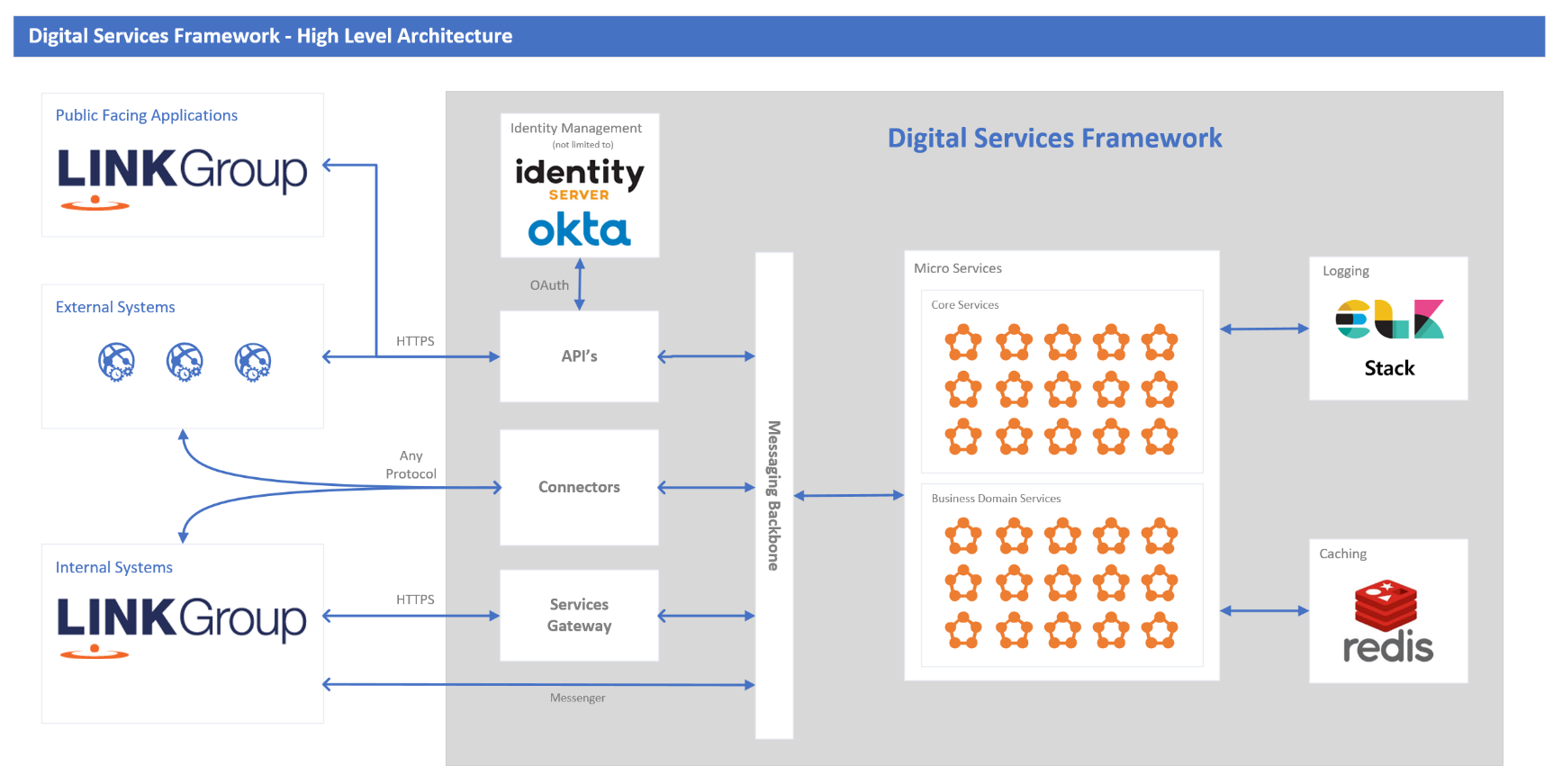
**Concepts**

The **Digital Services Framework** (usually abbreviated to ***DSF***) is a collection of **Micro-Services** hosted on the **Azure Service Fabric**. These **Micro-Services** are designed to be self-contained and as such are independently deployable, maintainable and scalable.

Typically, services implement business functionality, however there are some types of service that offer core and reusable features that can be used by any business unit:



Communications between services is achieved through the use of the **Message Oriented Middleware** pattern (commands and queries) utilising the **Azure Service Bus** and supported by the **Messenger** middleware component. The use of message based communication allows for offline and rolling upgrades, load smoothing and technology independence.

Holistic **Logging** is achieved through the use of a centralised function with high throughput and large capacity. This allows multi-service insight during development and production activities.

Services also have access to a shared **Fast Data Cache** allowing them to take advantage of static or slow moving data by bringing it closer to the consumer resulting in increased performance.

Services can utilize external capabilities of other systems, both internally or externally hosted, through **Connectors**. These have a one-to-one relationship with the connected system and abstract away implementation complexities offering the same standardised and familiar internal consumption via message based communications.

Consumers that exist outside the boundary of the fabric can access services through the **Services API**. The **Services API** accepts JSON representations of service messages in a RESTful manner. This itself is protected by the **Identity Server** ensuring functional ability is limited to those that have the right to invoke at a message level.

**Apps** and **APIs** build upon the Services API to offer user-experience through applications (web, mobile, etc.) alternatively inter-business (B2B) connectivity is achieved through APIs directly. This abstraction allows a decoupling of the DSF core technologies and allows the adoption of common web frameworks and protocols.

Overview

# Setting Up Your Test Rig

The test rig setup required for Developers in Test closely resembles the Developers' environment setup. Most of the steps detailed in the [Developer](https://10.243.110.58/platform/docs/developer/getting-started/setting-up-your-development-environment/overview) section apply to Testers as well. This section details the additional steps that are specific to testing such as configuring SpecFlow and installing DSF test templates.

## Prerequisites[​](https://10.243.110.58/platform/docs/tester/getting-started/setting-up-your-test-rig/overview#prerequisites)

Follow the instructions in the [Developer - Getting Started - Setting up your development environment](https://10.243.110.58/platform/docs/developer/getting-started/setting-up-your-development-environment/overview) section and perform the following steps:

1. Configure and connect your VDI
2. Install Git
3. Install a Git Client such as SourceTree
4. Install Visual Studio
5. Configure NuGet feeds
6. Install SonarLint
7. Install the required target frameworks

## Additional steps[​](https://10.243.110.58/platform/docs/tester/getting-started/setting-up-your-test-rig/overview#additional-steps)

1. Install the [SpecFlow extension](https://10.243.110.58/platform/docs/tester/getting-started/setting-up-your-test-rig/specflow-extension) for Visual Studio
2. Install the [DSF test templates](https://10.243.110.58/platform/docs/tester/getting-started/setting-up-your-test-rig/dsf-test-templates)
3. Install the [Visual Studio spell checker](https://10.243.110.58/platform/docs/tester/getting-started/setting-up-your-test-rig/visual-studio-spell-checker)

# Target Frameworks

Automation test projects require the same .Net Core and .Net Framework SDKs used for production code.

# SpecFlow Extension

This is required for integration with Visual Studio and is sponsored by TechTalk. It works with Visual Studio 2017 or higher. Earlier versions of Visual Studio are not supported.

The SpecFlow extension for Visual Studio should not be confused with SpecFlow NuGet packages. SpecFlow package references are automatically added by DSF Test Templates when a new project is created. The required packages are usually referenced indirectly via the Dsf.TestFramework.SpecFlow component. Both the extension for Visual Studio and the NuGet references are required to design and execute tests within the DSF test framework, but, unlike SpecFlow package references, which are automatically added to each test project, the extension needs to be manually installed only once for each version of Visual Studio on your VDI.

**DSF Test Templates**

DSF test templates are published as a single Visual Studio extension package (VSIX).

The package is compatible with Visual Studio 2017 and Visual Studio 2019.

The package includes the following:

* Project templates: service tests, API test, performance tests
* Item templates: feature files, step definition classes

# Test Code Management Overview

In this section we will look at how test code is managed within the DSF. For the most part, test code management follows the same guidelines as production code managment, but with some key differences. This section details the additional guidelines that are specific to test code. Topics include: creation of test code repositories, branching strategy, and merge requests.

We use GitLab to manage our test code repositories. As a technical tester or software developer in test, you will design new automated tests or update existing ones in a feature branch, and push your local commits to the feature branch on the remote Git server.

When you have finished working on a story or work item and all the tests pass, you should raise a merge request in GitLab so that you feature branch is merged into the parent develop branch.

When your team has finished working on the project and all the tests pass, including all regression tests, then a member of project team should be raise a merge request in GitLab so that the develop branch is merged into the release branch.

# Tester: Repositories Overview

The automated test for each DSF service, API or UI application (excluding unit tests) are contained in a separate Git repository. The structure of test code repositories closely mirrors that of production code repositories.

# Tester: Repositories Organization

Testers' repositories are grouped in GitLab and TeamCiy, in the same way as the corresponding developers' repositories. See [here](https://10.243.110.58/platform/docs/developer/code-management/repositories/organization).

## Repository name[​](https://10.243.110.58/platform/docs/tester/test-code-management/repositories/organization#repository-name)

This is the name of the Service, API or application under test, followed by **.tests**, all in lower case. For example, if you are testing the Calendar service, then you should push your scripts to the calendar.tests repository.

It is possible, but unusual, for two different repositories to have the same name but to belong to different groups. For example, there are fund.tests repositories both in the APAC and the EMEA jurisdiction.

## Repository type[​](https://10.243.110.58/platform/docs/tester/test-code-management/repositories/organization#repository-type)

To each type of repository there corresponds a type of automation test suite:

| **Repository type** | **Test type** |
| --- | --- |
| Applications | UI tests |
| APIs | REST API tests |
| Connectors | Connector message tests |
| Services | Service message tests |
| Components | Component tests |

Web application test repositories hold UI automation scripts, usually designed using the Selenium Web Driver.

API test repositories are used to store RestSharp, Postman and JMeter scripts.

Service and connector tests repositories are used for DSF message tests.

Component tests repositories are owned by the Test Architecture team and are not normally used as part of project work.

## Examples[​](https://10.243.110.58/platform/docs/tester/test-code-management/repositories/organization#examples)

core / services / calendar.tests

* Jurisdiction: Core
* Type: Service message tests

emea / bus / corporate-markets / connectors / fidelity.tests

* Jurisdiction: EMEA
* Business unit: Corporate Markets
* Type: Connector message tests

# Tester: Requesting Access to a Repository

All DSF repositories, including test code repositories, are set as private by default. You will not be able to see a repository you have no permissions for.

## Test code repositories[​](https://10.243.110.58/platform/docs/tester/test-code-management/repositories/requesting-access-to-a-repository#test-code-repositories)

To request access to a test code repository, send an email to the Test Architects, with the following information:

* Subject: [GitLab Role] access to [repository name]
* To:
  + Repository maintainer or owner, if known
  + Otherwise: [LGE-DL-TOUK-ITSD-TestArchitects@linkgroup.co.uk](mailto:LGE-DL-TOUK-ITSD-TestArchitects@linkgroup.co.uk)
* Type: service, connector, API or application tests
* Project code
* Full GIT path to the repository, if known

Example:

* Subject: Developer access to reportmanager.tests
* To: [LGE-DL-TOUK-ITSD-TestArchitects@linkgroup.co.uk](mailto:LGE-DL-TOUK-ITSD-TestArchitects@linkgroup.co.uk)
* Type: Service tests
* Project code: DCI
* GIT path: <https://gitlab01.casfs.co.uk/digital-services-framework/core/services/reporting.reportmanager.tests>

### Repository name and type[​](https://10.243.110.58/platform/docs/tester/test-code-management/repositories/requesting-access-to-a-repository#repository-name-and-type)

Usually the Service name, API name or application name will suffice to locate a repository. But repositories are also grouped by Jurisdiction (EMEA, APAC, Core) and Business Unit. If you know the repository group or full GIT path, provide this information when requesting access.

### Project code[​](https://10.243.110.58/platform/docs/tester/test-code-management/repositories/requesting-access-to-a-repository#project-code)

This is required to verify that the repository holds the develop branch relevant to your project. If not, it will be created and protected by the repository maintainer.

The project code is the first group of three letters in the build number of the pre-release build under test. Pre-release builds are usually deployed to a TST or PAT environment. Numbers of deployed builds can be found in Octopus for each test environment. For example, if you are testing the 2021.1.2-abc-dev build of the Fund service on a TST environment, then the project code is "abc".

### GitLab role[​](https://10.243.110.58/platform/docs/tester/test-code-management/repositories/requesting-access-to-a-repository#gitlab-role)

You will have different permissions depending on the GitLab role you have in an individual repository or group of repositories. If a you are a member of both the repository's group and the individual repository itself, then the highest role is used.

The commonly used roles are as follows:

| **GitLab role** | **Permissions** | **Tester role** |
| --- | --- | --- |
| Reporter | Read-only access | Test executor |
| Developer | Write-access | Test designer |
| Maintainer | Management-level access | Test certifier |

* The Reporter role allows to view, configure and existing tests.
* The Developer role allows to modify existing tests and design new ones.
* The Maintainer role allows to accept merge request, grant or restrict access to others, and protect branches.

Developers in test will normally need the Developer role.

Tester who only need to view or merely execute the existing tests can request the Reporter role.

If you believe you need the "Maintainer" role, provide a reason why.

### GIT path[​](https://10.243.110.58/platform/docs/tester/test-code-management/repositories/requesting-access-to-a-repository#git-path)

The full GIT path of a repository can be found in Team City, as explained below. This is useful when you do not have access to the repository but you need to locate it or verify it exists.

1. Log into the TeamCity portal, <http://teamcity01.casfs.co.uk/>
2. Enter the name of the service, API or application in the search box.
3. View the latest build.
4. Find the full GIT path in the build logs, under "VCS root details". Sometimes the GIT path is also visible in the change logs.

Alternatively, just note the Jurisdiction, Business Unit and Type of the relevant Team City project, and provide this information in the email to the Test Architecture team.

## Production code repositories[​](https://10.243.110.58/platform/docs/tester/test-code-management/repositories/requesting-access-to-a-repository#production-code-repositories)

Testers should never request write access ("developer role") to repositories used for production code or for candidate production code.

Occasionally, it might be useful for testers to request ready-only access ("reporter role"), for example, to use test clients created by developers.

If you believe that you would benefit from accessing a DSF service, API, or application repository, send an email to the Test Architects with the below details:

* Subject: Reporter access to [repository name]
* To: [LGE-DL-TOUK-ITSD-TestArchitects@linkgroup.co.uk](mailto:LGE-DL-TOUK-ITSD-TestArchitects@linkgroup.co.uk)
* Project code
* GIT path, if known
* Reason for requesting access

If you request is reasonable, it will be forwarded to the developer or architect who owns the repository.

# Requesting a New Test Repository

Test repositories should be created when the corresponding service, API, or application repository is created by the Architecture team.

If you cannot see a test repository, it may because you do not have access to it.

If you believe that the repository has not been created yet, you should still request access as detailed [here](https://10.243.110.58/platform/docs/tester/test-code-management/repositories/requesting-access-to-a-repository). In your request, note that a new repository may need to be created, and explain the reason why.

If the required repository does not exist, it will be created and you will be granted access to it. Otherwise you will be directed to the relevant existing repository.

## DON'Ts[​](https://10.243.110.58/platform/docs/tester/test-code-management/repositories/requesting-a-new-test-repository#donts)

If the required test repository does not seem to be available, you should always follow the process detailed above.

Do not attempt any of the following workarounds:

❌ Do not use a personal repository for project work.

❌ Do not leave your scripts on a local workspace, outside of source control. Push your work as often as you can, and follow the merge request process.

❌ Do not use another repository other than that associated to the service, API, application under test.

If you use the wrong repository, your merge requests will be rejected and you will be asked to migrate your test code to the correct repository.

If you are in doubt, confirm with the repository maintainer or the Test Architecture team before you create your feature branch.

# Configuring a Test Repository

This section is exclusively for maintainers of test code repositories. You need to read this section only if you intend to request maintainer role when a new repository is created.

Follow the below steps to configure a newly created repository in GitLab:

* Step 1: Add gitignore
* Step 2: Protect master and release branches
* Step 3: Set maintainers and developers
* Step 4: Create the develop branch

## Step 1: Add gitignore[​](https://10.243.110.58/platform/docs/tester/test-code-management/repositories/configuring-a-test-repository" \l "step-1-add-gitignore" \o "Direct link to Step 1: Add gitignore)

### How to obtain gitignore[​](https://10.243.110.58/platform/docs/tester/test-code-management/repositories/configuring-a-test-repository" \l "how-to-obtain-gitignore" \o "Direct link to How to obtain gitignore)

Get the latest version of gitignore for Visual Studio from: <https://github.com/github/gitignore/blob/master/VisualStudio.gitignore>

Alternatively, you can obtain gitignore from GitLab templates, as follows:

* Open the repository project in the GitLab portal
* View the root directory of the master branch
* Click the plus sign to add a new file, and select This directory > New file
* Click the Select a template type list box and select .gitignore
* Click Apply a template, type VisualStudio in the filter box, and select Languages > VisualStudio

### Ignore SpecFlow auto-generated files[​](https://10.243.110.58/platform/docs/tester/test-code-management/repositories/configuring-a-test-repository#ignore-specflow-auto-generated-files)

Add the following two lines at the top of the gitignore file, after the comment section:

#Specflow auto-generated files  
\*.feature.cs

### Push gitignore to master[​](https://10.243.110.58/platform/docs/tester/test-code-management/repositories/configuring-a-test-repository#push-gitignore-to-master)

Ensure that the file name is **.gitignore** and that the file is added to the root directory.

Commit and push the file to the master branch.

## Step 2: Protect master and release branches[​](https://10.243.110.58/platform/docs/tester/test-code-management/repositories/configuring-a-test-repository#step-2-protect-master-and-release-branches)

Branch release off the master branch.

The master and release branches need protecting so that only designated maintainers can merge into/change them.

This can be done in the Protected Branches section within the Settings > Repository menu option, as follows:

* Within the Protected branches section, select both the master and release branch in turn.
* Set Allowed to merge to Maintainers.
* Set Allowed to push to Maintainers.

## Step 3: Set maintainers and developers[​](https://10.243.110.58/platform/docs/tester/test-code-management/repositories/configuring-a-test-repository#step-3-set-maintainers-and-developers)

This can be done in the Members section within the Settings > Members menu option.

The Maintainer role should be given to senior or lead technical testers.

The Developer role should be given to any technical tester or developer in test who will be designing new tests within the project.

The Reporter role should be given to any tester or developer that who need to execute the tests but will not be making any changes within the project.

## Step 4: Create the develop branch[​](https://10.243.110.58/platform/docs/tester/test-code-management/repositories/configuring-a-test-repository#step-4-create-the-develop-branch)

The repository is now setup. You can now branch off the project develop branch from release.

Develop branches can be protected at maintainers' discretion.

# Tester: Personal Repositories

Personal repositories should never be used for project work. Instead, follow the process to [request access to a test repository](https://10.243.110.58/platform/docs/tester/test-code-management/repositories/requesting-access-to-a-repository).

* A **personal repository** is a GitLab repository associated to a specific username. Example: [git@gitlab01.casfs.co.uk](mailto:git@gitlab01.casfs.co.uk):JBloggs/mytesttool.git
* A **DSF repository** is a GitLab repository that belongs to the “digital-service-framework” group. Example: [git@gitlab01.casfs.co.uk](mailto:git@gitlab01.casfs.co.uk):digital-services-framework/core/services/reporting.reportmanager.tests.git

✅ You can use your own personal repository for learning purposes, proofs of concept, or for experimenting with new or unfamiliar tools outside of normal project work.

❌ Do not create personal repositories to test DSF services, APIs or applications.

## Creating a personal repository[​](https://10.243.110.58/platform/docs/tester/test-code-management/repositories/personal-repositories#creating-a-personal-repository)

Log into GitLab.

Click on the New Project button. It will take you the New Project creation page.

Enter the project name, all in lower case.

You can write a brief description of the project. This is optional.

Visibility level should be set to Private.

Tick the following checkbox: Initialize repository with a README.

Finally, click the Create project button.

## Requesting to move a personal repository to the DSF group[​](https://10.243.110.58/platform/docs/tester/test-code-management/repositories/personal-repositories#requesting-to-move-a-personal-repository-to-the-dsf-group)

If you believe that an existing personal repository should be moved to the DSF group, follow the steps below:

1. Ensure that the repository structure is in line with the existing DSF test repositories.
2. Ensure that the test code reasonably meets the [test coding standards](https://10.243.110.58/platform/docs/tester/test-coding-standards/overview).
3. Ensure that the release and master branches are up to date and that there are no stale branches.
4. Determine the candidate location, including Jurisdiction (EMEA, APAC or Core), Business Unit, and repository type such as Service tests or API tests.
5. Send an email to the Test Architecture team with the following information:

* Subject: Request to move personal repository to the DSF group
* To: [LGE-DL-TOUK-ITSD-TestArchitects@linkgroup.co.uk](mailto:LGE-DL-TOUK-ITSD-TestArchitects@linkgroup.co.uk)
* Full GIT path to the personal repository
* Proposed new location: Jurisdiction / Business Unit / Type
* Project code, if the repository migration is required for project work, otherwise note that this is required for continuous improvement or to remove test tech debt.

If the personal repository meets DSF testing guidelines, your request will be forwarded to a GitLab administrator or to the relevant project group owner.

## Disadvantages of personal repositories[​](https://10.243.110.58/platform/docs/tester/test-code-management/repositories/personal-repositories#disadvantages-of-personal-repositories)

Personal repositories are never suitable for project work, for the following reasons:

1. Lack of visibility: personal repositories do not fall under the DSF group permissions and, by default, are visible to only to their owners and to GitLab administrators.
2. Security risk: when the creator of a personal repository is removed from the list of active GitLab users, the owner role will remain associated to a blocked username.
3. No Continuous Integration: personal repositories cannot be used to create Continuous Integration pipelines and cannot be linked to TeamCity using the VCS Root configuration of DSF build templates.
4. Lack of test reporting: personal repositories are excluded from most DSF test reporting tools.
5. Quotas: there is a restriction of 10 personal repositories per user.

# Branching Strategy Overview

The test code base follows the same branching strategy as production code. This section details the branching structure, naming conventions and the branch types from a testing perspective.

# Test Code Management: Branching Structure

The branch tiers used by testers mirror those described in the [Developer](https://10.243.110.58/platform/docs/developer/code-management/branching-strategy/branches) section.

If you are the maintainer of a test code repository, enforce the below guidelines. Ensure that the names of project-specific release and develop branches match those in the corresponding production code repository used by developers.

## Branches and test environments[​](https://10.243.110.58/platform/docs/tester/test-code-management/branching-strategy/branching-structure#branches-and-test-environments)

To each branch type there corresponds a test environment type and a build number convention, as follows:

| **Level** | **Branch** | **Test environment** | **Build type** |
| --- | --- | --- | --- |
| Production | master | ppd | Release |
| Scheduled for release | release | uat, per | Release |
| Project release candidate | project/{code}/release | {code}pat | Pre-release: {code}-rel |
| Project development | project/{code}/develop | {code}tst | Pre-release: {code}-dev |
| Work item | project/{code}/feature/{work item} | {code}dev | Pre-release: {code}-dev |

In the above table {code} denotes the three-letter abbreviation of the project name. If the project consists of multiple phases or "drops", then the code will typically track a project stage using two letters followed by a number, for example: ab1, ab2, etc.

## Branches and build types[​](https://10.243.110.58/platform/docs/tester/test-code-management/branching-strategy/branching-structure#branches-and-build-types)

When testing a pre-release build, the same project code should be used in the branch name, the test environment name, and pre-release package references. Assuming, for example, that the project code is 'abc', then the convention is as follows:

| **Branch name** | **Test environment** | **Build under test** |
| --- | --- | --- |
| release | uat | 2022.03.03 |
| project/abc/release | abcpat | 2022.02.02-abc-rel |
| project/abc/develop | abctst | 2022.01.02-abc-dev |
| project/abc/feature/S-12345 | abctst | 2022.01.02-abc-dev |

## Branches and NuGet packages[​](https://10.243.110.58/platform/docs/tester/test-code-management/branching-strategy/branching-structure#branches-and-nuget-packages)

When restoring NuGet packages, do not reference pre-release packages that do not match the project code.

Example:

✅ DO reference either release packages or ab1-dev packages, whichever is the latest, in project/ab1 branches.

❌ DON'T reference ab1-dev packages in project/ab2 branches.

# Test Code Management: Branch Types

This section illustrates the purpose of each branch type from the perspective of test code management.

## Feature branch[​](https://10.243.110.58/platform/docs/tester/test-code-management/branching-strategy/branch-types#feature-branch)

This is a work-item level branch, i.e. the bottom level branch for work-in-progress: individual features or user stories that have not passed testing yet.

The branch naming convention is project/{code}/feature/{work item}\_{your initials}

You will usually work in feature branches and test pre-release builds.

Each time you commit a change locally, you should also push it to the remote feature branch. Commit and push often in order to avoid losing your changes.

Feature branches must be branched off the develop branch for the relevant project or project phase.

When you create a new feature branch, ensure that the project code in the branch name matches the project code in the pre-release suffix of the build number under test.

If the project code of the build under test does not match the test environment name (for example, because DevOps have recycled an existing test environment that was previously used by another project), then the project code in your branch name should still match the code in the build number rather than the test environment name.

Always create your feature branch from a develop branch with a matching project code. Do not branch feature directly off release or off the wrong develop branch. If the develop branch does not exist already, or you are not sure as to what it should be called, then contact the test code repository maintaner.

Unless testing and development proceed exactly in parallel, the tester's branch will usually lag one level behind the developer's branch.

Example: Story S-1 is ready for testing. The developer has merged his feature/S-1 branch into the develop branch and DevOps have deployed the dev build to the test environment. But the tester cannot merge his feature/S-1 branch into develop until it has passed testing. As the developer starts working on the next story, S-2, in the feature/S-2 branch, the tester runs his test in the S-1 feature branch.

A feature branch should be merged into the parent develop branch when the user story passes testing and the work item is marked as 'Done'.

## Develop branch[​](https://10.243.110.58/platform/docs/tester/test-code-management/branching-strategy/branch-types#develop-branch)

This is a project level branch. It is an intermediate branch for features that have passed system testing but are not ready yet for project acceptance testing. At this stage, the dev build under test should be deployed to a system test environment, known as TST (formerly known as SYS or INT).

The branch naming convention is project/{code}/develop

Develop branches are branched off releases branch.

If there are other project-specific branches in the same repository, relating to other projects, and they have not been merged yet into unified release yet, then always double-check with the repository maintainer. Do not branch develop off release unless you are sure that the release branch is up to date.

All regression tests in the develop branch should be automatically executed each time a feature branch is merged into develop. This can be set as continuous integration trigger in TeamCity.

When all testing for a project has completed and the release candidate has been signed off, the project-level develop branch should be merged into release.

## Release branches[​](https://10.243.110.58/platform/docs/tester/test-code-management/branching-strategy/branch-types#release-branches)

These are release candidate branches.

There are two types of release branches:

| **Type** | **Branch name** | **Test environment** |
| --- | --- | --- |
| Unified | release | uat, per |
| Project-specific | project/{code}/release | {code}pat |

At this stage, the release candidate should be deployed to a Project Acceptance Test environment (PAT), if project-specific, or the User Acceptance Test environment (UAT), if scheduled for a unified product release.

If applicable, release candidates should also be deployed to a DSF Certification environment that is suitable for performance testing. The conventional abbreviation for such a test environment is PER.

The automated tests in project-specific release branches should be suitable for PAT environments.

The automated tests in Unified (non project-specific) release should be suitable for PER or UAT environments.

## Master branch[​](https://10.243.110.58/platform/docs/tester/test-code-management/branching-strategy/branch-types#master-branch)

This is the production level branch. At this stage the release build should be in pre-production and production.

By default, the master branch is marked as protected in GitLab, e.g. it cannot be force pushed or deleted.

The master branch is not project-specific. Separate branches for different projects will converge into the same master branch.

You should never work directly in the master branch. You should never merge your test code directly into master.

When a release build is deployed to production, a senior tester should merge the release branch into master.

## Hotfix branches[​](https://10.243.110.58/platform/docs/tester/test-code-management/branching-strategy/branch-types#hotfix-branches)

These are short-lived branches, used to make last-resort changes that need to be urgently deployed to production.

There are two types of hotfix branches:

| **Type** | **Branch name** |
| --- | --- |
| Low-level | hotfix/{reference} |
| High-level | hotfix/release |

These types of branches are rarely used for production code.

Hotfix branches are even more rarely used used for test code. Hotfixes would normally go straight into production without waiting for test execution.

# Test Code Management: Branch Naming Convention

Before you push a new develop or feature branch note the following:

* The build number under test, especially the three-letter project code in the pre-release suffix. For example: 2021.10.245-**ab2**-dev.
* The User Story number corresponding to the feature.

The project-specific branch names should be:

* Parent branch: **project/[project code]/develop**
* Child branch: **project/[project code]/feature/**[user story number]-[your initials]

The part highlighted in bold is especially important. Any departures from the naming convention will cause test code maintenance overhead, will not be supported by TeamCity templates, and will create unnecessary work for repository maintainers.

## Project codes[​](https://10.243.110.58/platform/docs/tester/test-code-management/branching-strategy/branch-naming-convention#project-codes)

When naming a project branch, always refer to the **project code in the pre-release build number**. This uniquely determines the project-specific part of the develop or feature branch name.

The timesheet code or what Service Delivery Managers informally call the project is **not** relevant to the branch name.

Branch names should not be based on test environment names. The project code in the pre-release build number suffix will usually, but not always, match the project code of the TST or PAT environment used to deploy the build. However, if the test environment's project code does not match the build number suffix, always refer to the build's project code instead. Do not use test environments codes to name new branches.

Do not omit the project code when creating a project-specific branch.

Never push branches called 'project/develop' or 'project/feature'. Only product branches such as **master** and (unified) **release** are not associated to project codes.

## Example[​](https://10.243.110.58/platform/docs/tester/test-code-management/branching-strategy/branch-naming-convention#example)

Let's say that:

* you are testing the **ab2-dev** build on the xyztst environment,
* you are working on the US-1234 story,
* you are part of the project team for phase 2 of the so-called "ABC" project,
* and your name is Joe Bloggs.

✅ DO:

* project/ab2/develop
* project/ab2/feature/US-1234-JB

❌ DON'T:

* project/abc/feature/US-1234-JB
* project/ab1/feature/US-1234-JB
* project/phase-2/feature/US-1234-JB
* project/xyz/feature/US-1234-JB
* project/xyztst/feature/US-1234-JB
* project/ab2tst/develop/US-1234-JB
* project/develop/US-1234-JB
* project/abc/develop/feature/US-1234-JB
* project/feature/US-1234-JB

# Tester: Merging Overview

Master and release branches are always protected. Project develop branches may be protected as well, at the repository maintainer's discretion. Only repository maintainers can perform merges into protected branches. This is to ensure test code quality and enforce test code reviews.

# Tester: Requesting a Merge

The merge request process followed by testers is based on that followed by developers.

This section details the additional steps that are specific to testing.

## Prerequisites[​](https://10.243.110.58/platform/docs/tester/test-code-management/merging/requesting_a_merge#prerequisites)

Read the (Requesting a merge)[......\developer\code-management\merging\requesting-a-merge.md] instructions in the Developer section.

## Requesting a merge: from feature into develop[​](https://10.243.110.58/platform/docs/tester/test-code-management/merging/requesting_a_merge#requesting-a-merge-from-feature-into-develop)

Each feature branch should be merged into the parent develop branch when the User Story associated to the feature has passes testing, and all the tasks associated to the story or work item are marked as completed.

Once you have finished working on the story, raise a merge request in GitLab, assign your request to a repository maintainer, and notify all testers in the project team. To notify other team members (other than the merge assignee) add their usernames, prefixed with the @ sign, to the merge description text-box.

If there are any code review comments, address them as part of the existing merge request. Do not close the merge request and open a new one.

Usually you can mark the User Story as "Done" without having to wait for the merge request to be accepted. However, if your scripts do not pass code review and require extensive re-work (for example, if you are asked to fill a large gap in test coverage) you may need to move the User Story back into "Test" until all code review comments have been addressed.

Normally, each feature branch should be deleted after it has been merged into develop. When raising a merge request in GitLab ensure that you have ticked the following checkbox: Merge options > Delete source branch when merge request is accepted. If you do not want your feature branch to be deleted, let the merge request assignee know why.

Do not let unmerged branches become stale on the remote Git server. A stale branch is a branch that has not had any commits for three months. If you are no longer working in a feature branch ensure it has been merged and deleted from the remote Git server before you move on to the next feature.

### Merge-ready checklist[​](https://10.243.110.58/platform/docs/tester/test-code-management/merging/requesting_a_merge#merge-ready-checklist)

Ensure the following before requesting that a feature branch is merged into develop:

1. Update all NuGet references to the latest release package or latest pre-release package, whichever is most recent. Do not reference pre-release packages that belong to other projects. The project code in pre-release build number should match the project code in the branch name. For example, in **abc**/project/feature branches only reference release, **abc**-rel or **abc**-dev packages; do not reference, say, xyz-dev packages.
2. If your feature branch is one or more commits behind develop, carry out the converse merge from the develop branch into the feature branch.
   * Resolve merge conflicts if any. If you are the only tester working on the project, there should be no conflicts. But if you share the develop branch with other testers, and are unsure as to how to resolve the conflicts, ask a repository maintainer to help you.
3. Ensure that the solution builds with no errors or compiler warnings.
4. Ensure that all newly added tests pass.
   * If you intend to merge a failed test into a develop branch, tag the scenario in the feature file using the appropriate tag (for example, @techdebt, @issue, @defect or @wip) and explain why the test fails.

## Requesting a merge: from develop into release[​](https://10.243.110.58/platform/docs/tester/test-code-management/merging/requesting_a_merge#requesting-a-merge-from-develop-into-release)

A develop branch should be merged into the release branch (project release or unified product release) after the release candidate is signed-off in system testing, and a developer has merged their corresponding develop branch into release.

Failure to merge the test code after sign-off defeats the purpose of continuous integration, and causes tech debt and delays to future projects.

Each time a release candidate is signed off, the tester responsible for the sign-off should take ownership of the corresponding develop branch, ensure that the branch is ready for the merge into release, and raise a merge request in GitLab.

It the project team is collectively responsible for the sign-off of multiple services, APIs or applications, then team members should divide the labour between and designate an owner for each develop branch that needs merging.

When raising the merge request in GitLab, notify all other testers who work on the project, and assign the request to the repository maintainer for peer-review. Do not assign the merge request to yourself, unless you are the sole maintainer of the repository.

#### Merge-ready checklist[​](https://10.243.110.58/platform/docs/tester/test-code-management/merging/requesting_a_merge#merge-ready-checklist-1)

Ensure the following before requesting that a develop branch is merged into release:

* **Update NuGet references**: All pre-release dependencies should be upgraded to the release version, including service message packages references, if any. Open the solution in Visual Studio, select Manage Nuget Packages for solution and untick the following checkbox: Include Prerelease. Update all pre-release NuGet package references to the latest release packages. Pre-release package should never be referenced in release branches. By default, TeamCity will issue a warning when this constraint is violated.
* **Merge downwards**: Carry out the converse merge of the release branch into your project�s develop branch and resolve merge conflicts, if any. If you are unsure as to how to resolve the conflicts, ask a repository maintainer to help you.
* **Build locally**: Ensure that the solution builds on your computer with no errors in Visual Studio. Fix compiler warnings, code analysis warnings and SonarLint warnings if required. Otherwise suppress the warning and add a comment to justify why it can be suppressed.
* **Execute all tests**: execute all tests locally, or let TeamCity trigger the test run on the remote build agent. Ensure that all tests in the relevant Visual Studio project or solution are executed, including all regression tests, and review the results. Ensure that there are no false negatives or false positives, and that all genuine failures have been accounted for.
* **Add the test report**: use the extent report that is automatically generated in html format by Dsf.TestFramework.Reporting.Core. The report is saved locally it to the bin/ExtentReports directory. Alternatively, you can download the Extent Report from the TeamCity build artifacts. Push the latest report to the doc/TestReports/[build number] directory on the remote repository.

# Performing a Merge

Merges are performed by testers in the same way as they are performed by developers, using either the GitLab web portal or an alternative tool of your choice.

This section details the additional steps specific to testing.

## Prerequisites[​](https://10.243.110.58/platform/docs/tester/test-code-management/merging/performing_a_merge#prerequisites)

Read the (Performing a merge)[......\developer\code-management\merging\performing-a-merge.md] instructions in the Developer section.

## Review the test code[​](https://10.243.110.58/platform/docs/tester/test-code-management/merging/performing_a_merge#review-the-test-code)

1. Ensure that newly added or modified test scripts adhere to the guidelines detailed in the (Test Coding Standards)[test-coding-standards/overview.md] section. If there are deviations for the guidelines, highlight them in the code review comments. If applicable, refer the reviewee to the rule Id of the relevant Gherkin keyword rule, Feature style rules, or SpecFlow rule. Provide constructive feedback politely and offer a specific suggestion or solution.
2. Pay special attention to feature files written in Gherkin. Ensure that the business requirements are clearly documented and specified by example. The technical implementation of the step definitions can usually be refactored and optimised even with little knowledge of the business requirements, but poorly worded or unclear scenarios and steps may be harder to interpret and improve at a later date.
3. Review the test execution report before accepting a merge into a release branch. Ensure that the report for the sign-off of the release candidate has been submitted as part of the merge request, and that test coverage is adequate. The report should be located in the doc/TestReports/[Build number] directory.

## Merging into master[​](https://10.243.110.58/platform/docs/tester/test-code-management/merging/performing_a_merge#merging-into-master)

Repository maintainers are responsible for merging the release branch into the master branch after the product is signed-off in pre-production or deployed to production.

# Test Coding Standards: Overview

The languages used to write DSF tests are C# and Gherkin.

Unless otherwise specified, DSF test code should strive to adhere to the same standards as DSF production code.

This section details the guidelines that relate specifically to test code, with a special attention to the Gherkin syntax, feature files, and SpecFlow step definitions.

# Gherkin Guidelines

Gherkin is the language used to write feature files (.feature.cs).

The purpose of these guidelines is to ensure that Gherkin keywords are used consistently throughout all DSF test projects.

Each of the below Gherkin rule has a rule Id, for ease of reference in code reviews. Gherkin keyword rules (Id beginning with 'G') have higher priority than feature style rules (Id beginning with 'F').

## Gherkin Keywords[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/gherkin-guidelines#gherkin-keywords)

### Feature[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/gherkin-guidelines#feature)

* [G100] The Feature: keyword should be followed by the feature title in Title Case.
* [G110] Use the free-form text underneath the feature title to provide a brief description of the User Story or feature narrative.

### Background[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/gherkin-guidelines#background)

* [G130] Use the Background keyword to group simple Given steps that appear in all scenarios within the same feature.
* [G140] Do not use Background to set up complex states.
* [G150] Keep the Background short. If there are more than five background steps, consider grouping them into fewer high-level steps.

### Scenario[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/gherkin-guidelines#scenario)

* [G170] The Scenario: keyword should be followed by the scenario title in sentence case.

### Scenario Outline[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/gherkin-guidelines#scenario-outline)

* [G190] Use Scenario Outline to group similar scenarios that differ only in regards to test data combinations.
* [G200] The Scenario Outline: keyword should be followed by scenario outline title, in sentence case.

### Examples[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/gherkin-guidelines#examples)

* [G220] Use the Examples section of Scenario Outline to define each combination of values.
* [G230] Do not repeat the same value in all examples. If a value is the same for scenarios then it should not be placed in the Examples section.
* [G240] Do not duplicate values in the first column of the Examples section. The values in the first column are also used as scenario titles and should therefore be unique.
* [G250] If none of the columns in the Examples section provides a unique and clear description of each data row, then add the | \*Example\* | header in the first column and use it to define a unique title for each example.
* If the same scenario outline has multiple example sets:
  + [G261] ensure that the column headers are the same across all sets,
  + [G262] provide a unique and descriptive title, in sentence case, for each set.

✅ DO:

Scenario Outline: Generate report  
 Given the jurisdiction is <Jurisdiction>  
 When I request the jurisdiction report  
 Then the report is generated successfully  
  
Examples: Europe  
 | Jurisdiction |  
 | UK |  
 | Ireland |  
  
Examples: Asia  
 | Jurisdiction |  
 | India |  
 | China |

❌ DON'T:

Scenario Outline: Generate report  
 Given the jurisdiction is <Jurisdiction>  
 When I request the jurisdiction report  
 Then the report is generated successfully  
  
Examples:  
 | Jurisdiction |  
 | UK |  
 | Ireland |  
  
Examples:  
 | Jurisdiction |  
 | India |  
 | China |

❌DON'T:

Scenario Outline: Generate report  
 Given the jurisdiction is <Jurisdiction>  
 When I request the jurisdiction report  
 Then the report is generated successfully  
  
Examples: Set 01  
 | Jurisdiction |  
 | UK |  
 | Ireland |  
  
Examples: Set 02  
 | Jurisdiction |  
 | India |  
 | China |

### Given[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/gherkin-guidelines#given)

* [G280] Use Given steps to arrange the test pre-requisites.

Given the message is  
 | Generate report | Value |  
 | Run date | 1-Jan-2020 |

### When[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/gherkin-guidelines#when)

* [G300] Use When steps to perform or trigger the action under test.

When I send the message

### Then[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/gherkin-guidelines#then)

* [G320] Use Then steps to make assertions.
* [G330] Do not use Given or When steps to make assertions other than inconclusive assertions.

Then the reply is successful  
And the report Id is valid

* [G340] If there are many independent Then steps in a single scenario consider breaking the scenario down into smaller scenarios with fewer Then steps.

### And[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/gherkin-guidelines#and)

* [G360] Use the And keyword to join sequences of Given or Then steps.
* [G370] Instead of using the conjunction 'and' within a single step, consider breaking the step down into two steps.

✅ DO:

Given the report is generated  
And the notification is sent

❌ DON'T:

Given the report is generated and the notification is sent

### But[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/gherkin-guidelines#but)

* [G390] Use the But keyword to join sequences of Given steps where positive steps are followed by negative steps, for example steps that deliberately set an invalid state:

Given the report has been generated   
But the report type is invalid

* [G400] Use the But keyword to join sequences of Then steps where positive assertions are followed negative assertions, such as a validation failures:

✅ DO:

Then the reply is successful  
But the reply message is 'Invalid username'

❌ DON'T:

Then the reply is successful  
Then the reply message is 'Invalid username'

### @ (at sign)[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/gherkin-guidelines#-at-sign)

* [G420] Consider using tags, prefixed by the @ sign, to categorise scenarios.
* [G430] Consider using tags to categorise example blocks in scenario outlines.
* [G440] Avoid introducing new tags or ad-hoc tags. Use the existing tags defined in Dsf.TestFramework.SpecFlow.Tags when applicable.
* [G450] If the same scenario outline has multiple sets of examples, grouped under different tags, give a title to each example set.
* [G460] Use the @ignore tag to skip tests that are not suitable for regression testing. Add a comment above the tag to explain why the test is skipped.
* [G470] Use the @blocked tag to mark tests that cannot be executed due to blockers or impediments. Add a comment above the tag to explain why.
* [G480] Use the @manual tag to mark scenarios with unimplemented steps that require manual intervention. Add a comment to describe the manual step execution.
* [G490] Use the @techdebt tag when there are false positives or false negatives due to technical debt in the test code. Add a #ToDo comment above the tag to explain what should be done to remove the tech debt.
* [G500] Use the @deferred tag to mark tests that fail due to defects that will be fixed in a future release
* [G510] Use the @wip tag (work in progress) to mark unfinished features or scenarios that are merged into develop branches. Ensure that all work is completed and all @wip tags are removed before the develop branch is merged into the release branch. However, if your work in progress is not merged into the develop branch or shared with other testers, there is no need to use the @wip tag when working in a feature branch. Any work done in feature branches is considered to be implicitly in progress.
* [G520] Do not use the Tag keyword in step classes to restrict the binding scope of step definitions.
* [G530] Do not use tags to control the test flow.

### """ (Three double quotation marks)[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/gherkin-guidelines#-three-double-quotation-marks)

* [G560] Use multi-line arguments or 'docs string' to pass formatted text to step definitions.
* [G570] The text must be offset by delimiters consisting of three double-quote marks """ on lines of their own.
* [G580] If the formatted text is longer than five lines consider moving it from the feature file into a separate test data file.
* [G590] You can use multi-line arguments to pass short JSON strings to step definitions:

Then the response body is  
"""  
{  
 "Message": "Welcome to the DSF!",  
 "Success": true  
}  
"""

* [G600] If the JSON string is longer than five lines, consider placing it into a test data json file, instead of adding it directly to the feature file as a multi-line argument.

### \* (Asterisk)[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/gherkin-guidelines#-asterisk)

* [G620] Avoid using asterisks \* in place of step keywords. Gherkin supports asterisks as bullet points or as an alternative to the And keyword. However, it is usually better to consolidate lists of steps into a single data table.

### Rule[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/gherkin-guidelines#rule)

* [G640] Do not use the Rule keyword. Rules have been introduced in Gherkin v6 but are not supported by SpecFlow v3.

# Feature File Guidelines

Feature files (\*.feature.cs) are written using the Given-When-Then format of the Gherkin syntax.

This section describes the recommended style and naming conventions relating to feature files.

The primary goal of these rules is to maximise readability and consistency, and minimise merge conflicts.

Each feature style rule has a rule Id, beginning with 'F', for ease of reference in code reviews.

## Prerequisites[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/feature-file-guidelines#prerequisites)

This ruleset expands on the rules detailed in [Gherkin Guidelines](https://10.243.110.58/platform/docs/tester/test-coding-standards/gherkin-guidelines).

Feature style rules (rule Id beginning with F) have lower priority than Gherkin keyword rules (rule Id beginning with G). Ensure you familiarise yourself with the [Gherkin Guidelines](https://10.243.110.58/platform/docs/tester/test-coding-standards/gherkin-guidelines) before reading this section.

## Style and appearance[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/feature-file-guidelines#style-and-appearance)

### Writing style[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/feature-file-guidelines#writing-style)

* [F100] Write in simple and plain English.
  + [F101] Write full grammatical sentences.
  + [F102] Be concise and to the point.
  + [F103] Avoid pseudo-code.
  + [F104] Prioritise readability over exposition of technical details.
  + [F105] Do not use technical terms unless they are necessary to resolve ambiguities or are widely known within the business.
* [F110] Specify the business or technical requirements by example.
* [F120] Write each step from the perspective of the end-user, API user, or service consumer.
  + [F121] Do not describe the action from the tester or developer's perspective.
  + [F122] Do not describe the step definition logic using technical jargon.

✅DO:

Given the report is generated successfully  
And the notification is sent

❌DON'T:

Given I mock the Report Generator component  
And I intercept the Push Notification message

### Casing[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/feature-file-guidelines#casing)

* [F150] Write the feature title in Title Case. Use sentence case for everything else.
  + [F151] Title Case: capitalise the first letter of each word, including conjunctions, articles, and prepositions.
  + [F152] Sentence case: capitalise only the first letter of each sentence, proper noun or acronym. Use lower case for everything else.
  + [F153] If a dot (.), colon (:) or hyphen (-) is used as a sentence separator, capitalise the first letter after the dot, colon or hyphen.
* [F160] Avoid PascalCase and camelCase. Do not use underscores. Insert white spaces between words where possible.
* [F170] Do not capitalise letters inside words. Write 'Invalid' instead of 'InValid, or 'invalid' instead of 'inValid'.
* [F180] Capitalise the first letter after an hyphen when using the hyphen as a title separator in feature titles, scenarios titles, scenario outline titles, or example titles.
* [F190] Capitalise the first letter of each placeholder used in a Scenario Outline's steps and contained within angle brackets. For example: <User type>
* [F200] Capitalise the first letter of each abbreviation or acronym:
  + [F201] Write 'Id' instead of 'id' or 'ID',
  + [F202] Write 'Guid' instead of 'guid' or 'GUID'.
  + [F203] Write 'Api' instead of 'API'.

### Abbreviations and acronyms[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/feature-file-guidelines#abbreviations-and-acronyms)

* [F220] Do not use abbreviations or acronyms indiscriminately.
* [F230] Do not introduce new abbreviations or acronyms.
* [F240] Avoid using any existing abbreviation or acronym unless:
  + it is required in the test data,
  + it is used in the name of the service message under test,
  + it is used in the name of parameter of the service message under test,
  + it is used in the endpoint name of the API under test,
  + it is displayed in the front-end application under test,
  + it is widely known within the business,
  + it appears in the top ten results of a Google search.

Examples:

✅ DO write 'Isa' instead of 'Individual Saving Account'. This acronym is widely known and appears as the top result in a Google UK search.

❌ DON'T shorten 'Transfer In Authority Form' to 'Taf'.

### Indentation and formatting[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/feature-file-guidelines#indentation-and-formatting)

* [F270] When a feature is complete, open the feature file in Visual Studio and press CTRL + K + D. This shortcut will automatically apply the standard formatting rules to the entire feature file.
  + [F271] Do not change the default Formatting Options defined for the SpecFlow extension in Visual Studio 2019 > Tools > Options > SpecFlow > General\*.
  + [F272] Do not alter the standard indentation that is automatically applied when pressing CTRL + K + D.
  + [F273] Do not add any extra white lines at the start, middle or end of the feature file. The only white lines should be those that are automatically added when pressing CTRL + K + D.

## Naming conventions[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/feature-file-guidelines#naming-conventions)

### Feature file path[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/feature-file-guidelines#feature-file-path)

* [F300] Feature file names should be in PascalCase, followed by the '.feature' extension. For example: InvestorRegistration.feature
* [F310] All feature files in a test project should be grouped under the Features directory. This should be a top-level directory inside the test project directory.
  + [F311] Sub-groups of related features should be organised in sub-directories inside the Features directory. Sub-directory names should be in PascalCase, unless the sub-directory corresponds to a version number, as explained below:
  + [312] Service message feature and REST API features should always be grouped by version number. The version directory name is the letter 'v' in lower case followed by the version number. For example: Features\v1\GenerateRerpot.feature.
* [F320] In service message test projects, there should be a feature file for each command or query. The feature file name should match the service message name.
* [F330] In API test projects, there should be a feature file for each endpoint. The feature file name should match or closely resemble the endpoint name.
* [F340] If UI test projects, there should be a feature file for each end-user journey, and feature files should be grouped by user type.

### Feature title[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/feature-file-guidelines#feature-title)

* [F370] Write feature titles in Title Case.
* [F380] Ensure that each feature in the test project has a unique title.
* [F390] Do not add User Story Numbers or Work Item numbers to feature titles. These additional details can be placed in the free-text underneath the feature title.
* [F400] Do not start the feature title with a number.
* [F410] Do not attempt to introduce a numerical or alphabetical order. If the order changes you would have to rename all the features. Order is not relevant to test execution. Features should be executable in any order.
* [F420] Use hyphens as separators if you need to add qualifiers or disambiguate because there are other features that start with the same name.
  + [F421] Leave a blank space before and after the hyphen when using the hyphen as a separator:

Feature: Generate Report v1 - Daily  
Feature: Generate Report v1 - Quarterly

* [F430] Feature titles should reflect the feature file's name and place in the test project directory structure. The naming convention depends on the type of test project, as described below:
  + [F431] Titles of service message features should start with the service message name followed by the version number; for example: Feature: Generate Report v1
  + [F432] Titles of API features should start with, or closely resemble, the name of the endpoint under test, and should be followed by the API version number.
  + [F433] Titles of UI features should concisely describe the end-user journey and should match the feature file path, ignoring spaces and using hyphens instead of dashes as separators. For example, if the feature file path is 'Features/Investor/Registration' then the feature title should be: 'Investor - Registration';
  + [F434] Do not mention API method names in feature titles unless you need to disambiguate between endpoints with similar names; for example: 'Delete document' vs. 'Post document'.

### Scenario title[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/feature-file-guidelines#scenario-title)

* [F450] Write scenario titles in sentence case.
* [F460] Use hyphens as title separators, for example if you need to qualify different titles that start with the same words. Do not use dots or commas.
  + [F461] Leave a blank space before and after each hyphen, when using the hyphen as a title separator.
  + [F462] Capitalise the first letter after each hyphen, when using the hyphen as a title separator.

✅ DO:

Scenario title: Invalid user type - Null  
Scenario title: Invalid user type - Empty

❌ DON'T:

Scenario title: Invalid user type. Null  
Scenario title: Invalid user type, empty

* [F470] Do not start or end the scenario title with a number.
* [F480 Do not attempt to define a numerical or alphabetical order. If the order changes you would have to move or rename all the scenarios. Scenarios should be executable in any order. Order is irrelevant.
* [F490] Do not use titles that are vague, trivial, or non-descriptive.
  + [F491] Do not write 'Valid test'. All tests should be valid tests.
  + [F492] Do not write 'Invalid test'. There should not be any invalid test.
* [F500] Specify the invalid parameter name in validation failure scenarios.
  + [F501] If the invalid value is undefined, specify in the scenario title whether the value is null, empty or white space.
* [F510] Scenario titles should be specific but concise. Avoid repetitions and unnecessarily long titles.

✅ DO:

Scenario: Invalid withdrawal amount - Zero  
Scenario: Invalid withdrawal amount - Negative

❌ DON'T:

Scenario: Invalid parameter  
Scenario: Negative test  
Scenario: Sending an invalid request with a withdrawal parameter that is less than the minimum limit

### Scenario outline title[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/feature-file-guidelines#scenario-outline-title)

* [F530] Write the scenario outline title in sentence case.
* [F540] Bear in mind that a scenario outline is a blueprint or template for each scenario example, it is not a collection of examples. In test reports, the title of each scenario example starts with the scenario outline title. Therefore the scenario outline title should not be the pluralised title of a collection of scenarios.
  + [F541] Do not pluralise the scenario outline title unless the pluralised title accurately describes each scenario example taken individually. If, say, each scenario example generates only one report (singular), do not change the scenario outline title to "Generate reports" (plural).

✅ DO

Scenario Outline: Generate report  
Given the report type is <Report type>  
When I request the report  
Then the report is generated successfully

❌ DON'T pluralise the title

Scenario Outline: Generate reports  
Given the report type is <Report type>  
When I request the report  
Then the report is generated successfully

### Example title[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/feature-file-guidelines#example-title)

* [F560] If the scenario outline has only one set of examples, the Example keyword need not be followed by a title.
* [F570] If the scenario outline has multiple sets of examples, then each Example keyword should be followed by a unique and descriptive title, in sentence case.
* [F580] Do not pluralise the title of an example set, unless the pluralised title accurately describes each example taken individually. The stylistic rules for Scenario Outlines also apply to Example titles.

## Free-form text[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/feature-file-guidelines#free-form-text)

Free-form text underneath feature titles is ignored during test execution, but is available for reporting and is included in Extent Reports and SpecFlow+ Living Docs.

* [F590] Add a feature description using the following template:

In order to <achieve my goal>  
As a <type of user>  
I want <some functionality>.

- [F591] If the feature is complex and cannot be concisely described using the above template, consider adding a brief explanation or a list of business rules.

* [F580] Consider using the markdown syntax for bold, italic, lists etc. This is useful when generating test reports.
  + [F581] Pictures can be embedded using the markdown syntax:

![](folder/image.png)

## Steps[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/feature-file-guidelines#steps)

### Given steps[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/feature-file-guidelines#given-steps)

* [F600] Use Given steps to arrange the test prerequisites.
* [F610] Consider limiting the number of Given steps to five or less per scenario.
  + [F611] If the are too many Given steps in a scenario, consider consolidating them into fewer steps.
* [F620] If a Given step is used in all scenarios within a feature, move it to the Background block.
* [F630] Consider using hooks instead of Given steps if the prerequisite does not need to be explicitly stated in the feature.
* [F640] Use the active voice with the first-person 'I' pronoun if the action can be performed by the end-user or service consumer.

Given I have requested the report

* [F650] Use the passive voice with the first-person 'I' pronoun if the end-user or service consumer is the receiver of the action.

Given I have received a notification

* [F660] Use the passive impersonal voice if the action is not normally performed by the end-user or service consumer.

Given the cache is refreshed

### When steps[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/feature-file-guidelines#when-steps)

* [F680] Use When steps to describe the action that triggers the behaviour under test.
* [F690] Describe the action from the perspective of the end-user or service consumer.

When I request the report

* [F700] There should normally be exactly one When step in each scenario, unless the scenario involves a complex action with multiple triggers.
* [F710] Omit the When step only if the scenario does not require the user or service consumer to perform any actions.
* [F720] Include the When step even when the user action could be implied implicitly from other steps or from the scenario title. State the action of the When step explicitly.
* [F730] Avoid test data or input data in the When steps.
* [F740] Do not replace the When step with the last step in a sequence of Given steps.
* [F750] Always use the active voice and the first-person 'I' pronoun.

### Then steps[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/feature-file-guidelines#then-steps)

* [F770] Use Then steps to make assertions and compare the actual outcome to the expected outcome.
* [F780] Consider limiting the number of Then steps to five or less per scenario.
* [F790] Use the passive voice
* [F800] Do not use 'Should' or 'Would'. Make a factual 'Is' or 'Are' statement.

✅ DO:

Then the report is generated  
And the notifications are sent

❌ DON'T:

Then the report should be generated  
And the notifications should be sent

## Data tables[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/feature-file-guidelines#data-tables)

* [F820] Ensure that the property names in the data table match the model's property names. However, the text match is not case sensitive and spaces are ignored.
* [F830] Write property names in sentence case and insert spaces between words even if the corresponding C# property name is in PascaCase.
* [F840] Write the headers in sentence case.
* [F850] Use the first header of vertical tables to describe the type of data model, unless this is clear from the context or results in unnecessary repetition.
* [F860] Use vertical tables when creating or comparing instances.
  + [F861] Exception: Use a horizontal table if the instance has only one or two properties and the type of data model need not be explicitly described in a vertical table header.
* [F870] Always use horizontal tables when creating or comparing sets.
* [F880] Write Enum values as strings rather than numbers

## Value descriptors[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/feature-file-guidelines#value-descriptors)

The rules detailed below require the test project to reference the **Dsf.TestFramework.SpecFlow** component as an external binding assembly.

### Null values[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/feature-file-guidelines#null-values)

* [F900] Values are (usually) null by default. If you are testing a scenario with a null value, and this can implicitly implied from other steps or from the scenario title, then you do not need to add an additional step to set the value to null.
* [F910] Use the word 'Null' if you need to state explicitly that something is null. Do not use blank spaces or empty tables.
* [F920] The word 'Undefined' can be used in titles of Scenario Outlines to mean 'Either null or empty'. But do not use the word 'Undefined' to mean 'Null'.

✅ DO:

Given the user type is null

❌ DON'T:

Given the user type is  
 | User type |  
 | |

### Empty values[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/feature-file-guidelines#empty-values)

* [F930] Use the word 'Empty' to describe empty strings or empty collections. Do not use a blank space.
* [F940] When the data type is string, the term 'Empty' is automatically parsed to an empty string by the Value Retrievers and Step Argument Transformations of the Dsf.TestFramework.SpecFlow component.
* [F950] When the data type is a collection of string or numbers, the term 'Empty' is automatically parsed to an empty collection by the Value Retrievers and Step Argument Transformations of the Dsf.TestFramework.SpecFlow component.

✅DO:

Given the user type is empty

❌DON'T:

Given the user type is ''

### White spaces[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/feature-file-guidelines#white-spaces)

* [F960] Do not use whitespace-only values in data tables or step arguments.
* [F970] If you need to test a scenario where a non-empty string value consists only of white spaces, then add a custom step such as Given the <Value> consists only of white spaces, and set the value to white spaces directly in the step definition.

### Booleans[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/feature-file-guidelines#booleans)

* [F980] Use the words 'True' and 'False' to describe Boolean values. Do not use '1' and '0'.
* [F990] However, if using the terms 'True' and 'False' results in a poorly worded step, consider using the below alternatives. The following descriptors are automatically parsed to Boolean values by the Value Retrievers and Step Argument Transformations of the Dsf.TestFramework.SpecFlow component:
  + 'Enabled' / 'Disabled'
  + 'Active' / 'Inactive'
  + 'Successful' / 'Unsuccessful'

✅ DO:

Given the task is enabled

❌ DON'T:

Given IsTaskEnabled is set to 'True'

# SpecFlow Guidelines

This section details the coding standards for the implementation of SpecFlow bindings, that is, C# classes marked with the [Binding] attribute.

Bindings consist of:

* Step definitions: used to provide the connection between feature files and the system under test, such as services, APIs or front-end applications.
* Hooks: used to implement setup or teardown logic that needs to be executed before or after each scenario, feature or test run.

Each of the below SpecFlow rule has a rule Id, beginning with 'S', for ease of reference in code reviews.

## Prerequisites[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/specflow-guidelines#prerequisites)

* Ensure that the **Dsf.TestFramework.SpecFlow** component is included in the project file as a NuGet reference.
* Ensure that the **Dsf.TestFramework.SpecFlow** steps are included in the stepAssemblies section of the **specflow.json** file.

The above dependencies are automatically added when creating a new test project using DSF test project templates.

## Test Context[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/specflow-guidelines#test-context)

### Context injection[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/specflow-guidelines#context-injection)

* [S100] Use Context Injection to share test data between step definitions or hooks that belong to different classes. See the official SpecFlow documentation: <https://docs.specflow.org/projects/specflow/en/latest/Bindings/Sharing-Data-between-Bindings.html>
* [S110] If the test context cannot be modelled using a POCO class (simple .Net object), add a using directive to include **Dsf.TestFramework.SpecFlow.Extensions** and use the ScenarioContext extension method, Register, to add the required instance to the Scenario Container.
  + [S111] If the context instance implements an interface, register it using the interface:

[BeforeScenario]  
 public void InitializeWebDriver()  
 {  
 var chromeDriver = new ChromeDriver();  
 scenarioContext.Register<IWebDriver>(chromeDriver);  
 }

### Feature context[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/specflow-guidelines#feature-context)

* [S150] Avoid using FeatureContext.Current in step definition methods. Instead of using the current static context, consider injecting FeatureContext as the parameter of a static method marked with a BeforeFeature or AfterFeature hook attribute, for example:

✅ DO:

[BeforeFeature]  
 public static void SetFeatureTitle(FeatureContext featureContext)  
 {  
 \_featureTitle = featureContext.FeatureInfo.Title;  
 }

❌ DON'T:

\_featureTitle = FeatureContext.Current.FeatureInfo.Title;

* [S160] When using FeatureContext.Current.Get and FeatureContext.Current.Set ensure that all usages are strongly-typed, denoted by a globally unique key, and wrapped into properties with getters and setters.

### Scenario context[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/specflow-guidelines#scenario-context)

* [S200] Do not use ScenarioContext.Current. This has been deprecated since SpecFlow v3. Instead of using the current static context, use the injected or derived context as explained below.
* [S210] If the scenario context is required in shared steps, inject the Scenario context parameter into the step class constructor.
* [S220] If the scenario context is required only in a hook, add the ScenarioContext parameter to a BeforeScenario or AfterScenario method.
* [S230] If the scenario context is required in a derived step class, use the Scenario context from the base class.

## Hooks[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/specflow-guidelines#hooks)

* [S300] Use BeforeScenario hooks to implement setup logic that need not be explicitly defined in Background steps.
* [S310] Use AfterScenario hooks to implement the clean-up logic or any action that needs to be executed regardless of whether the test passes or fails.
* [S320] Consider specifying the Order property in the hook attribute. By default, hooks of the same type have the lowest priority, and are executed in a random order.
* [S330] Use enums to define the order for each hook type, and ensure that your custom ordering fits logically with the order of pre-existing hooks. The order of common test framework hooks is defined in **Dsf.TestFramework.SpecFlow.HookOrder**.
* [S340] Consider using hooks with parameter injection where applicable:

| **Hook type** | **Applicable parameters** |
| --- | --- |
| BeforeScenario, AfterScenario | ScenarioContext |
| BeforeFeature, AfterFeature | FeatureContext |
| BeforeTestRun, AfterTestRun | ITestRunManager, ITestRunner |

## Data tables[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/specflow-guidelines#data-tables)

* [S400] Use SpecFlow data tables to pass test data models to step definitions.
* [S410] Do not iterate through data tables to parse and retrieve the data. Instead of iterating, use the following Table extension methods:

| **Dsf.TestFramework.SpecFlow.Extensions** | **Description** |
| --- | --- |
| BuildInstance() | Similar to CreateInstance() but with validation options and support for nested objects |
| BuildSet() | Similar to CreateSet() but with validation options and support for nested objects |
| CreateList() | Creates a list of strings |
| CreateDictionary() | Creates a Dictionary |

* [S420] Do not iterate through data tables to make assertions. Instead of iterating, use the following Table extension methods:

| **TechTalk.SpecFlow.Assist** |
| --- |
| CompareToInstance() |
| CompareToSet() |

* [S430] When the default BuildInstance() or BuildSet() cannot be used to model the test data, consider registering a custom implementation of SpecFlow.Assist.IValueRetriever.
* [S440] When the default CompareToInstance() or CompareToInstance() cannot be used to make assertions about test data models, consider registering a custom implementation of SpecFlow.Assist.IValueComparer.
* [S450] Avoid using Tuples to model data tables, especially if the model contains more than three properties or is required in more than one step definition. Instead of Tuples, consider using a POCO class to model the test data.
  + [S451] If a Tuple is used to model a data table, ensure that the number and order of arguments in the Tuple match the number and order of parameters in the data table. Parameter names do not need to match and cannot be validated because Value Tuples do not hold argument names at runtime using reflection. However, for the sake of consistency, Tuple argument names should be in PascalCase and should match the property names in the data table, ignoring spaces and casing.
  + [S452] Never use Tuples to map eight or more parameters, as this will cause SpecFlow to throw an exception.

## Test execution order[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/specflow-guidelines#test-execution-order)

* [S500] Features and scenarios should be independent of each other and executable in any order.
* [S510] If pre-requisite data needs to be setup or configured before each test run, feature or scenario, then use the relevant hooks instead of enforcing an execution order.
* [S520] If pre-requisite data needs to be setup or configured only once per test environment, then it is acceptable to enforce a test execution order only for the first test run, i.e. only when the tests are executed for the first time on a new environment. The ordering should be irrelevant for all subsequent runs.
* [S530] If the test project requires the tests to be executed in a specific order, this should be documented in the ReadMe file or in the **doc** section of the test code repository.

## Environment-specific test data[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/specflow-guidelines#environment-specific-test-data)

* [S550] Do not hard-code environment-specific data in step definitions.
* [S560] Environment-specific static data should be added to the relevant test configuration file in JSON format.
* [S570] Test data that changes dynamically across test environments should be set up and retrieved programmatically.

## Step reusability[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/specflow-guidelines#step-reusability)

### Shared steps[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/specflow-guidelines#shared-steps)

* [S600] Do no restrict the binding scope when the step definition can be shared across all features within the same test project.
* [S610] Do not use shared steps from external binding assemblies other than those defined in Dsf.TestFramework components.

### Base steps[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/specflow-guidelines#base-steps)

* [S650] Use base steps for step definitions that apply to generic types such as service message types.
* [S660] Do not use base steps other than those defined in Dsf.TestFramework components. By default, SpecFlow step definitions are global and sealed from inheritance.

### Scoped bindings[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/specflow-guidelines#scoped-bindings)

* [S700] Use scoped bindings to resolve ambiguous step definitions.
* [S710] Use feature-scoped bindings when your step definition class derives from a base class.
* [S720] Ensure that the feature scope matches the feature title. The match is case sensitive and spaces are not ignored.
* [S730] Do not use scoped bindings in shared steps.
* [S740] Do not use tags to restrict binding scopes, unless there is a strong justification for doing so. Tag-scoped bindings should be documented and justified in the ReadMe file or doc section of the test code repository.

# Spelling guidelines

## Pre-requisites[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/spelling-guidelines#pre-requisites)

Install the [Visual Studio spell checker](https://10.243.110.58/platform/docs/tester/getting-started/setting-up-your-test-rig/visual-studio-spell-checker).

## Service message names[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/spelling-guidelines#service-message-names)

When signing-off a new service message, a new version of an existing message, or a new API endpoint, ensure that there are no spelling errors in the message name, endpoint name, parameter names, and model names.

Ensure that each reply name matches the corresponding query name followed by the "Reply" suffix.

If any spelling errors are found, raise an issue with the relevant developer **before** the release candidate is merged into the unified release branch.

If a service message is released to production with spelling errors, or deviations from the naming convention, then the errors can no longer be corrected, unless the misspelt message is marked as obsolete and a new version of the message is created.

If you regression test any existing or obsolete service messages that have been already released with spelling errors, use the Ignore spelling directive as explained below.

## SpecFlow bindings[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/spelling-guidelines#specflow-bindings)

Be especially careful to avoid spelling and grammatical errors in SpecFlow bindings, because correcting such errors can be tedious and time consuming, for the below reasons:

1. If a binding is amended in a step definition class, all binding usages in feature files will need to be identified and amended accordingly. A partial correction would result in orphaned steps.
2. A simple Find and Replace in Visual Studio will sometimes suffice to correct a typo in a binding. But, depending on how the binding is worded, a more complex Find and Replace tool with a Regular Expression pattern may be required instead.
3. In the case of frequently used steps that are shared across several features, you would also need to generate a Step Definition Report to identify all usages of the misspelt binding.

## Frequently used acronyms[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/spelling-guidelines#frequently-used-acronyms)

Optionally, you can add frequently used acronyms, such as "Guid" or "Dsf" to the user-specific Visual Studio dictionary setting or list of ignored words:

* Either in Tools > Spell Checker > Edit Global Configuration > Dictionary settings or Ignore Words
* Or using the Quick Action context-menu: Add to Dictionary or Add to ignored Word file.

## Ignore spelling directive[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/spelling-guidelines#ignore-spelling-directive)

Sometimes words are deliberately misspelt in automated tests for the following reasons:

* To test validation failures.
* To test obsolete service messages or API endpoints that have been released with spelling errors.

If a word is deliberately misspelt in a step, scenario or feature and it is not convenient to exclude that word in the user-specific dictionary or the list of ignored words, consider adding an Ignore spelling directive inside the feature file or step definition class.

The ignore spelling directive is added as a comment inside the file:

Gherkin (feature file):

#Ignore spelling: The Grauniad

C# (step definition class)

//Ignore spelling: The Grauniad

Feature file:

* If the word occurs in a single scenario, add the ignore spelling directive above the scenario title.
* If the word occurs in more than one scenarios within the same feature, add the ignore spelling directive above the feature title.

Step definition class:

* If the word occurs in single step binding, add the ignore spelling directive above the binding.
* If the word occurs more than once within the same class, add the ignore spelling directive above the class name.

If the reason for the exclusion is not obvious from the test context, consider adding a comment to explain why the spelling error can be ignored, for example:

//Reason: The parameter name was misspelt in obsolete message versions

# Test Coding Standards: Code Analysis

Automation test code written in C# should pass static code analysis. This complements, but does not replace, peer reviews carried out by Developers in Test.

## Prerequisites[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/code-analysis#prerequisites)

Install the [SonarLint](https://10.243.110.58/platform/docs/developer/getting-started/setting-up-your-development-environment/initial-simple-steps/installing-sonarlint) extension for Visual Studio.

## How to run code analysis[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/code-analysis#how-to-run-code-analysis)

To run code analysis on the entire solution in Visual Studio 2019, select Analyze > Run Code Analysis > On Solution, and view the warnings in the Error List window.

Pay special attention to violations that are specific to test coding practices, for example:

S2699: Add at least an assertion to this test case

If there are any warnings, refactor the test code to remove all code analysis violations, including Sonar scan warnings (ID beginning with 'S').

## Warning suppression[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/code-analysis#warning-suppression)

When code analysis guidelines are intentionally violated, the corresponding warning should be suppressed with a comment to explain why.

If you think you have a good reason to ignore a specific warning, suppress it in-line using the pragma warning disable directive and add a comment to justify the suppression; for example:

//Suppress 'parameter is never used' warning  
//Justification: The Assert parameter is required to integrate Assert extensions into the MS test framework  
#pragma warning disable RECS0154, IDE0060  
public static void AreEqual(  
 this Assert assert,  
 DateTime expected,  
 DateTime actual,  
 TimeSpan tolerance)

Ensure to re-enable the suppressed warning after the affected code line or code block, using the #pragma warning restore directive, unless the warning in question can be suppressed throughout the entire class file.

## Target framework: End-of-life warnings[​](https://10.243.110.58/platform/docs/tester/test-coding-standards/code-analysis#target-framework-end-of-life-warnings)

DotNet SDK end-of-life warnings such as the below can be usually ignored in existing DSF test projects:

NETSDK1138 The target framework 'netcoreapp2.2' is out of support and will not receive security updates in the future.

To suppress all end-of-life warnings set the <CheckEolTargetFramework> flag to false in the project file, below the target framework or frameworks, for example:

<PropertyGroup>  
 <TargetFrameworks>net472;netcoreapp2.2</TargetFrameworks>  
 <CheckEolTargetFramework>false</CheckEolTargetFramework>

# Service Tests Overview

DSF services are testable in isolation using service messages.

This section provides step-by-step instructions to create and configure test projects for DSF services.

# Adding a New Service Test Project

The addition of new service test projects is normally performed by repository maintainers.

Developers in test without maintainer role will usually add new or update service message features that belong to existing service test projects.

Read this section only if you are a repository maintainer or you need to create a new DSF service test project.

## Prerequisites[​](https://10.243.110.58/platform/docs/tester/service-tests/adding-a-new-service-test-project#prerequisites)

A dedicated test repository should be created by the Architecture team, for each DSF service, alongside the corresponding developers' repository.

You will need at least Developer-level access to the test repository. If you cannot see the repository, request access as explained [here](https://10.243.110.58/platform/docs/tester/test-code-management/repositories/requesting-access-to-a-repository).

## Create the test project using DSF templates[​](https://10.243.110.58/platform/docs/tester/service-tests/adding-a-new-service-test-project#create-the-test-project-using-dsf-templates)

1. Ensure you have installed the latest version of the DSF Test Templates for Visual Studio, as explained [here](https://10.243.110.58/platform/docs/tester/getting-started/setting-up-your-test-rig/dsf-test-templates).
2. Open Visual Studio and select File > New Project...
3. In the Add new project dialog:
   * Visual Studio 2017: select Visual C# > Test
   * Visual Studio 2019: type "DSF" in the template search box
4. Select the required DSF Service Test Project template:
   * Select the **.NET core** template when creating a new test project for a standard 5 (S5) DSF service. S5 services use the Messenger only to send and handle messages. All new services should be tested using the .Net core template.
   * Select the **.NET Framework.** template when creating a test project for a Standard 4 (S4) or 4+ (S4+) service. These services use the Message Client instead of the Messenger (S4) or in addition to the Messenger (S4+). You will rarely, if ever, use the .NET Framework template. This template should be used only for legacy services that are not .NET core compliant.
5. Enter the project name: **Dsf.[ServiceName].ServiceTests**

Ensure that the project name is correct before pushing to the GIT repository.

By default, the project name is used as the root namespace.

Ensure you type the correct name. TeamCity build templates expects the naming convention to be followed. Late corrections to project names or namespaces are tedious and time-consuming, and can cause avoidable merge conflicts.

1. Click Browse... and select the repository location on your computer.

If the solution does not already exist, select Create new solution from the Solution drop-down menu, and tick the "Create directory for solution" checkbox.

The naming convention is as follows:

* + GIT repository in lower case: **servicename.tests**
  + Solution file in Pascal case: **Dsf.[ServiceName].Tests**
  + Project file in Pascal case: **Dsf.[ServiceName].ServiceTests**

The project file suffix is .ServiceTest to distinguish it from .PerformanceTests, .ApiTests and .TestClient projects that might belong to the same .Tests solution

1. Click Ok.
2. Rename to src the top-level directory. This stands for "source code". TeamCity build templates are configured to build only the content of the src directory.
3. Ensure that the new solution has been created correctly. The directory structure should be as follows:

* **src** (top-level directory with solution file)
  + **Dsf.ServiceName.ServiceTests** (project folder with project file)
    - **Features**
      * v1
      * v2
      * etc.
    - **Steps**
      * v1
      * v2
      * etc.

The v2 directories are shown only for illustration purposes. If the DSF service is brand new, it should only have v1 messages.

1. Generate a new Guid and use it populate the Messagging:Messenger:ServiceCode value in appsettings.json. This is a one-off, to ensure that the ServiceCode used in the test project will never match any of the actual ServiceCodes used by DSF services. The Guid can be generated using an on-line Guid generator tool.
2. Populate the Messagging:Messenger:ServiceName value in appsettings.json. This should be the name of the DSF service under test, followed by ".tests", all in lower case. For example if the service is called "Report Manager" then the ServiceName should be "reportmanager.tests".

## Raise a merge request[​](https://10.243.110.58/platform/docs/tester/service-tests/adding-a-new-service-test-project#raise-a-merge-request)

If you are not the repository maintainer, raise a merge request before you start adding service message features to the newly created project. Assign your request to a repository maintainer for code review.

Ensure that the directory structure is correct before requesting the merge. Late corrections to the folder paths are tedious and time-consuming, can cause avoidable merge conflicts, and can make it difficult to trace the complete history of changes in source control.

Repository maintainers should ensure that the new project is being added to the correct repository and there are no deviations from the naming convention.

### Target branch for new test projects[​](https://10.243.110.58/platform/docs/tester/service-tests/adding-a-new-service-test-project#target-branch-for-new-test-projects)

The newly created test project should be merged into a develop branch, so that it can be used to add new features.

At this stage the newly created project is merely a featureless blue-print, and should not hold any references to pre-release NuGet packages.

At the discretion of repository maintainers, the empty test project can also be merged into the release branch, especially if it will need to be shared across multiple projects branches.

# Configuring Service Tests: Overview

This section details how to populate the configuration values required to execute DSF service tests.

DSF service tests are configured using appsettings files in JSON format.

Tests for legacy DSF services may also require app.config files in XML format.

# service-configuration-files

## Prerequisite: Service Configuration Files[​](https://10.243.110.58/platform/docs/tester/service-tests/configuring-service-tests/service-configuration-files#prerequisite-service-configuration-files)

Each service test project's configuration is based on the configuration of the DSF service under test.

Before you can configure your test project, you will need to obtain to the DSF service's configuration files. These can be requested from DevOps or, if you have access to Octopus, from the list of deployment artifacts in the Octopus portal.

In order to locate the configuration files, you will need to know the service name and the name of the test environment used to deploy the service.

## Services with a single configuration file[​](https://10.243.110.58/platform/docs/tester/service-tests/configuring-service-tests/service-configuration-files#services-with-a-single-configuration-file)

In a typical scenario where the DSF service under test is a Standard 5 service (Messenger only) and has a single component, you will just need to obtain a single appsettings.json file.

## Services with multiple configuration files[​](https://10.243.110.58/platform/docs/tester/service-tests/configuring-service-tests/service-configuration-files#services-with-multiple-configuration-files)

If a DSF service consists of multiple components, there will be distinct configuration file for each component.

The set of configuration files consist of

* One appsettings.json file for each service's component.
* One app.config file (XML) for each service's component, if the service is of Standard 4 or Standard S4+; that is, if it uses the Message Client and targets .NET Framework.

# Test Configuration Files

Unlike DSF services, DSF test projects use environment-specific configuration files - for example, appsettings.tst.json - in addition to the default appsetting.json file.

Configuration values that are unlikely to change across test environment should be placed in the default appsettings.json file.

Environment-specific configuration values should be placed in the relevant environment-specific configuration file, usually appsettings.tst.json.

## DSF service configuration vs Test configuration[​](https://10.243.110.58/platform/docs/tester/service-tests/configuring-service-tests/test-configuration-files#dsf-service-configuration-vs-test-configuration)

While the test configuration is based on the DSF service configuration, the contents of DSF services's appsettings.json files cannot be simply copy/pasted into the test project's appsettings.json file.

For standard configuration sections used by all DSF services, such as the "Messaging", "Logging" or "TableStorage" sections, the key names used by the DSF Test Framework will always match those used by the DSF service under test.

For service-specific configuration sections, the key names used by the DSF service under test may not match the default configuration key names used in the DSF test framework.

The following sections explain how to map the service configuration to the test configuration.

# Messaging Configuration

The service test configuration should nearly always include the Messaging configuration section.

The Messaging:Messenger section is required to send, intercept and mock service messages.

The Messenger configuration consists of the following keys:

* **"ServiceCode"**: a globally unique identifier (GUID). There are no constraints on the GUID format, but, for the sake of consistency, the preferred format is 32 digits without hyphens, braces or parentheses.
* **"ServiceName"**: the name of the service under tests, followed by '.tests', all in lower case.
* **"MessageManagerEndpoint"**: the base URL of the Message Manager API. This is a core DSF component, and is required to manage topics and subscriptions in the Azure Service Bus.
* **"Scope"**: either six or three letter, all in lower case, used to identify the test environment.
  + Six-letter codes are used for project-specific environments such as TST or PAT. The first three letters are the project code, followed by type of test environment. For example: "abctst".
  + Three-letter codes are used for unified product environments. For example: "per" or "uat".

The **ServiceCode** and **ServiceName** are global values. They should not change across test environment. These values should have already been set as a one-off configuration in the default appsettings.json file when the test project was created from a DSF template. Once these global values are set, as explained is steps 10 and 11 of the [Adding a New Service Test Project](https://10.243.110.58/platform/docs/tester/service-tests/adding-a-new-service-test-project) article, you should not change them, unless you have a good reason for doing so.

The **MessageManagerEndpoint** and the **Scope** are environment-specific values and should be placed in primary configuration files such as appsettings.tst.json

## Example of messenger configuration[​](https://10.243.110.58/platform/docs/tester/service-tests/configuring-service-tests/messaging-configuration#example-of-messenger-configuration)

appsettings.json:

"Messaging": {  
 "Messenger": {  
 "ServiceCode": "ccd26ebf1ec242e8bc41d226db3319ed",  
 "ServiceName": "reportmanagerservice.tests",  
 "MessageManagerEndpoint": "#MessageManagerEndpoint",  
 "Scope": "#Scope"  
 }

appsetting.tst.json:

"Messaging": {  
 "Messenger": {  
 "MessageManagerEndpoint": "http://auesysdsf.messaging.messagemanager.application:7998/api",  
 "Scope": "auetst"  
 }

## Message Client configuration[​](https://10.243.110.58/platform/docs/tester/service-tests/configuring-service-tests/messaging-configuration#message-client-configuration)

This is required only when testing legacy DSF service of Standard 4 or Standard 4+.

The Message Client configuration is in XML format. It should be copied and pasted into the environment-specific message client configuration file, for example messageclient.tst.config.

If all tests involving the Message Client consist in sending and receiving messages, without any interception or mocking, then the original MessageClient configuration can be re-used in the test project without making any changes.

If the one or more scenarios in the test project require service messages to be intercepted or mocked using the Message Client, then append ".tests" to the value of the serviceName key in the serviceBus configuration section. This change is required to avoid interferences between the Azure service bus subscription used by the test setup and that used by the service under test.

# Data Store Connection Strings

Data store connection strings may be required when configuring service tests for stateful services. Stateful services are services that persist a state to a data store such as Azure Table or an Azure SQL database.

Data store connection strings should be used only when it is not possible to set up or validate the service state using service messages.

In any case, the service test configuration should never include connection strings that are not present in the service's original configuration files.

## Default connection string names[​](https://10.243.110.58/platform/docs/tester/service-tests/configuring-service-tests/Data-Store-Connection-Strings#default-connection-string-names)

By default, the DSF test framework expects Azure connection strings to be placed in the "ConnectionStrings" configuration section. However, DSF services do not universally follow this convention; Azure connection strings may be found in other configuration sections, other than the "ConnectionStrings" section.

It is not necessary to enforce an exact correspondence between the DSF service configuration keys and the test project's configuration keys. You can simply copy the connection strings used by the DSF service into the default configuration section used by the DSF test framework, as follows:

| **Type of data store** | **Default JSON Configuration key** |
| --- | --- |
| Azure table, blob storage, file Shares | "ConnectionStrings":"AzureStorage" |
| MsSQL database | "ConnectionStrings":"MsSqlDatabase" |
| Azure SQL | "ConnectionStrings":"AzureSql" |
| Oracle database | "ConnectionStrings":"OracleDatabase" |
| Data warehouse | "ConnectionStrings":"DataWarehouse" |

## Binding the Service Configuration to the Test Settings[​](https://10.243.110.58/platform/docs/tester/service-tests/configuring-service-tests/Data-Store-Connection-Strings#binding-the-service-configuration-to-the-test-settings)

Alternatively, you can preserve the service's original configuration section, and programmatically override the default configuration keys expected by the DSF test framework. In order to do so, you will need to bind the service configuration to the test settings using the **SettingsBinder** class of the **Dsf.TestFramework.Configuration** component.

For example, you can bind a connection string in the "ConnectionString" configuration using an alternative key name, as follows:

var settingsBinder = new SettingsBinder();  
string connectionString = settingsBinder.GetConnectionString("MyAzureDatabase");

Other examples can be found in Dsf.TestFramework.Configuration.Core.UnitTests.SettingBinderTests.

However, using the default configuration keys is usually easier than binding a custom configuration section.

# Sample Service Test Configuration

The below example illustrates the content of sample configuration files for a core DSF service, the Communications.Email service:

Default configuration **appsettings.json** sample:

{  
 "Environment": "tst",  
 "Logging": {  
 "Kibana": {  
 "SourcePrefix": "#SourcePrefix",  
 "Source": "communicationsemailservice"  
 }  
 },  
 "Messaging": {  
 "Messenger": {  
 "ServiceCode": "9dd38c610c6a48aaa736be7cb4b8f85c",  
 "ServiceName": "emailservice.tests",  
 "MessageManagerEndpoint": "#MessageManagerEndpoint",  
 "Scope": "#Scope"  
 }  
 },  
 "ConnectionStrings": {  
 "AzureStorage": "#Email storage connection string"  
 },  
 "TestData": {  
 "ConnectionKey": "#Row key of the ConnectionCredential table. This is located in the Azure storage of the Receiver component of the Email service."  
 }  
}

Primary configuration **appsettings.tst.json** sample:

{  
 "Logging": {  
 "Kibana": {  
 "SourcePrefix": "abctst\_"  
 }  
 },  
 "Messaging": {  
 "Messenger": {  
 "MessageManagerEndpoint": "https://genpatmessagemanager.casfs.co.uk/api",  
 "Scope": "abctst"  
 }  
 },  
 "ConnectionStrings":{  
 "AzureStorage": "DefaultEndpointsProtocol=https;AccountName=JoeBloggs;AccountKey=a123Bc/d/4/D56efghIlmnopqr7StuVxYZaBCDeFgHIj8klMnO9PQr/sTUVWX/YZABCdEFgHIJkLMNOP01QRs==;EndpointSuffix=core.windows.net"  
 },  
 "TestData": {  
 "ConnectionKey": "f7bb6185-b2e1-4032-8c37-80bc97fc712b"  
 }  
}

* The **"Environment"** key is used to set appsettings.tst.json file as the primary configuration source.
* The **"Messaging"** configuration section is mandatory. It is required to send and receive messages.
* The **"Logging"** configuration is usually optional. It is used to capture the Kibana logs. Splunk log capture is not supported in the above example.
* The **"ConnectionStrings"** configuration section holds the service data store's connection string. This should be optional whenever possible, and should never be required to execute smoke tests or basic regression tests.
* The **"TestData"** configuration section holds environment-specific static data or pre-requisite data needed for the test setup.

# Testing Services in Isolation: Overview

This section describes how to design and execute isolation tests for DSF services, and explains what are the advantages of doing so.

# Isolation Test Execution

The golden rule of DSF testing is that each service should be tested in maximum isolation whenever possible.

Testing a service in �maximum isolation� means to test it independently of:

1. Other services
2. REST APIs
3. Front-end applications

## Partial isolation[​](https://10.243.110.58/platform/docs/tester/service-tests/testing-services-in-isolation/isolation-test-execution#partial-isolation)

If maximum isolation is not achievable in specific cases, then the service should tested in as much isolation as practically possible.

Examples:

* Dependencies on legacy systems.
* Connector services with dependencies on third-party APIs that we can't mock.

## Self-contained deployments[​](https://10.243.110.58/platform/docs/tester/service-tests/testing-services-in-isolation/isolation-test-execution#self-contained-deployments)

Isolation tests should be executable even if nothing is deployed to the test environment other than the service under test.

If your test setup requires additional services to be deployed, or relies on REST APIs or front-end applications, then you are not testing the service in maximum isolation.

## Isolation test configuration[​](https://10.243.110.58/platform/docs/tester/service-tests/testing-services-in-isolation/isolation-test-execution#isolation-test-configuration)

Isolation tests should be configurable without using the configuration files of other services, APIs or applications.

If your test configuration requires connection strings that are not present in the service�s appsettings.json file, then you are not testing the service in isolation.

# Isolation Test Design

Functional tests should treat each service as a single black box.

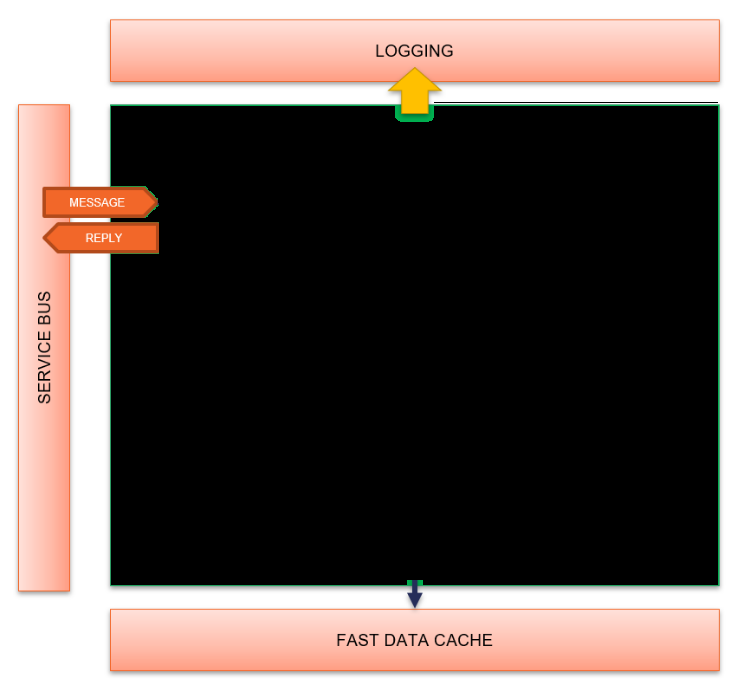
Isolation tests care only about the boundaries of the black box and are agnostic about anything else that goes inside or outside of the box.

If your test design requires the test to “know” about other services, a REST API or front-end applications, then you are not testing the service in isolation.

## Black-box testing[​](https://10.243.110.58/platform/docs/tester/service-tests/testing-services-in-isolation/isolation-test-design#black-box-testing)

Below is an illustration of the boundaries of a stateful DSF service with a single component:

The isolation test should be agnostic as to the inner computational workings of each component, and should treat the service as a **black box**, as illustrated below:



## Service message test design[​](https://10.243.110.58/platform/docs/tester/service-tests/testing-services-in-isolation/isolation-test-design#service-message-test-design)

Typically, isolation tests interact with DSF services by sending service messages:

* Fire-and-forget commands, or
* Query/reply pairs

Commands, queries and replies are serialisable POCO classes, and are published using NuGet packages.

Service messages are at the boundaries of the black box. Think of messages as the interfaces between the test and the service.

### Arrange - Act - Assert[​](https://10.243.110.58/platform/docs/tester/service-tests/testing-services-in-isolation/isolation-test-design#arrange---act---assert)

Use services messages to

1. **Arrange**: set up the test pre-requisites.
2. **Act**: perform the action that triggers the required behaviour.
3. **Assert**: make assertions about the expected outcomes.

The Arrange - Act - Assert pattern should be always used to implement the test logic, and should correspond to the Given - When - Then syntax in SpecFlow features.

## Dependencies on messages of other services[​](https://10.243.110.58/platform/docs/tester/service-tests/testing-services-in-isolation/isolation-test-design#dependencies-on-messages-of-other-services)

Wherever possible, avoid dependencies on messages that do not belong to the service message package under test.

If you cannot avoid a dependency on external messages, the tests should still be executable in isolation, without relying on other services’ message handlers.

### Examples of external message dependencies[​](https://10.243.110.58/platform/docs/tester/service-tests/testing-services-in-isolation/isolation-test-design#examples-of-external-message-dependencies)

#### 1. The service under test issues a command to another service[​](https://10.243.110.58/platform/docs/tester/service-tests/testing-services-in-isolation/isolation-test-design#1-the-service-under-test-issues-a-command-to-another-service)

This is a common occurrence, and is how services talk to each other. Normally the message belongs to the receiver's namespace, not the sender's. In this case the isolation test for the sender only needs to intercept the message, as explained further down this section, and assert that the message was sent as expected. When testing the sender service in isolation, the actual receiver service is not required to handle the command.

#### 2. The service under test sends a query to another service and the reply affects the test outcome[​](https://10.243.110.58/platform/docs/tester/service-tests/testing-services-in-isolation/isolation-test-design#2-the-service-under-test-sends-a-query-to-another-service-and-the-reply-affects-the-test-outcome)

To illustrate this scenario, let us say that you are testing a Reporting service that sends a message to the Notification service when a new report is generated. However, the Reporting service marks the report status as "completed" only after it receives a "success" reply from the Notification service. In this case, you can still achieve test isolation by mocking the Notification service. This involves intercepting the query sent to the Notification service and using a mock message handler to send a canned reply; for example, always reply "success" or always reply "failure".

#### 3. The service under test handles the messages of another service[​](https://10.243.110.58/platform/docs/tester/service-tests/testing-services-in-isolation/isolation-test-design#3-the-service-under-test-handles-the-messages-of-another-service)

This is uncommon. In this scenario the external messages should be included in the test suite.

The following example is based on EMEA Fund Solution services: the File Extracted v1 message belongs to the Dsf.DataPlatform.Extract.FileHandler.ServiceMessages package but is handled by the Dsf.DataPlatform.Monitoring service. The Data Platform Monitoring service tests project will therefore have a dependency on the File Handler service messages, and will include the feature with the File Extracted scenarios.

### Interception[​](https://10.243.110.58/platform/docs/tester/service-tests/testing-services-in-isolation/isolation-test-design#interception)

If the service under test issues a command to another service:

1. Intercept the command
2. Assert that the command was sent with the correct parameters

Intercepting a message is simple step. In the simplest cases, you can just derive your step definitions from the base InterceptMessageSteps<TMessage, TIntercept> class in Dsf.TestFramework.ServiceTests.Core, and re-use the existing SpecFlow bindings such as "Then the message <message name> <version> is sent to the <sevice name>".

The interception logic and associated assertions is implemented in the below classes of the Dsf.TestFramework components:

* Dsf.TestFramework.MessageClient.Interception.Interceptor (Message Client)
* Dsf.TestFramework.Messaging.Interpcetion.Interceptor (Messenger)
* Dsf.TestFramework.ServiceTests.InterceptMessageSteps (Message Client)
* Dsf.TestFramework.ServiceTests.Core.InterceptMessageSteps (Messenger)
* Dsf.TestFramework.Messaging.Assertions.InterceptionAssert (Message Client and Messenger)

### Mocking[​](https://10.243.110.58/platform/docs/tester/service-tests/testing-services-in-isolation/isolation-test-design#mocking)

If the service under test sends a query to another service:

1. Intercept the query
2. Assert that the query was sent with the correct parameters
3. Mock the reply from the other service

Mocking is somewhat more complex that mere interception, because it requires some knowledge of the mocked service and the ability to simulate a relevant reply. The mocking logic extends the interception logic and is implemented in the below re-usable classes of the Dsf.TestFramework components:

* Dsf.TestFramework.MessageClient.Interception.Mocker (Message Client)
* Dsf.TestFramework.Messaging.Interpcetion.Mocker (Messenger)

## Data store dependencies[​](https://10.243.110.58/platform/docs/tester/service-tests/testing-services-in-isolation/isolation-test-design#data-store-dependencies)

A stateful service is a service that persists a state in a data store such as an Azure Table Storage or an Azure Sql database.

A stateless service is a service that is not stateful. Tests for stateless services should never depend on external data stores.

Even when testing stateful services, dependencies on data stores (SQL or Oracle databases, Azure storages, Cosmos Db, etc.) should still be avoided wherever possible.

If the service is stateful and the data store dependency is unavoidable (for example, because the state cannot be validated using service messages) then the data store tests should be separated from the full isolation tests.

# Testing Services in Isolation: NuGet Dependencies

Isolation tests should have no dependencies other than the following:

1. Service messages handled or sent by the service under test.
2. Messaging components used to handle or send the messages.
3. Test framework components required to execute the tests.

## Standard 5 services[​](https://10.243.110.58/platform/docs/tester/service-tests/testing-services-in-isolation/nuget-dependencies#standard-5-services)

DSF services of Standard 5 (S5) target .Net Core and use the Messenger only to send and receive messages.

Test projects created from S5 DSF templates target .Net Core and should reference the following:

1. Dsf.[ServiceName].ServiceMessage
2. Dsf.Messaging, either directly, or indirectly via the below component
3. Dsf.TestFramework.ServiceTests.Core
4. Microsoft.NET.Test.Sdk
5. Optionally, Dsf.TestFramework components for test reporting and execution metrics

The reference to the service message package under test should be updated to the latest project's pre-release version, if available, or the latest product release version.

Regression tests for services that originally belonged to EMEA Fund Solutions may also need to reference service messages in the legacy Capita namespace, in addition to the newer messages in the Dsf namespace.

References to additional services messages, other than those of the service under test, should be added only if necessary. These additional messages should be intercepted or mocked as explained in the [Isolation Test Design](https://10.243.110.58/platform/docs/tester/service-tests/testing-services-in-isolation/isolation-test-design) section.

References to additional Dsf.TestFramework components, other than those listed above, may be required, for example, to connect to Azure storages (using Dsf.TestFramework.Azure.Storage components) or Sql database (using Dsf.TestFramework.DataAccess components). However, these additional dependencies should be avoided whenever possible.

## Third-party NuGet packages[​](https://10.243.110.58/platform/docs/tester/service-tests/testing-services-in-isolation/nuget-dependencies#third-party-nuget-packages)

You should never reference any third-party NuGet packages other than those already referenced by the Dsf.TestFramework components.

Third-party packages may require licensing, pose security risk, or introduce technical debt, especially if they are obsolete, poorly maintained, or at odds with DSF standards.

If you believe that your tests would benefit from referencing a third-party package, other than those supported by the DSF test framework, send an email to the Test Architects ([LGE-DL-TOUK-ITSD-TestArchitects@linkgroup.co.uk](mailto:LGE-DL-TOUK-ITSD-TestArchitects@linkgroup.co.uk)) to request approval for the new package.

# Advantages of Test Isolation

Testability in isolation is one of the main advantages of DSF micro-services, for the following reasons:

### Risk-based regression testing[​](https://10.243.110.58/platform/docs/tester/service-tests/testing-services-in-isolation/advantages-of-test-isolation#risk-based-regression-testing)

Barring deployment issues or orchestration/compatibility issues involving external dependencies, test isolation can prove that a service works the same as before without having to run any end-to-end tests. End-to-end testing is still required to catch deployment and certain integration issues, but it can be reduced to simple sanity checks when all isolation tests have passed.

### Robustness[​](https://10.243.110.58/platform/docs/tester/service-tests/testing-services-in-isolation/advantages-of-test-isolation#robustness)

Isolation tests are less fragile, are not vulnerable to side-effects from external dependencies, and are less likely to need revision when production code changes.

### Speed and coverage[​](https://10.243.110.58/platform/docs/tester/service-tests/testing-services-in-isolation/advantages-of-test-isolation#speed-and-coverage)

Test execution is faster and covers more in less time.

### Test code maintenance[​](https://10.243.110.58/platform/docs/tester/service-tests/testing-services-in-isolation/advantages-of-test-isolation#test-code-maintenance)

Test code is easier to maintain when external dependencies are eliminated or reduced to a bare minimum.

### Finding issues earlier[​](https://10.243.110.58/platform/docs/tester/service-tests/testing-services-in-isolation/advantages-of-test-isolation#finding-issues-earlier)

Non-integration issues can be found in lower environments, before the service is deployed to PAT or UAT.

### Parallel development[​](https://10.243.110.58/platform/docs/tester/service-tests/testing-services-in-isolation/advantages-of-test-isolation#parallel-development)

We can start testing the service as soon as it is deployed, even when the other services, APIs, and front applications are under development and are not ready for testing yet.

### Focussed diagnosis[​](https://10.243.110.58/platform/docs/tester/service-tests/testing-services-in-isolation/advantages-of-test-isolation#focussed-diagnosis)

If an isolation test fails it is easier to pinpoint the cause, because the failure should not have been caused by an external dependency.

### Modularity[​](https://10.243.110.58/platform/docs/tester/service-tests/testing-services-in-isolation/advantages-of-test-isolation#modularity)

Isolation test suites, and the associated documentation, are granular and self-contained. The test suites can be organised in separate solutions, repositories or CI pipeLines, in a way that tracks and mirrors the structure of production code.

### Sign-off of individual builds[​](https://10.243.110.58/platform/docs/tester/service-tests/testing-services-in-isolation/advantages-of-test-isolation#sign-off-of-individual-builds)

Releases of complex products will normally require us to sign off multiple release candidates, one for each service, API or application required by the new product. Test isolation allows us to sign-off each build independently of the others.

### Lean deployments[​](https://10.243.110.58/platform/docs/tester/service-tests/testing-services-in-isolation/advantages-of-test-isolation#lean-deployments)

It should be possible to test a DSF service without deploying any REST API or front-end application.

# API Tests Overview

This section provides step-by-step instructions to create and configure test projects for REST APIs developed within the DSF.

The how-to steps are followed by introductory guidelines to testing DSF APIs with RestSharp, Postman and JMeter.

# Adding a New API Test Project

The addition of new API test projects is normally performed by repository maintainers.

Testers without maintainer role will usually add or update endpoint tests that belong to existing API test projects.

Read this section only if you are a repository maintainer or you need to create a test project for a new DSF API.

## Prerequisites[​](https://10.243.110.58/platform/docs/tester/api-tests/adding-a-new-api-test-project#prerequisites)

A dedicated repository for API tests should be created by the Architecture team for each new DSF API, alongside the developers' repository.

New endpoints that belong to the Dsf.ServicesGateway or the Dsf.Services.Api are an exception to the above rule. When there is a one-to-one correspondence between API endpoints and service messages, then the API test project and the associated service test project can be located in the same repository and can belong to the same solution.

You will need at least Developer-level access to the relevant repository. If you cannot see the repository, request access as explained [here](https://10.243.110.58/platform/docs/tester/test-code-management/repositories/requesting-access-to-a-repository).

## Create the test project using DSF templates[​](https://10.243.110.58/platform/docs/tester/api-tests/adding-a-new-api-test-project#create-the-test-project-using-dsf-templates)

1. Ensure you have installed the latest version of the DSF Test Templates for Visual Studio.
2. Open Visual Studio and select File > New Project...
3. In the Add new project dialog:
   * Visual Studio 2017: select Visual C# > Test
   * Visual Studio 2019: type "DSF" in the template search box
4. Select DSF API Test Project (.NET Core).
5. Enter the project name: **Dsf.[ApiName].ApiTests**
6. Click Browse... and select the repository location on your computer.
7. If the solution does not already exist, select Create new solution from the Solution drop-down menu, and tick the "Create directory for solution" checkbox.

8 .The naming convention is as follows:

* Solution file in Pascal case: **Dsf.[ApiName].Tests**
* Project file in Pascal case: **Dsf.[ApiName].ApiTests**. The project file suffix is .ApiTest in order to distinguish it from .PerformanceTests .ServiceTests and .TestClient projects that might belong to the same .Tests solution.

1. Click Ok.
2. Rename to src the top-level directory. TeamCity build templates are configured to build only the content of the src directory.
3. Ensure that the new solution has been created correctly. The folder structure should be as follows:

* **src** (top-level folder with solution file)
  + **Dsf.ApiName.ApiTests** (project folder with project file)
    - **Features**
      * v1
      * v2
      * etc.
    - **Steps**
      * v1
      * v2
      * etc.

The v2 directories are shown only for illustration purposes. If the DSF API is brand new, it should only have v1 endpoints.

## Raise a merge request[​](https://10.243.110.58/platform/docs/tester/api-tests/adding-a-new-api-test-project#raise-a-merge-request)

If you are not the repository maintainer, raise a merge request before you start adding endpoint features to the newly created project. Assign your request to a repository maintainer for code review.

Repository maintainers should ensure that the new project is being added to the correct repository, that the directory structure is correct, and there are no deviations from the naming convention.

# Configuring API Test Settings

Every API test project should include the **BaseUrl** and the **Messaging** configuration sections. The **Security** configuration section is required only for Services Gateway tests.

The **Messaging:Messenger** section is required to intercept and mock service messages triggered by API calls.

The **Security:Users** section is used get access tokens when authentication is enabled in the Services Gateway.

The **Logging** configuration section is optional. It is used to capture Kibana logs. In the below example, Splunk log capture is not supported.

Example of default configuration: **appsettings.json**

{  
 "Environment": "tst",  
 "Logging": {  
 "Kibana": {  
 "SourcePrefix": "#SourcePrefix",  
 "Source": "servicesgateway\_api ",  
 }  
 },  
 "Messaging": {  
 "Messenger": {  
 "ServiceCode": "9dd38c610c6a48aaa736be7cb4b8f85c",  
 "ServiceName": "documentstorage.apitests",  
 "MessageManagerEndpoint": "#MessageManagerEndpoint",  
 "Scope": "#Scope"  
 }  
 },  
 "BaseUrl": {  
 "ServicesGateway": "#Services Gatewawy base url"  
 },  
 "Security":{  
 "Enabled": true  
 }  
}

Example of primary configuration: **appsettings.tst.json**

{  
 "Logging": {  
 "Kibana": {  
 "SourcePrefix": "abctst\_"  
 }  
 },  
 "Messaging": {  
 "Messenger": {  
 "MessageManagerEndpoint": "https://genpatmessagemanager.casfs.co.uk/api",  
 "Scope": "abctst"  
 }  
 },  
 "BaseUrl": {  
 "ServicesGateway": "https://abctstservicesgateway.casfs.co.uk"  
 },  
 "Security": {  
 "Users": [  
 {  
 "Id": 1,  
 "Name": "Pega",  
 "SecurityKey": "1gd10b8508894c33802e1e04f752126g"  
 },  
 {  
 "Id": 2,  
 "Name": "SAP",  
 "SecurityKey": "4546701875da4f5697bcaf175a02c0d8"  
 }  
 ]  
 }  
}

# Postman

The Postman API platform can be optionally used, in addition to the Dsf.TestFramework, for exploratory testing, sanity checks, or to troubleshoot test failures.

However, **Postman scripts should never replace the usual DSF automated tests**, for the reasons outlined below under the **Limitations of Postman** header.

## Prerequisite: Installing Postman[​](https://10.243.110.58/platform/docs/tester/api-tests/postman#prerequisite-installing-postman)

Download Postman from the below location and install it on your VDI:

<https://www.postman.com/downloads/>

## Postman collections[​](https://10.243.110.58/platform/docs/tester/api-tests/postman#postman-collections)

Postman collections are a group of saved REST requests, in JSON format.

If you have used Postman in addition to the usual DSF automated tests, the Postman collection should be pushed to the same remote repository where DSF API tests are located, but in a separate directory called 'postman'.

### Naming convention[​](https://10.243.110.58/platform/docs/tester/api-tests/postman#naming-convention)

The naming convention is a follows:

**postman/<CollectionName>\_collection.json**

1. The **postman** directory should be a top-level directory, outside of the **src** directory.
2. The name of the **postman** directory should be in lower case.
3. The collection name should be in PascalCase, followed by **\_collection.json** in lower case.
4. The collection name should be based on the endpoint names and endpoint places in the resource hierarchy.

For example, a collection consisting of investor/address/main and investor/address/correspondence requests should be saved to:

postman/InvestorAddress\_collection.json

## Advantages of Postman[​](https://10.243.110.58/platform/docs/tester/api-tests/postman#advantages-of-postman)

Postman is easier to learn, compared to C#/RestSharp. It also a good practical introduction to technical testing for those who are unfamiliar with REST API automation.

### Exploratory testing[​](https://10.243.110.58/platform/docs/tester/api-tests/postman#exploratory-testing)

Postman makes it easy to inspect REST responses, view status codes or response times, and share your findings with other testers or developers.

Developers sometimes create Postman scripts and add them to API repositories. If a developer has created a Postman collection, ask them to share it with the testing team, as this can speed up test planning and test design.

### RestSharp integration[​](https://10.243.110.58/platform/docs/tester/api-tests/postman#restsharp-integration)

RestSharp is a widely used HTTP client library for .Net. The DSF test framework for API tests is based on RestSharp.

Postman scripts can be converted to RestSharp as follows:

1. Open the rest request in Postman
2. Select the Code option to the right-hand side
3. View the Generate Code Snippet window, and type 'C#' in the Filter language... text box.
4. Ensure that C# - RestSharp is selected. The Settings for C# - RestSharp can be left to the default values.
5. Click the Copy button to the right-hand side to save the code snippet to the clipboard.

Use the code snippets with caution. Do not simply copy and paste the auto-generated code into the DSF test project. For example, the base url or the bearer token should never be hard-coded. The code snippet generated by Postman should be used only as a preliminary guide, to see how the RestRequest object can be defined programmatically using C# with RestSharp.

## Limitations of Postman[​](https://10.243.110.58/platform/docs/tester/api-tests/postman#limitations-of-postman)

While Postman is useful for reasons outlined above, it has the following limitations from a DSF automation perspective:

* It cannot be used to test DSF APIs in isolation.
* It cannot be used to test the serialisation/deserialisation logic that maps REST Responses or REST Requests to DSF service messages.
* It is not included in continuous integration. Postman scripts are ignored by DSF build templates configured in TeamCity.
* It does not lend itself to BDT (Behaviour Driven Testing) as well as SpecFlow does.

Testing DSF APIs exclusively with Postman is therefore not recommended.

# JMeter

Apache JMeter is an open source application for load, stress and performance testing.

Even though it is a pure Java application, JMeter is the recommended performance tool for Azure-based APIs, including DSF APIs.

Microsoft themselves have deprecated the load test functionality built in Visual Studio 2019 and recommend using JMeter instead. For example, Microsoft's [Azure Load Testing](https://docs.microsoft.com/en-us/azure/architecture/framework/scalability/performance-test) tool is a cloud-based service that runs test scripts based on JMeter.

## Prerequisites[​](https://10.243.110.58/platform/docs/tester/api-tests/jmeter#prerequisites)

1. JMeter requires Java 8 (1.8) or higher. To find out whether java is installed or you Vdi, open the command prompt and type java -version. The build number should start with 1.8.0 or higher.
2. If Java 8 or higher is not already installed on your Vdi, download an install the latest Java Development Kit (JDK) for Windows from the official Oracle website.
3. Download the latest version of JMeter from <https://jmeter.apache.org/download_jmeter.cgi>
4. Simply unzip the compressed files into the directory where you want JMeter to be installed.
5. Ensure that your VDI, or the server that runs JMeter scripts, is powerful enough in terms of CPU and RAM. The minimum specifications will depend on the payload and the number of concurrent threads in your load test plan. Consider increasing he Java Heap size as explained further down this section.

## JMeter test plans[​](https://10.243.110.58/platform/docs/tester/api-tests/jmeter#jmeter-test-plans)

JMeter scripts are called "test plans" and are saved in XML format.

A test plan typically consist of one or more thread groups, samplers, listeners, timers, assertions, and configuration elements.

Thread groups are the starting points of any test plan, and define the number of concurrent users, the ramp-up period and the number of iterations.

A single test plan may cover multiple endpoints.

If JMeter test plans have been designed in addition to the usual functional tests, the .jmeter files should be pushed to the same remote repository where the DSF API tests are located, but in a separate directory called **jmeter**.

### Naming convention[​](https://10.243.110.58/platform/docs/tester/api-tests/jmeter#naming-convention)

The naming convention is a follows:

**jmeter/[TestPlanName].jmeter**

1. The **jmeter** directory should be a top-level directory, outside of the src directory.
2. The name of the **jmeter** directory should be in lower case.
3. The test plan name should be in PascalCase, followed by the **.jmeter** file extension.
4. The test plan name should be based on the API name, endpoint names or endpoint places in the resource hierarchy.

For example, a test plan for the investor/orders/{id}, investor/orders/pending, and investor/orders/{id}/status endpoints should be saved to

jmeter/InvestorOrders.jmeter

## Limitations of JMeter[​](https://10.243.110.58/platform/docs/tester/api-tests/jmeter#limitations-of-jmeter)

* JMeter cannot be used to performance test DSF services in isolation.
* Because JMeter is Java-based, its test plans cannot be easily integrated with functional tests designed within the .NET ecosystem.

## Best practices: Basic concepts[​](https://10.243.110.58/platform/docs/tester/api-tests/jmeter#best-practices-basic-concepts)

### GUI mode vs CLI mode[​](https://10.243.110.58/platform/docs/tester/api-tests/jmeter#gui-mode-vs-cli-mode)

JMeter can be started in either GUI (window) mode or CLI (command-line) mode. The mode is important because it affects the speed of text execution and the test results.

### Do and Don'ts[​](https://10.243.110.58/platform/docs/tester/api-tests/jmeter#do-and-donts)

✅ Use the GUI mode to design and debug JMeter test plans.

✅ Use the CLI mode to execute JMeter test plans.

✅ Parametrise the number of threads (users), the ramp-up period in seconds, and the loop count in each thread group.

❌ Do not use the GUI mode to execute load tests.

❌ Do not add custom plugins or extensions to the standard JMeter installation.

❌ Do not use commercial products based on JMeter, such as BlazeMeter.

### Troubleshooting "out of memory" errors on your VDI[​](https://10.243.110.58/platform/docs/tester/api-tests/jmeter#troubleshooting-out-of-memory-errors-on-your-vdi)

1. Ensure that no instances of JMeter are running in GUI mode.
2. Close all other windows applications, other than JMeter in CLI mode.
3. Increase the Java Heap size. By default, this is set to 1 Gb.

### How to increase the Heap size[​](https://10.243.110.58/platform/docs/tester/api-tests/jmeter#how-to-increase-the-heap-size)

Edit the following line in bin/jmeter.bat:

HEAP=-Xms1g -Xmx1g

* Xms is the amount of RAM allocated at startup to the Java virtual machine (JVM)
* Xmx is the maximum amount of RAM that is allowed for the Heap.

Increase the -Xmx value as required. For example: if you need to set the maximum heap size to 4 gigabytes, change the line to:

HEAP="-Xms1g -Xmx4gm"

Restart JMeter to apply the change.

## Best practices : Advanced topics[​](https://10.243.110.58/platform/docs/tester/api-tests/jmeter#best-practices--advanced-topics)

Detailed discussion of how to build and execute JMeter test plans falls outside the scope of the DSF documentation.

As a further reading, consult the [Best practices](https://jmeter.apache.org/usermanual/best-practices.html) page from the official Apache JMeter documentation.

If you need advice on designing and executing performance tests at the DSF API layer, contact the Test Architecture team via email:

* **Subject**: API performance testing
* **To**: [LGE-DL-TOUK-ITSD-TestArchitects@linkgroup.co.uk](mailto:LGE-DL-TOUK-ITSD-TestArchitects@linkgroup.co.uk)

In the email body, provide the project code, the API name, and, if known, the Git path to the repository that should store the .jmeter files.