
    Assert.Equal(HttpStatusCode.Created, response.StatusCode);
    JsonElement user = await ReadAsync(response);
    Assert.Equal(new[] { "createdAt", "customerId", "email", "id", "roles", "status"
}, user.EnumerateObject().Select(x => x.Name).Order());
    Guid id = user.GetProperty("id").GetGuid();
    if (activate)
    {
        string credential = await InvitationAsync(id);
        using var activated = await Client.PostAsJsonAsync("/api/v1/auth/activate",
new { Token = credential, Password });
        Assert.Equal(HttpStatusCode.NoContent, activated.StatusCode);
    }
    return id;
}
/// <summary>
/// Reads the account invitation created for the test user.
/// </summary>
private async Task<string> InvitationAsync(Guid userId)
{
    await using var db = await OpenVendorAsync(vendorId);
    Guid id = await db.AccountActivations.Where(x => x.UserId == userId && x.UsedAt ==
null && x.RevokedAt == null).Select(x => x.Id).SingleAsync();
}

```

```

        string message = await
File.ReadAllTextAsync(Path.Combine(Factory.InvitationDirectory, $"{id:N}.eml"));
        return message.Split('\n').Single(x => x.StartsWith("Activation token:
")) [18..].Trim();
    }

    /// <summary>
    /// Verifies that users invite normalize retry and edit without changing ownership.
    /// </summary>
    [Fact]
    public async Task Users_InviteNormalizeRetryAndEditWithoutChangingOwnership()
    {
        await PrepareAsync(); Guid customer = await CustomerAsync("First");
        Guid user = await UserAsync(" PERSON@EXAMPLE.TEST ", "CustomerUser", customer,
activate: false);
        string original = await InvitationAsync(user);
        Assert.Equal(user, await UserAsync("person@example.test", "CustomerUser",
customer, activate: false));
        Assert.Equal(original, await InvitationAsync(user));
        using var edited = await SendAsync(HttpMethod.Patch, $"/api/v1/users/{user}", new
{ Email = " new@example.test " });
        Assert.Equal(HttpStatusCode.OK, edited.StatusCode);
        Assert.Equal(customer, (await
ReadAsync(edited)).GetProperty("customerId").GetGuid());
        string replacement = await InvitationAsync(user);
        Assert.NotEqual(original, replacement);
        using var rejected = await Client.PostAsJsonAsync("/api/v1/auth/activate", new {
Token = original, Password });
        Assert.Equal(HttpStatusCode.Unauthorized, rejected.StatusCode);
        using var active = await Client.PostAsJsonAsync("/api/v1/auth/activate", new {
Token = replacement, Password });
        Assert.Equal(HttpStatusCode.NoContent, active.StatusCode);
        await LoginAsync("new@example.test");
        using var omitted = await SendAsync(HttpMethod.Patch, $"/api/v1/users/{user}", new
{ });
        Assert.Equal("new@example.test", (await
ReadAsync(omitted)).GetProperty("email").GetString());
        using var duplicate = await SendAsync(HttpMethod.Post, "/api/v1/users", new {
Email = "NEW@EXAMPLE.TEST", CustomerId = customer, RoleIds = new[] { roles["CustomerUser"] } });
        Assert.Equal(HttpStatusCode.Conflict, duplicate.StatusCode);
    }

    /// <summary>
    /// Verifies that role boundaries apply to collections and details.
    /// </summary>
    [Theory]
    [InlineData("VendorAdministrator", 200, 200, 200)]
    [InlineData("VendorTechnician", 403, 200, 403)]
    [InlineData("CustomerAdministrator", 200, 200, 200)]
    [InlineData("CustomerUser", 403, 200, 403)]
    public async Task RoleBoundaries_ApplyToCollectionsAndDetails(string role, int
usersStatus, int customersStatus, int catalogStatus)
    {
        await PrepareAsync(); Guid first = await CustomerAsync("First"); Guid second =
await CustomerAsync("Second");
        Guid? customer = role.StartsWith("Customer") ? first : null;
        Guid user = await UserAsync("role@example.test", role, customer);
        if (role == "VendorTechnician")
        {
            using var assigned = await SendAsync(HttpMethod.Put,
$/api/v1/users/{user}/customer-access", new { CustomerIds = new[] { first } });
            Assert.Equal(HttpStatusCode.NoContent, assigned.StatusCode);

```

```

    }
    string token = await LoginAsync("role@example.test");
    using var users = await SendAsync(HttpMethod.Get, "/api/v1/users?pageSize=1",
token: token);
    Assert.Equal(usersStatus, (int)users.StatusCode);
    using var customers = await SendAsync(HttpMethod.Get,
"/api/v1/customers?pageSize=1", token: token);
    Assert.Equal(customersStatus, (int)customers.StatusCode);
    Assert.Equal(role == "VendorAdministrator" ? 2 : 1, (await
ReadAsync(customers)).GetProperty("totalCount").GetInt32());
    using var direct = await SendAsync(HttpMethod.Get, $"/api/v1/customers/{second}",
token: token);
    Assert.Equal(role == "VendorAdministrator" ? HttpStatusCode.OK :
HttpStatusCode.NotFound, direct.StatusCode);
    using var catalog = await SendAsync(HttpMethod.Get, "/api/v1/roles", token:
token);
    Assert.Equal(catalogStatus, (int)catalog.StatusCode);
    if (catalogStatus == 200)
    { Assert.Equal(role == "VendorAdministrator" ? 4 : 2, (await
ReadAsync(catalog)).GetProperty("totalCount").GetInt32()); }
    }

    /// <summary>
    /// Verifies that customer admin cannot switch customers or grant vendor roles.
    /// </summary>
    [Fact]
    public async Task CustomerAdmin_CannotSwitchCustomersOrGrantVendorRoles()
    {
        await PrepareAsync(); Guid first = await CustomerAsync("First"); Guid second =
await CustomerAsync("Second");
        await UserAsync("admin@example.test", "CustomerAdministrator", first);
        string token = await LoginAsync("admin@example.test");
        Guid user = await UserAsync("own@example.test", "CustomerUser", first, token);
        Guid foreign = await UserAsync("foreign@example.test", "CustomerUser", second);
        using var otherUser = await SendAsync(HttpMethod.Get, $"/api/v1/users/{foreign}",
token: token);
        Assert.Equal(HttpStatusCode.NotFound, otherUser.StatusCode);
        using var outside = await SendAsync(HttpMethod.Post, "/api/v1/users", new { Email
= "new@example.test", CustomerId = second, RoleIds = new[] { roles["CustomerUser"] } }, token);
        Assert.Equal(HttpStatusCode.NotFound, outside.StatusCode);
        using var elevated = await SendAsync(HttpMethod.Put,
$"/api/v1/users/{user}/roles", new { RoleIds = new[] { roles["VendorAdministrator"] } }, token);
        Assert.Equal(HttpStatusCode.Conflict, elevated.StatusCode);
        using var filtered = await SendAsync(HttpMethod.Get,
$"/api/v1/users?customerId={second}", token: token);
        Assert.Equal(0, (await ReadAsync(filtered)).GetProperty("totalCount").GetInt32());
        using var permissions = await SendAsync(HttpMethod.Get, "/api/v1/permissions",
token: token);
        Assert.DoesNotContain((await
ReadAsync(permissions)).GetProperty("items").EnumerateArray(), x =>
x.GetProperty("code").GetString() == "customers.manage");
    }

    /// <summary>
    /// Verifies that user patch rejects non editable and invalid fields.
    /// </summary>
    [Theory]
    [InlineData("{\"customerId\":null}")]
    [InlineData("{\"password\":\"admin-chosen-password\"}")]
    [InlineData("{\"status\":\"Active\"}")]
    [InlineData("{\"email\":null}")]
    public async Task UserPatch_RejectsNonEditableAndInvalidFields(string json)

```

```

    {
        await PrepareAsync();
        using var request = new HttpRequestMessage(HttpMethod.Patch,
$/api/v1/users/{Guid.NewGuid()})
        { Content = new StringContent(json, System.Text.Encoding.UTF8, "application/json")
    };

        request.Headers.Authorization = new("Bearer", adminToken);
        using var response = await Client.SendAsync(request);
        Assert.Equal(HttpStatusCode.BadRequest, response.StatusCode);
    }

    /// <summary>
    /// Verifies that role replacement is idempotent and reduces existing session permissions.
    /// </summary>
    [Fact]
    public async Task RoleReplacement_IsIdempotentAndReducesExistingSessionPermissions()
    {
        await PrepareAsync(); Guid user = await UserAsync("admin@example.test",
"VendorAdministrator"); string token = await LoginAsync("admin@example.test");
        using var changed = await SendAsync(HttpMethod.Put, $"/api/v1/users/{user}/roles",
new { RoleIds = new[] { roles["VendorTechnician"] } });
        Assert.Equal(HttpStatusCode.NoContent, changed.StatusCode);
        using var repeated = await SendAsync(HttpMethod.Put,
$/api/v1/users/{user}/roles", new { RoleIds = new[] { roles["VendorTechnician"] } });
        Assert.Equal(HttpStatusCode.NoContent, repeated.StatusCode);
        using var reduced = await SendAsync(HttpMethod.Get, "/api/v1/users", token:
token);

        Assert.Equal(HttpStatusCode.Forbidden, reduced.StatusCode);
        await using var db = await OpenVendorAsync(vendorId);
        Assert.Equal(1, await db.UserRoles.CountAsync(x => x.UserId == user));
        Assert.Equal(1, await db.VendorAuditLogs.CountAsync(x => x.EntityId == user &&
x.Action == "UserRoleGranted"));
    }

    /// <summary>
    /// Verifies that technician replacement is atomic and removal immediately conceals
    customers.
    /// </summary>
    [Fact]
    public async Task TechnicianReplacement_IsAtomicAndRemovalImmediatelyConcealsCustomers()
    {
        await PrepareAsync(); Guid first = await CustomerAsync("First"); Guid second =
await CustomerAsync("Second");
        Guid user = await UserAsync("tech@example.test", "VendorTechnician"); string token
= await LoginAsync("tech@example.test");
        object body = new { CustomerIds = new[] { first, second } };
        var responses = await Task.WhenAll(SendAsync(HttpMethod.Put,
$/api/v1/users/{user}/customer-access", body), SendAsync(HttpMethod.Put,
$/api/v1/users/{user}/customer-access", body));
        foreach (var response in responses) { Assert.Equal(HttpStatusCode.NoContent,
response.StatusCode); response.Dispose(); }
        using var removed = await SendAsync(HttpMethod.Put,
$/api/v1/users/{user}/customer-access", new { CustomerIds = new[] { second } });
        Assert.Equal(HttpStatusCode.NoContent, removed.StatusCode);
        using var denied = await SendAsync(HttpMethod.Get, $"/api/v1/customers/{first}",
token: token);
        Assert.Equal(HttpStatusCode.NotFound, denied.StatusCode);
        using var list = await SendAsync(HttpMethod.Get, "/api/v1/customers?pageSize=1",
token: token);
        Assert.Equal(1, (await ReadAsync(list)).GetProperty("totalCount").GetInt32());
        using var selfGrant = await SendAsync(HttpMethod.Put,
$/api/v1/users/{user}/customer-access", body, token);

```

```

        Assert.Equal(HttpStatusCode.Forbidden, selfGrant.StatusCode);
        await using var db = await OpenVendorAsync(vendorId);
        Assert.Equal(2, await db.VendorAuditLogs.CountAsync(x => x.EntityId == user &&
x.Action == "CustomerAccessGranted"));
    }

    /// <summary>
    /// Verifies that customer lifecycle revokes sessions and keeps records on reactivation.
    /// </summary>
    [Fact]
    public async Task CustomerLifecycle_RevokesSessionsAndKeepsRecordsOnReactivation()
    {
        await PrepareAsync(); Guid customer = await CustomerAsync("First"); Guid user =
await UserAsync("customer@example.test", "CustomerAdministrator", customer);
        string token = await LoginAsync("customer@example.test");
        using var disabled = await SendAsync(HttpMethod.Post,
$/api/v1/customers/{customer}/deactivate");
        Assert.Equal(HttpStatusCode.NoContent, disabled.StatusCode);
        using var enabled = await SendAsync(HttpMethod.Post,
$/api/v1/customers/{customer}/reactivate");
        Assert.Equal(HttpStatusCode.NoContent, enabled.StatusCode);
        using var stale = await SendAsync(HttpMethod.Get, "/api/v1/users", token: token);
        Assert.Equal(HttpStatusCode.Unauthorized, stale.StatusCode);
        await LoginAsync("customer@example.test");
        await using var db = await OpenVendorAsync(vendorId);
        Assert.Equal("Active", (await db.UserAccounts.SingleAsync(x => x.Id ==
user)).Status);
        Assert.Equal(1, await db.Customers.CountAsync());
    }

    /// <summary>
    /// Verifies that customer patch changes only supplied name and preserves lifecycle.
    /// </summary>
    [Fact]
    public async Task CustomerPatch_ChangesOnlySuppliedNameAndPreservesLifecycle()
    {
        await PrepareAsync(); Guid customer = await CustomerAsync(" Original ");
        using var updated = await SendAsync(HttpMethod.Patch,
$/api/v1/customers/{customer}", new { Name = " Changed " });
        Assert.Equal(HttpStatusCode.OK, updated.StatusCode);
        Assert.Equal("Changed", (await
ReadAsync(updated)).GetProperty("name").GetString());
        using var omitted = await SendAsync(HttpMethod.Patch,
$/api/v1/customers/{customer}", new { });
        Assert.Equal("Changed", (await
ReadAsync(omitted)).GetProperty("name").GetString());
        using var status = await SendAsync(HttpMethod.Patch,
$/api/v1/customers/{customer}", new { Status = "Disabled" });
        Assert.Equal(HttpStatusCode.BadRequest, status.StatusCode);
    }

    /// <summary>
    /// Verifies that installation assignments enforce customer ownership and removal on
existing sessions.
    /// </summary>
    [Fact]
    public async Task
InstallationAssignments_EnforceCustomerOwnershipAndRemovalOnExistingSessions()
    {
        await PrepareAsync(); Guid first = await CustomerAsync("First"); Guid second =
await CustomerAsync("Second");
        await UserAsync("admin@example.test", "CustomerAdministrator", first);

```

```

        string admin = await LoginAsync("admin@example.test");
        Guid user = await UserAsync("ordinary@example.test", "CustomerUser", first,
admin);

        string ordinary = await LoginAsync("ordinary@example.test");
        await using var db = await OpenVendorAsync(vendorId);
        var product = new Product { Id = Guid.NewGuid(), Name = "Product", Status =
"Active", CreatedAt = DateTimeOffset.UtcNow };
        var one = new Installation { Name = "One", CustomerId = first, ProductId =
product.Id };
        var other = new Installation { Name = "Other", CustomerId = second, ProductId =
product.Id };
        db.AddRange(product, one, other); await db.SaveChangesAsync();
        using var foreign = await SendAsync(HttpMethod.Put,
$"/api/v1/users/{user}/installation-access", new { InstallationIds = new[] { one.Id, other.Id }
}, admin);

        Assert.Equal(HttpStatusCode.NotFound, foreign.StatusCode);
        Assert.Empty(await db.CustomerUserInstallationAccessGrants.ToListAsync());
        for (int i = 0; i < 2; i++)
        {
            using var granted = await SendAsync(HttpMethod.Put,
$"/api/v1/users/{user}/installation-access", new { InstallationIds = new[] { one.Id } }, admin);
            Assert.Equal(HttpStatusCode.NoContent, granted.StatusCode);
        }
        using var listed = await SendAsync(HttpMethod.Get,
$"/api/v1/users/{user}/installation-access", token: admin);
        Assert.Equal(one.Id, (await
ReadAsync(listed)).GetProperty("installationIds")[0].GetGuid());
        var access = (await TenantAccess.LoadAsync(db, user, default))!;
        Assert.Equal(one.Id, (await access.Installations(db).SingleAsync()).Id);
        using var self = await SendAsync(HttpMethod.Put,
$"/api/v1/users/{user}/installation-access", new { InstallationIds = new[] { other.Id } },
ordinary);
        Assert.Equal(HttpStatusCode.Forbidden, self.StatusCode);
        using var removed = await SendAsync(HttpMethod.Put,
$"/api/v1/users/{user}/installation-access", new { InstallationIds = Array.Empty<Guid>() },
admin);
        Assert.Equal(HttpStatusCode.NoContent, removed.StatusCode);
        Assert.Empty(await access.Installations(db).ToListAsync());
        Assert.Equal(1, await db.VendorAuditLogs.CountAsync(x => x.EntityId == user &&
x.Action == "InstallationAccessGranted"));
    }

    /// <summary>
    /// Verifies that cross vendor resources are concealed from direct collection and
assignment requests.
    /// </summary>
    [Fact]
    public async Task
CrossVendorResources_AreConcealedFromDirectCollectionAndAssignmentRequests()
    {
        await PrepareAsync(); Guid technician = await UserAsync("tech@example.test",
"VendorTechnician");
        Guid second = await CreateVendorAsync("Second vendor");
        await using var other = await OpenVendorAsync(second);
        var foreign = new Customer { Id = Guid.NewGuid(), Name = "Private", Status =
"Active", CreatedAt = DateTimeOffset.UtcNow };
        other.Customers.Add(foreign); await other.SaveChangesAsync();
        using var direct = await SendAsync(HttpMethod.Get,
$"/api/v1/customers/{foreign.Id}");
        Assert.Equal(HttpStatusCode.NotFound, direct.StatusCode);
        using var list = await SendAsync(HttpMethod.Get,
"/api/v1/customers?name=Private");

```

```

        Assert.Equal(0, (await ReadAsync(list)).GetProperty("totalCount").GetInt32());
        using var grant = await SendAsync(HttpMethod.Put,
$/api/v1/users/{technician}/customer-access", new { CustomerIds = new[] { foreign.Id } });
        Assert.Equal(HttpStatusCode.NotFound, grant.StatusCode);
        using var edited = await SendAsync(HttpMethod.Patch,
$/api/v1/customers/{foreign.Id}", new { Name = "Changed" });
        Assert.Equal(HttpStatusCode.NotFound, edited.StatusCode);
        Assert.Empty(await other.VendorAuditLogs.ToListAsync());
    }

    /// <summary>
    /// Verifies that assignments reject duplicate or missing identifiers without partial
changes.
    /// </summary>
    [Theory]
    [InlineData(true)]
    [InlineData(false)]
    public async Task
Assignments_RejectDuplicateOrMissingIdentifiersWithoutPartialChanges(bool duplicates)
    {
        await PrepareAsync(); Guid customer = await CustomerAsync("First"); Guid user =
await UserAsync("tech@example.test", "VendorTechnician");
        Guid[] ids = duplicates ? [customer, customer] : [Guid.Empty];
        using var invalid = await SendAsync(HttpMethod.Put,
$/api/v1/users/{user}/customer-access", new { CustomerIds = ids });
        Assert.Equal(HttpStatusCode.BadRequest, invalid.StatusCode);
        await using var db = await OpenVendorAsync(vendorId);
        Assert.Empty(await db.TechnicianCustomerAccessGrants.ToListAsync());
    }

    /// <summary>
    /// Verifies that removing role clears delegated grants and does not restore them when
role returns.
    /// </summary>
    [Fact]
    public async Task RemovingRole_ClearsDelegatedGrantsAndDoesNotRestoreThemWhenRoleReturns()
    {
        await PrepareAsync(); Guid customer = await CustomerAsync("First"); Guid user =
await UserAsync("tech@example.test", "VendorTechnician");
        using var grant = await SendAsync(HttpMethod.Put,
$/api/v1/users/{user}/customer-access", new { CustomerIds = new[] { customer } });
        Assert.Equal(HttpStatusCode.NoContent, grant.StatusCode);
        using var removed = await SendAsync(HttpMethod.Put, $"/api/v1/users/{user}/roles",
new { RoleIds = Array.Empty<Guid>() });
        Assert.Equal(HttpStatusCode.NoContent, removed.StatusCode);
        using var restored = await SendAsync(HttpMethod.Put,
$/api/v1/users/{user}/roles", new { RoleIds = new[] { roles["VendorTechnician"] } });
        Assert.Equal(HttpStatusCode.NoContent, restored.StatusCode);
        using var current = await SendAsync(HttpMethod.Get,
$/api/v1/users/{user}/customer-access");
        Assert.Empty((await
ReadAsync(current)).GetProperty("customerIds").EnumerateArray());
    }

    /// <summary>
    /// Verifies that assignment audit failure rolls back replacement and keeps the previous
grant.
    /// </summary>
    [Fact]
    public async Task AssignmentAuditFailure_RollsBackReplacementAndKeepsThePreviousGrant()
    {

```

```

        await PrepareAsync(); Guid first = await CustomerAsync("First"); Guid second =
await CustomerAsync("Second");
        Guid user = await UserAsync("tech@example.test", "VendorTechnician");
        using var granted = await SendAsync(HttpMethod.Put,
$/api/v1/users/{user}/customer-access", new { CustomerIds = new[] { first } });
        Assert.Equal(HttpStatusCode.NoContent, granted.StatusCode);
        await using var db = await OpenVendorAsync(vendorId);
        await db.Database.ExecuteSqlRawAsync("CREATE TRIGGER TestAssignmentAuditFailure ON
VendorAuditLogs AFTER INSERT AS THROW 51012, 'Test audit failure.', 1;");
        using var failed = await SendAsync(HttpMethod.Put,
$/api/v1/users/{user}/customer-access", new { CustomerIds = new[] { second } });
        Assert.Equal(HttpStatusCode.InternalServerError, failed.StatusCode);
        Assert.Equal(first, (await
db.TechnicianCustomerAccessGrants.AsNoTracking().SingleAsync()).CustomerId);
    }

    /// <summary>
    /// Verifies that customer disable audit failure rolls back status and session revocation.
    /// </summary>
    [Fact]
    public async Task CustomerDisableAuditFailure_RollsBackStatusAndSessionRevocation()
    {
        await PrepareAsync(); Guid customer = await CustomerAsync("First"); Guid user =
await UserAsync("person@example.test", "CustomerUser", customer);
        await LoginAsync("person@example.test");
        await using var db = await OpenVendorAsync(vendorId);
        await db.Database.ExecuteSqlRawAsync("CREATE TRIGGER TestCustomerAuditFailure ON
VendorAuditLogs AFTER INSERT AS THROW 51013, 'Test audit failure.', 1;");
        using var failed = await SendAsync(HttpMethod.Post,
$/api/v1/customers/{customer}/deactivate");
        Assert.Equal(HttpStatusCode.InternalServerError, failed.StatusCode);
        Assert.Equal("Active", (await db.Customers.SingleAsync()).Status);
        Assert.Null((await db.AuthenticationSessions.SingleAsync(x => x.AccountId ==
user)).RevokedAt);
    }

    /// <summary>
    /// Verifies that invitation delivery failure retries same user and credential.
    /// </summary>
    [Fact]
    public async Task InvitationDeliveryFailure_RetrievesSameUserAndCredential()
    {
        await PrepareAsync();
        string blocked = Path.Combine(Factory.PrivateRoot, "blocked-invitation"); await
File.WriteAllTextAsync(blocked, "Force directory failure");
        var options =
Microsoft.Extensions.DependencyInjection.ServiceProviderServiceExtensions.GetRequiredService<Micr
osoft.Extensions.Options.IOptions<BridgeControl.Options.VendorProvisioningOptions>>(Services).Val
ue;

        string original = options.InvitationDirectory; options.InvitationDirectory =
blocked;

        using var failed = await SendAsync(HttpMethod.Post, "/api/v1/users", new { Email =
"tech@example.test", CustomerId = (Guid?)null, RoleIds = new[] { roles["VendorTechnician"] } });
        Assert.Equal(HttpStatusCode.InternalServerError, failed.StatusCode);
        await using var db = await OpenVendorAsync(vendorId);
        var user = await db.UserAccounts.AsNoTracking().SingleAsync(x => x.Email ==
"tech@example.test");
        var pending = await db.AccountActivations.AsNoTracking().SingleAsync(x => x.UserId
== user.Id);
        Assert.NotNull(pending.ProtectedToken); Assert.Null(pending.DeliveredAt);
        options.InvitationDirectory = original;

```

```

        Assert.Equal(user.Id, await UserAsync("tech@example.test", "VendorTechnician",
activate: false));
        var delivered = await db.AccountActivations.AsNoTracking().SingleAsync(x =>
x.UserId == user.Id);
        Assert.Equal(pending.Id, delivered.Id); Assert.Null(delivered.ProtectedToken);
Assert.NotNull(delivered.DeliveredAt);
    }

    /// <summary>
    /// Verifies that database concurrency rejects stale user and customer edits.
    /// </summary>
    [Fact]
    public async Task DatabaseConcurrency_RejectsStaleUserAndCustomerEdits()
    {
        await PrepareAsync(); Guid customer = await CustomerAsync("First");
        await using var stale = await OpenVendorAsync(vendorId);
        var originalCustomer = await stale.Customers.SingleAsync();
        var originalUser = await stale.UserAccounts.SingleAsync();
        using var updated = await SendAsync(HttpMethod.Patch,
$/api/v1/customers/{customer}", new { Name = "Winner" });
        Assert.Equal(HttpStatusCode.OK, updated.StatusCode);
        originalCustomer.Name = "Stale";
        await Assert.ThrowsAsync<DbUpdateConcurrencyException>(() =>
stale.SaveChangesAsync());
        stale.Entry(originalCustomer).State = EntityState.Detached;
        // Public login updates LastLoginAt, advancing the account's concurrency token.
        await LoginAsync("invited@example.test");
        originalUser.Email = "stale@example.test";
        await Assert.ThrowsAsync<DbUpdateConcurrencyException>(() =>
stale.SaveChangesAsync());
    }

    /// <summary>
    /// Verifies that disabled customer blocks activation and operational installations while
preserving history.
    /// </summary>
    [Fact]
    public async Task
DisabledCustomer_BlocksActivationAndOperationalInstallationsWhilePreservingHistory()
    {
        await PrepareAsync(); Guid customer = await CustomerAsync("First"); Guid user =
await UserAsync("person@example.test", "CustomerUser", customer, activate: false);
        string credential = await InvitationAsync(user);
        await using var db = await OpenVendorAsync(vendorId);
        var product = new Product { Id = Guid.NewGuid(), Name = "Product", Status =
"Active", CreatedAt = DateTimeOffset.UtcNow };
        db.AddRange(product, new Installation { Name = "History", CustomerId = customer,
ProductId = product.Id }); await db.SaveChangesAsync();
        Guid adminId = await db.UserAccounts.Where(x => x.Email ==
"invited@example.test").Select(x => x.Id).SingleAsync();
        var admin = (await TenantAccess.LoadAsync(db, adminId, default))!;
        using var disabled = await SendAsync(HttpMethod.Post,
$/api/v1/customers/{customer}/deactivate");
        Assert.Equal(HttpStatusCode.NoContent, disabled.StatusCode);
        using var activation = await Client.PostAsJsonAsync("/api/v1/auth/activate", new {
Token = credential, Password });
        Assert.Equal(HttpStatusCode.Unauthorized, activation.StatusCode);
        Assert.Equal(1, await admin.Installations(db).CountAsync());
        Assert.Equal(0, await admin.OperationalInstallations(db).CountAsync());
    }
}

```

Integration/VendorAccessStorageTests.cs

```
using BridgeControl.Data;
using BridgeControl.Models.Shared;
using BridgeControl.Models.Vendor;
using BridgeControl.Services.Vendor;
using BridgeControl.Tests.Infrastructure;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;

namespace BridgeControl.Tests.Integration;

/// <summary>
/// Verifies vendor access storage behavior and regression scenarios.
/// </summary>
[Trait("Category", "Integration")]
public class VendorAccessStorageTests : VendorTest
{
    /// <summary>
    /// Verifies that seed async is repeatable and keeps predefined grants within their role.
    /// </summary>
    [Fact]
    public async Task SeedAsync_IsRepeatableAndKeepsPredefinedGrantsWithinTheirRole()
    {
        await SignInAsync();
        Guid id = await CreateVendorAsync();
        await using VendorDbContext database = await OpenVendorAsync(id);
        Guid[] roleIds = await database.Roles.OrderBy(role => role.Id).Select(role =>
role.Id).ToArrayAsync();
        int grants = await database.RolePermissions.CountAsync();
        await Services.GetRequiredService<VendorRoleSeeder>().SeedAsync(database);
        Assert.Equal(roleIds, await database.Roles.OrderBy(role => role.Id).Select(role =>
role.Id).ToArrayAsync());
        Assert.Equal(grants, await database.RolePermissions.CountAsync());
        Assert.False(await database.RolePermissions.AnyAsync(grant => grant.Role.Name ==
"CustomerUser" && grant.Permission.Code.EndsWith(".manage")));
        Assert.False(await database.RolePermissions.AnyAsync(grant =>
grant.Permission.Code.StartsWith("platform.")));
    }

    /// <summary>
    /// Verifies that save changes enforces customer ownership without requiring customers for
    vendor users.
    /// </summary>
    [Fact]
    public async Task
SaveChanges_EnforcesCustomerOwnershipWithoutRequiringCustomersForVendorUsers()
    {
        await SignInAsync();
        Guid id = await CreateVendorAsync();
        await using VendorDbContext database = await OpenVendorAsync(id);
        Customer first = new() { Id = Guid.NewGuid(), Name = "First", Status = "Active",
CreatedAt = DateTimeOffset.UtcNow };
        Customer second = new() { Id = Guid.NewGuid(), Name = "Second", Status = "Active",
CreatedAt = DateTimeOffset.UtcNow };
        Product product = new() { Id = Guid.NewGuid(), Name = "Product", Status =
"Active", CreatedAt = DateTimeOffset.UtcNow };
        UserAccount user = new() { Id = Guid.NewGuid(), Customer = first, Email =
"customer@example.test", Status = "Active", PasswordHash = "test-only", CreatedAt =
DateTimeOffset.UtcNow };
    }
}
```

```

        UserAccount administrator = new() { Id = Guid.NewGuid(), Email =
"vendor@example.test", Status = "Active", PasswordHash = "test-only", CreatedAt =
DateTimeOffset.UtcNow };
        Installation installation = new() { Name = "Installation", Customer = first,
Product = product };
        database.AddRange(first, second, product, user, administrator, installation);
        await database.SaveChangesAsync();
        CustomerUserInstallationAccess access = new()
        {
            Id = Guid.NewGuid(),
            CustomerId = first.Id,
            UserId = user.Id,
            InstallationId = installation.Id,
            GrantedByUserId = administrator.Id,
            GrantedAt = DateTimeOffset.UtcNow
        };
        database.CustomerUserInstallationAccessGrants.Add(access);
        await database.SaveChangesAsync();
        Assert.Null(administrator.CustomerId);
        await Assert.ThrowsAsync<Microsoft.Data.SqlClient.SqlException>(() =>
database.UserAccounts.Where(account => account.Id == user.Id)
        .ExecuteUpdateAsync(setters => setters.SetProperty(account =>
account.CustomerId, second.Id)));
        await Assert.ThrowsAsync<Microsoft.Data.SqlClient.SqlException>(() =>
database.Installations.Where(value => value.Id == installation.Id)
        .ExecuteUpdateAsync(setters => setters.SetProperty(value =>
value.CustomerId, second.Id)));
        database.ChangeTracker.Clear();
        database.CustomerUserInstallationAccessGrants.Add(new
CustomerUserInstallationAccess
        {
            Id = Guid.NewGuid(),
            CustomerId = first.Id,
            UserId = administrator.Id,
            InstallationId = installation.Id,
            GrantedByUserId = administrator.Id,
            GrantedAt = DateTimeOffset.UtcNow
        });
        await Assert.ThrowsAsync<DbUpdateException>(() => database.SaveChangesAsync());
        database.ChangeTracker.Clear();
        database.AuthenticationSessions.Add(new AuthenticationSession
        {
            Id = Guid.NewGuid(),
            AccountId = Guid.NewGuid(),
            CreatedAt = DateTimeOffset.UtcNow,
            ExpiresAt = DateTimeOffset.UtcNow.AddHours(1)
        });
        await Assert.ThrowsAsync<DbUpdateException>(() => database.SaveChangesAsync());
    }
}

```

Integration/VendorDiscoveryTests.cs

```

using System.Net;
using System.Text.Json;
using BridgeControl.Models.Platform;
using BridgeControl.Tests.Infrastructure;

namespace BridgeControl.Tests.Integration;

```

```

/// <summary>
/// Verifies vendor discovery behavior and regression scenarios.
/// </summary>
[Trait("Category", "Integration")]
public sealed class VendorDiscoveryTests : SqlServerTest
{
    /// <summary>
    /// Verifies that anonymous discovery returns only active names in ordered pages.
    /// </summary>
    [Fact]
    public async Task AnonymousDiscoveryReturnsOnlyActiveNamesInOrderedPages()
    {
        Database.SoftwareVendors.AddRange(
            Vendor("Zulu", "Active"), Vendor("Alpha", "Active"), Vendor("Hidden",
            "Inactive"), Vendor("Pending", "Provisioning"));
        await Database.SaveChangesAsync();
        using var response = await
Client.GetAsync("/api/v1/auth/vendors?page=1&pageSize=1");
        Assert.Equal(HttpStatusCode.OK, response.StatusCode);
        Assert.True(response.Headers.CacheControl?.NoStore);
        using var json = JsonDocument.Parse(await response.Content.ReadAsStringAsync());
        var root = json.RootElement;
        Assert.Equal(2, root.GetProperty("totalCount").GetInt32());
        Assert.Equal(1, root.GetProperty("page").GetInt32());
        Assert.Equal(1, root.GetProperty("pageSize").GetInt32());
        var item = Assert.Single(root.GetProperty("items").EnumerateArray());
        Assert.Equal("name", Assert.Single(item.EnumerateObject()).Name);
        Assert.Equal("Alpha", item.GetProperty("name").GetString());
        using var next = JsonDocument.Parse(await
Client.GetStringAsync("/api/v1/auth/vendors?page=2&pageSize=1");
        Assert.Equal("Zulu",
next.RootElement.GetProperty("items")[0].GetProperty("name").GetString());
        Assert.Equal(HttpStatusCode.Unauthorized, (await
Client.GetAsync("/api/v1/platform/vendors")).StatusCode);
    }

    /// <summary>
    /// Verifies that empty discovery is successful and invalid pagination is rejected.
    /// </summary>
    [Fact]
    public async Task EmptyDiscoveryIsSuccessfulAndInvalidPaginationIsRejected()
    {
        using var empty = JsonDocument.Parse(await
Client.GetStringAsync("/api/v1/auth/vendors"));
        Assert.Empty(empty.RootElement.GetProperty("items").EnumerateArray());
        Assert.Equal(0, empty.RootElement.GetProperty("totalCount").GetInt32());
        Assert.Equal(HttpStatusCode.BadRequest, (await
Client.GetAsync("/api/v1/auth/vendors?page=0")).StatusCode);
        Assert.Equal(HttpStatusCode.BadRequest, (await
Client.GetAsync("/api/v1/auth/vendors?pageSize=101")).StatusCode);
    }

    /// <summary>
    /// Builds a vendor entity for discovery tests.
    /// </summary>
    private static SoftwareVendor Vendor(string name, string status) => new()
    { Id = Guid.NewGuid(), Name = name, Status = status, CreatedAt = DateTimeOffset.UtcNow };
}

```

Integration/VendorInvitationTests.cs

```
using System.Net;
using System.Net.Http.Json;
using BridgeControl.Data;
using BridgeControl.Models.Vendor;
using BridgeControl.Services.Shared;
using BridgeControl.Tests.Infrastructure;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;

namespace BridgeControl.Tests.Integration;

/// <summary>
/// Verifies vendor invitation behavior and regression scenarios.
/// </summary>
[Trait("Category", "Integration")]
public class VendorInvitationTests : VendorTest
{
    /// <summary>
    /// Verifies that invite and activate create one user and consume the private credential
    once.
    /// </summary>
    [Fact]
    public async Task InviteAndActivate_CreateOneUserAndConsumeThePrivateCredentialOnce()
    {
        await SignInAsync();
        Guid id = await CreateVendorAsync();
        string token = await InviteAsync(id, " INVITED@EXAMPLE.TEST ");
        Assert.Equal(token, await InviteAsync(id));
        Assert.Single(Directory.GetFiles(Factory.InvitationDirectory, "*.eml"));
        using HttpResponseMessage existingVendor = await
Client.PostAsJsonAsync("/api/v1/platform/vendors", new { Name = "Test vendor" });
        Assert.Equal(HttpStatusCode.Conflict, existingVendor.StatusCode);
        await using VendorDbContext database = await OpenVendorAsync(id);
        UserAccount invited = await database.UserAccounts.AsNoTracking().SingleAsync();
        Assert.Equal("Invited", invited.Status);
        Assert.Equal(string.Empty, invited.PasswordHash);
        Assert.Null((await
database.AccountActivations.AsNoTracking().SingleAsync()).ProtectedToken);
        Assert.Equal("Active", (await
Database.SoftwareVendors.AsNoTracking().SingleAsync()).Status);
        using HttpResponseMessage other = await
Client.PostAsJsonAsync($"/api/v1/platform/vendors/{id}/administrators", new { Email =
"other@example.test" });
        Assert.Equal(HttpStatusCode.Conflict, other.StatusCode);
        Client.DefaultRequestHeaders.Authorization = null;
        const string password = " 12345678 ";
        using HttpResponseMessage activated = await
Client.PostAsJsonAsync("/api/v1/auth/activate", new { Token = token, Password = password });
        Assert.Equal(HttpStatusCode.NoContent, activated.StatusCode);
        using HttpResponseMessage reused = await
Client.PostAsJsonAsync("/api/v1/auth/activate", new { Token = token, Password = "other-password"
});
        Assert.Equal(HttpStatusCode.Unauthorized, reused.StatusCode);
        UserAccount active = await database.UserAccounts.AsNoTracking().SingleAsync();
        Assert.Equal("Active", active.Status);
        PasswordService<UserAccount> passwords =
Services.GetRequiredService<PasswordService<UserAccount>>();
```

```

        Assert.Equal(PasswordVerificationResult.Success, passwords.Verify(active,
active.PasswordHash, password));
        Assert.Equal(PasswordVerificationResult.Failed, passwords.Verify(active,
active.PasswordHash, password.Trim()));
        Assert.Equal(1, await database.VendorAuditLogs.CountAsync(audit => audit.Action ==
"AccountActivated"));
    }

    /// <summary>
    /// Verifies that invite retries encrypted pending delivery without duplicating users or
credentials.
    /// </summary>
    [Fact]
    public async Task
Invite_RetriesEncryptedPendingDeliveryWithoutDuplicatingUsersOrCredentials()
    {
        await SignInAsync();
        Guid id = await CreateVendorAsync();
        await File.WriteAllTextAsync(Factory.InvitationDirectory, "Prevent directory
creation.");
        using HttpResponseMessage failure = await
Client.PostAsJsonAsync($"api/v1/platform/vendors/{id}/administrators", new { Email =
"invited@example.test" });
        Assert.Equal(HttpStatusCode.InternalServerError, failure.StatusCode);
        await using VendorDbContext database = await OpenVendorAsync(id);
        AccountActivation pending = await
database.AccountActivations.AsNoTracking().SingleAsync();
        Assert.NotNull(pending.ProtectedToken);
        Assert.Null(pending.DeliveredAt);
        Assert.Equal(1, await Database.PlatformAuditLogs.CountAsync(log => log.Action ==
"InitialVendorInvitationDeliveryFailed"));
        Assert.Equal("AwaitingAdministrator", (await
Database.SoftwareVendors.AsNoTracking().SingleAsync()).Status);
        File.Delete(Factory.InvitationDirectory);
        string token = await InviteAsync(id);
        Assert.DoesNotContain(token, pending.ProtectedToken!);
        Assert.Equal(1, await database.UserAccounts.CountAsync());
        AccountActivation sent = await
database.AccountActivations.AsNoTracking().SingleAsync();
        Assert.Equal(pending.Id, sent.Id);
        Assert.Null(sent.ProtectedToken);
        Assert.NotNull(sent.DeliveredAt);
    }

    /// <summary>
    /// Verifies that invite replaces an expired token but cannot reopen completed bootstrap.
    /// </summary>
    [Fact]
    public async Task Invite_ReplacesAnExpiredTokenButCannotReopenCompletedBootstrap()
    {
        await SignInAsync();
        Guid id = await CreateVendorAsync();
        string previous = await InviteAsync(id);
        await using VendorDbContext database = await OpenVendorAsync(id);
        await database.AccountActivations.ExecuteUpdateAsync(setters => setters
            .SetProperty(value => value.CreatedAt, DateTimeOffset.UtcNow.AddDays(-2))
            .SetProperty(value => value.ExpiresAt, DateTimeOffset.UtcNow.AddDays(-1)));
        string replacement = await InviteAsync(id);
        Assert.NotEqual(previous, replacement);
        Assert.Equal(1, await database.UserAccounts.CountAsync());
        Assert.Equal(2, await database.AccountActivations.CountAsync());
    }

```

```

        using HttpResponseMessage revoked = await
Client.PostAsJsonAsync("/api/v1/auth/activate", new { Token = previous, Password = "1234567890"
});
        Assert.Equal(HttpStatusCode.Unauthorized, revoked.StatusCode);
        await database.UserRoles.ExecuteDeleteAsync();
        using HttpResponseMessage restarted = await
Client.PostAsJsonAsync($"/api/v1/platform/vendors/{id}/administrators", new { Email =
"other@example.test" });
        Assert.Equal(HttpStatusCode.Conflict, restarted.StatusCode);
        Assert.Equal(1, await database.UserAccounts.CountAsync());
    }

    /// <summary>
    /// Verifies that activate rejects unavailable credentials without changing the user.
    /// </summary>
    [Theory]
    [InlineData("expired")]
    [InlineData("revoked")]
    [InlineData("disabled-user")]
    [InlineData("disabled-vendor")]
    [InlineData("wrong-secret")]
    public async Task Activate_RejectsUnavailableCredentialsWithoutChangingTheUser(string
change)
    {
        await SignInAsync();
        Guid id = await CreateVendorAsync();
        string token = await InviteAsync(id);
        await using VendorDbContext database = await OpenVendorAsync(id);
        if (change == "expired")
        {
            await database.AccountActivations.ExecuteUpdateAsync(setters => setters
                .SetProperty(value => value.CreatedAt,
DateTimeOffset.UtcNow.AddDays(-2))
                .SetProperty(value => value.ExpiresAt,
DateTimeOffset.UtcNow.AddDays(-1)));
        }
        if (change == "revoked") { await
database.AccountActivations.ExecuteUpdateAsync(setters => setters.SetProperty(value =>
value.RevokedAt, DateTimeOffset.UtcNow)); }
        if (change == "disabled-user") { await
database.UserAccounts.ExecuteUpdateAsync(setters => setters.SetProperty(value => value.Status,
"Disabled")); }
        if (change == "disabled-vendor") { await
Database.SoftwareVendors.ExecuteUpdateAsync(setters => setters.SetProperty(value => value.Status,
"Disabled")); }
        if (change == "wrong-secret") { token = token[..66] +
Services.GetRequiredService<RefreshCredentialService>().Create(); }
        using HttpResponseMessage response = await
Client.PostAsJsonAsync("/api/v1/auth/activate", new { Token = token, Password = "1234567890" });
        Assert.Equal(HttpStatusCode.Unauthorized, response.StatusCode);
        Assert.Empty((await
database.UserAccounts.AsNoTracking().SingleAsync()).PasswordHash);
        Assert.Null((await
database.AccountActivations.AsNoTracking().SingleAsync()).UsedAt);
    }

    /// <summary>
    /// Verifies that activate concurrent requests allow only one password and audit event.
    /// </summary>
    [Fact]
    public async Task Activate_ConcurrentRequestsAllowOnlyOnePasswordAndAuditEvent()
    {

```

```

        await SignInAsync();
        Guid id = await CreateVendorAsync();
        string token = await InviteAsync(id);
        HttpResponseMessage[] responses = await Task.WhenAll(
            Client.PostAsJsonAsync("/api/v1/auth/activate", new { Token = token,
Password = "1234567890" })),
            Client.PostAsJsonAsync("/api/v1/auth/activate", new { Token = token,
Password = "0987654321" }));
        try
        {
            Assert.Single(responses, response => response.StatusCode ==
HttpStatusCode.NoContent);
            Assert.Single(responses, response => response.StatusCode ==
HttpStatusCode.Unauthorized);
            await using VendorDbContext database = await OpenVendorAsync(id);
            Assert.Equal(1, await database.VendorAuditLogs.CountAsync(audit =>
audit.Action == "AccountActivated"));
        }
        finally { foreach (HttpResponseMessage response in responses) {
response.Dispose(); } }
    }

    /// <summary>
    /// Verifies that activate audit failure rolls back the password and token consumption.
    /// </summary>
    [Fact]
    public async Task Activate_AuditFailureRollsBackThePasswordAndTokenConsumption()
    {
        await SignInAsync();
        Guid id = await CreateVendorAsync();
        string token = await InviteAsync(id);
        await using VendorDbContext database = await OpenVendorAsync(id);
        await database.Database.ExecuteSqlRawAsync("CREATE TRIGGER
TestActivationAuditFailure ON VendorAuditLogs AFTER INSERT AS THROW 51008, 'Test activation audit
failure.', 1;");
        using HttpResponseMessage response = await
Client.PostAsJsonAsync("/api/v1/auth/activate", new { Token = token, Password = "1234567890" });
        Assert.Equal(HttpStatusCode.InternalServerError, response.StatusCode);
        Assert.Empty(await
database.UserAccounts.AsNoTracking().SingleAsync()).PasswordHash);
        Assert.Null(await
database.AccountActivations.AsNoTracking().SingleAsync()).UsedAt);
    }

    /// <summary>
    /// Verifies that activate rejects invalid new passwords before consuming credentials.
    /// </summary>
    [Theory]
    [InlineData("123456789")]
    [InlineData("")]
    public async Task Activate_RejectsInvalidNewPasswordsBeforeConsumingCredentials(string
password)
    {
        using HttpResponseMessage response = await
Client.PostAsJsonAsync("/api/v1/auth/activate", new { Token = "not-a-token", Password = password
});
        Assert.Equal(HttpStatusCode.BadRequest, response.StatusCode);
    }

    /// <summary>
    /// Verifies that activate rate limits malformed credential attempts.
    /// </summary>

```

```

[Fact]
public async Task Activate_RateLimitsMalformedCredentialAttempts()
{
    for (int attempt = 0; attempt < 10; attempt++)
    {
        using HttpResponseMessage response = await
Client.PostAsJsonAsync("/api/v1/auth/activate", new { Token = "not-a-token", Password =
"1234567890" });
        Assert.Equal(HttpStatusCode.Unauthorized, response.StatusCode);
    }
    using HttpResponseMessage limited = await
Client.PostAsJsonAsync("/api/v1/auth/activate", new { Token = "not-a-token", Password =
"1234567890" });
    Assert.Equal(HttpStatusCode.TooManyRequests, limited.StatusCode);
    Assert.NotNull(limited.Headers.RetryAfter);
}
}

```

Integration/VendorProvisioningTests.cs

```

using System.Net;
using System.Net.Http.Json;
using System.Text;
using System.Text.Json;
using BridgeControl.Data;
using BridgeControl.Models.Platform;
using BridgeControl.Services;
using BridgeControl.Services.Shared;
using BridgeControl.Tests.Infrastructure;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;

namespace BridgeControl.Tests.Integration;

/// <summary>
/// Verifies vendor provisioning behavior and regression scenarios.
/// </summary>
[Trait("Category", "Integration")]
public class VendorProvisioningTests : VendorTest
{
    /// <summary>
    /// Verifies that create provisions only on request and retries the same incomplete
    vendor.
    /// </summary>
    [Fact]
    public async Task Create_ProvisionsOnlyOnRequestAndRetriesTheSameIncompleteVendor()
    {
        Assert.Empty(await Database.SoftwareVendors.ToListAsync());
        Assert.False(Directory.Exists(Factory.ArchiveRoot));
        await SignInAsync();
        Guid id = await CreateVendorAsync(" Nordic Software ");
        await using VendorDbContext vendor = await OpenVendorAsync(id);
        Assert.Equal(14, (await vendor.Database.GetAppliedMigrationsAsync()).Count());
        Assert.Equal(4, await vendor.Roles.CountAsync());
        Assert.Equal(18, await vendor.Permissions.CountAsync());
        Assert.False(await vendor.Permissions.AnyAsync(permission =>
permission.Code.StartsWith("platform.")));
        Assert.True(Directory.Exists(Path.Combine(Factory.ArchiveRoot,
id.ToString("N"))));
    }
}

```

```

        Assert.Null(await
Services.GetRequiredService<ITenantResolver>().ResolveForLoginAsync("nordic software"));
        Assert.Equal(id, await CreateVendorAsync("NORDIC SOFTWARE"));
        Assert.Equal(1, await Database.SoftwareVendors.CountAsync());
        VendorProvisioning progress = await
Database.VendorProvisioning.AsNoTracking().SingleAsync();
        Assert.NotNull(progress.DatabaseReadyAt);
        Assert.NotNull(progress.SeededAt);
        Assert.Null(progress.InvitationEstablishedAt);
    }

    /// <summary>
    /// Verifies that create resumes after archive failure without duplicating the database.
    /// </summary>
    [Fact]
    public async Task Create_ResumesAfterArchiveFailureWithoutDuplicatingTheDatabase()
    {
        await SignInAsync();
        PrivateDirectory.Ensure(Factory.PrivateRoot);
        await File.WriteAllTextAsync(Factory.ArchiveRoot, "Force the archive directory
creation to fail.");
        using HttpResponseMessage failure = await
Client.PostAsJsonAsync("/api/v1/platform/vendors", new { Name = "Retry vendor" });
        Assert.Equal(HttpStatusCode.InternalServerError, failure.StatusCode);
        SoftwareVendor pending = await
Database.SoftwareVendors.AsNoTracking().Include(vendor => vendor.Provisioning).SingleAsync();
        Assert.Equal("Failed", pending.Status);
        Assert.Equal("archive-failed", pending.Provisioning!.FailureCode);
        Assert.NotNull(pending.Provisioning.DatabaseReadyAt);
        File.Delete(Factory.ArchiveRoot);
        Assert.Equal(pending.Id, await CreateVendorAsync("Retry vendor"));
        Assert.Equal("AwaitingAdministrator", (await
Database.SoftwareVendors.AsNoTracking().SingleAsync()).Status);
        Assert.Null((await
Database.VendorProvisioning.AsNoTracking().SingleAsync()).FailureCode);
    }

    /// <summary>
    /// Verifies that create concurrent same name requests share one provisioned vendor.
    /// </summary>
    [Fact]
    public async Task Create_ConcurrentSameNameRequestsShareOneProvisionedVendor()
    {
        await SignInAsync();
        Guid[] ids = await Task.WhenAll(CreateVendorAsync("Concurrent vendor"),
CreateVendorAsync("CONCURRENT VENDOR"));
        Assert.Equal(ids[0], ids[1]);
        Assert.Equal(1, await Database.SoftwareVendors.CountAsync());
    }

    /// <summary>
    /// Verifies that reads and patch keep safe fields stable and enforce name uniqueness.
    /// </summary>
    [Fact]
    public async Task ReadsAndPatch_KeepSafeFieldsStableAndEnforceNameUniqueness()
    {
        await SignInAsync();
        Guid first = await CreateVendorAsync("First");
        await CreateVendorAsync("Second");
        using HttpResponseMessage list = await
Client.GetAsync("/api/v1/platform/vendors?page=1&pageSize=1");
        JsonElement page = await ReadAsync(list);

```

```

        Assert.Equal(2, page.GetProperty("totalCount").GetInt32());
        Assert.Single(page.GetProperty("items").EnumerateArray());
        using HttpResponseMessage renamed = await
Client.PatchAsJsonAsync($"/api/v1/platform/vendors/{first}", new { Name = " Renamed " });
        Assert.Equal(HttpStatusCode.OK, renamed.StatusCode);
        JsonElement body = await ReadAsync(renamed);
        Assert.Equal(new[] { "createdAt", "deactivatedAt", "id", "name", "status" },
body.EnumerateObject().Select(property => property.Name).Order());
        Assert.Equal(first, body.GetProperty("id").GetGuid());
        Assert.Equal("Renamed", body.GetProperty("name").GetString());
        Assert.Equal("AwaitingAdministrator", body.GetProperty("status").GetString());
        using HttpResponseMessage duplicate = await
Client.PatchAsJsonAsync($"/api/v1/platform/vendors/{first}", new { Name = " SECOND " });
        Assert.Equal(HttpStatusCode.Conflict, duplicate.StatusCode);
        using HttpResponseMessage omitted = await
Client.PatchAsJsonAsync($"/api/v1/platform/vendors/{first}", new { });
        Assert.Equal("Renamed", (await
ReadAsync(omitted)).GetProperty("name").GetString());
        Assert.Equal(1, await Database.PlatformAuditLogs.CountAsync(audit => audit.Action
== "VendorRenamed"));
        using HttpResponseMessage missing = await
Client.GetAsync($"/api/v1/platform/vendors/{Guid.NewGuid()}");
        Assert.Equal(HttpStatusCode.NotFound, missing.StatusCode);
    }

    /// <summary>
    /// Verifies that patch rejects invalid names and non editable fields.
    /// </summary>
    [Theory]
    [InlineData("{\"name\":\"null\"}")]
    [InlineData("{\"name\":\"  \"}")]
    [InlineData("{\"status\":\"Active\"}")]
    [InlineData("{\"databaseName\":\"other\"}")]
    public async Task Patch_RejectsInvalidNamesAndNonEditableFields(string json)
    {
        await SignInAsync();
        using HttpRequestMessage request = new(HttpMethod.Patch,
$/api/v1/platform/vendors/{Guid.NewGuid()})
        {
            Content = new StringContent(json, Encoding.UTF8, "application/json")
        };
        using HttpResponseMessage response = await Client.SendAsync(request);
        Assert.Equal(HttpStatusCode.BadRequest, response.StatusCode);
    }

    /// <summary>
    /// Verifies that vendor administration requires platform authentication for every new
route.
    /// </summary>
    [Fact]
    public async Task VendorAdministration_RequiresPlatformAuthenticationForEveryNewRoute()
    {
        Guid id = Guid.NewGuid();
        foreach ((HttpMethod method, string path) in new[]
        {
            (HttpMethod.Get, "/api/v1/platform/vendors"), (HttpMethod.Post,
"/api/v1/platform/vendors"),
            (HttpMethod.Get, $"/api/v1/platform/vendors/{id}"), (HttpMethod.Patch,
$/api/v1/platform/vendors/{id}"),
            (HttpMethod.Post, $"/api/v1/platform/vendors/{id}/administrators")
        })
        {
            {

```

```

        using HttpRequestMessage request = new(method, path) { Content =
JsonContent.Create(new { }) };
        using HttpResponseMessage response = await Client.SendAsync(request);
        Assert.Equal(HttpStatusCode.Unauthorized, response.StatusCode);
    }
    Assert.Empty(await Database.SoftwareVendors.ToListAsync());
}

/// <summary>
/// Verifies that create rejects server owned fields before provisioning.
/// </summary>
[Fact]
public async Task Create_RejectsServerOwnedFieldsBeforeProvisioning()
{
    await SignInAsync();
    using HttpResponseMessage response = await
Client.PostAsJsonAsync("/api/v1/platform/vendors", new
    {
        Name = "Overposted vendor",
        Id = Guid.NewGuid(),
        Status = "Active",
        DatabaseName = "arbitrary-database"
    });
    Assert.Equal(HttpStatusCode.BadRequest, response.StatusCode);
    Assert.Empty(await Database.SoftwareVendors.ToListAsync());
    Assert.False(Directory.Exists(Factory.ArchiveRoot));
}
}

```

Integration/VersionPublishingTests.cs

```

using System.Net;
using System.Security.Cryptography;
using System.Text;
using BridgeControl.Api.Exceptions;
using BridgeControl.Repositories.Vendor;
using BridgeControl.Services.Vendor;
using BridgeControl.Tests.Infrastructure;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;

namespace BridgeControl.Tests.Integration;

/// <summary>
/// Verifies version publishing behavior and regression scenarios.
/// </summary>
[Trait("Category", "Integration")]
public class VersionPublishingTests : ConfigurationTest
{
    /// <summary>
    /// Verifies that publishing requires artifacts but not defaults and retries atomically.
    /// </summary>
    [Fact]
    public async Task PublishingRequiresArtifactsButNotDefaultsAndRetriesAtomically()
    {
        await PrepareAsync(); var (product, version) = await CatalogAsync();
        using var missingInstaller = await PublishAsync(product, version);
        Assert.Equal(HttpStatusCode.Conflict, missingInstaller.StatusCode);
        byte[] msi = MsiFixture.Create(Factory.PrivateRoot);
    }
}

```

```

        using var uploaded = await UploadAsync(product, version, "installer", msi);
Assert.Equal(HttpStatusCode.OK, uploaded.StatusCode);
        using var removed = await SendAsync(HttpMethod.Delete,
$"/api/v1/products/{product}/versions/{version}/schema"); Assert.Equal(HttpStatusCode.NoContent,
removed.StatusCode);
        using var missingSchema = await PublishAsync(product, version);
Assert.Equal(HttpStatusCode.Conflict, missingSchema.StatusCode);
        using var restored = await UploadAsync(product, version, "schema",
Encoding.UTF8.GetBytes(Schema)); Assert.Equal(HttpStatusCode.OK, restored.StatusCode);
        var responses = await Task.WhenAll(PublishAsync(product, version),
PublishAsync(product, version));
        string? timestamp = null;
        foreach (var response in responses)
        {
            Assert.Equal(HttpStatusCode.OK, response.StatusCode); var json = await
ReadAsync(response); Assert.Equal("Published", json.GetProperty("status").GetString());
            string time = json.GetProperty("publishedAt").GetString()!; if (timestamp
is not null) { Assert.Equal(timestamp, time); }
            timestamp = time; response.Dispose();
        }
        await using var db = await OpenVendorAsync(VendorId); var published = await
db.ProductVersions.SingleAsync();
        Assert.NotNull(published.PublishedSchemaArtifactId);
        Assert.NotNull(published.PublishedInstallerArtifactId);
        Assert.Null(published.PublishedDefaultRevisionId);
        Assert.Equal(1, await db.VendorAuditLogs.CountAsync(x => x.Action ==
"ProductVersionPublished")); Assert.Equal(0, await db.ConfigurationRevisions.CountAsync());
        using var defaultAfterRelease = await
SetAsync($"/api/v1/products/{product}/configuration", version, new { Port = 80 });
Assert.Equal(HttpStatusCode.Conflict, defaultAfterRelease.StatusCode);
        using var patch = await SendAsync(HttpMethod.Patch,
$"/api/v1/products/{product}/versions/{version}", new { Version = "changed" });
Assert.Equal(HttpStatusCode.Conflict, patch.StatusCode);
        foreach (string kind in new[] { "schema", "installer" })
        {
            using var replacement = await UploadAsync(product, version, kind, kind ==
"schema" ? Encoding.UTF8.GetBytes(Schema) : msi); Assert.Equal(HttpStatusCode.Conflict,
replacement.StatusCode);
            using var delete = await SendAsync(HttpMethod.Delete,
$"/api/v1/products/{product}/versions/{version}/{kind}"); Assert.Equal(HttpStatusCode.Conflict,
delete.StatusCode);
            using var download = await SendAsync(HttpMethod.Get,
$"/api/v1/products/{product}/versions/{version}/{kind}/content"); Assert.Equal(HttpStatusCode.OK,
download.StatusCode);
            Assert.Equal(kind == "schema" ? Encoding.UTF8.GetBytes(Schema) : msi, await
download.Content.ReadAsByteArrayAsync());
        }
        Guid customer = await CustomerAsync(); using var overrideAfterRelease = await
SetAsync($"/api/v1/customers/{customer}/products/{product}/configuration", version, new { Port =
443 });
        Assert.Equal(HttpStatusCode.OK, overrideAfterRelease.StatusCode);
    }
    /// <summary>
    /// Verifies that corrupt or structurally invalid stored artifacts cannot publish.
    /// </summary>
    [Theory]
    [InlineData("Schema", false)]
    [InlineData("Installer", false)]
    [InlineData("Schema", true)]
    [InlineData("Installer", true)]
    public async Task CorruptOrStructurallyInvalidStoredArtifactsCannotPublish(string kind,
bool matchingHash)

```

```

        {
            await PrepareAsync(); var (product, version) = await CatalogAsync(installer:
true); await using var db = await OpenVendorAsync(VendorId);
            var artifact = await db.FileArtifacts.SingleAsync(x => x.ArtifactType == kind);
            byte[] invalid = Encoding.UTF8.GetBytes("not a supported artifact");
            await File.WriteAllBytesAsync(Path.Combine(ArchiveRoot, "blobs",
artifact.StorageIdentifier + ".blob"), invalid);
            if (matchingHash) { artifact.Sha256 =
Convert.ToHexStringLower(SHA256.HashData(invalid)); artifact.SizeBytes = invalid.Length; await
db.SaveChangesAsync(); }
            using var failed = await PublishAsync(product, version);
            Assert.Equal(HttpStatusCode.Conflict, failed.StatusCode);
            var draft = await db.ProductVersions.AsNoTracking().SingleAsync();
            Assert.Equal("Draft", draft.Status); Assert.Null(draft.PublishedAt);
            Assert.Equal(0, await db.VendorAuditLogs.CountAsync(x => x.Action ==
"ProductVersionPublished"));
            Assert.DoesNotContain(ArchiveRoot, await failed.Content.ReadAsStringAsync());
        }
        /// <summary>
        /// Verifies that incompatible default does not add an undocumented publication gate.
        /// </summary>
        [Fact]
        public async Task IncompatibleDefaultDoesNotAddAnUndocumentedPublicationGate()
        {
            await PrepareAsync(); var (product, version) = await CatalogAsync(installer:
true); string path = $"api/v1/products/{product}/configuration";
            using var original = await SetAsync(path, version, new { Port = 80 });
            Assert.Equal(HttpStatusCode.OK, original.StatusCode);
            Guid revision = (await ReadAsync(original)).GetProperty("id").GetGuid();
            string changedSchema = Schema.Replace($"\"minimum\":1", $"\"minimum\":100");
            using var replacement = await UploadAsync(product, version, "schema",
Encoding.UTF8.GetBytes(changedSchema)); Assert.Equal(HttpStatusCode.OK, replacement.StatusCode);
            using var publish = await PublishAsync(product, version);
            Assert.Equal(HttpStatusCode.OK, publish.StatusCode);
            await using var db = await OpenVendorAsync(VendorId); var release = await
db.ProductVersions.SingleAsync(); Assert.Null(release.PublishedDefaultRevisionId);
            Assert.Equal("PublishedWithoutDefault", (await db.VendorAuditLogs.SingleAsync(x =>
x.Action == "ProductVersionPublished")).Outcome);
            var historic = await db.ConfigurationRevisions.SingleAsync();
            Assert.Equal(revision, historic.Id); Assert.NotEqual(release.PublishedSchemaArtifactId,
historic.SchemaArtifactId);
            using var history = await SendAsync(HttpMethod.Get, path + "/revisions");
            Assert.Equal(80, (await
ReadAsync(history)).GetProperty("items")[0].GetProperty("values").GetProperty("port").GetInt32());
        }
        /// <summary>
        /// Verifies that publishing and upload reauthorize without losing frozen references.
        /// </summary>
        [Fact]
        public async Task PublishingAndUploadReauthorizeWithoutLosingFrozenReferences()
        {
            await PrepareAsync(); var (product, version) = await CatalogAsync(installer:
true); await using var db = await OpenVendorAsync(VendorId);
            var session = await db.AuthenticationSessions.SingleAsync(); TenantActor actor =
new(VendorId, session.AccountId, session.Id);
            var artifacts = Services.GetRequiredService<ProductArtifactRepository>();
            using MemoryStream stream = new(Encoding.UTF8.GetBytes(Schema.Replace("65535",
"10000")));
            await Assert.ThrowsAsync<ApiConflictException>(() => artifacts.PutAsync(actor,
product, version, "Schema", "new.json", stream,

```

```

        async ct => { using var published = await PublishAsync(product, version);
Assert.Equal(HttpStatusCode.OK, published.StatusCode); }, default));
        var release = await db.ProductVersions.SingleAsync();
        Assert.Equal(release.PublishedSchemaArtifactId, await
db.ProductVersionArtifacts.Where(x => x.ArtifactType == "Schema").Select(x =>
x.FileArtifactId).SingleAsync());
        Assert.Equal(2, await db.FileArtifacts.CountAsync());
Assert.Empty(Directory.GetFiles(Path.Combine(ArchiveRoot, "staging")));
        var tech = await UserAsync("tech@example.test", "VendorTechnician");
        using var denied = await SendAsync(HttpMethod.Post,
$"/api/v1/products/{product}/versions/{version}/publish", token: tech.Token);
Assert.Equal(HttpStatusCode.Forbidden, denied.StatusCode);
        using var wrongParent = await SendAsync(HttpMethod.Post,
$"/api/v1/products/{Guid.NewGuid()}/versions/{version}/publish");
Assert.Equal(HttpStatusCode.NotFound, wrongParent.StatusCode);
    }
    /// <summary>
    /// Verifies that audit failure rolls back publication and preserves default.
    /// </summary>
    [Fact]
    public async Task AuditFailureRollsBackPublicationAndPreservesDefault()
    {
        await PrepareAsync(); var (product, version) = await CatalogAsync(installer:
true);
        using var configuration = await
SetAsync($"/api/v1/products/{product}/configuration", version, new { Port = 80 });
Assert.Equal(HttpStatusCode.OK, configuration.StatusCode);
        Guid revision = (await ReadAsync(configuration)).GetProperty("id").GetGuid();
await using var db = await OpenVendorAsync(VendorId);
        await db.Database.ExecuteSqlRawAsync("CREATE TRIGGER TestPublishAuditFailure ON
VendorAuditLogs AFTER INSERT AS THROW 51022, 'Test publication audit failure.', 1;");
        using var failure = await PublishAsync(product, version);
Assert.Equal(HttpStatusCode.InternalServerError, failure.StatusCode);
        var draft = await db.ProductVersions.AsNoTracking().SingleAsync();
Assert.Equal("Draft", draft.Status); Assert.Null(draft.PublishedDefaultRevisionId);
Assert.Null(draft.PublishedAt);
        await db.Database.ExecuteSqlRawAsync("DROP TRIGGER TestPublishAuditFailure;");
        using var retry = await PublishAsync(product, version);
Assert.Equal(HttpStatusCode.OK, retry.StatusCode);
        Assert.Equal(revision, (await
db.ProductVersions.AsNoTracking().SingleAsync()).PublishedDefaultRevisionId);
    }
}

```

Unit/AccountValidationTests.cs

```

using BridgeControl.Services.Shared;

namespace BridgeControl.Tests.Unit;

/// <summary>
/// Verifies account validation behavior and regression scenarios.
/// </summary>
[Trait("Category", "Unit")]
public class AccountValidationTests
{
    /// <summary>
    /// Verifies that validate new account enforces password length.
    /// </summary>
    [Theory]

```

```

[InlineData(9, false)]
[InlineData(10, true)]
[InlineData(128, true)]
[InlineData(129, false)]
public void ValidateNewAccount_EnforcesPasswordLength(int length, bool valid)
{
    string? error = AccountValidation.ValidateNewAccount("user@example.test", new
string('a', length));
    Assert.Equal(valid, error is null);
}

/// <summary>
/// Verifies that validate new account preserves spaces but rejects an entirely blank
password.
/// </summary>
[Theory]
[InlineData(" ", false)]
[InlineData(" 12345678 ", true)]
public void ValidateNewAccount_PreservesSpacesButRejectsAnEntirelyBlankPassword(string
password, bool valid)
{
    Assert.Equal(valid, AccountValidation.ValidateNewAccount("user@example.test",
password) is null);
}
}

```

Unit/ArtifactValidationTests.cs

```

using System.Text;
using BridgeControl.Api.Exceptions;
using BridgeControl.Options;
using BridgeControl.Services.Shared;
using BridgeControl.Services.Vendor;
using Microsoft.Extensions.Options;

namespace BridgeControl.Tests.Unit;

/// <summary>
/// Verifies artifact validation behavior and regression scenarios.
/// </summary>
[Trait("Category", "Unit")]
public class ArtifactValidationTests : IAsyncDisposable
{
    private readonly SchemaArtifactValidationService schemas = new([new
Draft202012SchemaArtifactValidator()]);
    private readonly string root = PrivateDirectory.Ensure(Path.Combine(Path.GetTempPath(),
"BridgeControl_ArtifactTests_" + Guid.NewGuid().ToString("N")));
    /// <summary>
    /// Parses the schema fixture used by the validation test.
    /// </summary>
    private static string Schema(string members) => $"{"$schema\":\"" +
Draft202012SchemaArtifactValidator.DialectUri + "\", " + members + "}";
    /// <summary>
    /// Stages and validates the artifact fixture for the test.
    /// </summary>
    private async Task ValidateAsync(string json)
    {
        string path = Path.Combine(root, Guid.NewGuid().ToString("N") + ".json");
        await File.WriteAllTextAsync(path, json); await schemas.ValidateAsync(path,
default);
    }
}

```

```

    }
    /// <summary>
    /// Verifies that supported schemas are accepted.
    /// </summary>
    [Theory]

    [InlineData("\"type\": \"object\", \"$defs\": {\"port\": {\"type\": \"integer\"}}, \"properties\": {\"port\": {\"$ref\": \"#/defs/port\"}})")]

    [InlineData("\"type\": [\"object\", \"null\"], \"properties\": {\"enabled\": true}, \"required\": [\"enabled\"], \"additionalProperties\": false)")]

    [InlineData("\"allOf\": [{\"type\": \"object\"}, {\"properties\": {\"names\": {\"type\": \"array\", \"items\": {\"type\": \"string\", \"minLength\": 1}, \"uniqueItems\": true}}}]")]]
    public Task SupportedSchemasAreAccepted(string members) => ValidateAsync(Schema(members));

    /// <summary>
    /// Verifies that unsupported or malformed schemas are rejected.
    /// </summary>
    [Theory]
    [InlineData("\"type\": \"invalid\"")]
    [InlineData("\"type\": [\"string\", \"string\"]")]
    [InlineData("\"required\": [\"a\", \"a\"]")]
    [InlineData("\"properties\": [])")]
    [InlineData("\"minimum\": \"1\"")]
    [InlineData("\"minLength\": -1")]
    [InlineData("\"multipleOf\": 0")]
    [InlineData("\"enum\": [1, 1.0]")]
    [InlineData("\"type\": \"object\", \"type\": \"string\"")]
    [InlineData("\"$ref\": \"https://attacker.invalid/schema\"")]
    [InlineData("\"$ref\": \"file:///C:/private.json\"")]
    [InlineData("\"$ref\": \"#/$defs/missing\"")]
    [InlineData("\"$defs\": {\"loop\": {\"$ref\": \"#/$defs/loop\"}})")]
    [InlineData("\"$ref\": \"#/default\", \"default\": {})")]
    [InlineData("\"pattern\": \"[\"")]
    [InlineData("\"pattern\": \"(?:=a)a\"")]
    [InlineData("\"format\": \"email\"")]
    [InlineData("\"x-unknown\": true")]
    [InlineData("\"maximum\": 1e1000")]
    public async Task UnsupportedOrMalformedSchemasAreRejected(string members)
        => await Assert.ThrowsAsync<ArtifactValidationException>(() =>
        ValidateAsync(Schema(members)));

    /// <summary>
    /// Verifies that missing dialect truncation and deep documents are rejected.
    /// </summary>
    [Fact]
    public async Task MissingDialectTruncationAndDeepDocumentsAreRejected()
    {
        foreach (string json in new[] { "{", "{", "true", Schema("\"$defs\": \" + new
        string('[' + 65) + "true" + new string(']', 65)) })
        {
            await Assert.ThrowsAsync<ArtifactValidationException>(() =>
            ValidateAsync(json));
        }
    }

    /// <summary>
    /// Verifies that interrupted and oversized streams remove their private stages.
    /// </summary>
    [Fact]
    public async Task InterruptedAndOversizedStreamsRemoveTheirPrivateStages()
    {

```

```

        VendorArtifactArchive archive =
new(Microsoft.Extensions.Options.Options.Create(new ArtifactArchiveOptions { SchemaMaxBytes = 16
}), schemas);
        using MemoryStream oversized = new(Encoding.UTF8.GetBytes(new string('x', 100)));
        await Assert.ThrowsAsync<ArtifactValidationException>(() =>
archive.StageAsync(root, "Schema", "schema.json", oversized, default));
        using InterruptedStream interrupted = new();
        await Assert.ThrowsAsync<IOException>(() => archive.StageAsync(root, "Schema",
"schema.json", interrupted, default));
        using CancellationTokenSource cancellation = new(); cancellation.Cancel();
        using MemoryStream bytes = new([1, 2, 3]);
        await Assert.ThrowsAnyAsync<OperationCanceledException>(() =>
archive.StageAsync(root, "Schema", "schema.json", bytes, cancellation.Token));
        Assert.Empty(Directory.GetFiles(Path.Combine(root, "staging")));
        Assert.Empty(Directory.GetFiles(Path.Combine(root, "blobs")));
    }
    /// <summary>
    /// Verifies that reference depth limit does not depend on definition order.
    /// </summary>
    [Fact]
    public async Task ReferenceDepthLimitDoesNotDependOnDefinitionOrder()
    {
        string definitions = @"\n0":true," + string.Join(',', Enumerable.Range(1,
70).Select(i => $"\n{i}\":{{\"$ref\":"#\n{i - 1}\n}}"));
        await Assert.ThrowsAsync<ArtifactValidationException>(() =>
ValidateAsync(Schema($"\"$defs\":" + definitions + "}"));
    }
    /// <summary>
    /// Verifies that stale stage cleanup only removes generated old unlocked uploads.
    /// </summary>
    [Fact]
    public async Task StaleStageCleanupOnlyRemovesGeneratedOldUnlockedUploads()
    {
        string staging = PrivateDirectory.Ensure(Path.Combine(root, "staging"));
        string old = Path.Combine(staging, Guid.NewGuid().ToString("N") + ".upload");
        string unrelated = Path.Combine(staging, "operator-note.upload");
        string locked = Path.Combine(staging, Guid.NewGuid().ToString("N") + ".upload");
        foreach (string path in new[] { old, unrelated, locked }) { await
File.WriteAllTextAsync(path, "stage"); File.SetLastWriteTimeUtc(path,
DateTime.UtcNow.AddDays(-2)); }
        using FileStream lease = new(locked, FileMode.Open, FileAccess.ReadWrite,
FileShare.None);
        VendorArtifactArchive archive =
new(Microsoft.Extensions.Options.Options.Create(new ArtifactArchiveOptions()), schemas);
        using MemoryStream source =
new(Encoding.UTF8.GetBytes(Schema($"\"type\":"object\")));
        await using (var staged = await archive.StageAsync(root, "Schema", "schema.json",
source, default))
        { Assert.False(File.Exists(old)); Assert.True(File.Exists(unrelated));
Assert.True(File.Exists(locked)); }
        Assert.Equal(2, Directory.GetFiles(staging).Length);
    }
    /// <summary>
    /// Simulates a stream failure during artifact upload.
    /// </summary>
    private sealed class InterruptedStream : MemoryStream
    {
        private int reads;
        /// <summary>
        /// Throws the configured interruption when an upload attempts to read the stream.
        /// </summary>

```

```

        public override ValueTask<int> ReadAsync(Memory<byte> buffer, CancellationToken
cancellationToken = default)
        {
            if (reads++ != 0) { throw new IOException("Interrupted upload"); }
            buffer.Span[0] = 1; return ValueTask.FromResult(1);
        }
    }
    /// <summary>
    /// Asynchronously releases the resources owned by this instance.
    /// </summary>
    public ValueTask DisposeAsync()
    {
        string fullPath = Path.GetFullPath(root);
        if (Path.GetDirectoryName(fullPath) !=
Path.GetTempPath().TrimEnd(Path.DirectorySeparatorChar)
            || !Path.GetFileName(fullPath).StartsWith("BridgeControl_ArtifactTests_",
StringComparison.Ordinal))
        { throw new InvalidOperationException("Unsafe test cleanup target."); }
        Directory.Delete(fullPath, recursive: true); return ValueTask.CompletedTask;
    }
}

```

Unit/ConfigurationMigrationRulesTests.cs

```

using System.Text.Json;
using BridgeControl.Api.Exceptions;
using BridgeControl.DTOs.Vendor;
using BridgeControl.Services.Vendor;

namespace BridgeControl.Tests.Unit;

/// <summary>
/// Verifies configuration migration rules behavior and regression scenarios.
/// </summary>
public class ConfigurationMigrationRulesTests
{
    private readonly ConfigurationMigrationRules rules = new(new ConfigurationValues(new
SchemaArtifactValidationService([new Draft202012SchemaArtifactValidator()]));
    /// <summary>
    /// Parses the supplied JSON into an independent test value.
    /// </summary>
    private static JsonElement Json(string text) =>
JsonDocument.Parse(text).RootElement.Clone();
    /// <summary>
    /// Builds a configuration migration rule for the test scenario.
    /// </summary>
    private static MigrationRuleDto Rule(string type, string source = "", string target = "",
string transform = "", bool destructive = false) => new()
    { Id = Guid.NewGuid(), Sequence = 1, RuleType = type, SourcePath = source, TargetPath =
target, Transform = transform, IsDestructive = destructive };
    /// <summary>
    /// Verifies that layers preserve unmentioned values and treat arrays as atomic.
    /// </summary>
    [Fact]
    public void Layers_PreserveUnmentionedValuesAndTreatArraysAsAtomic()
    {
        var result = rules.Apply(Json("""{"old":[1,2],"keep":true}"""), [Rule("Move",
"/old", "/nested/new")], "Customer");
        Assert.Empty(result.Unresolved); Assert.Equal(2,
result.Values.GetProperty("nested").GetProperty("new").GetArrayLength());
    }
}

```

```

        Assert.True(result.Values.GetProperty("keep").GetBoolean());
Assert.False(result.Values.TryGetProperty("old", out _));
    }
    /// <summary>
    /// Verifies that collisions and array traversal are never silently overwritten.
    /// </summary>
    [Fact]
    public void CollisionsAndArrayTraversal_AreNeverSilentlyOverwritten()
    {
        var collision = rules.Apply(Json("'{\"old\":1,\"new\":2}'"), [Rule("Move", "/old",
"/new")], "Installation");
        Assert.Single(collision.Unresolved); Assert.Equal(1,
collision.Values.GetProperty("old").GetInt32()); Assert.Equal(2,
collision.Values.GetProperty("new").GetInt32());
        var array = rules.Apply(Json("'{\"old\":1,\"array\":[]}'"), [Rule("Move", "/old",
"/array/0")], "Installation"); Assert.Single(array.Unresolved);
    }
    /// <summary>
    /// Verifies that conversion is explicit and culture independent.
    /// </summary>
    [Theory]
    [InlineData("integer", "\"123\"", "123")]
    [InlineData("boolean", "\"false\"", "false")]
    [InlineData("number", "\"1.25\"", "1.25")]
    [InlineData("string", "123", "\"123\"")]
    public void Conversion_IsExplicitAndCultureIndependent(string type, string input, string
expected)
    {
        var result = rules.Apply(Json("{\"value\":\"" + input + "\"}"), [Rule("Convert",
"/value", transform: type, destructive: true)], "Customer");
        Assert.Empty(result.Unresolved); Assert.Equal(expected,
result.Values.GetProperty("value").GetRawText());
    }
    /// <summary>
    /// Verifies that invalid conversion remains unresolved.
    /// </summary>
    [Theory]
    [InlineData("\"1.5\"")]
    [InlineData("\"9007199254740992\"")]
    [InlineData("{}")]
    public void InvalidConversion_RemainsUnresolved(string input)
    {
        var result = rules.Apply(Json("{\"value\":\"" + input + "\"}"), [Rule("Convert",
"/value", transform: "integer", destructive: true)], "Customer");
Assert.Single(result.Unresolved);
    }
    /// <summary>
    /// Verifies that definition validation rejects unsafe and ambiguous rules.
    /// </summary>
    [Fact]
    public void DefinitionValidation_RejectsUnsafeAndAmbiguousRules()
    {
        Assert.Throws<ConfigurationValidationException>(() => rules.Validate(new(1,
"Delete", "/value", null, null, false)));
        Assert.Throws<ConfigurationValidationException>(() => rules.Validate(new(1,
"Move", "/value", "/value/nested", null, false)));
        Assert.Throws<ConfigurationValidationException>(() => rules.Validate(new(1,
"Preserve", "/bad~2", null, null, false)));
        Assert.Throws<ConfigurationValidationException>(() => rules.Validate(new(1,
"Script", "/value", null, "code", false)));
        Assert.Throws<ConfigurationValidationException>(() => rules.Validate(new(1,
"Introduce", null, "/value", "null", false)));
    }

```

```

        rules.Validate([new(1, "Preserve", "/a~1b/~0key", null, null, false)]);
    }
}

```

Unit/ConfigurationValuesTests.cs

```

using System.Text;
using System.Text.Json;
using BridgeControl.Api.Exceptions;
using BridgeControl.Services.Vendor;
using Org.Webpki.JsonCanonicalizer;

namespace BridgeControl.Tests.Unit;

/// <summary>
/// Verifies configuration values behavior and regression scenarios.
/// </summary>
[Trait("Category", "Unit")]
public class ConfigurationValuesTests
{
    private readonly ConfigurationValues service = new(new
SchemaArtifactValidationService([new Draft202012SchemaArtifactValidator()]));
    /// <summary>
    /// Parses the supplied JSON into an independent test value.
    /// </summary>
    private static JsonElement Json(string json) =>
JsonDocument.Parse(json).RootElement.Clone();
    /// <summary>
    /// Parses the schema fixture used by the validation test.
    /// </summary>
    private static JsonElement Schema(string members) => Json("{\"schema\":\"" +
Draft202012SchemaArtifactValidator.DialectUri + "\",\"members\": \"" + members + "\"}");
    /// <summary>
    /// Verifies that cleanup merge and canonical bytes preserve strings arrays and
inheritance.
    /// </summary>
    [Fact]
    public void CleanupMergeAndCanonicalBytesPreserveStringsArraysAndInheritance()
    {
        JsonElement defaults =
Json("""{"network":{"port":80,"host":"local"},"tags":["base"]}""");
        JsonElement customer =
Json("""{"tags":["site",null],"network":{"host":null,"port":443},"name":" spaced "}""");
        JsonElement installation = Json("""{"network":{"port":8443}}""");
        Assert.Equal("""{"name":" spaced
", "network":{"host":"local", "port":8443}, "tags":["site", null]}""",
Encoding.UTF8.GetString(service.Merge(defaults, customer, installation)));
        Assert.Equal(service.Normalize(Json("""{"a":1.0, "b":2, "gone":null}""")),
service.Normalize(Json("""{"b":2e0, "a":1}""")));
    }
    /// <summary>
    /// Verifies that canonicalizer matches rfc8785 example and unicode sort vector.
    /// </summary>
    [Fact]
    public void CanonicalizerMatchesRfc8785ExampleAndUnicodeSortVector()
    {
        const string input =
""""{"numbers": [333333333.33333329, 1E30, 4.50, 2e-3, 0.000000000000000000000001], "string": "\u20ac$
\u000F\u000aA'\u0042\u0022\u005c\\\"/\", \"literals\": [null, true, false]}""";

```

```

        const string expected =
            "\"{\\\"literals\\\":[null,true,false],\\\"numbers\\\":[333333333.3333333,1e+30,4.5,0.002,1e-27],\\\"string\\\":\\\"€$\\n\\u000f\\nA'B\\\"\\\\\\\\\\\\\\\\\\\"}\"";
        Assert.Equal(expected, new JsonCanonicalizer(input).GetEncodedString());
        JsonElement sorted = Json(new
            JsonCanonicalizer("\"\"{\\\"u20ac\\\":1,\\\"r\\\":2,\\\"ufb33\\\":3,\\\"1\\\":4,\\\"ud83d\\ude00\\\":5,\\\"u0080\\\":6,\\\"u00f6\\\":7}\"")
            .GetEncodedString());
        Assert.Equal(new[] { "\\r", "1", "\\u0080", "ö", "€", "👨", "Ṛ" },
            sorted.EnumerateObject().Select(x => x.Name));
    }
    /// <summary>
    /// Verifies that invalid configuration values are rejected.
    /// </summary>
    [Theory]
    [InlineData("{\"a\\\":1,\\\"a\\\":2}\"")]
    [InlineData("{\"a\\\":9007199254740992}\"")]
    [InlineData("{\"a\\\":1e999}\"")]
    [InlineData("{\"a\\\":\\\"\\\\uDEAD\\\"}\"")]
    [InlineData("{\"\"}")]
    public void InvalidConfigurationValuesAreRejected(string input)
        => Assert.Throws<ConfigurationValidationException>(() =>
            service.Normalize(Json(input)));

    /// <summary>
    /// Verifies that supported schema keywords have draft202012 semantics.
    /// </summary>
    [Theory]

    [InlineData("{\"type\\\":\\\"object\\\",\\\"required\\\":[\\\"port\\\"],\\\"properties\\\":{\\\"port\\\":{\\\"type\\\":\\\"integer\\\",\\\"minimum\\\":1,\\\"maximum\\\":65535}},\\\"additionalProperties\\\":false\", \"{\\\"port\\\":443}\",
        true)]
    [InlineData("{\"type\\\":\\\"object\\\",\\\"required\\\":[\\\"port\\\"]\", \"{\\\"\\\", false)"]
    [InlineData("{\"properties\\\":{\\\"port\\\":{\\\"type\\\":\\\"integer\\\"}}\", \"{\\\"port\\\":\\\"443\\\"}\",
        false)]
    [InlineData("{\"additionalProperties\\\":false\", \"{\\\"unknown\\\":1}\", false)]

    [InlineData("{\"$defs\\\":{\\\"num\\\":{\\\"type\\\":\\\"number\\\",\\\"multipleOf\\\":0.1}},\\\"properties\\\":{\\\"n\\\":{\\\"$ref\\\":\\\"#/$defs/num\\\"}}\", \"{\\\"n\\\":0.3}\", true)]

    [InlineData("{\"properties\\\":{\\\"s\\\":{\\\"type\\\":\\\"string\\\",\\\"minLength\\\":1,\\\"maxLength\\\":1}}\",
        \"{\\\"s\\\":\\\"👨\\\"}\", true)]

    [InlineData("{\"properties\\\":{\\\"a\\\":{\\\"type\\\":\\\"array\\\",\\\"prefixItems\\\":[{\\\"const\\\":1}],\\\"items\\\":
        false}}\", \"{\\\"a\\\":[1,2]}\", false)]
    [InlineData("{\"properties\\\":{\\\"a\\\":{\\\"uniqueItems\\\":true}}\", \"{\\\"a\\\":[1,1.0]}\", false)]
    [InlineData("{\"oneOf\\\":[{\\\"required\\\":[\\\"a\\\"]},{\\\"required\\\":[\\\"b\\\"]}]",
        \"{\\\"a\\\":1,\\\"b\\\":1}\", false)]

    [InlineData("{\"if\\\":{\\\"required\\\":[\\\"a\\\"]},\\\"then\\\":{\\\"required\\\":[\\\"b\\\"]},\\\"else\\\":{\\\"required\\\":
        [\\\"c\\\"]}\", \"{\\\"a\\\":1}\", false)]

    [InlineData("{\"allOf\\\":[{\\\"minProperties\\\":1},{\\\"maxProperties\\\":2}],\\\"not\\\":{\\\"required\\\":[\\\"bad\\\"
        ]}\", \"{\\\"good\\\":true}\", true)]
    [InlineData("{\"properties\\\":{\\\"a\\\":{\\\"anyOf\\\":[{\\\"enum\\\":[1,2]},{\\\"type\\\":\\\"null\\\"}]}}\",
        \"{\\\"a\\\":3}\", false)]
    public void SupportedSchemaKeywordsHaveDraft202012Semantics(string schema, string input,
        bool valid)
    {
        if (valid) { service.Validate(schema, Json(input), default); }
        else { Assert.Throws<ConfigurationValidationException>(() =>
            service.Validate(schema, Json(input), default)); }
    }
}

```

```

    /// <summary>
    /// Verifies that exponential references are rejected before library evaluation.
    /// </summary>
    [Fact]
    public void ExponentialReferencesAreRejectedBeforeLibraryEvaluation()
    {
        string defs = "\"n0\":true,\" + string.Join(',', Enumerable.Range(1, 25).Select(i
=> $"\"n{i}\":{{\"allOf\":[{{\"$ref\": \"#/$defs/n{i - 1}\"}},{{\"$ref\": \"#/$defs/n{i -
1}\"}}]}})\"));
        Assert.Throws<ConfigurationValidationException>(() =>
service.Validate(Schema($"\"$defs\":{{\" + defs + \"}},\"$ref\": \"#/$defs/n25\"}), Json("{}"),
default));
    }
    /// <summary>
    /// Verifies that merge replaces scalars and arrays and preserves accepted empty objects.
    /// </summary>
    [Fact]
    public void Merge_ReplacesScalarsAndArrays_AndPreservesAcceptedEmptyObjects()
    {
        Assert.Equal("""{"empty":{},"nested":{"x":2},"scalar":false,"tags":[]}""",
Encoding.UTF8.GetString(service.Merge(
        Json("""{"nested":1,"scalar":{"x":1},"tags":[1,2]}"""),
Json("""{"nested":{"x":2},"scalar":false,"tags":[],"empty":{"gone":null}}""")));
        Assert.Equal({}, Encoding.UTF8.GetString(service.Merge()));
    }
}

```

Unit/EncryptedBackupTests.cs

```

using System.Security.Cryptography;
using BridgeControl.Services.Platform;

namespace BridgeControl.Tests.Unit;

/// <summary>
/// Verifies encrypted backup behavior and regression scenarios.
/// </summary>
[Trait("Category", "Unit")]
public sealed class EncryptedBackupTests
{
    /// <summary>
    /// Verifies that authenticates entire envelope before writing plaintext.
    /// </summary>
    [Theory]
    [InlineData("content")]
    [InlineData("header")]
    [InlineData("tag")]
    [InlineData("truncated")]
    [InlineData("wrong-key")]
    public async Task AuthenticatesEntireEnvelopeBeforeWritingPlaintext(string corruption)
    {
        string directory = Path.Combine(Path.GetTempPath(), "BridgeControl_CryptoTests_" +
Guid.NewGuid().ToString("N")); Directory.CreateDirectory(directory);
        string source = Path.Combine(directory, "source"), encrypted =
Path.Combine(directory, "encrypted"), restored = Path.Combine(directory, "restored");
        byte[] key = RandomNumberGenerator.GetBytes(32), data =
RandomNumberGenerator.GetBytes(200001);
        try
        {
            await File.WriteAllBytesAsync(source, data);

```

```

        await EncryptedBackup.EncryptAsync(source, encrypted, "key-1", key, 300000,
default);
        await EncryptedBackup.DecryptAsync(encrypted, restored, new
Dictionary<string, string> { ["key-1"] = Convert.ToBase64String(key) }, 300000, default);
Assert.Equal(data, await File.ReadAllBytesAsync(restored));
File.Delete(restored);
byte[] bytes = await File.ReadAllBytesAsync(encrypted);
switch (corruption)
{
    case "content": bytes[1000] ^= 1; break;
    case "header": bytes[75] ^= 1; break;
    case "tag": bytes[^1] ^= 1; break;
    case "truncated": bytes = bytes[..^32]; break;
    case "wrong-key": key = RandomNumberGenerator.GetBytes(32); break;
}
await File.WriteAllBytesAsync(encrypted, bytes);
await Assert.ThrowsAsync<InvalidDataException>(() =>
EncryptedBackup.DecryptAsync(encrypted, restored, new Dictionary<string, string> { ["key-1"] =
Convert.ToBase64String(key) }, 300000, default));
Assert.False(File.Exists(restored));
}
finally { foreach (string file in new[] { source, encrypted, restored }) {
File.Delete(file); } Directory.Delete(directory); }
}
}

```

Unit/EntityMapperTests.cs

```

using BridgeControl.DTOS.Platform;
using BridgeControl.Mappings.Shared;
using BridgeControl.Models.Platform;

namespace BridgeControl.Tests.Unit;

/// <summary>
/// Verifies entity mapper behavior and regression scenarios.
/// </summary>
[Trait("Category", "Unit")]
public class EntityMapperTests
{
    /// <summary>
    /// Verifies that map dto create request leaves server owned fields untouched.
    /// </summary>
    [Fact]
    public void MapDto_CreateRequestLeavesServerOwnedFieldsUntouched()
    {
        CreateVendorRequest request = new() { Name = " Example " };
        Guid id = Guid.NewGuid();
        SoftwareVendor vendor = new() { Id = id, Status = "Active" };

        vendor.MapDto(request);

        Assert.Equal("Example", vendor.Name);
        Assert.Equal(id, vendor.Id);
        Assert.Equal("Active", vendor.Status);
    }

    /// <summary>
    /// Verifies that map dto copies exact values including null and defaults.
    /// </summary>
}

```

```

[Theory]
[InlineData(" keep spaces ", true)]
[InlineData(null, false)]
public void MapDto_CopiesExactValuesIncludingNullAndDefaults(string? name, bool enabled)
{
    Input request = new() { Name = name, Enabled = enabled };
    Entity entity = new() { Name = "Before", Enabled = !enabled };

    entity.MapDto(request);

    Assert.Equal(name, entity.Name);
    Assert.Equal(enabled, entity.Enabled);
}

/// <summary>
/// Verifies that map dto skips relationships incompatible types and nonpublic setters.
/// </summary>
[Fact]
public void MapDto_SkipsRelationshipsIncompatibleTypesAndNonpublicSetters()
{
    Input request = new() { Count = 42, Children = ["new"], ReadOnly = "attempted
change" };
    Entity entity = new();

    entity.MapDto(request);

    Assert.Equal("unchanged", entity.Count);
    Assert.Equal(new[] { "existing" }, entity.Children);
    Assert.Equal("unchanged", entity.ReadOnly);
}

/// <summary>
/// Supplies input properties for entity mapping tests.
/// </summary>
public sealed class Input
{
    public string? Name { get; set; }
    public bool Enabled { get; set; }
    public int Count { get; set; }
    public List<string> Children { get; set; } = [];
    public string ReadOnly { get; set; } = "";
}

/// <summary>
/// Supplies destination properties for entity mapping tests.
/// </summary>
public sealed class Entity
{
    public string? Name { get; set; }
    public bool Enabled { get; set; }
    public string Count { get; set; } = "unchanged";
    public List<string> Children { get; set; } = ["existing"];
    public string ReadOnly { get; private set; } = "unchanged";
}
}

```

Unit/LicenseValidityTests.cs

```

using BridgeControl.Models.Vendor;
using BridgeControl.Services.Vendor;

```

```

namespace BridgeControl.Tests.Unit;

/// <summary>
/// Verifies license validity behavior and regression scenarios.
/// </summary>
[Trait("Category", "Unit")]
public class LicenseValidityTests
{
    /// <summary>
    /// Verifies that validity start is inclusive end is exclusive.
    /// </summary>
    [Theory]
    [InlineData(-1, false)]
    [InlineData(0, true)]
    [InlineData(59, true)]
    [InlineData(60, false)]
    [InlineData(61, false)]
    public void Validity_StartIsInclusive_EndIsExclusive(int seconds, bool expected)
    {
        DateTimeOffset start = new(2026, 9, 16, 12, 0, 0, TimeSpan.Zero);
        License license = new() { ValidFrom = start, ValidUntil = start.AddSeconds(60) };
        Assert.Equal(expected, LicenseEligibility.ValidAt(license,
start.AddSeconds(seconds)));
        license.Status = "Disabled"; Assert.False(LicenseEligibility.ValidAt(license,
start.AddSeconds(seconds)));
    }
}

```

Unit/PagedResponseExtensionsTests.cs

```

using BridgeControl.DTOs.Shared;
using BridgeControl.Services.Shared;

namespace BridgeControl.Tests.Unit;

public class PagedResponseExtensionsTests
{
    [Fact]
    public async Task MappingPreservesMetadataAndAwaitsEachItemBeforeStartingTheNext()
    {
        PagedResponse<int> source = new() { Items = [3, 4], Page = 2, PageSize = 2,
TotalCount = 9 };
        List<int> completed = [];
        bool active = false;
        var result = await source.MapAsync(async (item, ct) =>
        {
            Assert.False(active);
            active = true;
            await Task.Yield();
            completed.Add(item);
            active = false;
            return item.ToString();
        });
        Assert.Equal(new[] { 3, 4 }, completed);
        Assert.Equal(new[] { "3", "4" }, result.Items);
        Assert.Equal(2, result.Page);
        Assert.Equal(2, result.PageSize);
        Assert.Equal(9, result.TotalCount);
        Assert.Equal(new[] { 3, 4 }, source.Items);
    }
}

```

```

    }

    [Fact]
    public async Task CancellationStopsBeforeMappingTheNextItem()
    {
        using CancellationTokenSource cancellation = new();
        PagedResponse<int> source = new() { Items = [1, 2] };
        int calls = 0;
        await Assert.ThrowsAnyAsync<OperationCanceledException>(() =>
source.MapAsync((item, ct) =>
    {
        Assert.Equal(cancellation.Token, ct);
        calls++;
        cancellation.Cancel();
        return Task.FromResult(item);
    }, cancellation.Token));
        Assert.Equal(1, calls);
    }
}

```

Unit/PasswordServiceTests.cs

```

using BridgeControl.Models.Platform;
using BridgeControl.Services.Shared;
using Microsoft.AspNetCore.Identity;

namespace BridgeControl.Tests.Unit;

/// <summary>
/// Verifies password service behavior and regression scenarios.
/// </summary>
[Trait("Category", "Unit")]
public class PasswordServiceTests
{
    /// <summary>
    /// Verifies that hash and verify preserve the exact password and use distinct salts.
    /// </summary>
    [Fact]
    public void HashAndVerify_PreserveTheExactPasswordAndUseDistinctSalts()
    {
        PasswordService<PlatformAdmin> passwords = new(new
PasswordHasher<PlatformAdmin>());
        PlatformAdmin account = new();
        const string password = " 12345678 ";

        string hash = passwords.Hash(account, password);

        Assert.NotEqual(password, hash);
        Assert.NotEqual(hash, passwords.Hash(account, password));
        Assert.Equal(PasswordVerificationResult.Success, passwords.Verify(account, hash,
password));
        Assert.Equal(PasswordVerificationResult.Failed, passwords.Verify(account, hash,
password.Trim()));
    }

    /// <summary>
    /// Verifies that verify missing or malformed stored hash fails.
    /// </summary>
    [Theory]
    [InlineData(null)]
}

```

```

        [InlineData("")]
        [InlineData("invalid hash")]
        public void Verify_MissingOrMalformedStoredHashFails(string? hash)
        {
            PasswordService<PlatformAdmin> passwords = new(new
PasswordHasher<PlatformAdmin>());
            Assert.Equal(PasswordVerificationResult.Failed, passwords.Verify(new
PlatformAdmin(), hash, "1234567890"));
        }
    }
}

```

Unit/PrivateDirectoryTests.cs

```

using System.Security.AccessControl;
using System.Security.Principal;
using BridgeControl.Services.Shared;

namespace BridgeControl.Tests.Unit;

/// <summary>
/// Verifies private directory behavior and regression scenarios.
/// </summary>
[Trait("Category", "Unit")]
public class PrivateDirectoryTests
{
    /// <summary>
    /// Verifies that ensure rejects an existing shared directory without changing its
permissions.
    /// </summary>
    [Fact]
    public void Ensure_RejectsAnExistingSharedDirectoryWithoutChangingItsPermissions()
    {
        string path = Path.Combine(Path.GetTempPath(), "BridgeControl_DirectoryTest_" +
Guid.NewGuid().ToString("N"));
        try
        {
            {
                DirectoryInfo directory = new(PrivateDirectory.Ensure(path));
                Assert.Equal(path, PrivateDirectory.Ensure(path));
                if (OperatingSystem.IsWindows())
                {
                    DirectorySecurity shared = directory.GetAccessControl();
                    shared.AddAccessRule(new FileSystemAccessRule(new
SecurityIdentifier(WellKnownSidType.BuiltinUsersSid, null), FileSystemRights.Read,
AccessControlType.Allow));
                    directory.SetAccessControl(shared);
                    string before =
directory.GetAccessControl().GetSecurityDescriptorSddlForm(AccessControlSections.Access);
                    Assert.Throws<InvalidOperationException>(() =>
PrivateDirectory.Ensure(path));
                    Assert.Equal(before,
directory.GetAccessControl().GetSecurityDescriptorSddlForm(AccessControlSections.Access));
                }
                else
                {
                    UnixFileMode shared = File.GetUnixFileMode(path) |
UnixFileMode.GroupRead;
                    File.SetUnixFileMode(path, shared);
                    Assert.Throws<InvalidOperationException>(() =>
PrivateDirectory.Ensure(path));
                    Assert.Equal(shared, File.GetUnixFileMode(path));
                }
            }
        }
    }
}

```

```

        }
    }
    finally
    {
        // The exact fresh directory above contains no files or subdirectories.
        if (Directory.Exists(path)) { Directory.Delete(path); }
    }
}
}

```

Unit/RefreshCredentialServiceTests.cs

```

using BridgeControl.Services.Shared;

namespace BridgeControl.Tests.Unit;

/// <summary>
/// Verifies refresh credential service behavior and regression scenarios.
/// </summary>
[Trait("Category", "Unit")]
public class RefreshCredentialServiceTests
{
    /// <summary>
    /// Verifies that create and try hash produce distinct credentials and repeatable
    verification data.
    /// </summary>
    [Fact]
    public void CreateAndTryHash_ProduceDistinctCredentialsAndRepeatableVerificationData()
    {
        RefreshCredentialService credentials = new();
        string credential = credentials.Create();

        Assert.NotEqual(credential, credentials.Create());
        Assert.True(credentials.TryHash(credential, out byte[] hash));
        Assert.True(credentials.TryHash(credential, out byte[] repeated));
        Assert.Equal(32, hash.Length);
        Assert.Equal(hash, repeated);
    }

    /// <summary>
    /// Verifies that try hash rejects malformed or noncanonical credentials.
    /// </summary>
    [Theory]
    [InlineData(null)]
    [InlineData("")]
    [InlineData("too-short")]
    [InlineData("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!")]
    [InlineData("AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAB")]
    public void TryHash_RejectsMalformedOrNoncanonicalCredentials(string? credential)
    {
        RefreshCredentialService credentials = new();
        Assert.False(credentials.TryHash(credential, out byte[] hash));
        Assert.Empty(hash);
    }
}

```

Unit/SchemaArtifactValidationServiceTests.cs

```

using System.Text.Json;
using BridgeControl.Api.Exceptions;
using BridgeControl.Services.Vendor;
using Microsoft.Extensions.DependencyInjection;

namespace BridgeControl.Tests.Unit;

/// <summary>
/// Verifies schema artifact validation service behavior and regression scenarios.
/// </summary>
[Trait("Category", "Unit")]
public class SchemaArtifactValidationServiceTests
{
    /// <summary>
    /// Verifies that additional dialect can be registered without changing callers.
    /// </summary>
    [Fact]
    public async Task AdditionalDialectCanBeRegisteredWithoutChangingCallers()
    {
        ServiceCollection services = new();
        services.AddSingleton<ISchemaArtifactValidator,
Draft202012SchemaArtifactValidator>();
        services.AddSingleton<ISchemaArtifactValidator, TestDialectValidator>();
        services.AddSingleton<SchemaArtifactValidationService>();
        using ServiceProvider provider = services.BuildServiceProvider();
        SchemaArtifactValidationService validation =
provider.GetRequiredService<SchemaArtifactValidationService>();
        using JsonDocument current =
JsonDocument.Parse("""{"$schema":"https://json-schema.org/draft/2020-12/schema","type":"object"}"
""");
        validation.Validate(current.RootElement, default);

        string path = Path.Combine(Path.GetTempPath(),
$"BridgeControl_schema_{Guid.NewGuid():N}.json");
        try
        {
            await File.WriteAllTextAsync(path,
""""{"$schema":"urn:bridgecontrol:test-dialect","testKeyword":true}""");
            await validation.ValidateAsync(path, default);
            await File.WriteAllTextAsync(path,
""""{"$schema":"urn:bridgecontrol:test-dialect"}""");
            await Assert.ThrowsAsync<ArtifactValidationException>(() =>
validation.ValidateAsync(path, default));
        }
        finally { File.Delete(path); }
    }

    /// <summary>
    /// Verifies that missing or unregistered dialect does not fall back to another validator.
    /// </summary>
    [Theory]
    [InlineData("{}")]
    [InlineData("{\"$schema\":null\"}")]
    [InlineData("{\"$schema\":\"https://unknown.example/schema\"}")]
    public void MissingOrUnregisteredDialectDoesNotFallBackToAnotherValidator(string json)
    {
        SchemaArtifactValidationService validation = new([new
Draft202012SchemaArtifactValidator()]);
        using JsonDocument document = JsonDocument.Parse(json);
        Assert.Throws<ArtifactValidationException>(() =>
validation.Validate(document.RootElement, default));
    }
}

```

```

    /// <summary>
    /// Verifies that ambiguous dialect registrations are rejected.
    /// </summary>
    [Fact]
    public void AmbiguousDialectRegistrationsAreRejected()
        => Assert.Throws<InvalidOperationException>(() => new
SchemaArtifactValidationService(
        [new Draft202012SchemaArtifactValidator(), new
Draft202012SchemaArtifactValidator()]));

    /// <summary>
    /// Records schema validation calls for dialect delegation tests.
    /// </summary>
    private sealed class TestDialectValidator : ISchemaArtifactValidator
    {
        public string Dialect => "urn:bridgecontrol:test-dialect";
        /// <summary>
        /// Records a call to the test schema dialect validator.
        /// </summary>
        public void Validate(JsonElement schema, CancellationToken ct)
        {
            ct.ThrowIfCancellationRequested();
            if (!schema.TryGetProperty("testKeyword", out JsonElement keyword) ||
keyword.ValueKind != JsonValueKind.True)
                { throw new ArtifactValidationException("The test dialect requires
testKeyword."); }
        }
    }
}

```

Unit/WidgetSchemaTests.cs

```

using System.Text.Json;
using System.Text.Json.Nodes;
using BridgeControl.Api.Exceptions;
using BridgeControl.Services.Vendor;

namespace BridgeControl.Tests.Unit;

/// <summary>
/// Verifies widget schema behavior and regression scenarios.
/// </summary>
[Trait("Category", "Unit")]
public class WidgetSchemaTests
{
    private readonly SchemaArtifactValidationService schemas = new([new
Draft202012SchemaArtifactValidator()]);

    /// <summary>
    /// Parses the supplied JSON into an independent test value.
    /// </summary>
    private static JsonElement Json(string text) =>
JsonDocument.Parse(text).RootElement.Clone();

    private static JsonElement Schema =>
Json(File.ReadAllText(Path.Combine(AppContext.BaseDirectory, "Fixtures",
"pslWidget.schema.json")));

    /// <summary>
    /// Verifies that exact uploaded schema is accepted.
    /// </summary>
    [Fact]
}

```

```

        public async Task ExactUploadedSchemaIsAccepted() => await
schemas.ValidateAsync(Path.Combine(AppContext.BaseDirectory, "Fixtures",
"pslWidget.schema.json"), default);

    /// <summary>
    /// Verifies that configuration rules are enforced.
    /// </summary>
    [Theory]
    [InlineData("{} ", true)]

    [InlineData("""{"ScriptSources":[{"Path":"C:\\Scripts","Extensions":[".ps1","*.custom","cmd"]}],"
KeyboardNavigation":{"ActivationShortcut":"Ctrl+Alt+L","UpKey":"DeadCharProcessed"},"Position":{"
Offset":-1.5e+2}}""", true)]

    [InlineData("""{"WebsiteLinks":{"Items":[{"Kind":"Folder","Name":"Tools","Children":[{"Kind":"Fol
der","Name":"Nested","Children":[{"Kind":"Link","Name":"Docs","Url":"https://example.com/docs"}]}
]}]}""", true)]
        [InlineData("""{"ScriptSources":[{"Path":"  "}]""", false)]
        [InlineData("""{"ScriptSources":[{"Path":"x","Extensions":["*.bad ext"]}""", false)]
        [InlineData("""{"KeyboardNavigation":{"ActivationShortcut":"Ctrl+Unknown"}}""", false)]
        [InlineData("""{"KeyboardNavigation":{"UpKey":"Unknown"}}""", false)]
        [InlineData("""{"Position":{"Offset":"one"}}""", false)]

    [InlineData("""{"WebsiteLinks":{"Items":[{"Kind":"Folder","Name":"Tools","Children":[{"Kind":"Lin
k","Name":"Docs"}]}]}""", false)]

    [InlineData("""{"WebsiteLinks":{"Items":[{"Kind":"Link","Name":"Docs","Url":"https://example.com/
%ZZ"}]}]}""", false)]

    [InlineData("""{"WebsiteLinks":{"Items":[{"Kind":"Link","Name":"Docs","Url":"file:///local"}]}]}""
", false)]

    [InlineData("""{"WebsiteLinks":{"Items":[{"Kind":"Folder","Name":"Tools","Url":"https://example.c
om"}]}]}""", false)]
        public void ConfigurationRulesAreEnforced(string values, bool valid)
        {
            ConfigurationValues validator = new(schemas);
            if (valid) { validator.Validate(Schema, Json(values), default); }
            else { Assert.Throws<ConfigurationValidationException>(() =>
validator.Validate(Schema, Json(values), default)); }
        }
    /// <summary>
    /// Verifies that recursive values still have depth limits.
    /// </summary>
    [Fact]
    public void RecursiveValuesStillHaveDepthLimits()
    {
        JsonObject node = new() { ["Kind"] = "Folder", ["Name"] = "Nested" };
        for (int i = 0; i < 18; i++) { node = new() { ["Kind"] = "Folder", ["Name"] =
"Nested", ["Children"] = new JsonArray(node) }; }
        JsonElement values = Json(new JsonObject { ["WebsiteLinks"] = new JsonObject {
["Items"] = new JsonArray(node) } }).ToJsonString();
        Assert.Throws<ConfigurationValidationException>(() => new
ConfigurationValues(schemas).Validate(Schema, values, default));
    }
    /// <summary>
    /// Verifies that non descending cycles and missing references are rejected.
    /// </summary>
    [Theory]

    [InlineData("""{"$defs":{"a":{"allOf":[{"$ref":"#/$defs/b"}]},"b":{"$ref":"#/$defs/a"}}""")]
    [InlineData("""{"$defs":{"a":{"properties":{"child":{"$ref":"#/$defs/missing"}}}}""")]

```

```

    public void NonDescendingCyclesAndMissingReferencesAreRejected(string body)
    {
        var schema = JsonNode.Parse(body)!.AsObject(); schema["$schema"] =
Draft202012SchemaArtifactValidator.DialectUri;
        Assert.Throws<ArtifactValidationException>(() =>
schemas.Validate(Json(schema.ToJsonString()), default));
    }
    /// <summary>
    /// Verifies that nested quantifiers use bounded matching.
    /// </summary>
    [Fact]
    public void NestedQuantifiersUseBoundedMatching()
    {
        var schema =
Json("""{"$schema":"https://json-schema.org/draft/2020-12/schema","properties":{"text":{"pattern":
:"(a+)+$"}}}""");
        Assert.Throws<ConfigurationValidationException>(() => new
ConfigurationValues(schemas).Validate(schema, JsonSerializer.SerializeToElement(new { text = new
string('a', 10000) + "!" }, default));
    }
    /// <summary>
    /// Verifies that enum and pattern size limits remain enforced.
    /// </summary>
    [Fact]
    public void EnumAndPatternSizeLimitsRemainEnforced()
    {
        var schema =
JsonNode.Parse("""{"$schema":"https://json-schema.org/draft/2020-12/schema"}""").AsObject();
        schema["enum"] = JsonSerializer.SerializeToNode(Enumerable.Range(0, 512));
        schemas.Validate(Json(schema.ToJsonString()), default);
        schema["enum"]!.AsArray().Add(512);
        Assert.Throws<ArtifactValidationException>(() =>
schemas.Validate(Json(schema.ToJsonString()), default));
        schema.Remove("enum"); schema["pattern"] = new string('a', 4097);
        Assert.Throws<ArtifactValidationException>(() =>
schemas.Validate(Json(schema.ToJsonString()), default));
    }
    /// <summary>
    /// Verifies that recursive branch expansion still has a work limit.
    /// </summary>
    [Fact]
    public void RecursiveBranchExpansionStillHasAWorkLimit()
    {
        JsonElement schema =
Json("""{"$schema":"https://json-schema.org/draft/2020-12/schema","$ref":"#/$defs/tree","$defs":{
"tree":{"allOf":[{"properties":{"child":{"$ref":"#/$defs/tree"}}},{ "properties":{"child":{"$ref":
"#/$defs/tree"}}}]}}""");
        JsonObject value = new();
        for (int i = 0; i < 24; i++) { value = new() { ["child"] = value }; }
        Assert.Throws<ConfigurationValidationException>(() => new
ConfigurationValues(schemas).Validate(schema, Json(value.ToJsonString()), default));
    }
    /// <summary>
    /// Verifies that cancellation is honored before recursive validation.
    /// </summary>
    [Fact]
    public void CancellationIsHonoredBeforeRecursiveValidation()
    {
        using CancellationTokenSource cancellation = new(); cancellation.Cancel();
        Assert.Throws<OperationCanceledException>(() => new
ConfigurationValues(schemas).Validate(schema, Json("{}"), cancellation.Token));
    }
}

```

