Guidewire PolicyCenter™

ISBTF and GUnit Testing Guide

Release 10.1.2

GUIDEWIRE
Adapt and succeed™

Product Name: Guidewire PolicyCenter
Product Release: 10.1.2
Document Name: ISBTF and GUnit Testing Guide
Document Revision: 14-June-2021

Contents

Part 2
Mapping steps to implementation code

Part 5
Platform-level testing functionality

Guidewire Documentation

About PolicyCenter documentation

The following table lists the documents in PolicyCenter documentation:

Document	Purpose
InsuranceSuite Guide	If you are new to Guidewire InsuranceSuite applications, read the *InsuranceSuite Guide* for information on the architecture of Guidewire InsuranceSuite and application integrations. The intended readers are everyone who works with Guidewire applications.
Application Guide	If you are new to PolicyCenter or want to understand a feature, read the *Application Guide*. This guide describes features from a business perspective and provides links to other books as needed. The intended readers are everyone who works with PolicyCenter.
Upgrade Guide	Describes the overall upgrade process, and describes how to upgrade your configuration and database. The intended readers are implementation engineers who must merge base application changes into existing application extensions and integrations.
Configuration Upgrade Tools Guide	Describes the tools and functionality provided by the Guidewire InsuranceSuite Configuration Upgrade Tools. The intended readers are implementation engineers who must merge base application changes into existing application extensions and integrations. Visit the Guidewire Community to access the *Configuration Upgrade Tools Guide*, which is available for download, separately from the main documentation set, with the Configuration Upgrade Tools.
Installation Guide	Describes how to install PolicyCenter. The intended readers are everyone who installs the application for development or for production.
System Administration Guide	Describes how to manage a PolicyCenter system. The intended readers are system administrators responsible for managing security, backups, logging, importing user data, or application monitoring.
Configuration Guide	The primary reference for configuring initial implementation, data model extensions, and user interface (PCF) files for PolicyCenter. The intended readers are all IT staff and configuration engineers.
PCF Format Reference	Describes PolicyCenter PCF widgets and attributes. The intended readers are configuration engineers. See the *Configuration Guide*
Data Dictionary	Describes the PolicyCenter data model, including configuration extensions. The dictionary can be generated at any time to reflect the current PolicyCenter configuration. The intended readers are configuration engineers.
Security Dictionary	Describes all security permissions, roles, and the relationships among them. The dictionary can be generated at any time to reflect the current PolicyCenter configuration. The intended readers are configuration engineers.
Globalization Guide	Describes how to configure PolicyCenter for a global environment. Covers globalization topics such as global regions, languages, date and number formats, names, currencies, addresses, and phone numbers. The intended readers are configuration engineers who localize PolicyCenter.
Rules Guide	Describes business rule methodology and the rule sets in Guidewire Studio for PolicyCenter. The intended readers are business analysts who define business processes, as well as programmers who write business rules in Gosu.
Guidewire Contact Management Guide	Describes how to configure Guidewire InsuranceSuite applications to integrate with ContactManager and how to manage client and vendor contacts in a single system of record. The intended readers are PolicyCenter implementation engineers and ContactManager administrators.
Best Practices Guide	A reference of recommended design patterns for data model extensions, user interface, business rules, and Gosu programming. The intended readers are configuration engineers.
Integration Guide	Describes the integration architecture, concepts, and procedures for integrating PolicyCenter with external systems and extending application behavior with custom programming code. The intended

Document	Purpose
	readers are system architects and the integration programmers who write web services code or plugin code in Gosu or Java.
REST API Client Guide	Describes how to use the InsuranceSuite REST API Client to make outbound HTTP calls to internal or third-party REST services.
Java API Reference	Javadoc-style reference of PolicyCenter Java plugin interfaces, entity fields, and other utility classes. The intended readers are system architects and integration programmers.
Gosu Reference Guide	Describes the Gosu programming language. The intended readers are anyone who uses the Gosu language, including for rules and PCF configuration.
Gosu API Reference	Javadoc-style reference of PolicyCenter Gosu classes and properties. The reference can be generated at any time to reflect the current PolicyCenter configuration. The intended readers are configuration engineers, system architects, and integration programmers.
ISBTF and GUnit Testing Guide	Describes the tools and functionality provided by InsuranceSuite for testing application behavior during an initial implementation or an upgrade. The guide covers functionality related to Behavior Testing Framework, GUnit, and Gosu functionality designed specifically for application testing. There are two sets of intended readers: business analysts who will assist in writing tests that describe the desired application behavior; and technical developers who will write implementation code that executes the tests.
Glossary	Defines industry terminology and technical terms in Guidewire documentation. The intended readers are everyone who works with Guidewire applications.
Advanced Product Designer Guide	Advanced Product Designer is a tool that helps you, a business user, design, simulate, and deploy an insurance product. The intended readers are business analysts who understand insurance products, business systems analysts who liaise between business analysts and IT, project managers, and IT who provides technical expertise in areas such as programming, testing, and databases.
Product Model Guide	Describes the PolicyCenter product model. The intended readers are business analysts and implementation engineers who use PolicyCenter or Product Designer. To customize the product model, see the *Product Designer Guide*.
Product Designer Guide	Describes how to use Product Designer to configure lines of business. The intended readers are business analysts and implementation engineers who customize the product model and design new lines of business.
REST API Framework	Describes the Guidewire InsuranceSuite framework that provides the means to define, implement, and publish REST API contracts. It also describes how the Guidewire REST framework interacts with JSON and Swagger objects. The intended readers are system architects and integration programmers who write web services code or plugin code in Gosu or Java.

Conventions in this document

Text style	Meaning	Examples
italic	Indicates a term that is being defined, added emphasis, and book titles. In monospace text, italics indicate a variable to be replaced.	A *destination* sends messages to an external system. Navigate to the PolicyCenter installation directory by running the following command: `cd installDir`

bold	Highlights important sections of code in examples.	`for (i=0, i<someArray.length(), i++) {`
		`  newArray[i] = someArray[i].getName()`
		`}`

narrow bold	The name of a user interface element, such as a button name, a menu item name, or a tab name.	Click **Submit**.

Text style	Meaning	Examples
`monospace`	Code examples, computer output, class and method names, URLs, parameter names, string literals, and other objects that might appear in programming code.	The `getName` method of the `IDoStuff` API returns the name of the object.
`monospace italic`	Variable placeholder text within code examples, command examples, file paths, and URLs.	Run the `startServer` *`server_name`* command. Navigate to `http://`*`server_name`*`/index.html`.

Support

For assistance, visit the Guidewire Community.

Guidewire customers

`https://community.guidewire.com`

Guidewire partners

`https://partner.guidewire.com`

Part 1

Getting started

Behavior Testing Framework is a Guidewire product that enhances the capabilities of PolicyCenter by providing resources and an infrastructure so that insurers can implement Behavior-Driven Development.

PolicyCenter provides Behavior Testing framework in a compressed file. You must install the contents of this file into PolicyCenter.

chapter 1

Behavior Testing Framework overview

Behavior Testing Framework is a Guidewire product that enhances the capabilities of PolicyCenter by providing resources and an infrastructure so that insurers can implement Behavior-Driven Development (BDD).

Behavior Testing Framework extends the GUnit testing framework in the base application. The GUnit testing framework provides:

- An environment that supports writing and running the tests for your Gosu configuration code.
- The ability to run individual tests and complete test suites within Guidewire Studio.
- The ability to run test suites at a command prompt or in a continuous integration work flow.

Behavior Testing Framework provides the following additional functionality:

- The ability to describing software behavior in natural language that is directly tied to implementation code.
- A framework that makes it possible to share implementation code common to broad business contexts, as well as isolate code specific to more narrow business contexts.
- The ability to test pages and screens using the native page model.

Behavior Testing Framework provides both the testing framework and examples of how to use the framework.

Goals of Behavior Testing Framework

Behavior Testing Framework was designed to support several goals.

Support for Behavior-Driven Development

Behavior-Driven Development (BDD) is a software development approach where the focus is placed on describing user behavior and verifying that the software produces the expected results from that behavior. Ideally, you write these descriptions of behavior before the software is developed, as well as tests that verify the behavior. Initially, the tests will fail because the functionality has not yet been implemented. As functionality is developed, the tests pass. This ensures the software supports the desired behaviors.

Behavior-Driven Development emphasizes collaboration between business resources and technical resources. A fundamental belief is that software development is more effective and more efficient when the stakeholders and the agile team discuss requirements using a common language.

- Analysts contribute by analyzing their business and communicating the business behavior expected from the application.
- Software developers and QA developers contribute by clarifying the technical implementation on the user behavior and implementing the step definitions

In Behavior-Driven Development, software behaviors are written as a set of *scenarios*. Each scenario describes user behavior and its impact on the state of the system.

Behavior Testing Framework supports this goal in the context of InsuranceSuite by providing an integration with Cucumber and enhancing the GUnit framework, a tool that reads executable specifications written in natural language.

Separation of business details from implementation details

Some testing frameworks use a single layer of resources to define testing requirements. This approach tends to be highly technical. As a result, business experts are forced to play a secondary role in testing development because they do not usually express business requirements in a highly technical manner.

Behavior Testing Framework uses multiple layers of resources. This includes:

- *The business layer*, which consists primarily of scenarios. Scenarios are written in Gherkin, an executable language that reads like natural language. Both business experts and technical experts can write scenarios in Gherkin.
- *The context implementation layer*, which consists primarily of Gosu classes that specify implementation details in a highly technical manner.
- *The mapping layer*, which connects the business layer and the context implementation layer, and provides the freedom to modify resources in one layer without needing to make changes to resources in the other layer.

Sharing and isolating implementations

In some circumstances, different scenarios must perform a given action in the same way. For example, consider the action of assigning a user to a group. This action is always performed in the same way, regardless of the scenario that executes it.

In other circumstances, different scenarios must perform a given action in different ways. For example, consider the action of adding an optional coverage during a submission. The possible values vary based on the policy's type of product. In other words, this action is performed in different ways, depending on the scenario that executes it.

Behavior Testing Framework is designed to support both of these use cases. When a test step behaves the same way in all circumstances, all scenarios using this test step execute the same implementation code. When a test step must behave differently based on the scenario, multiple implementations exist, and Behavior Testing Framework selects the correct implementation for each scenario.

Provide for easy extensibility

Behavior Testing Framework comes with a set of business scenarios and implementations to test these scenarios. These resources provide examples for how to write and implement scenarios. They are based on the base configuration of PolicyCenter, and all scenarios will pass when run against the base configuration.

However, during an implementation of PolicyCenter, insurers configure the application to meet their business needs. Behavior Testing Framework can similarly be modified to test an insurer's configurations.

- New scenarios can be added.
- New implementation resources can be added.
- Existing implementation resources can be extended.
- New implementation strategies can be added, such as conducting tests with REST APIs.

Technologies used in Behavior Testing Framework

The bulk of the technology used by Behavior Testing Framework is identical to that of PolicyCenter . Behavior Testing Framework uses Gosu classes and interfaces created through the Guidewire Studio IDE.

Behavior Testing Framework also uses the following technologies.

Cucumber and Gherkin

Cucumber is an open-source software testing tool that is capable of running automated acceptance tests. Cucumber is designed to run tests created in a behavior-driven development environment. Studio includes a plugin to support Cucumber.

Gherkin is a language used by Cucumber to define tests. It uses natural language so that test writing can be a collaborative effort between business experts and technical experts.

For more information, refer to the Cucumber Documentation at https://docs.cucumber.io/guides/.

Google Guice

To share and isolate implementations appropriately, Behavior Testing Framework uses dependency injection. *Dependency injection* is a software design approach in which one object provides the dependencies of another object. The act of passing the dependency object into another object is referred to as *injecting*. The primary goal of dependency injection is to isolate objects so that no dependent object needs to be changed solely because an object that it depends on is being changed. This approach adheres to the Open/Closed principle of object-oriented programming.

Cucumber supports several dependency injection frameworks. Behavior Testing Framework uses Google Guice. *Google Guice* (pronounced like "juice") is an open-source Java framework that supports dependency injection. Dependencies can be injected into dependent objects using the @Inject annotation.

The following is an example of the `AdminSteps` class. It defines a `createUserWithRoleAndSetAsCurrentUser` method, which must identify and execute the correct implementation of `createAndSetUserWithRole`. The method is able to do this using an instance of `ContextFactory`, which has been injected into the class.

```
class AdminSteps {
  @Inject
  var _contextFactory : ContextFactory

  @Given("^I am a user with the \"([^\"]*)\" role$")
  function createUserWithRoleAndSetAsCurrentUser(roleString: String) {
    _contextFactory.getAdminContext().createAndSetUserWithRole(roleString)
  }
```

For more information, refer to the Guice Documentation at https://github.com/google/guice.

chapter 2

Installing Behavior Testing Framework

Guidewire provides the Behavior Testing Framework extension pack in a compressed file. You must extract the contents of this compressed file. The following directions refer to the extracted contents as the "extracted directory".

Note: Each release of Behavior Testing Framework supports a specific version of the core application. You must ensure that the version of Behavior Testing Framework is compatible with your version of the core application.

To install Behavior Testing Framework for ContactManager, see “Install Behavior Testing Framework for ContactManager” on page 107.

Install Behavior Testing Framework

About this task

The following instructions are for installing Behavior Testing Framework for PolicyCenter. To install Behavior Testing Framework for ContactManager, see “Installing Behavior Testing Framework for ContactManager” on page 107.

Procedure

1. Extract the Behavior Testing Framework distribution file.
   - The distribution file is named `pc-isbtf.zip`.
   - The distribution file is located in the core product's `isbtf` subdirectory.
2. Install the framework files.
   a. In the extracted directory, navigate to the `framework` directory. This directory contains a `modules` directory.
   b. Merge the `modules` directory into the PolicyCenter directory.
   c. If messages appear that ask whether you want to merge `modules` and any other directories, select the option that merges the directory.
3. Enable code generation of the native page model. This lets Behavior Testing Framework interact with PCF elements and screens.
   a. In the PolicyCenter directory, open the `gradle.properties` file in a text editor.
   b. Locate the `isPcfTestCodegenEnabled` property and set its value to `true`.
   c. Save and close the file.
4. Add a command for running a Cucumber test suite to the `build.gradle` file.

a. In the PolicyCenter directory, navigate to the `modules/configuration` directory.
b. Open both the `build.gradle` file and the `build.gradle_isbtf` file in a text editor.
c. In the `build.gradle_isbtf` file, locate the line that contains the following text:

```
/** ISBTF: ONLY MERGE THE LINE BELOW */
```

d. Copy the line between the `/** ISBTF: ONLY MERGE THE LINE BELOW */` and `/** ISBTF: ONLY MERGE THE LINE ABOVE */` lines.
e. Paste the copied line at the end of the `build.gradle` file.
f. Save and close the `build.gradle` file.
g. Close the `build.gradle_isbtf` file.

5. Some files in Behavior Testing Framework refer to generated PCFs. If the generated files do not exist, the application will not start. To ensure the files exist.
   a. Open Guidewire Studio.
   b. Click **Codegen→Generate Page Configuration Classes**. When the class generation is complete, Studio shows a "Page configuration codegen done..." message in the lower left corner.

Install the behavior testing suite

Procedure

1. In the extracted directory you created in “Install Behavior Testing Framework” on page 17, navigate to the `behavior` directory. This directory contains a `modules` directory.
2. Merge the `modules` directory to the PolicyCenter directory.
3. If messages appear that ask whether you want to merge `modules` or any other directories, select the option that merges the directory.
4. If other messages that ask whether you want to replace existing files appear, select the options that replace the existing files.

Install the behavior testing example

Procedure

1. In the extracted directory that you created in “Install Behavior Testing Framework” on page 17, navigate to the `example` directory. This directory contains a `modules` directory.
2. Merge the `modules` directory to the PolicyCenter directory.
3. If messages appear that ask whether you want to merge the `modules` and any other directories, select the option that merges the directory.
4. If other messages that ask whether you want to replace existing files appear, select the options that replace the existing files.

Verify the Behavior Testing Framework installation

About this task

Check that you installed the framework correctly. You can do this by: (1) verifying that the appropriate directories and files have been added, (2) running a base configuration feature file, and (3) running a base configuration suite.

The scenarios provided in Behavior Testing Framework will pass when executed against a stand-alone base configuration instance of PolicyCenter. If the instance has been integrated with other Guidewire applications or configured, some tests may not pass.

Procedure

1. Open Guidewire Studio.
2. Optionally verify that the following directories and files exist:
   - In `configuration/gsrc/gw/api/test`, a `pcSmokeTestClassBase` file.
   - A `configuration/gsrc/gw/smoketest.pl` directory.
   - A `configuration/gtest/gw/cucumber` directory.
   - A `configuration/gtest/gw/enhancement` directory.
   - In `configuration/gtest/gw/suites`, a `pcExampleSmokeSuite` file.
   - A `configuration/gtest/gw/smoketest` directory.
   - In `configuration/res/cucumber`, multiple subfolders that contain Cucumber feature files.
3. If you are installing Behavior Testing Framework on a configured instance of PolicyCenter, you may see compile errors in Studio. These errors are caused by impl methods in Behavior Testing Framework that reference PCF elements in the base configuration that have been either modified or removed in the configured instance. Resolve these compile issues before proceeding to the next step. You can resolve each issue by doing the following:
   a. Identify the impl method that is causing the compile error.
   b. Comment out the code in the method.
   c. Add a line of code that throws an UnsupportedOperationException. For example: `throw new UnsupportedOperationException("Not yet implemented")`

   If you resolve the issues in this way, then all code will compile. When you run a scenario that references one of these impl methods, the scenario will fail due to the `UnsupportedOperationException`.
4. Test a base configuration feature file.

   The feature file assumes that PolicyCenter has been installed in stand-alone mode and has not been configured. When run on a stand-alone base configuration of PolicyCenter, all scenarios in the feature file will pass. If PolicyCenter has been integrated with other Guidewire applications or configured, one or more scenarios may fail.
   a. Navigate to `configuration/res/cucumber/account`.
   b. Right-click the `AccountManagement_Agent.feature` file, and then click **Run 'Feature: AccountManagement_Agent'**.

   Studio opens a **Run Feature <FeatureName>** tab. The left pane contains icons that identify the tests that pass or fail. The right pane contains a console with the server output during testing.
5. Test a base configuration suite.

   A suite is a collection of feature files that are executed together. The suite assumes that PolicyCenter has not been configured. When run on a base configuration of PolicyCenter, all scenarios in the suite will pass. If PolicyCenter has been configured, one or more scenarios may fail.

   There is a known issue with Behavior Testing Framework that prevents a test suite from running if there is a running test server. As a workaround, if there is a running test server, stop it before executing the following steps.
   a. Navigate to `configuration/gtest/gw/suites`.
   b. Right-click the `PCBehaviorCucumberSuite` file, and then click **Run 'PCBehaviorCucumberSuite'**.

   Studio opens a **Run <SuiteName>** tab. The left pane contains icons that identify the tests that pass or fail. The right pane contains a console with the server output during testing.

Resolving errors that occur during verification

Errors that occur when executing a test suite

The following table lists errors that you might see when executing a suite, as well as possible causes and recommend fixes.

Error message	Possible cause	Suggested fix
Studio throws an `ArrayIndexOutOfBounds` exception	The test server might already be running.	Stop the test server before executing the file.

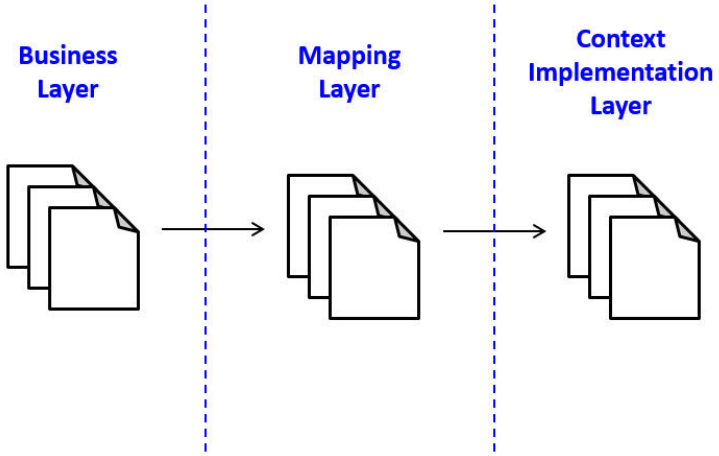
chapter 3

Behavior Testing Framework architecture

This topic provides an overview of the Behavior Testing Framework architecture. It provides a high-level understanding of scenarios, step definitions, Context Factory, the context APIs, and the way in which these components interact.

Behavior Testing Framework layers

Behavior Testing Framework consist of a set of resources that can be grouped into three layers:



The business layer

The business layer is the set of resources that define the behaviors to be tested. The fundamental building blocks of this layer are the scenarios. A *scenario* is a description of user or system behavior that can be tested. Scenarios are written in the Gherkin language. They typically have three sections:

- The Given section defines pre-conditions.
- The When section defines the actions to test.
- The Then section defines the expected outcome.

The following is an example of a test written in Gherkin.

```
Scenario: Set a user's email address
    Given I am a user with the "Underwriting Supervisor" role
    When I set my email address to "ssmith@customer.com"
    Then my primary email address should be "ssmith@customer.com"
```

Other than the scenario name, every line in a scenario is referred to as a step. A *step* is a reusable description of one of the following:

- A pre-condition that describes the environment (stated as a Given), or
- An action to test within that environment (stated as a When), or
- An expected outcome of the action (stated as a Then).

Any given step can be used in multiple scenarios. This gives you the ability to use the same natural language description in multiple scenarios, even if the underlying implementations must be different.

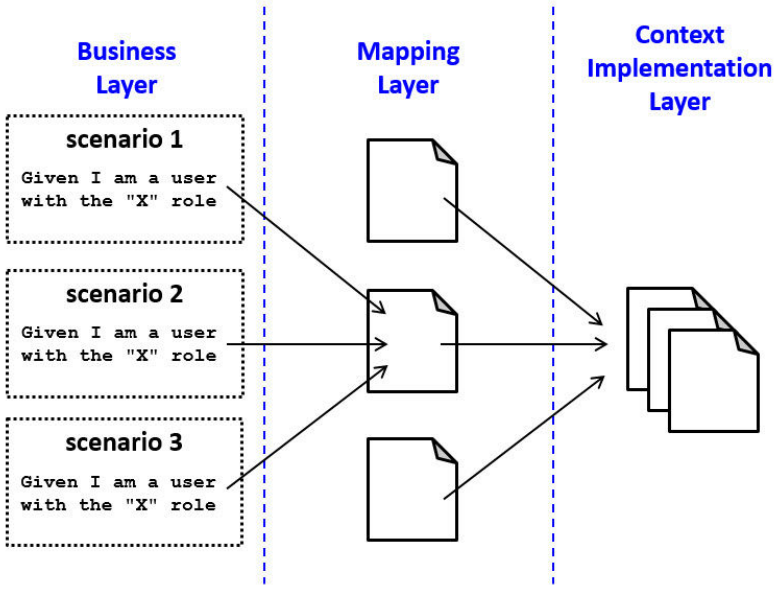
Every step maps to a step method. A *step method* is a method that locates and executes the correct implementation code for the associated scenario steps.

Scenarios are typically grouped together into features. A *feature* is a Cucumber file that contains a set of related scenarios. Features typically contain scenarios for a single type of user that test a single aspect of application behavior. Features can also contain a *Background*, which is a set of pre-condition steps that apply to all scenarios in the feature. The same step method can be used in multiple scenarios, even when those scenarios are in different features.

> **Note:** The remainder of this documentation refers to Cucumber features as "feature files". This is to prevent confusion with the more common use of the term "feature" to refer to application behavior, such as the "assignment feature".

Scenarios can be executed individually from Studio. Scenarios can also be grouped and run together using a testing suite. For more information on executing scenarios, see “Running scenarios” on page 41.

The following diagram depicts an example of resources in the business layer. Specifically, this example is three separate feature files that all use the same step method: `Given I am a user with the "X" role`. All three step methods point to a single resource in the mapping layer.



For more information on writing scenarios and creating feature files, see “Writing scenarios” on page 31.

The mapping layer

The mapping layer is the set of resources that map the scenarios to the implementation code that executes them. The resources in this layer are sometimes referred to as glue code. (*Glue code* is an programming term code that exists to connect different sets of code that are otherwise incompatible.)

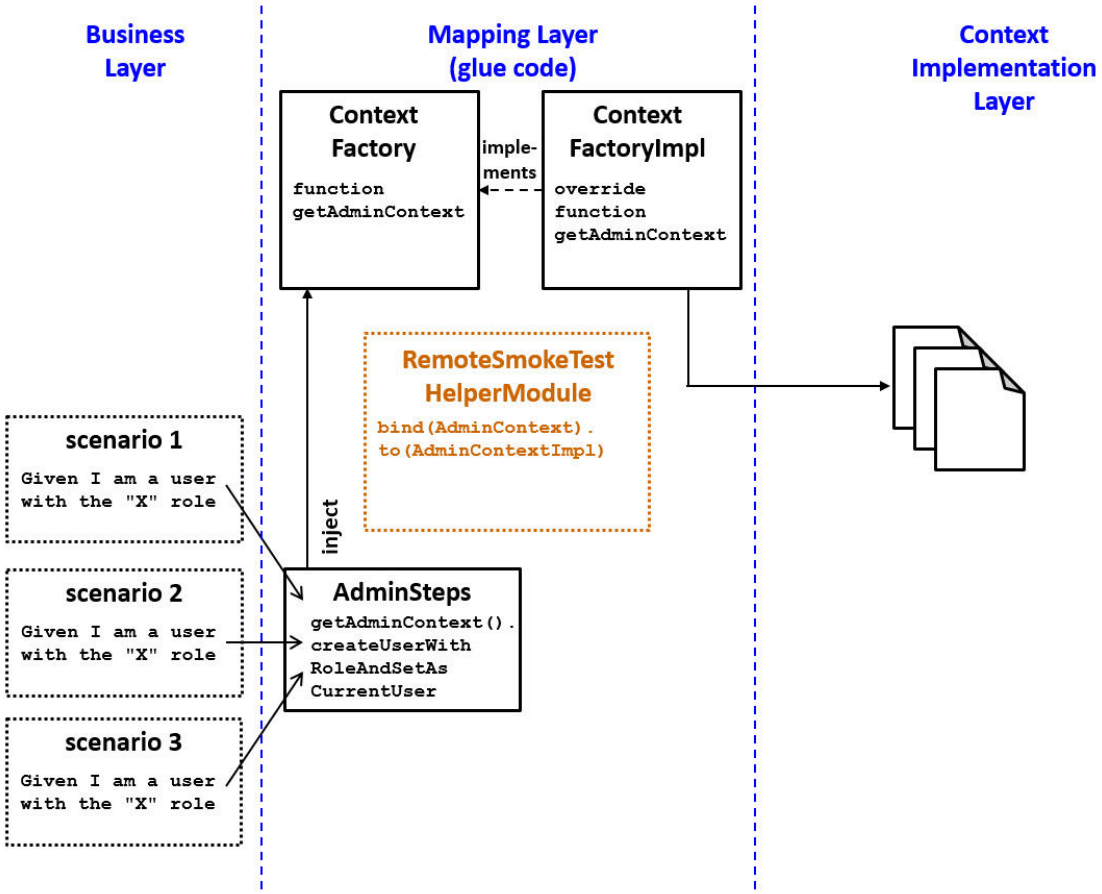
The fundamental building blocks of this layer are the step methods, Context Factory, and the RemoteSmokeTestHelperModule class.

A *step method* is a Gosu method that maps a scenario step to its implementation. Step methods are grouped together in step files.

Context Factory is the name used to refer to two Gosu classes: `ContextFactory` (an interface) and `ContextFactoryImpl` (an implementation of the interface). All of the implementations needed for testing (such as implementations for admin testing, activity testing, and so on) are injected into Context Factory. Context Factory is injected into every step file. In this way, Context Factory acts as a dependency object that provides every step definition with the ability to identify and call the appropriate implementation.

By itself, Context Factory does not have any information on which impl class to go to find the implementation for a particular method. The logic that identifies this is primarily stored in the *RemoteSmokeTestHelperModule*. This class consists of a set of custom bindings that bind every interface to the appropriate impl. For example, RemoteSmokeTestHelperModule binds the `AdminContext` interface to `AdminContextImpl`. Whenever a step method needs to access a method in a specific impl, it requests an instance of the interface and then accesses the method through the binding between the interface and the impl.

The following diagram depicts an example of resources in the mapping layer. Specifically, this example is the AdminSteps file, which provides the step method for the `Given I am a user with the "X" role` scenario step. The step file injects Context Factory, which provides access to the class that declares the implementation code.



For more information on writing step methods, see “Creating step methods” on page 49. For more information on Context Factory, see “Extending context APIs” on page 57.

The context implementation layer

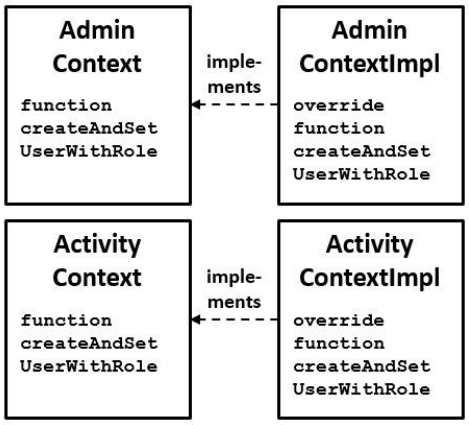
The context implementation layer is the set of resources that contain the implementation code. The fundamental building blocks of this layer are the context APIs.

Context APIs

A *context API* is a set of resources that provide the implementations to test a given business context. For example:

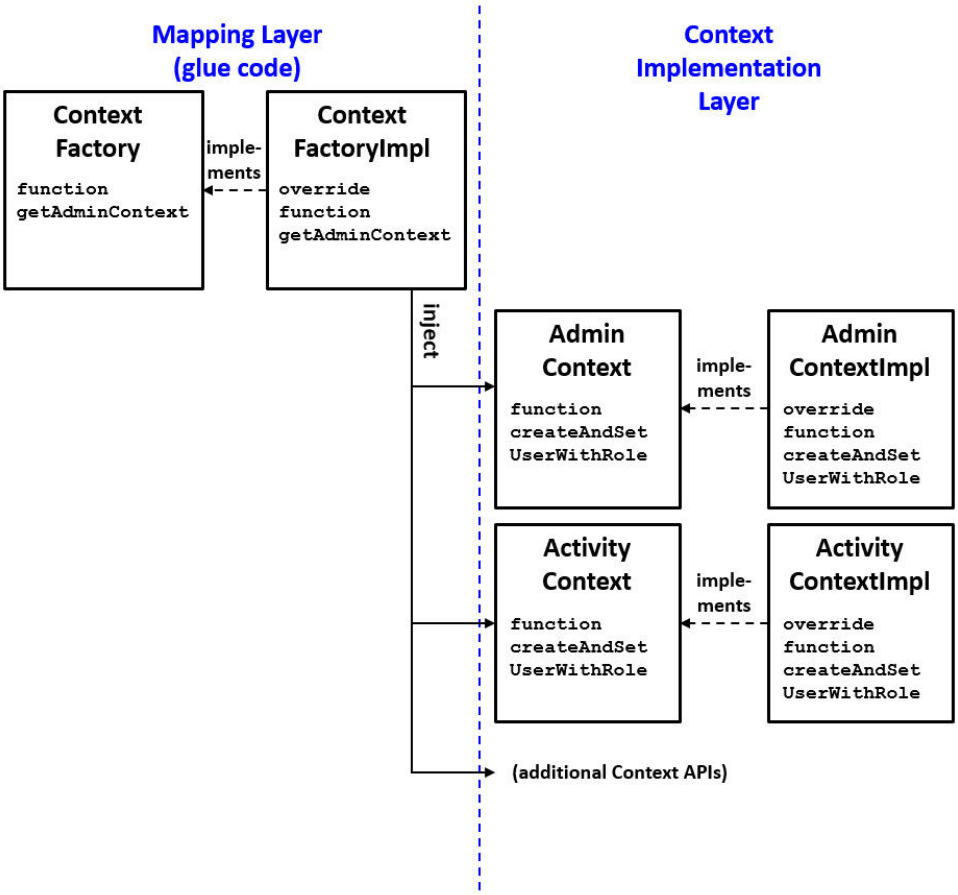
- `AdminContext` contains the resources for scenarios that test administration behavior.
- `ActivityContext` contains the resources for scenarios that test activity behavior.

A context API consists of one or more pairs of classes. Each pair consists of an interface (which defines testing method signatures) and an impl (which implements those methods). The impl always implements the corresponding interface. The following diagram illustrates two context APIs.

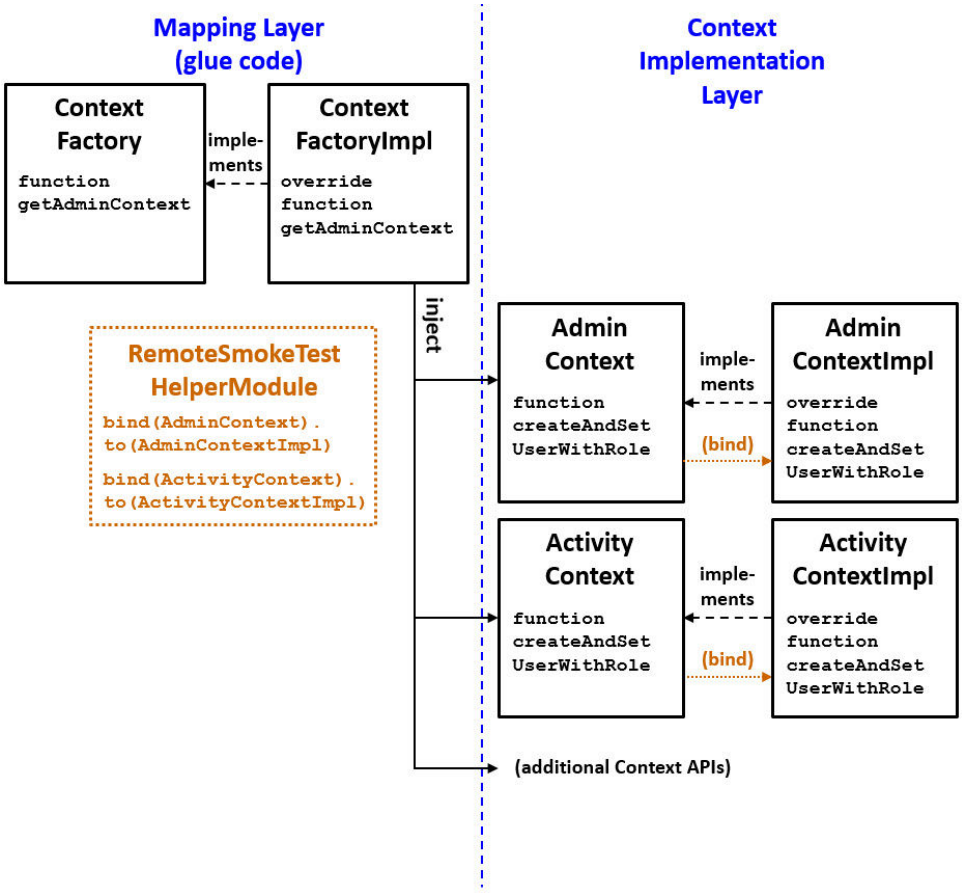


Injecting context APIs into Context Factory

All of the context APIs are injected into Context Factory. This gives the mapping layer access to all of the implementation code so that it can select the correct implementation method for each step method. This is illustrated in the following diagram.



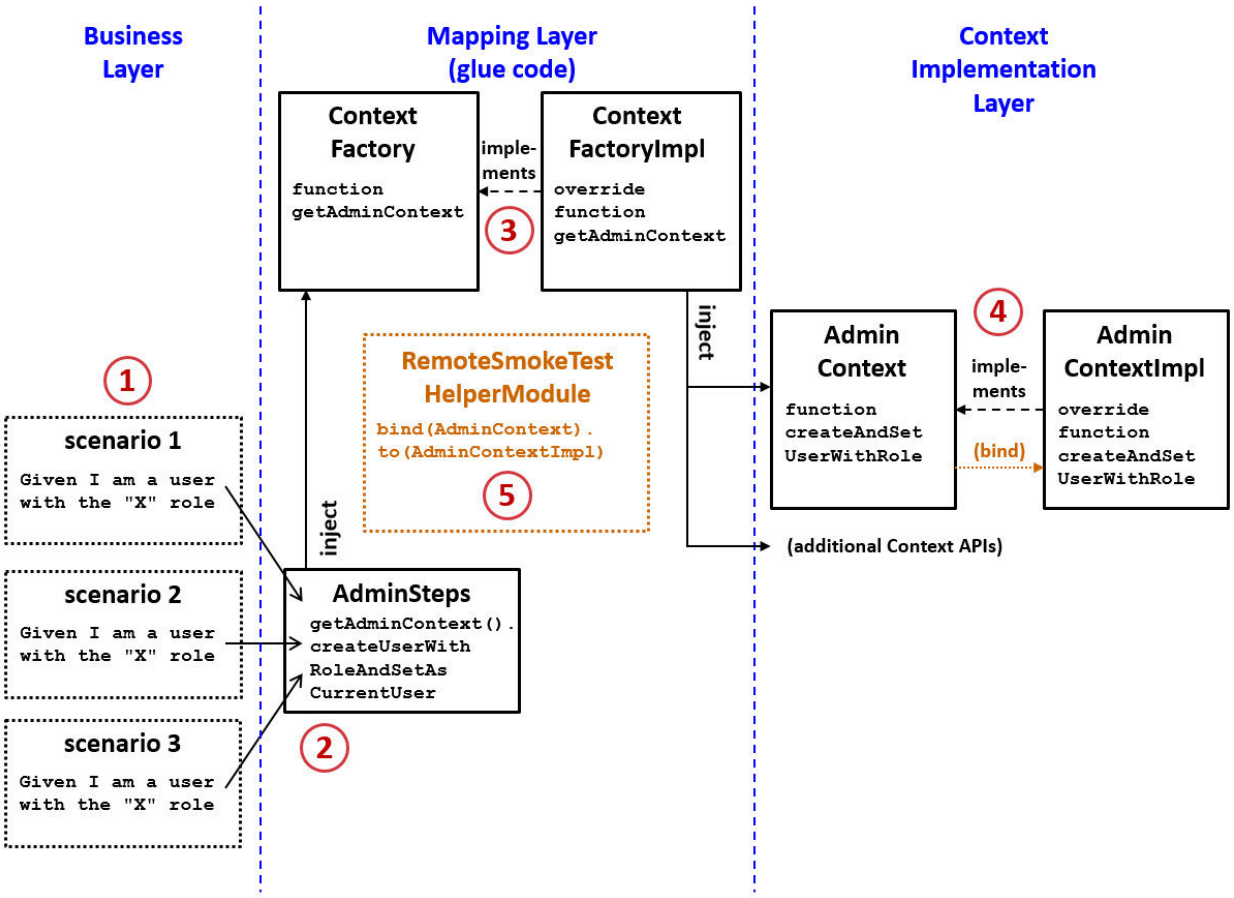
Finally, RemoteSmokeTestHelperModule binds the context API interfaces to their impls so that the correct impl can be provided to each step method. This is illustrated in the following diagram.



Summary of the resources in each layer

The following diagram depicts an example of the resources used in all three layers to enable the `Given I am a user with the "X" role` step method. Specifically:

1. Several scenarios reference the step method named `Given I am a user with the "X" role`.
2. The definition for this step method is defined in the `AdminSteps` class. This class injects `ContextFactory`, which means it has access to all the implementations known to `ContextFactory`. The step definition maps the step method to the `createUserWithRoleAndSetAsCurrentUser` method from the `AdminContext` interface.
3. `ContextFactor` has every context API injected into it. This allows it to give every step class access to every implementation.
4. Implementation details are stored in context APIs. Every context API is comprised of interface/impl pairs. The implementation code is stored in the impls.
5. RemoteSmokeTestHelperModule binds every context API interface to its impl. This provides the logic that allows Context Factory to select the correct impl for every context and method referenced by a step method definition.



For more information on extending base configuration context APIs, see “Extending context APIs” on page 57.
For more information on creating new context APIs, see “Creating context APIs” on page 67.

LOB-specific context APIs

All of the previous examples of context APIs have been LOB-agnostic. An LOB-agnostic context API is a context API for a business context that functions the same way, regardless of the line of business of the associated policy. An LOB-agnostic context API has a single interface/impl pair.

Some business contexts have testing requirements that vary based on the line of business and based on job. Within these business contexts, there is sometimes a need to share implementation code, and sometimes a need to isolate it. For example:

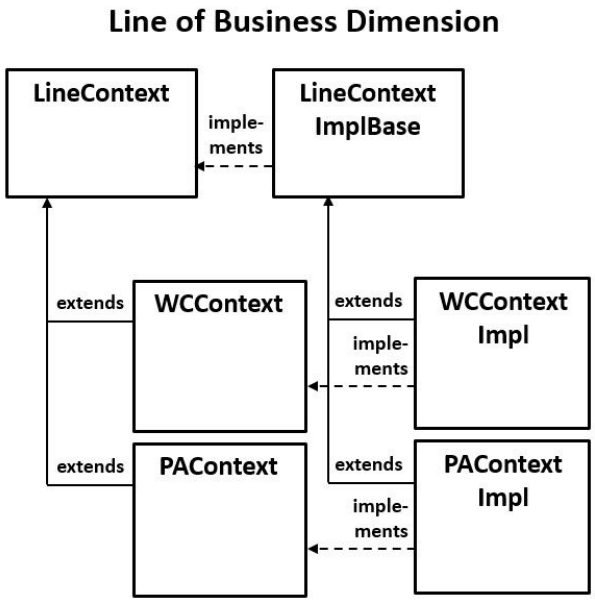
- A method to add line level coverages is needed for all policy-related tests, but:
  - A method to add drivers is needed only for personal auto policy tests.
  - A method to add covered workers is needed only for workers' compensation policy tests.
- A method to quote a job is needed for all job-related tests, but:
  - A method to start a renewal is needed only for renewal tests.
  - A method to apply preemption changes is needed only for policy change tests.

To ensure that there is a way to share code when needed and isolate code when needed, these business contexts conduct their tests using *LOB-specific context APIs*. In PolicyCenter, there is one LOB-specific context API, the LOB/Job context API. It has a multi-dimensional, multi-level structure. This structure provides a way to share code as needed and isolate code as needed.

There are two dimensions to the LOB/Job API: one for Line of Business and one for Job.

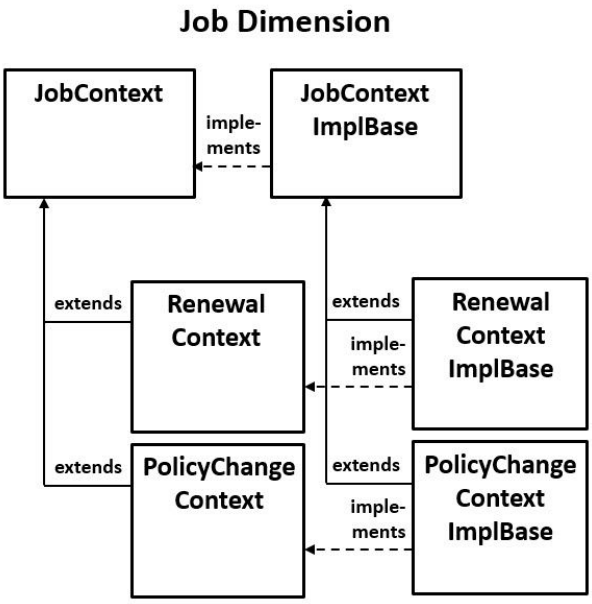
The line of business dimension has a top-level, generic `LineContext` level and multiple line-specific child levels. Methods that are required for every line (such as `addLineLevelCoverage`) are declared at the top level. Methods that are specific to a given line are declared at the child levels. For example:

- `addCoveredEmployees` is declared at the workers' compensation level.
- `addDriver` is declared at the personal auto level.

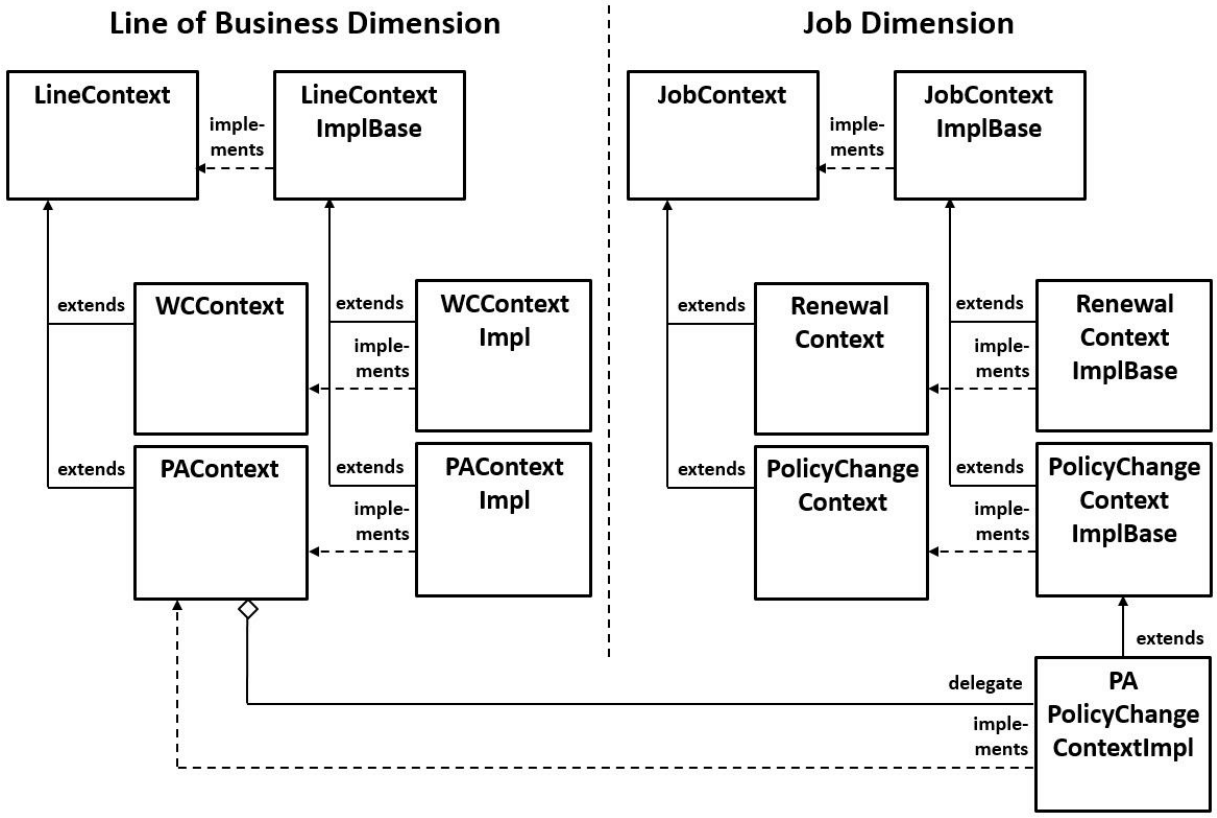


Similarly, the job dimension has a top-level, generic `JobContext` level and multiple job-specific child levels. Methods that are required for every job (such as `quote`) are declared at the top level. Methods that are specific to a given job are declared at the child levels. For example:

- `startRenewal` is declared at the Renewal level.
- `applyPreemptionChanges` is declared at the PolicyChange level.



Finally, there is an impl for every line/job combination that requires testing. This impl combines methods from the appropriate line of business level and the appropriate job level. For example, the following diagram depicts an example of `PAPolicyChangeContextImpl`. This impl combines methods from the line-specific `PAContext` and the job-specific PolicyChangeImpl.



Like Java, Gosu does not support multiple inheritance. To create a *Line/Job*`ContextImpl` class that inherits information from multiple, non-related parent classes:

- The job dimension is considered the "primary" dimension. *Line/Job*`ContextImpl` class extends the appropriate *Line/Job*`ContextImplBase` class and inherits its methods.
- The methods from the appropriate <Line>ContextImpl are made available to the *Line/Job*`ContextImpl` class through the use of a delegate named `_lineContextDelegate`.

For more information on the LOB-specific context API structure, see “The Line/Job Context API” on page 77.

LOB-specific testing also makes use of tags and a `TagInfoProcessor` class.

- Every feature file that has LOB-specific or job-specific scenarios starts with an LOB tag followed by a job tag, such as `@personal_auto` `@policy_change`.
- `TagInfoProcessor` contains the code that selects the policy type and job type during scenario testing based on the tags in the feature file.

chapter 4

Writing scenarios

Behavior Testing Framework consist of a set of resources that can be grouped into three layers. The first layer is the business layer. It consists of a set of resources that define the behaviors to be tested.

The fundamental building blocks of this layer are the *scenarios*. This topic describes the structure of a scenario, the way that they are grouped into feature files, and recommended best practices for how to design and write them.

Scenarios overview

Within the context of Behavior Testing Framework, a *scenario* is a description of a user or system behavior that can be tested. Scenarios typically have three sections:

- A Given section, which describes conditions that exist before the behavior is executed.
- A When section, which describes the behavior to be executed.
- A Then section, which describes the outcome expected from the behavior.

Scenarios are written in the Gherkin language. Gherkin is designed to enhance collaboration between business experts and technical experts by allowing scenarios to be written in natural language. Scenarios can be written collaboratively by a group of business analysts, technical consultants, and QA developers. After they have been written, the technical consultants can create Gosu methods and implementation code to execute the scenarios.

The following is a scenario written in Gherkin. The structure of Gherkin scenarios is discussed in detail later in this topic.

```
Scenario: Set a user's email address
  Given I am the supervisor of "Chris Jones"
  When I set the user's email address to "cjones@customer.com"
  Then the user's primary email address should be "cjones@customer.com"
```

Guidewire recommends that each scenario focus on the behavior of a single persona. You may also want to test behaviors that are executed by the system, such as the execution and outcome of a batch process. For these tests, you may want to use a persona of "system".

Scenarios are grouped together in feature files. Behavior Testing Framework comes with a set of base configuration feature files. You can view the scenarios in them from Studio by navigating to `configuration/res/cucumber`. If you do not have access to Studio, you can open them using a text editor. The files are located in subdirectories of `modules/configuration/res/cucumber`.

Scenario structure

A typical scenario has the following components:

- Name
- Given
- When
- Then

Scenario names

Every scenario has a name. For example:

```
Scenario: Issue a bound Personal Auto policy
```

Certain characters cannot be used in a scenario name because they prevent Studio from being able to correct parse the feature file. This includes the following ten characters:

```
= & ( ) [ ] < > / \
```

If a scenario name includes one of the characters, Behavior Testing Framework throws the following exception when trying to run the scenario: IllegalStateException: Remote Cucumber invocation: Could not find single matching Scenario/Step combination.

If necessary, use alphanumeric alternates for these characters, such as the word "equals" for "=".

Scenario steps overview

Scenarios are written as a series of steps. A *step* is a description of either a pre-condition that is true before the behavior is executed, the behavior to execute, or the outcome that is expected from the behavior.

Steps are intended to be reusable. You can use the exact same step in multiple scenarios when those scenarios require the same pre-condition, execute the same action, or expect the same outcome. For example, consider the following step:

```
Given an activity
And the activity's priority is "Urgent"
```

These steps can be used in any scenario in which one of the pre-conditions is that there is an urgent activity.

Steps with parameters

Some steps use parameters, which appear as text between quotation marks. These steps have increased usability because they combine multiple descriptions that differ only by a single value. For example:

```
Given an Personal Auto submission
And the job status is "Quoted"
```

In this example, "the job status is *status*" is a step that uses a parameter of *status*.

There are also steps that do not use parameters. These steps provide a single description that does not vary. For example:

```
Given a Personal Auto submission
And the submission is bound
```

Step methods

Every step maps to a step method. A *step method* is a method written in Gosu that identifies the implementation code that corresponds to the step. Step methods are part of the second layer of Behavior Testing Framework, the mapping layer. (Step methods are also referred to as the "glue code" because they glue the behavior descriptions to the implementation code.)

Given, When, and Then steps

Steps in a scenario are grouped into one sections: Given, When, and Then. Each section starts with a keyword (`Given`, `When`, or `Then`) and a step. If multiple steps are needed for the section, the subsequent steps start with the keyword `And`.

Given steps

Given steps sets up the state of the system for this scenario before any behavior is executed. For example:

```
Given a Personal Auto submission
And the job status is "Quoted"
```

When steps

When steps describe the behavior to execute. For example:

```
When I apply the pending changes to the unbound renewal
And I recall "Renewal term" as the current job
```

Then steps

Then steps describe the outcome expected as a result of the behavior. For example:

```
Then the submission status should be "Bound"
And the status of the contingency is "Pending"
```

Data tables

A *data table* is a set of values for a step that are specified in a rows-and-columns format. Data tables make it easier to specify multiple values for a single step where it would not be practical to use parameters.

A data table can optionally have a set of headers in the first row or in the first column.

- If the first row is a header row, then each row of the table is considered a single record of data.
- If the first column is a header column, then each column of the table is considered a single record of data.

For example, consider the following:

```
When I create a note with the following:
  |  Topic           |  Pre-renewal direction                    |
  |  Subject         |  My Subject                               |
  |  Security Level  |  Unrestricted                             |
  |  Text            |  Refer to customer service representative  |
```

In this example, the When step specifies that a note will be created. The note has four fields, and a value is specified for each field.

Feature file overview

A *feature file* is a Cucumber file that contains a set of related scenarios for a single set of behaviors executed by a single type of persona.

Note: Within the context of Cucumber, the term used to describe a group of scenarios is a "feature". This documentation refers to them as "feature files". This is to prevent confusion with the more common use of the term "feature" to refer to application behavior, such as the "assignment feature".

Feature file structure

Feature files are comprised of the following elements. Note that most elements appear in most files, but some elements may not be needed for every feature or scenario:

- One or more optional categorization tags
- Feature name
- Feature description
- Background
- One or more scenarios

The examples in the following sections come from different feature files in the base configuration.

File-level information

Guidewire recommends limiting each feature so that it has scenarios for only a single persona.

Tags

A *tag* is a text string that identifies the purpose or category of a feature file. Tags have two uses:

- A *test suite* is a collection of feature files that are executed at the same time. You can use tags to help define the scope of a test suite.
- If a feature file has scenarios that are specific to a given line of business, you must use tags to identify the line of business.

A tag must start with an @, but it can be followed by any text. For example:

```
@personal_auto
```

A feature file can have any number of tags. Each tag must be separate by a space. For example:

```
@personal_auto @issuance
```

Feature name and description

The feature file starts with the feature name and a brief description. Guidewire recommends this information be formatted as:

```
As a <persona/system>
I want to <action>
```

For example:

```
Feature: Activity

  As an agent,
  I want to create a new activity or manage an activity assigned to me on the policy.
```

Background

The *background* is a set of steps that describe the pre-conditions for all scenarios in the feature file. Background steps are executed before each scenario in addition to each scenario's Given steps. The purpose of the background is to prevent repetition of the same requirements in every scenario, such as the relevant persona.

Every feature file can have only one background. The background is declared before the first scenario.

For example:

```
Background:
  Given I am a user with the "Producer" role
```

```
  And today's date is "07/01/2018"
  And an in-force Personal Auto policy
```

Attributes available throughout a feature file

The following elements can be used multiple times in a feature file.

Comments

Feature files can have comments. In Gherkin, a *comment* is any line that starts with a #. Comments are ignored when executing a feature file. Comments can be used to provide additional clarification about why a scenario was written in a given way.

For example:

```
# For these tests to succeed, multicurrency must be enabled.
```

Scenario outlines

A scenario tests a single behavior and has a single outcome. However, there can be circumstances where a set of scenarios have nearly identical behaviors and nearly identical outcomes. Each scenario may vary by only a single value.

A *scenario outline* is a Gherkin feature that captures multiple, nearly identical scenarios into a single set of steps. A scenario outline is always followed by an Examples table. This table defines the different values to use for each execution of the outline.

For example, consider the following scenario outlines:

```
Scenario Outline: Add coverages without terms to a submission
  Given a Personal Auto submission
  And the job status is "Draft"
  When I add the coverage "<coverage>" line-level coverage
  Then the "<coverage>" line-level coverage should exist

  Examples:
    | coverage                              |
    | Medical Payments                      |
    | Underinsured Motorist - Property Damage |
    | Mexico Coverage - Limited             |
```

This scenario outline defines three separate scenarios. All three scenarios involve adding a line-level coverage to a submission. They differ only in which coverage is being added.

- The first iteration of this scenario outline will test adding the Medical Payments coverage.
- The second iteration of this scenario outline will test adding the Underinsured Motorist - Property Damage coverage.
- The third iteration of this scenario outline will test adding the Mexico Coverage - Limited coverage.

Note the following syntax conventions and restrictions for scenario outlines:

- A scenario outline starts with the text "Scenario Outline:", not "Scenario:".
- When a value in a step comes from the Examples table, the value must be written as "<*header*>", where:
  - *header* is the label at the top of the appropriate Examples table column, and
  - *header* is delimited by a < and a >.
- The values must use only alphanumeric characters. They cannot include special characters such as the slash (/) character.

Best practices for scenario design

When designing scenarios and features, Guidewire recommends that you adopt the following best practices.

Conduct scenario grooming meetings

A *scenario grooming meeting* is a meeting in which business people and technical people collaboratively define the required scenarios. The goal of each meeting is to review a set of scenarios to ensure that there is a common understanding of what each scenario is testing and whether there is enough information to develop it.

If you are implementing Behavior Testing Framework for an implementation that is at or near inception, you can conduct scenario grooming as part of your user story grooming meetings. If the implementation is already significantly complete, then it may make more sense to have meetings devoted to scenario grooming.

Guidewire recommends that insurers design scenarios in a series of scenario groom meetings.

- Each meeting includes at least one business analyst, one developer, and one QA analyst. This mix of people helps ensure that the multiple perspectives of the problem are discussed, including the business view, technical limitations, and the aspects that require testing the most.
- At the beginning of the implementation, business analysts write scenario descriptions using a standard template. The following section contains an example template. As the implementation progresses, business analysts can switch to writing scenario descriptions directly in Gherkin.
- By the end of the meeting, all participants have a clear understanding of the goal of each scenario.

For additional information on problems that can occur with too little cross-team collaboration, refer to the anti-patterns section of the Cucumber documentation.

Include both narrow and end-to-end tests

Across an entire implementation, there is typically a need to have tests that are narrow in focus as well as tests that cover broad end-to-end functionality. The general guideline is:

- If you are testing a specific unit of functionality, such as form inference during a submission, write a scenario with a narrow scope that tests the specific functionality only.
- If you are testing a user's journey through a series of screens or steps in a wizard, write a scenario with a broad scope that tests the flow end-to-end.

The best practice is to write a combination of tests with narrow scope and broad scope. The two extremes compliment each other. Narrow tests ensure the several small pieces of functionality work as intended, and they can be run quickly. On the other hand, end-to-end tests take longer to run, but they are useful to make sure that all the small pieces of functionality work as intended when they are combined together in a single user flow.

Multiple Then sections

All scenarios are limited to a single Given section.

Scenarios with a narrow scope typically have a single When section followed by a single Then section.

An end-to-end test may have multiple When sections, each followed by its own Then section. This structure is used to help define the state the system should be at important points during the end-to-end flow.

Best practices for scenario language

When writing scenarios and features, Guidewire recommends that you adopt the following best practices.

Title

- Use a short label that provides a high-level description.
- Avoid using CamelCase words without spaces in the title. For example, "Customer Service Rep" is preferred to "CustomerServiceRep".

Description

- Put the description at the top of the file, immediately after the feature name.
- Format the description using the following template:
  - As a (role or persona)
  - I want to (feature)

This approach helps to provide clarity on the user's identity, the reason for having the feature, and the feature's business value.

Limit each feature file to one persona. If the same behavior must be tested for multiple persons, create separate feature files for each persona. In this case, name each file *feature_persona*.feature. For example, `RegionalFormat_CSR.feature`.

Background

- Provide context and incidental details for the scenario.
- Write it so that it is common to most or all of the scenarios in the feature.
- Make it as short as possible.
- Limit each feature to only one background section.

Backgrounds ought to be used with caution. Keep in mind that that anyone reading the scenario must remember that there is background information at the beginning of the feature file that complements each scenario. Avoid moving information from scenario steps into the background if this makes the background less readable. It is preferable to have lines repeated in multiple scenarios than it is to have an unreadable background.

Scenario

- Every scenario either:
  - Tests one and only one behavior, or
  - Is an end-to-end test that covers an entire workflow
- Write and capitalize terms as they appear in the user interface. For example, refer to an activity's priority as "Low", "Normal", "High", or "Urgent".
- If a given term does not appear consistently in the user interface:
  - Use the version that best matches the industry standard, or
  - Use the version that best matches the section of the user interface that is being tested, or
  - Use the version that appears most often in the user interface
- Avoid making references to specific user interface controls. Focus on behaviors (such as "When I complete the activity") as opposed to user interface elements (such as "When I click the Complete button").

Scenario title

- Use short and objective descriptions that identify the general behavior.
- Avoid repeating the primary persona, as this already appears in the feature description.
  - However, if the action performed in a scenario requires a persona in addition to the one mentioned in the feature description, then it may be necessary to repeat the primary persona. For example, a scenario that tests a customer service representative behavior after supervisor approval has been granted may require explicit mention of the customer service representative.
- Avoid using Gherkin keywords, such as "given", "when", and "then".
- Avoid using words such as "verify", "assert", or "should". The scenario is testing a behavior, and therefore these types of words are redundant.

Given steps

- Use affirmative statements to represent the initial state or precondition. For example, do this:

```
Given an activity
And the activity's priority is "Urgent"
```

- Avoid the redundant phrases "there is", "there are", or "already". For example, avoid this:

```
Given there is already an activity
```

- Avoid using actions or talking about user interactions with the system. For example, avoid this:

```
Given an activity
And I reassign it to the "supervisor"
```

Do this instead:

```
Given an activity
And the activity has been reassigned to the "supervisor"
```

When steps

- Use an action verb
- Use the present tense
- Use active voice to clarify the person is the actor
- Use the first person pronoun ("I") as needed to add clarity. For example, do this:

```
Given an activity
And the activity's priority is not "Urgent"
When I set the priority to "Urgent"
```

Then steps

- Use "should" to make the Then step more assertive. For example:

```
Then the activity's status is "Complete"
```

Instead, do this:

```
Then the activity's status should be "Complete"
```

Evaluate the usability of the base configuration scenarios

Some of the base configuration scenarios may be applicable to your implementation. You can use these scenarios to test product behavior without modification.

Other base configuration scenarios may not be applicable to your implementation exactly as coded. There are two possible reasons for this:

- The scenario will pass in your implementation, but its conditions are not rigorous enough to constitute a meaningful test.
- The scenario will fail in your implementation because the behavior to be tested has been altered in your implementation or the scenario's data is not applicable.

Guidewire recommends the following for base configuration tests that are not applicable to your implementation:

- If the scenario will pass, but its conditions are not rigorous enough to constitute a meaningful test:
  - Keep the scenario test as is in the original file, as this minimizes work during upgrade.
  - Create a new scenario with sufficient rigor in a new file.
- If the scenario will fail, modify the original file and remove the scenario. This creates a more reliable environment for QA because all scenarios are ultimately expected to pass. Modifying the original file will cause a conflict during any future upgrade. Therefore, Guidewire recommends you also keep records of modifications made to base configuration scenarios so that future upgrades can rapidly identify the source of any conflict.

Working with feature files

Feature file location

Within the Guidewire application, feature files are stored in the subdirectories of `modules/configuration/res/cucumber`. They can be accessed from Studio by navigating to `configuration/res/cucumber`.

If a new file contains modification related to an existing file, Guidewire recommends that you either:

- Store each new file in the same folder as the original file, or
- Store all new files in a new folder, such as `res/cucumber/example_insurance`.

Feature file names

To ensure that your customizations do not create any upgrade conflicts, Guidewire recommends that you declare new scenarios in new feature files.

Prefix

If a file pertains to a specific line of business, Guidewire recommends starting the file name with the LOB code. For example: **`PA`**`_Activities_CSR_Ext.feature`

Suffix

Guidewire recommends including the feature's persona in the file name. For example: `PA_Activities_`**`CSR`**`_Ext.feature`

Guidewire recommends ending the file name with an _Ext suffix. For example: `PA_Activities_CSR_`**`Ext`**`.feature`

File extension

All feature files must have the .feature file extension. For example: `PA_Activities_CSR_Ext.`**`feature`**

Create a feature file

About this task

Drafts of feature files can be created using any type of text editor or word processor. However, to create a functional feature file, the user must have access to Studio.

Procedure

1. In Studio, navigate to the `configuration/res/cucumber directory`.
2. If necessary, create a new sub-directory for the feature file.
3. Right-click the appropriate sub-directory and select **New→File**.
4. In the **New File** dialog box, name the file and click **OK**. Guidewire opens the blank feature file.
5. On the first line of the file, optionally add a tag.
   - The syntax is: *@Tag*.
   - Tags can be used to narrow the scope of test suites. For more information, see “Running groups of feature files” on page 43.
6. Specify the feature name.
   - The syntax is: "Feature: *Feature Name*"
7. Provide a description of the feature.
8. Provide the feature background.

Adding scenarios to a feature file

Once a feature file exists, you can add scenarios to it by entering them after the Background section.

As you start to type the step text, Studio shows a popup that lists all of the defined step methods. As you enter text, Studio filters the choices in the popup to those that match the entered text. You can use this feature to select defined step methods. This can help promote reusability by making it easier to identify when there is an existing step with the desired functionality.

When you enter a step with no defined step method, Studio highlights the step in yellow. When you mouse over the step, a tooltip appears that starts with "Undefined step reference:".

Step methods are created in step files and require knowledge of Gosu. Business analysts can create feature files without having all of the associated steps defined. However, before the feature can be executed, the associated steps must be defined. For more information, see “Creating step methods” on page 49.

chapter 5

Running scenarios

The following topic provides information on how Behavior Testing Framework runs scenarios, and how you can initiate the running of scenarios.

Overview of running scenarios

Every Cucumber scenario describes a behavior that the application is ultimately expected to have. The behavior can be tested by *running the scenario*. This produces a *test result*, which indicates if, under the given pre-conditions, the behavior resulted in the expected outcome. The test result is either a pass or a fail.

There are several different options for running scenarios.

Which scenarios to run

You can choose to run an individual scenario, a feature file, or a test suite.

A *feature file* is a Cucumber file that contains a set of related scenarios. Features typically contain scenarios for a single type of user that test a particular aspect of application behavior.

A *test suite* is a Gosu class that defines a group of feature files to run at the same time.

How to initiate the testing

Individual scenarios and feature files can be run only through Guidewire Studio. For more information, see “Running feature files from Studio” on page 43.

Test suites can be run in one of two ways:

- Through Guidewire Studio
- From a command prompt

For more information, see “Running a test suite from Studio” on page 45 and “Running a test suite from a command prompt” on page 45.

Test servers

Scenarios must be executed on a running server. A server that is used to run scenarios is often referred to as a *test server*. Note that there is no flag, setting, or configuration parameter that inherently marks a server as a test server. A test server is simply a server on which tests are run.

Test server configuration

Test servers are often configured to run with settings that are appropriate for running tests, as opposed to settings appropriate for a production environment. The following describes some common test server settings.

Test server modes

A server can be started in one of three modes: development, test, and production. In development mode, all implementation and testing features are enabled. In test mode, all implementation and testing features are disabled, except for the ability to enable and advance the testing clock. Test mode is designed for testing functionality in a production-like environment with the one exception of being able to advance the clock to test time-based processes. In production mode, all implementation and testing features are disabled.

Test servers are typically started in either development mode or test mode.

For more information on server modes, refer to the section on server modes in the *System Administration Guide*.

Test server database

Test servers are often configured to store the database in memory only. When the server is started, a new database is created. When the server is stopped, that database is discarded. An "in memory" database has the advantage of providing a more stable testing environment. The database is always discarded and recreated, which means there is less chance that a valid test will fail because of the presence of unexpected data from a previous test or from some other process.

The `database-config.xml` file specifies different database configurations using the `<database>` tag. In the base configuration, test servers are run using the `<database>` configuration with the `env="h2mem"` property. This configuration specifies an H2 server whose database is in memory. Specifically, the `jdbc-url` property starts with `jdbc:h2:mem:...`.

Assertions enabled

An *assertion* is a method that tests to see whether a desired behavior has been correctly executed within an application. Gosu includes assertions, which are an extension of assertions in Java.

When an assertion is enabled, it checks to see whether a given outcome has occurred. If the outcome has occurred, the assertion does nothing. If the outcome has not occurred, the assertion throws an error, which can be captured by a testing framework and included in a report or shown to a user.

By default, assertions are disabled in InsuranceSuite applications. This means that whenever an assertion is reached during run-time, it does nothing.

Typically, a test server is configured to have assertions enabled. This means that whenever an assertion is reached during run-time, it is executed. If the assertion fails, the exception can be logged or shown to a user.

To start a server with assertions enable, the server must be started with the flag `-ea`. This is the same flag used to enable assertions in Java programs.

Test server ports

Test servers are often configured to run on a unique port. This helps to isolate the test environment and to ensure that other processes or users do not inadvertently connect to the test server.

Starting and stopping test servers

You can initiate the running of a scenario, feature file, or test suite whether there is a test server running or not.

If there is a test server running, Behavior Testing Framework runs the scenarios on that test server. When the scenarios are complete, the server is left running.

If there is no test server running, Behavior Testing Framework starts a test server, runs the scenarios on that test server, and then stops the test server.

If you plan to run multiple feature files or test suites in an ad hoc manner, you can start a test server and then run the scenarios as needed. This may be more efficient as the test server needs to execute the start-up process only once.

IMPORTANT There is a known issue with Behavior Testing Framework 10.0.1. If there is a running test server and you run a test suite, the test suite is not able to run properly and Behavior Testing Framework throws an `ArrayOutOfBounds` exception. As a workaround, stop the test server before running a test suite.

Running feature files from Studio

From Studio, you can run scenarios and feature files on demand. This is useful when you want to ensure a newly written scenario or feature file is working as expected. When appropriate, you can also run all of the feature files in a single directory.

The test results tab

When you run scenarios from Studio, Studio opens a testing tab at the bottom of the Studio interface. The test results appear on this tab.

The left pane shows test results in a hierarchical format.

- If all scenarios pass, a green check mark icon with an "All Tests Passed" label appears.
- If one or more scenarios fail, a yellow "X" icon with a "Test Results" label appears.

From these icons, you can drill down to get more explicit information on the scenarios that passed and failed.

The right pane shows console output generated during the running of the test. This output includes the number of scenarios that passed and failed. If there are any failures, additional information may appear to help identify the cause of the failure, such as exceptions and stack traces.

Run an individual scenario from Studio

Procedure

1. In Studio, navigate to `configuration/res/cucumber`. This directory contains subdirectories that contain the feature files.
2. Open the feature file containing the scenario.
3. To run a scenario, right-click the scenario name and select **Run 'Scenario: <scenario>'**.

Run feature files from Studio

Procedure

1. In Studio, navigate to `configuration/res/cucumber`. This directory contains subdirectories that contain the feature files.
2. To run an individual feature file:
   - Expand the subdirectory containing the feature file.
   - Right-click the file and select **Run 'Feature: <feature>'**.
3. To run all feature files in a given directory:
   - Right-click the directory and select **Run→All features in: <directory>**

Running groups of feature files

A *test suite* is a Gosu class that identifies a set of feature files to run at the same time. Test suites can be run either from Studio or from a command prompt. Test suites make it easier to run related feature files automatically as a part of continuous integration.

Behavior Testing Framework includes one example suite in the `configuration/gtest` folder, the `PCBehaviorCucumberSuite` test suite.

Narrowing the scope of a test suite

Every test suite must identify a single parent directory. By default, the test suite executes all feature files in that directory and its subdirectories.

You can narrow the scope of a test suite by using tags. In Behavior Testing Framework, a *tag* is a string added to a feature file that identifies a category to which the feature file belongs. Tags always start with an "@". For example, `@supervisor`, `@personal_auto`, and `@renewal` are all valid tags.

When you define the test suite, you can include one or more tags to the suite definition. For each tag, you can specify that either a feature file must have the tag to be included in the suite, or that a feature file must not have the tag to be incuded in the suite. This narrows the scope of the test suite to only the feature files in the given directory (and its subdirectories) that have all of the specified tags. For example:

- A test suite with `@supervisor` includes all feature files with the `@supervisor` tag. (Whether they have other tags or not is irrelevant.)
- A test suite with `@supervisor` and `@personal_auto` includes all feature files with both the `@supervisor` tag and the `@personal_auto` tag. (Whether they have other tags or not is irrelevant.)
- A test suite with `@supervisor` and not `@personal_auto` includes all feature files with the `@supervisor` tag and without the `@personal_auto` tag. (Whether they have other tags or not is irrelevant.)

In both feature files and test suites, tags can be listed in any order.

Tags are case-sensitive. For example, `@PA_submission` and `@pa_submission` are considered to be different tags.

A single feature file can belong to any number of test suites. For example, suppose you have a feature file with the following tags: `@supervisor @personal_auto`. You then create two test suites: one for all `@supervisor` tests, and one for all `@personal_auto` tests. The initial feature file will be included in both test suites.

Test suite reports

When you run a test suite, by default, Behavior Testing Framework generates two reports that identify test outcomes:

- An HTML report named `index.html` in `build/test-results/<testSuiteName>/CucumberHTMLReport`.
- A JUnit report named `CucumberResults.xml` in `build/test-results/<testSuiteName>`.

You can modify the name and location of the report in the test suite definition.

Create a test suite class

Procedure

1. Create a Gosu class.
   - Guidewire recommends ending the class name with `Suite_Ext`.
2. Navigate to the existing `PCBehaviorCucumberSuite` class. Copy its contents into your new class.
3. Modify the contents as needed.
   a. Optionally modify the `html:` portion of the `:plugin` value to specify a different name or location for the HTML test results report.
   b. Optionally modify the `junit:` portion of the `:plugin` value to specify different name or location for the JUnit test results report.
   c. Optionally modify the `:features` value to specify a different parent directory. By default, the test suite includes all feature files in the parent directory and its subdirectories.
   d. Optionally modify the `:tags` value to specify one or more tags. This restricts the suite to including only feature files that have all the listed tags. If you include multiple tags, they must be in a comma-delimited list, such as `"@tag1","@tag2"`. The tags can be listed in any order.

e. Optionally modify the `:tags` value to exclude one or more tags. This restricts feature files that have the listed tags from the test suite. To indicate a tag must be excluded, prefix it with a tilda. For example, `"~@tag3"`.

Example test suite

The following code is an example suite named `CalendarAdminSuite`. It runs all feature files in the `modules/configuration/res/cucumber` directory (in other words, all feature files in Behavior Testing Framework) that have both the `@calendar` tag and the `@admin` tag.

```
package gw.suites
uses cucumber.api.CucumberOptions
uses cucumber.api.SnippetType
uses cucumber.api.junit.Cucumber
uses org.junit.runner.RunWith
@Export
@RunWith(Cucumber)
@CucumberOptions(
    :dryRun = false,
    :plugin = {"pretty",
    "html:build/test-results/PCBehaviorCucumberSuite/CucumberHTMLReport",
    "junit:build/test-results/PCBehaviorCucumberSuite/CucumberResults.xml"},
    :monochrome = true,
    :glue = {"gw.cucumber"},
    :snippets = SnippetType.CAMELCASE,
    :strict = true,
    :features = {"modules/configuration/res/cucumber"},
    :tags = {"@calendar","@admin"}
)
class CalendarAdminSuite {}
```

Running a test suite from Studio

To run a test suite from Studio, navigate to the directory that contains the test suite class (such as `gtest/gw/suites`). Then, right-click the class and select **Run '<suite>'**.

IMPORTANT There is a known issue with Behavior Testing Framework 10.0.1. If there is a running test server and you run a test suite, the test suite is not able to run properly and Behavior Testing Framework throws an `ArrayOutOfBounds` exception. As a workaround, stop the test server before running a test suite.

The test results tab

When you run scenarios from Studio, Studio opens a testing tab at the bottom of the Studio interface. The test results appear on this tab.

The left pane shows test results in a hierarchical format.

- If all scenarios pass, a green check mark icon with an "All Tests Passed" label appears.
- If one or more scenarios fail, a yellow "X" icon with a "Test Results" label appears.

From these icons, you can drill down to get more explicit information on the scenarios that passed and failed.

The right pane shows console output generated during the running of the test. This output includes the number of scenarios that passed and failed. If there are any failures, additional information may appear to help identify the cause of the failure, such as exceptions and stack traces.

Running a test suite from a command prompt

Compiling resources required for running tests suites from a command prompt

Test suites can be run using the `gwb runCucumberSuite` command. However, before this command can be used, you must compile resources needed by the processes that `gwb runCucumberSuite` uses. This is done by first executing the `gwb compile` command from the application directory. For example:

```
gwb compile
```

Typically, you need to execute this command only once. You typically do not need to execute it prior to each test. However, these is no harm in executing this command multiple times.

If a test execution fails with a `java.lang.IllegalStateException` that starts "no modules found at...", you may need to execute or re-execute `gwb compile`.

Running tests suites from a command prompt

Once the necessary resources have been compiled, you can run a test suite from a command prompt by executing the following from the application directory:

`gwb runCucumberSuite -Dsuite=`*`pathToSuite.suiteName`*

For example:

```
gwb runCucumberSuite -Dsuite=gw.suites.CalendarAdminSuite
```

This command compiles and runs the feature files in the specified test suite.

IMPORTANT There is a known issue with Behavior Testing Framework that prevents a test suite from running if there is a running test server. When this is attempted, Behavior Testing Framework throws an `ArrayOutOfBounds` exception. As a workaround, if there is a running test server, stop it before executing the test suite.

Viewing test results

Behavior Testing Framework generates information on the test results. You can view these results by navigating to `modules/configuration/build/reports/tests/executeCucumberSuite` and clicking `index.html`.

Part 2

Mapping steps to implementation code

The mapping layer of Behavior Testing Framework maps scenarios to the code that runs the steps. In the implementation layer of the framework, you use context APIs in writing your step methods. You can extend the context APIs that the framework provides or create your own new context APIs.

chapter 6

Creating step methods

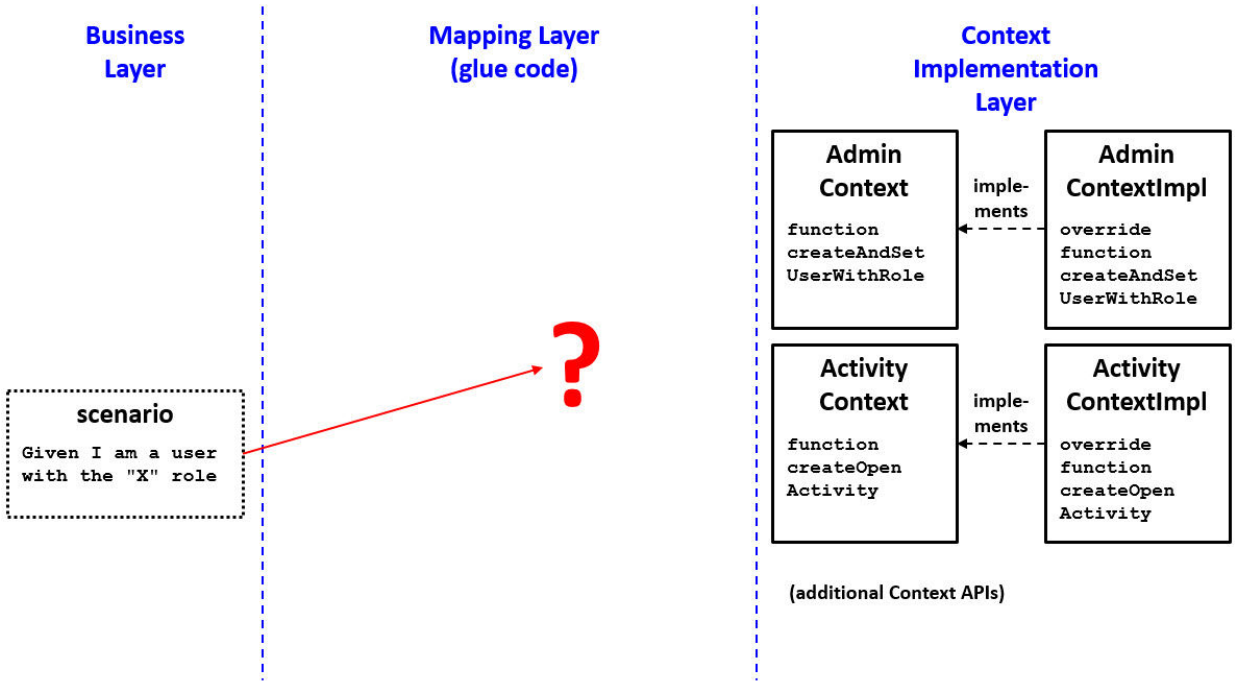
Behavior Testing Framework consist of a set of resources that can be grouped into three layers. The second layer is the mapping layer. The resources in this layer map the scenarios to the implementation code that executes them. The resources in this layer are sometimes referred to as glue code. (*Glue code* is an programming term code that exists to connect different sets of code that are otherwise incompatible.)

There are three fundamental resources for this layer: Context Factory, RemoteSmokeTestHelperModule, and step methods. This topic describes these resources and how to create step methods.

Glue code in Behavior Testing Framework

One of the goals of Behavior Testing Framework is to create a strong separation between the business description of behaviors and the implementation details for those behaviors. This provides flexibility because you can change a scenario without having to worry about the implementation details, and you can change implementations without having to worry about the impact on scenarios.

However, this separation comes with a question. How can each scenario be associated with the correct implementation code? For example, when there is a step such as "Given I am a user with the "X" role", how does Behavior Testing Framework identify the correct impl method to execute?



Behavior Testing Framework answers this question with the resource in the mapping layer. These are also referred to as the *glue code*, which means this code exists to connect different sets of code that are otherwise incompatible.

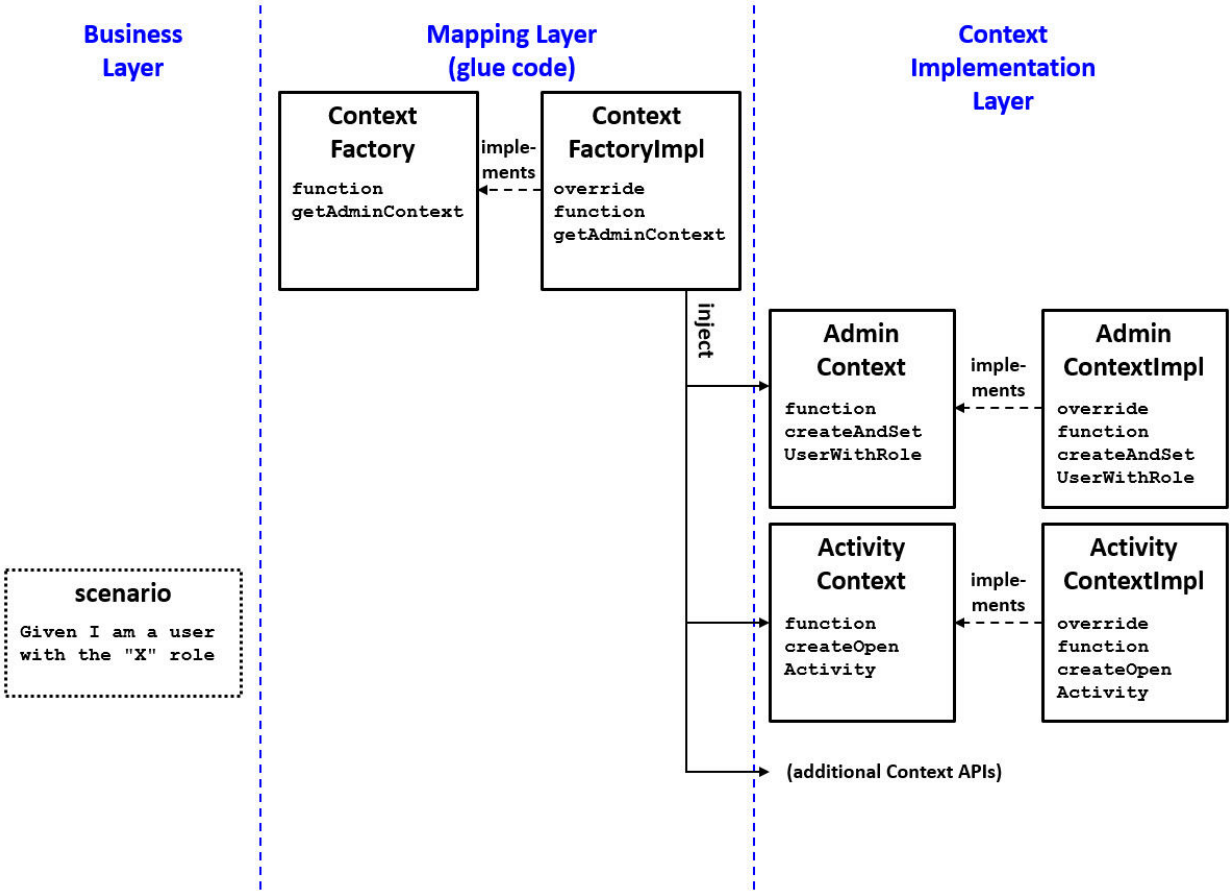
There are three types of resources that form the glue code:

- Context Factory
- RemoteSmokeTestHelperModule
- The step methods

Context Factory

All of the implementation details are stored in a set of context APIs. For example, Admin Context is a context API that contains implementation details related to the administration business context. All of these context APIs must be known to the glue code so that the context AI with the correct implementation can be selected.

Context Factory is a set of resources whose primary role is to make all of the context APIs available within the glue code. Context Factory consists of two Gosu classes: `ContextFactory` (an interface) and `ContextFactoryImpl` (an implementation of the interface). All of the context API interfaces are injected into `ContextFactoryImpl`.



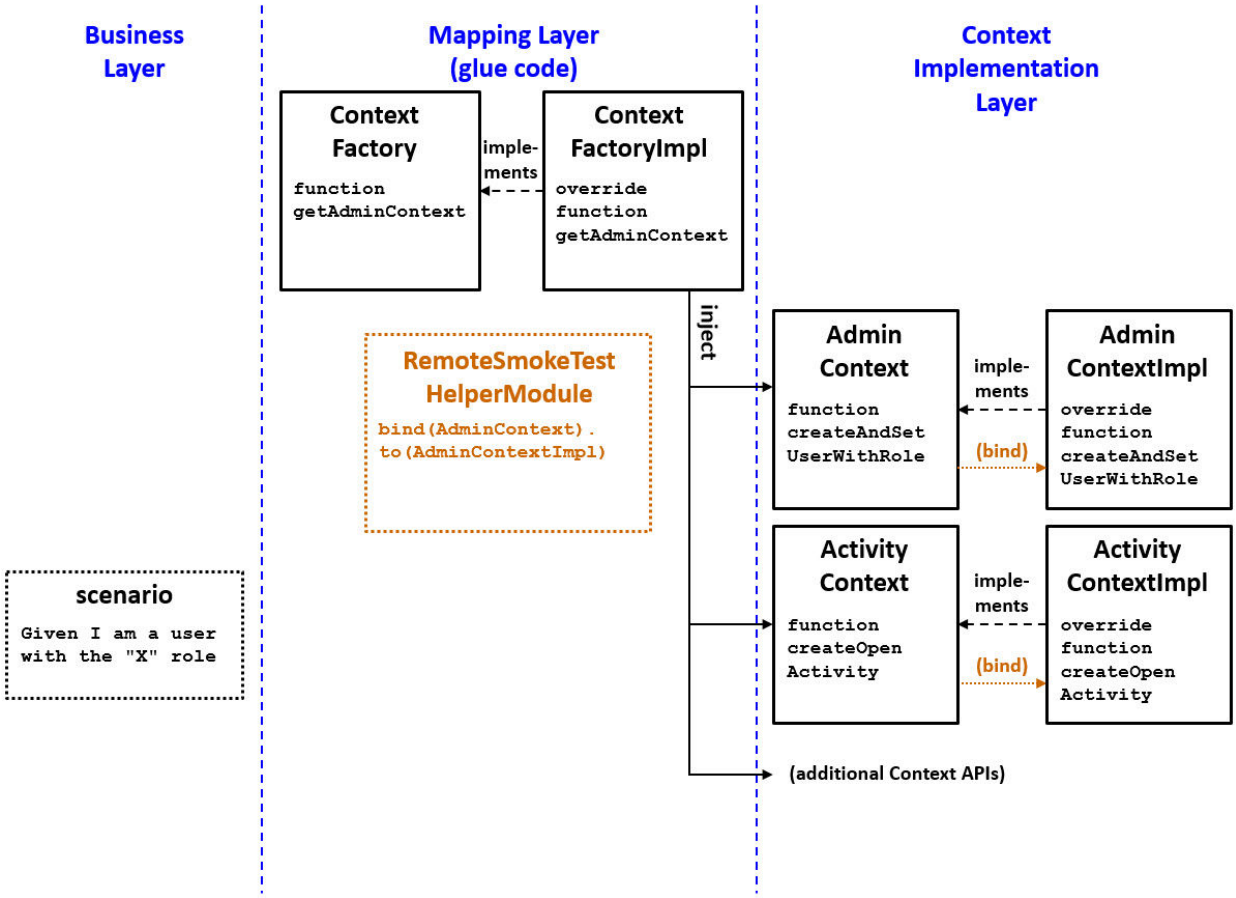
RemoteSmokeTestHelperModule

Through Context Factory, the glue code has access to all of the interfaces in the context APIs. But, Context Factory alone does not have information on which impl implements a given interface.

The glue code identifies which impl to use for a given interface through a series of binders. These binders are declared in RemoteSmokeTestHelperModule.

For example, the following line of code binds the AdminContext interface to the AdminContextImpl impl. If any code references a method from the AdminContextInterface, the glue code will look in AdminContextImpl to find the implementation of the method.

```
// Admin Context binder
bind(AdminContext).to(AdminContextImpl)
```



Furthermore, some scenarios are LOB-specific. In these cases, there could be multiple impls that declare a method with the same name, but each impl has an LOB-specific implementation of the method. For these situations, RemoteSmokeTestHelperModule uses a map binder that binds every policy type to the appropriate impl.

For example, the following lines of code create a map binder for Quick Quote submissions. Each of the policy types (`PERSONALAUTO` and `WORKERS_COMPENSATION`) is mapped to an LOB-specific impl (`PASubmissionContextQuickQuoteImpl` and `WCSubmissionContextQuickQuoteImpl`, respectively).

```
// Quick Quote Submission Contexts
var quickQuoteSubmissionContexts = MapBinder.newMapBinder(binder(), ProductCode, SubmissionContext,
Names.named("QuickQuote"))
quickQuoteSubmissionContexts.addBinding(ProductCode.PERSONAL_AUTO).to(PASubmissionContextQuickQuoteImpl)
quickQuoteSubmissionContexts.addBinding(ProductCode.WORKERS_COMPENSATION).to(WCSubmissionContextQuickQuoteImpl)
```

Step methods

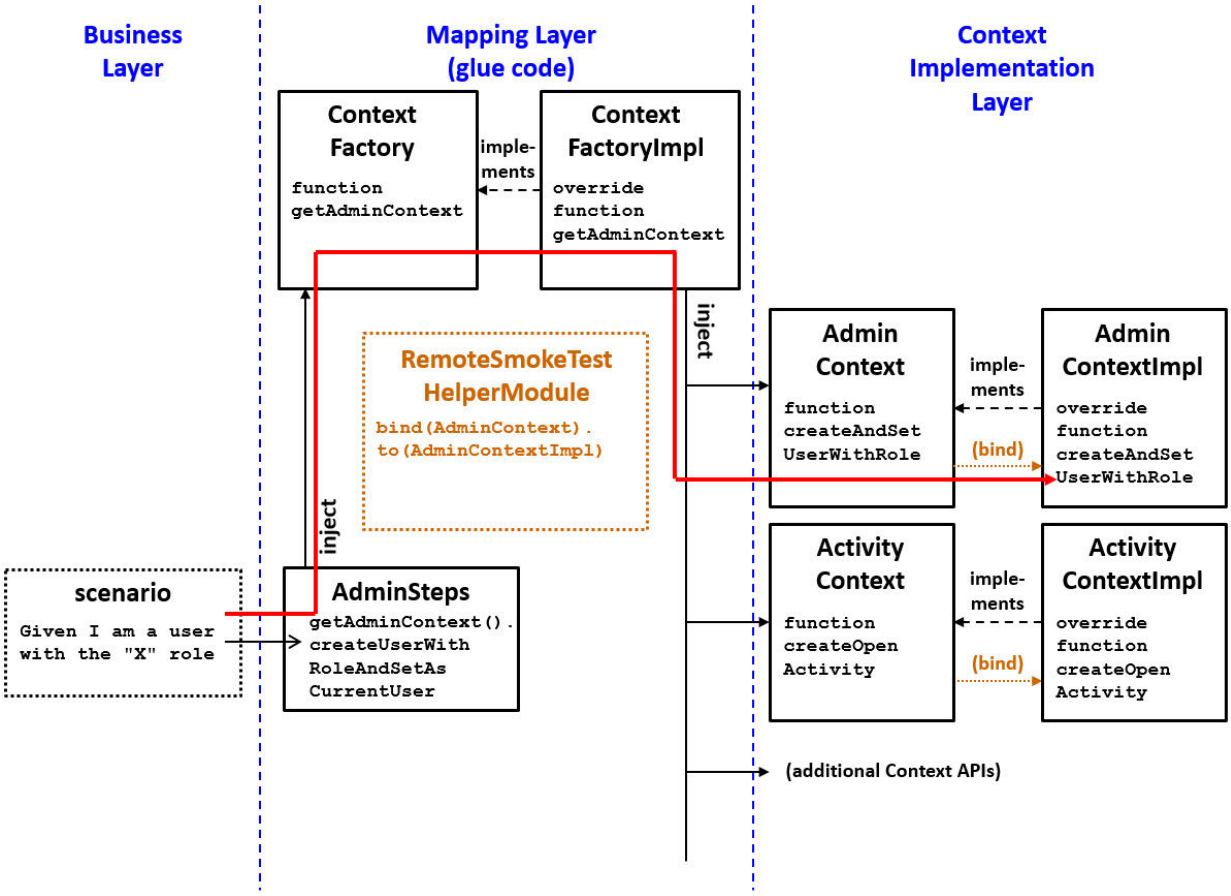
Finally, every step is associated with a step method. A *step method* is a Gosu method that maps a scenario step to a method in a specific interface.

Step methods are declared inside step classes. Every step class injects Context Factory. Therefore, every step method has access to all of the context API interfaces. And, because RemoteSmokeTestHelperModule binds every interface to one or more impls, the step method can name the method to implement.

For example, the following code is a step method from the `AdminSteps` class.

```
@Given("^I am a user with the \"([^\"]*)\" role$")
function createUserWithRoleAndSetAsCurrentUser(roleString: String) {
  _contextFactory.getAdminContext().createAndSetUserWithRole(roleString)
}
```

The step method is: `createUserWithRoleAndSetAsCurrentUser`. It maps the `"I am a user with the "X" role"` step to the `createAndSetUserWithRole` method from the `AdminContext` interface. (RemoteSmokeTestHelperModule maps this interface to the `AdminContextImpl` impl.)



The remainder of this topic discusses step methods in detail. For information on when and how to modify Context Factory and RemoteSmokeTestHelperModule, see:

- “Extending context APIs” on page 57
- “Creating context APIs” on page 67
- “The Line/Job Context API” on page 77

Step method structure

A *step method* is a Gosu method that maps a scenario step to a method in a specific interface.

Step methods have a three-part structure:

- The method connector
- The method declaration
- The method body

The scenario step connector

Step methods start with a line that connects the scenario step to the step method itself. In Cucumber, the scenario step is referred to as a *Cucumber expression*.

The scenario step connector has the following syntax:

```
@keyword("^scenarioStep$")
```

where:

- @ is a literal character.
- *keyword* is a Gherkin keyword, typically Given, Then, or When.
- ("^ and $") are literal characters that delimit the method name.
- *scenarioStep* is the Cucumber expression for the corresponding scenario step.

For example:

```
@When("^I complete the activity$")
```

A step method can be used anywhere in a scenario, regardless of the Gherkin keyword used in the scenario step connector. For example:

- You could use the Given keyword in a scenario step connector, and then use that method in a When step.
- You could use the And keyword in a scenario step connector, and then use that method in a Then step.

Wherever possible, Guidewire recommends using only the Given, Then, or When keywords in scenario step connectors. This helps to clarify if the primary purpose of the method is to create pre-conditions, execute actions, or verify that actions executed correctly.

Input parameters

When a method has an input parameter, the parameter is named in the scenario step connector and delimited by a special set of characters. The meaning of these characters is defined by the Gherkin language. The following table lists common character sets used in the base configuration step methods. For a complete description of character meanings, refer to the Cucumber documentation.

Character String	Meaning	Example
("^	Start of the method name	`@When("^I complete the activity$")`
$")	End of the method name	`@When("^I complete the activity$")`
\"([^\"]*)\"	String parameter whose value is any text (Technically, this expression means “any character - except a double quote - that occurs zero or more times inside double quotes”.)	`@When("^the payment plan of the policy is changed to \"([^\"]*)\"$")`
\"(X|Y)\":	Portion of a string parameter whose value is either X or Y	`@Given("^an account of type \"(Person|Company)\":$")`
(?:X|Y)	Portion of a string parameter whose value is either X or Y	`@When("^(?:a|1) policy is added to this account$")`
\"(\\d+)\"	Numeric parameter (any set of digits)	`@Then("^a maximum of \"(\\d+)\" invoices are generated$")`
\"(\\d+%)\"	Percent parameter (any set of digits followed by a "%")	`@Then("^one invoice with \"(\\d+%)\" down is created$")`

The method declaration

As is the case with all Gosu methods, you use the Gosu keyword `function` to declare a step method.

If the step method takes one or more parameters, then those values are typically included in the method declaration. For example:

```
@And("^the submission has an effective date of \"([^\"]+)\"$")
function submissionEffectiveOn(effectiveDate : String) {
_contextFactory.getSubmissionContext().setEffectiveDate(DateTransformer.transform(effectiveDate))
}
```

This scenario step provides a String parameter that identifies the submission's effective date. This string is then passed into the method as `effectiveDate`.

The method body

Step methods always end by calling one or more implementation methods on the appropriate context API. The `_contextFactory` object is used to access the context API and implementation method.

For example, consider the following:

```
@When("^I quote the submission$")
function quoteTheSubmission() {
  _contextFactory.getSubmissionContext().quote()
}
```

The final line of the method calls the `quote` implementation method. To do this, it uses the `_contextFactory` object to access the `SubmissionContext` interface. This interface is bound to an impl, and therefore the step method can reference any method in that impl.

Step classes

A *step class* is a class that contains definitions for step methods used in features.

Step class structure

Steps classes contain a set of injected variables followed by a set of step method definitions.

The @Export annotation

The base configuration step classes start with an `@Export` annotation. This is an internal Guidewire annotation that allows a file to be editable in Studio. You do not need to include this annotation in any custom Step classes.

Injected dependencies

The `@Inject` annotation identifies a dependency that is being injected into the steps class. All step classes require the following dependency:

```
@Inject
var _contextFactory : ContextFactory
```

This code makes the Context Factory dependency object available to each step method definition. The method definitions uses this object to invoke the appropriate context API and call the appropriate method on that API.

Some step classes include additional injected dependencies. For example, `RatingSteps` also includes:

```
@Inject
var _policyPeriod : PolicyPeriodWrapper
```

The remainder of the class is a list of step method definitions.

Best practices for creating step class files

To minimize upgrade conflicts, Guidewire recommends that you avoid making modifications to the base configuration step classes. Instead, add testing extensions to your own files.

The base configuration step class files are stored in `gtest/gw/cucumber/steps`.

Guidewire recommends storing all of the custom step class files, API files, and impl files in a single directory, such as `gtest/gw/cucumber/customer`, with the following subdirectories:

- `context/api`
- `context/impl`
- `steps`

You can use subdirectories in `gtest/gw/cucumber/customer/steps` to organize step class files. There are no technical restrictions for how to group step class files together. A feature can use any step method, regardless of the location of the step class that defines it. And, a step class can call implementation code from any impl file. Therefore, group files together in a way that makes the most sense to the business.

Guidewire recommends adding an `_Ext` suffix to all new files, including step class files.

chapter 7

Extending context APIs

For Behavior Testing Framework, there are three primary implementation tasks related to context APIs:

- Extending a base configuration context API, which can include:
  - Adding new methods
  - Overriding existing methods
- Creating a custom context API
- Adding resources for a specific line of business to the LOB/Job context API

This topic focuses on the process of extending a base configuration context API. For information on creating a custom context API, see “Creating context APIs” on page 67. For information on adding resources for a line of business, see “The Line/Job Context API” on page 77.

Overview of LOB-agnostic context APIs

In PolicyCenter, some business contexts function the same way, regardless of the policy's line of business. One example of this is the administration business context. Supervisors and administrators typically perform their tasks in the same way. Their work does not vary with any job's line of business. These business contexts are referred to as *LOB-agnostic business contexts*. Behavior Testing Framework supports them using LOB-agnostic context APIs.

An *LOB-agnostic context API* is a context API that declares its logic in a single interface/impl pair. There is no need to create multiple impls because the same set of methods are used for every scenario.

Extending LOB-agnostic context APIs

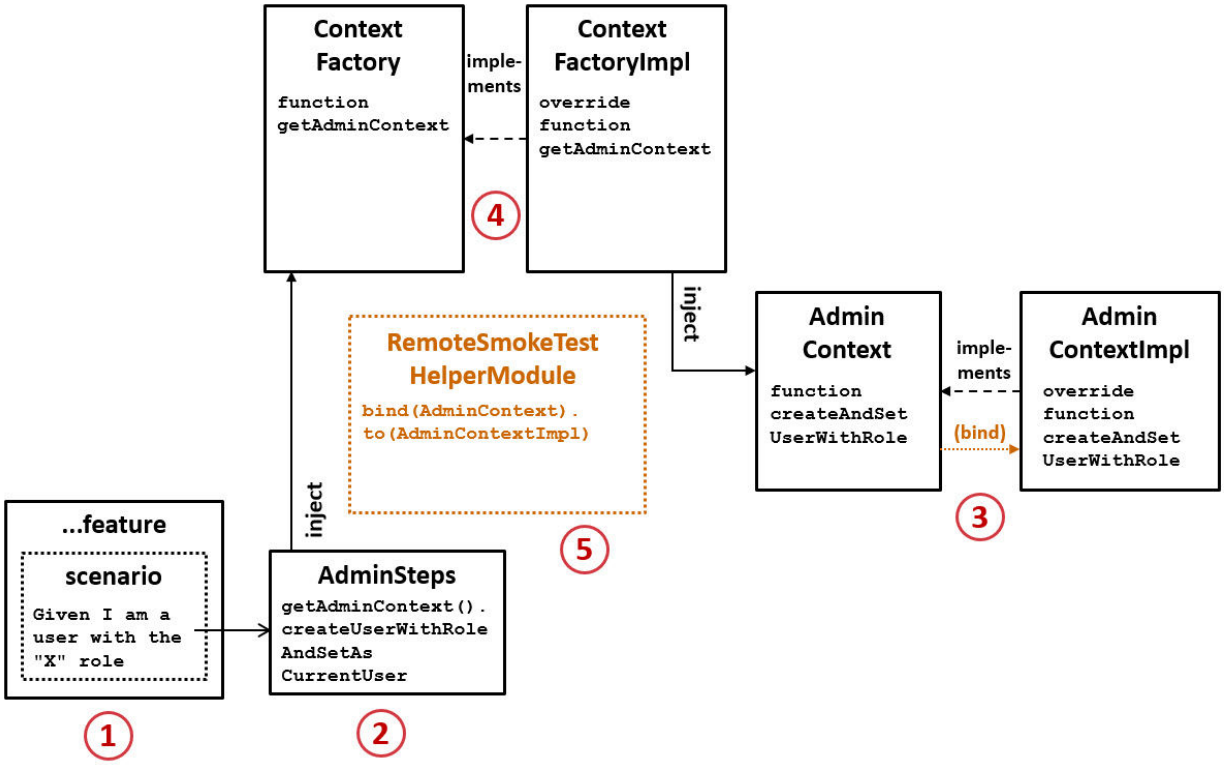
During an implementation of Behavior Testing Framework, you may need to extend an existing LOB-agnostic context API. This is necessary when the base configuration includes a context API, but it does not include all of the functionality you need for testing your implementation of PolicyCenter.

For example, your implementation may require testing of certain aspects of supervisor functionality, such as setting the email of a user that is in one of the supervisor's groups. This is administration functionality, but there are no methods to test this in the base configuration Admin Context. However, you can extend the Admin Context to add it.

LOB-agnostic context API resources

Base configuration resources for LOB-agnostic business contexts

The following diagram summarizes an example of the base configuration resources that support each LOB-agnostic business context. This example is for Admin Context.



First (#1), there are one or more scenarios that contain steps written in Gherkin. Every scenario is stored in a feature file.

Every step links to a step method. Step methods are defined in step classes, such as `AdminSteps` (#2). Step methods are written in Gosu. A step may require multiple implementations. Therefore, every step class injects an instance of ContextFactory. This instance is used to identify the correct implementation for each situation that uses the step. To make it easier to modify implementations without needing to modify scenarios, the implementations are not stored in the step classes themselves.

Admin Context (#3) is defined by two resources:

- `AdminContext`, which is an interface that defines methods relevant to this context.
- `AdminContextImpl`, which is an impl that implements the `AdminContext` interface. This is where the implementation code is stored.

Similarly, there are two resources that define ContextFactory (#4):

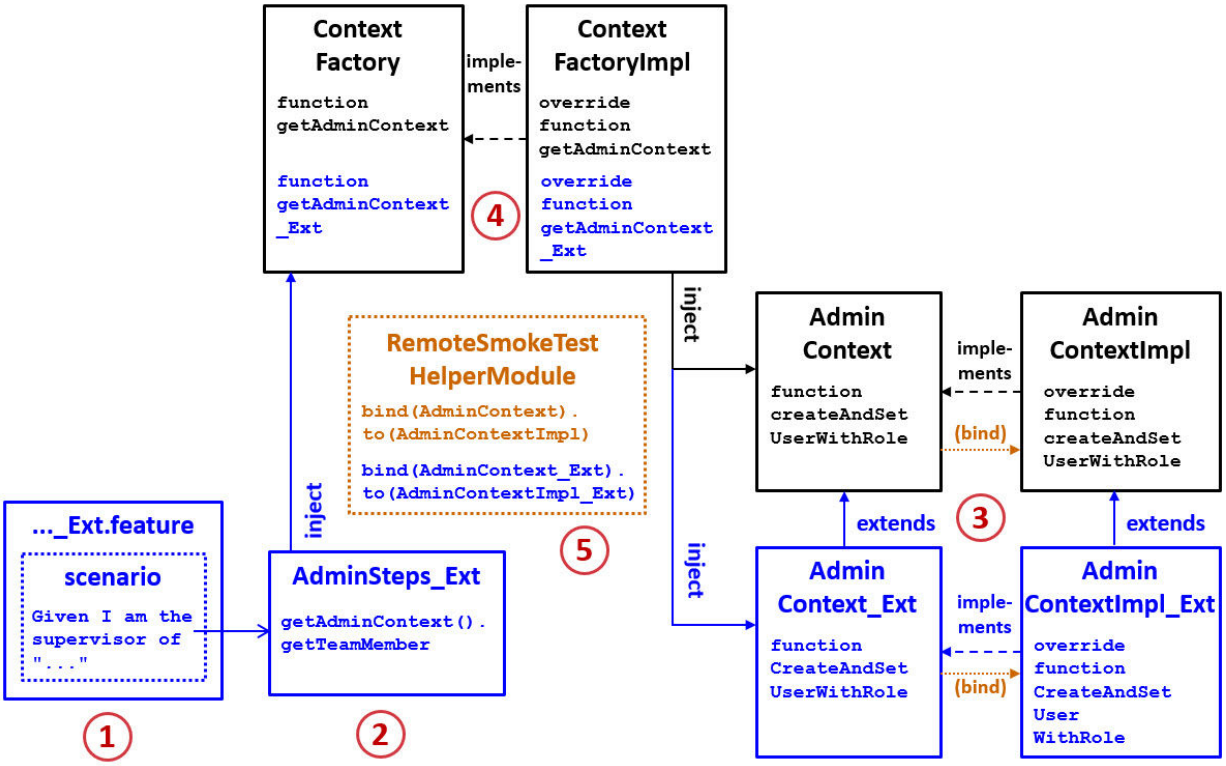
- `ContextFactory`, which is an interface that defines methods with return values of the different context APIs.
- `ContextFactoryImpl`, which is an impl that implements the ContextFactory interface. All of the different context APIs are injected into this impl. In turn, this impl is injected into every step file.

Finally, there is the RemoteSmokeTestHelperModule (#5): `PCRemoteSmokeTestHelperModule`. This file binds each interface to its corresponding impl.

Extending base configuration context APIs

When you extend a LOB-agnostic context API, you must create several new files and add code to existing files. The following diagram illustrates an example of this. In the diagram, base configuration resources appear in black. New files and code added to existing files appears in blue.

In this example, there are new scenarios testing a supervisor's ability to administer people on their team. To test these scenarios, AdminContext must be extended to add a step that creates a user who is a member of the supervisor's team.



First, you must create one or more new scenarios in new feature files (#1). These files specify new steps, such as "Given I am the supervisor of "*user*".

Then, you must create a new step definition in a new step class (#2). Guidewire recommends naming it *ExistingStepClass*_Ext, such as `AdminSteps_Ext`. Like the base configuration step classes, the new class must inject an instance of `ContextFactory`.

You must add two new resources for the extension context API itself (#3):

- `AdminContext_Ext`, which is an interface that extends the base configuration interface and defines the new methods needed.
- `AdminContextImpl_Ext`, which is an impl that extends the base configuration impl and implements the extension interface. The new implementation code is stored here.

You must also add code to Context Factory (#4):

- `ContextFactory` - Add a method the specifies a return value of the extension context API.
- `ContextFactoryImpl` - Inject the extension context API into this impl, and override the new interface method.

Finally, in PCRemoteSmokeTestHelperModule (#5), you must bind the extension interface to the extension impl.

Extending a base configuration context API

The following section identifies the high-level procedure for extending a base configuration LOB-agnostic context API. It is followed by an example that reiterates each step in a detailed manner.

The code in the examples can be copied and pasted into Studio to creating a working extension. Keep in mind that, in some cases, additional `uses` statements are required but are not present in the example code. If Studio states it "cannot resolve symbol" for a given type, you can add a uses statement by clicking inside the type name and pressing ALT + ENTER.

Extend a base configuration context API

About this task

This procedure is followed by a detailed example with code that you can copy and paste into Studio to form a working context API extension.

Procedure

1. Create new scenarios and feature files that specify new steps.
   - For more information on creating scenarios, see “Writing scenarios” on page 31.
2. Create a new step class that provides step methods for the new steps.
   - For more information on creating step methods, see “Creating step methods” on page 49.
3. Create a new context API interface.
   - It must extend the existing interface. (For example: `AdminContext_Ext extends AdminContext`)
   - It must define the new methods that are being added to the base configuration API.
4. Create a new context API impl.
   - It must extend the existing impl. (For example: `AdminContextImpl_Ext extends AdminContextImpl`)
   - It must implement the new interface. (For example: `implements AdminContext_Ext`)
   - It must provide implementations for the new methods.
5. Modify `ContextFactory`.
   - Add a new method that returns the extended context API. (For example: `AdminContext_Ext`).
6. Modify `ContextFactoryImpl`.
   - Inject the extended context API. (For example: `AdminContext_Ext`).
   - Add a new override that returns the extended context API. (For example: `AdminContext_Ext`).
7. Modify the RemoteSmokeTestHelperModule class.
   - Add code to bind the extended context API's interface to its impl.

Step 1: Creating scenarios that have new steps

Write new scenarios and feature files that contain new steps that extend the base configuration context API.

The following text is an example of a new `Email_Supervisor_Ext.feature` file. It contains several new steps, including `"I am the supervisor of "<user>"`.

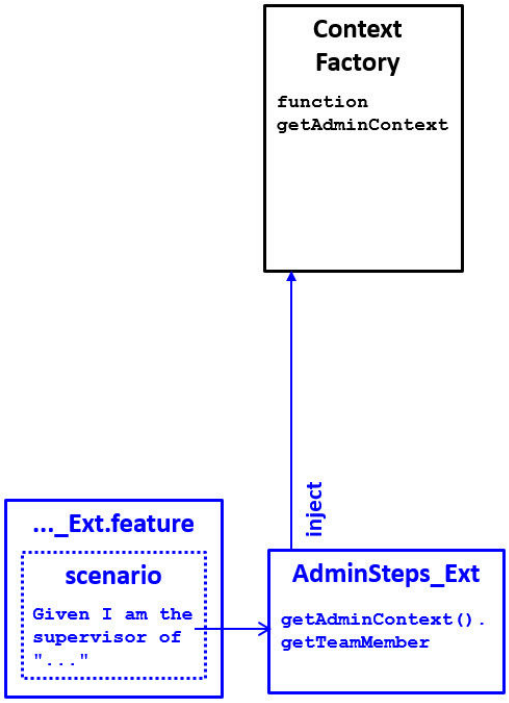
```
Feature: Email functionality

  As a supervisor,
  I want to administer my team's access to email functionality

  Background:
    Given I am a user with the "Underwriting Supervisor" role

  Scenario: Set a user's email address
    Given I am the supervisor of "Alice Applegate"
    When I set the user's email address to "aapplegate@customer.com"
    Then the user's primary email address should be "aapplegate@customer.com"
```

For more information on creating scenarios, see “Writing scenarios” on page 31

Step 2: Creating the step method

Create a new step class. The class must:

- Provide step methods for the new steps.
- Inject `ContextFactory`

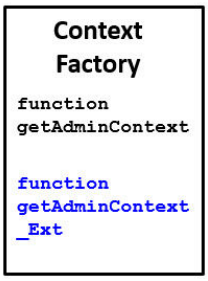


The following text is an example of a new `EmailSteps_Ext` class.

```
package gw.cucumber.customer.steps.admin

uses com.google.inject.Inject
uses cucumber.api.java.en.Given
uses cucumber.api.java.en.Then
uses cucumber.api.java.en.When
uses gw.cucumber.context.api.ContextFactory
// additional uses statements may be needed

class EmailSteps_Ext {

   @Inject
  var _contextFactory : ContextFactory

   @Given ("^I am the supervisor of \"([^\"]*)\"$")
  public function getTeamMember(userName : String) {
    _contextFactory.getAdminContext_Ext().getTeamMember(userName)
  }

   @When ("^I set the user's email address to \"([^\"]*)\"$")
  public function iSetTheUserEmailAddress(emailAddress : String) {
    _contextFactory.getAdminContext_Ext().setUserEmailAddress(emailAddress)
  }

   @Then ("^the user's primary email address should be \"([^\"]*)\"$")
  public function theEmailAddressShouldBe(emailAddress : String) {
    _contextFactory.getAdminContext_Ext().verifyEmailAddress(emailAddress)
  }

}
```

For more information on creating step methods, see “Creating step methods” on page 49.

Step 3: Extending the context API interface

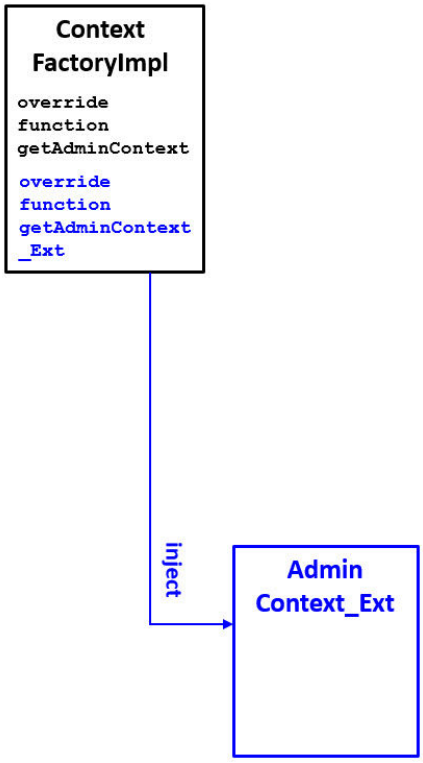


Create a new interface that extends the existing context API interface.

The following text is an example of a new `AdminContext_Ext` interface.

```
package gw.cucumber.customer.context.api

// additional uses statements may be needed
uses gw.cucumber.context.api.admin.AdminContext

interface AdminContext_Ext extends AdminContext {

  /**
   * Get a user who belongs to a team that the supervisor supervises
   *
   * @param userName
   */
  function getTeamMember(userName : String)

  /**
   * Set the user's email address.
   *
   * @param emailAddress
   */
  function setUserEmailAddress(emailAddress : String)

  /**
   * Verify the user's email address.
   *
   * @param emailAddress
   */
  function verifyEmailAddress(emailAddress : String)

}
```

Step 4: Extending the context API impl



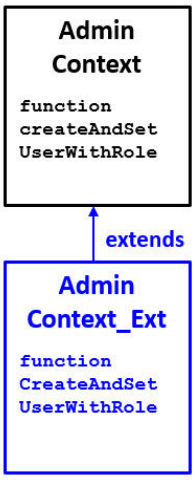
Create a new impl that extends the existing context API impl and implements the extension interface.

For each method, Guidewire recommends writing initial implementations that simply throw an `UnsupportedOperationException`. When the initial work is complete, you can test the structure by running a test and verifying that this type of exception is thrown. Later, you can replace each implementation with actual code.

The following text is an example of a new `AdminContextImpl_Ext` impl.

```
package gw.cucumber.customer.context.impl.smoketest

// additional uses statements may be needed
uses gw.cucumber.context.impl.smoketest.common.AdminContextImpl
uses gw.cucumber.customer.context.api.AdminContext_Ext

class AdminContextImpl_Ext extends AdminContextImpl implements AdminContext_Ext {

  override function getTeamMember(userName : String) {
    throw new UnsupportedOperationException("Not yet implemented")
  }

  override function setUserEmailAddress(emailAddress : String) {
    throw new UnsupportedOperationException("Not yet implemented")
  }

  override function verifyEmailAddress(emailAddress : String) {
    throw new UnsupportedOperationException("Not yet implemented")
  }
}
```

Step 5: Modifying ContextFactory

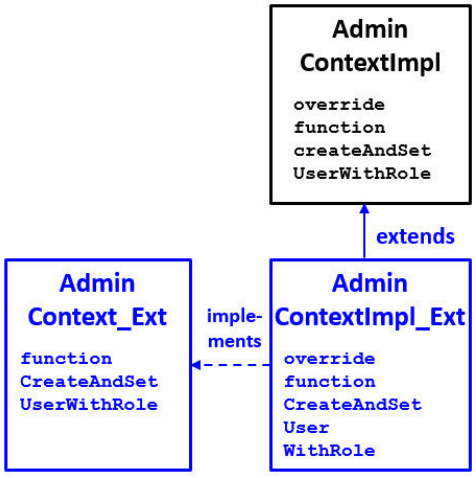


Modify `ContextFactory` so that it defines a method that returns the extension context API.

The following text is an example of code that can be added to `ContextFactory.gs`.

```
/**
 * Returns the api implementation object casted to {@link gw.cucumber.customer.steps.context.api.AdminContext_Ext}.
 * This method should be used on contexts that are LOB agnostic.
 */
function getAdminContext_Ext() : AdminContext_Ext
```

Step 6: Modifying ContextFactoryImpl

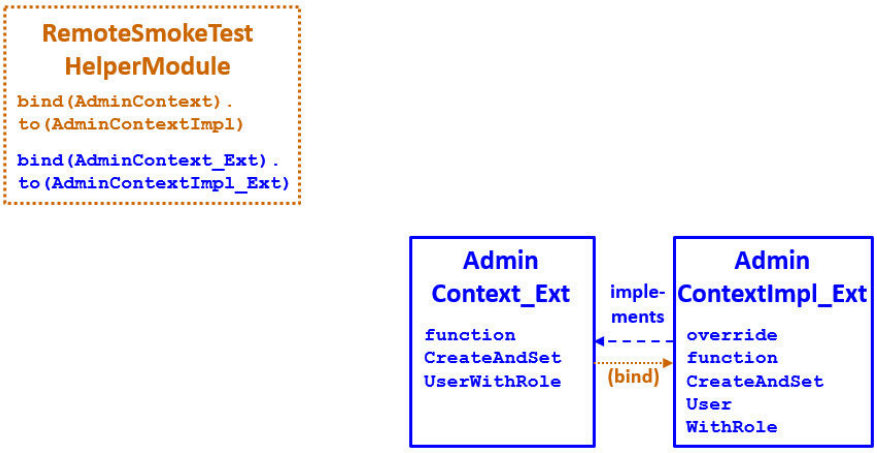


Modify `ContextFactoryImpl` so that it has an implementation for the new interface method and so that it injects the new context API.

The following text is an example of code that can be added to `ContextFactoryImpl`.

```
// Contexts
...
@Inject var _adminContext_Ext : AdminContext_Ext
...
override function getAdminContext_Ext(): AdminContext_Ext {  return _adminContext_Ext  }
```

Step 7: Binding the extension interface to the impl



Modify `PCRemoteSmokeTestHelperModule` so that it binds the extension interface and the extension impl.

The following is an example of code that can be added to `PCRemoteSmokeTestHelperModule.gs` to bind the new interface and impl.

```
...
bind(AdminContext_Ext).to(AdminContextImpl_Ext)
```

Running the first test

At this point, you can run the scenarios created in the first step. The scenario will fail because it will invoke a method whose implementation simply throws an `UnsupportedOperationException`. But the test will not result in an error. This ensures that you have the technical structure in place and you can begin to write the actual implementation code.

Summary of best practices

To minimize upgrade conflicts, Guidewire recommends that you avoid making modifications to the base configuration Behavior Testing Framework files. Instead, add your testing extensions to your own files. Exceptions to this recommendation include the following files, which are singleton files and present minimal effort when resolving upgrade conflicts:

- `ContextFactory.gs`
- `ContextFactoryImpl.gs`
- `PCRemoteSmoketestHelperModule.gs`

To ensure that your customizations do not create any upgrade conflicts, Guidewire recommends the following:

- End all file names with an `_Ext` suffix.
- Create context API interfaces in `gw.cucumber.customer.context.api.<approach>`. The name of `<approach>` identifies the technology used in the implementations, such as smoketest or rest.
- Create context API impls in `gw.cucumber.customer.context.impl`.
- For you initial scenario, implement the interface methods so that they throw an `UnsupportedOperationException`. Once you have confirmed that the scenario fails and throw this exception, you can replace impl methods with actual code as needed.

chapter 8

Creating context APIs

For Behavior Testing Framework, there are three primary implementation tasks related to context APIs:

- Extending a base configuration context API, which can include:
  - Adding new methods
  - Overriding existing methods
- Creating a custom context API
- Adding resources for a specific line of business to the LOB/Job context API

This topic focuses on the process of creating a custom context API. For information on extending a base configuration context API, see "Extending context APIs" on page 57. For information on adding resources for a line of business, see "The Line/Job Context API" on page 77.

Overview of LOB-agnostic context APIs

In PolicyCenter, some business contexts method the same way, regardless of the policy's line of business. One example of this is the administration business context. Supervisors and administrators typically perform their tasks in the same way. Their work does not vary with any job's line of business. These business contexts are referred to as *LOB-agnostic business contexts*. Behavior Testing Framework supports them using LOB-agnostic context APIs.

An *LOB-agnostic context API* is a context API that declares its logic in a single interface/impl pair. There is no need to create multiple impls because the same set of methods are used for every scenario.

Creating new LOB-agnostic context APIs

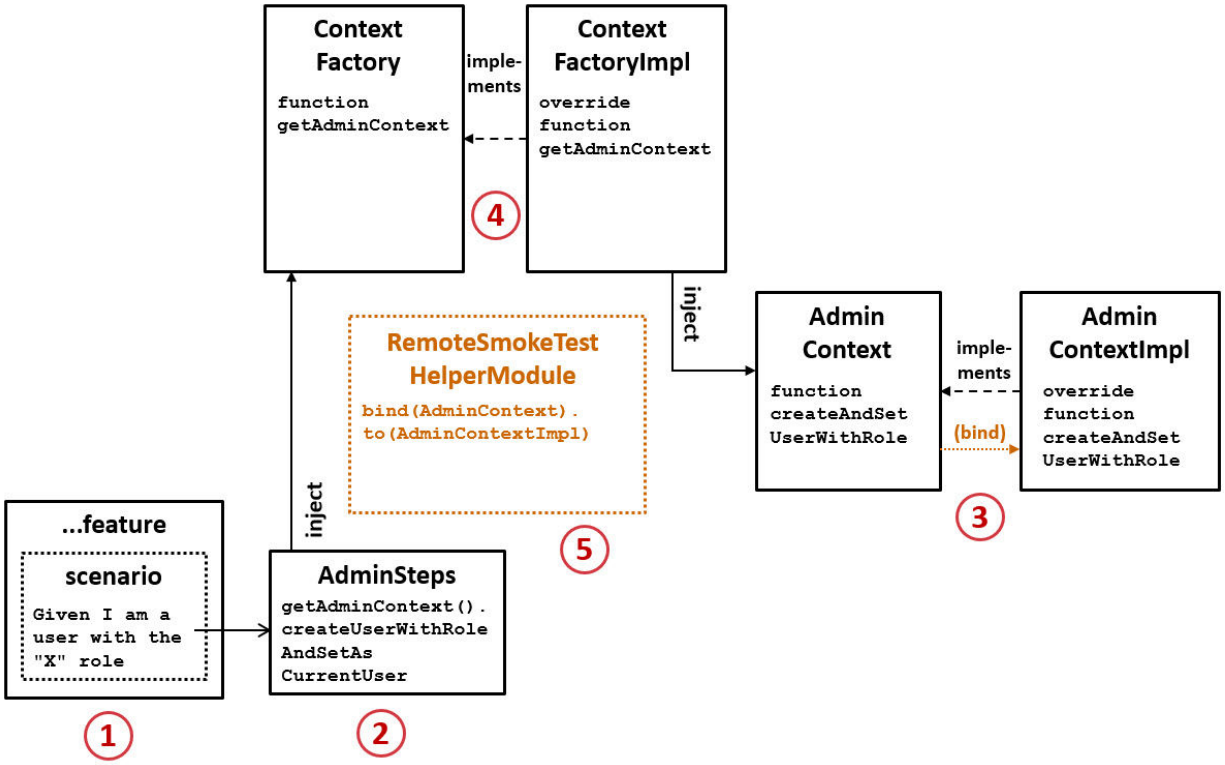
During an implementation of Behavior Testing Framework, you may need to create a new LOB-agnostic context API. This is necessary when you need functionality for testing your implementation of PolicyCenter that does not map to any base configuration context API.

For example, your implementation may require testing of certain aspects of calendar functionality, such as defining a holiday and ensuring that PolicyCenter takes the holiday into account when calculating an activity's due date. This type of functionality does not correspond to any of the base configuration context APIs. It may require a new context API, such as CalendarContext.

LOB-agnostic context API resources

Base configuration resources for LOB-agnostic business contexts

The following diagram summarizes an example of the base configuration resources that support each LOB-agnostic business context. This example is for Admin Context.



First (#1), there are one or more scenarios that contain steps written in Gherkin. Every scenario is stored in a feature file.

Every step links to a step method. Step methods are defined in step classes, such as `AdminSteps` (#2). Step methods are written in Gosu. A step may require multiple implementations. Therefore, every step class injects an instance of ContextFactory. This instance is used to identify the correct implementation for each situation that uses the step. To make it easier to modify implementations without needing to modify scenarios, the implementations are not stored in the step classes themselves.

Admin Context (#3) is defined by two resources:

- `AdminContext`, which is an interface that defines methods relevant to this context.
- `AdminContextImpl`, which is an impl that implements the `AdminContext` interface. This is where the implementation code is stored.

Similarly, there are two resources that define ContextFactory (#4):

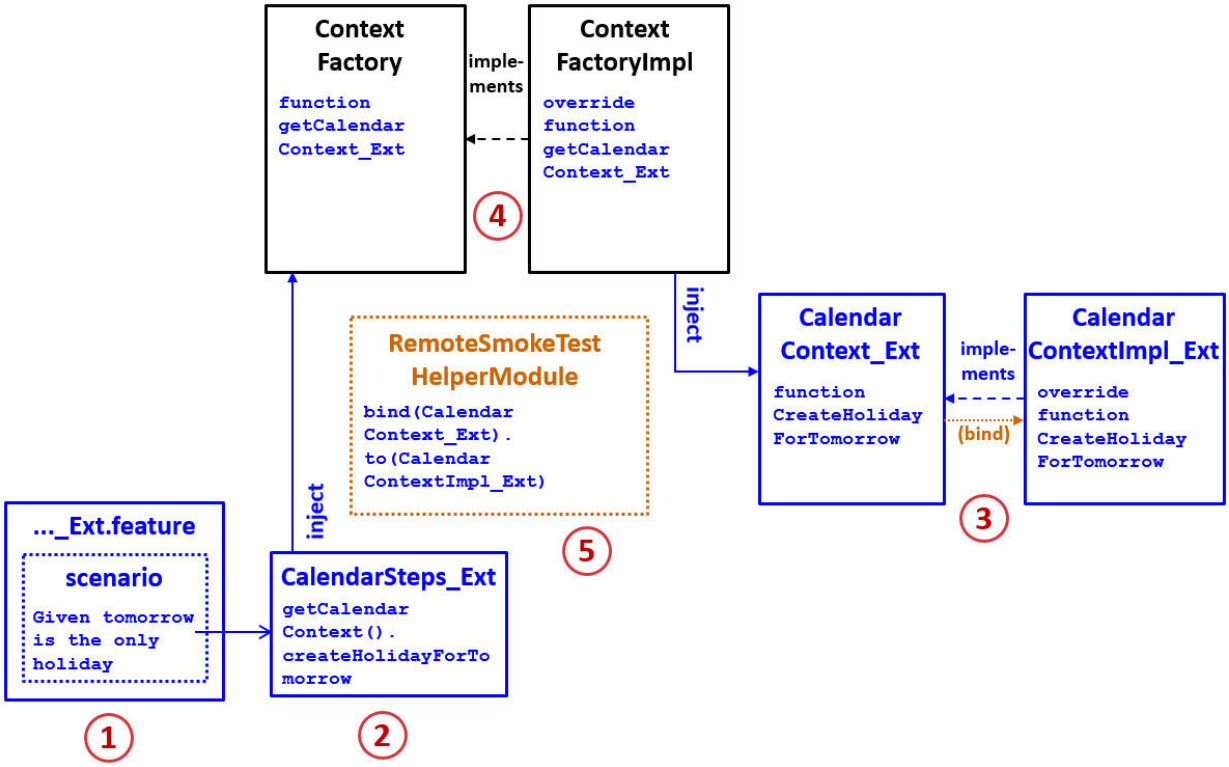
- `ContextFactory`, which is an interface that defines methods with return values of the different context APIs.
- `ContextFactoryImpl`, which is an impl that implements the ContextFactory interface. All of the different context APIs are injected into this impl. In turn, this impl is injected into every step file.

Finally, there is the RemoteSmokeTestHelperModule (#5): `PCRemoteSmokeTestHelperModule`. This file binds each interface to its corresponding impl.

Creating new context APIs

When you create a new LOB-agnostic context APIs, you must create several new files and add code to existing files. The following diagram illustrates an example of this. In the diagram, base configuration resources appear in black. New files and code added to existing files appears in blue.

In this example, there are new scenarios testing the impact of holidays on activity due dates. To test these scenarios, a new CalendarContext must be created to add new step methods that create holidays, create activities, and verify activity due dates.



First, you must create one or more new scenarios in new feature files (#1). These files specify new steps, such as "Given tomorrow is the only holiday".

Then, you must create a new step definition in a new step class (#2). Guidewire recommends naming it *ContextSteps*_Ext, such as `CalendarSteps_Ext`. Like the base configuration step classes, the new class must inject an instance of `ContextFactory`.

You must add two new resources for the new context API itself (#3):

- `CalendarContext_Ext`, which is an interface that defines the new methods needed.
- `CalendarContextImpl_Ext`, which is an impl that implements the new interface. The new implementation code is stored here.

You must also add code to Context Factory (#4):

- `ContextFactory` - Add a method the specifies a return value of the new context API.
- `ContextFactoryImpl` - Inject the new context API into this impl, and override the new interface method.

Finally, in PCRemoteSmokeTestHelperModule (#5), you must bind the new interface to the new impl.

Creating a new context API

The following section identifies the high-level procedure for creating a new LOB-agnostic context API. It is followed by an example that reiterates each step in a detailed manner.

The code in the examples can be copied and pasted into Studio to creating a working context API. Keep in mind that, in some cases, additional `uses` statements are required but are not present in the example code. If Studio states it "cannot resolve symbol" for a given type, you can add a uses statement by clicking inside the type name and pressing ALT + ENTER.

Create a new context API

About this task

This procedure is followed by a detailed example with code that you can copy and paste into Studio to form a working custom context API.

Procedure

1. Create new scenarios and feature files that specify new steps.
   - For more information on creating scenarios, see “Writing scenarios” on page 31.
2. Create a new step class that provides step methods for the new steps.
   - For more information on creating step methods, see “Creating step methods” on page 49.
3. Create a new context API interface.
   - It must define the new methods that are being added to the base configuration API.
4. Create a new context API impl.
   - It must implement the new interface. (For example: `implements CalendarContext_Ext`)
   - It must provide implementations for the new methods.
5. Modify `ContextFactory`.
   - Add a new method that returns the new context API. (For example: `CalendarContext_Ext`).
6. Modify `ContextFactoryImpl`.
   - Inject the new context API. (For example: `CalendarContext_Ext`).
   - Add a new override that returns the new context API. (For example: `CalendarContext_Ext`).
7. Modify the RemoteSmokeTestHelperModule class.
   - Add code to bind the new context API's interface to its impl.

Step 1: Creating scenarios that use new steps

Write new scenarios and feature files that contain new steps that extend the base configuration context API.

The following text is an example of a new `Calendar_Supervisor_Ext.feature` file. It contains several new steps, including `"I am the supervisor of "<user>"`.

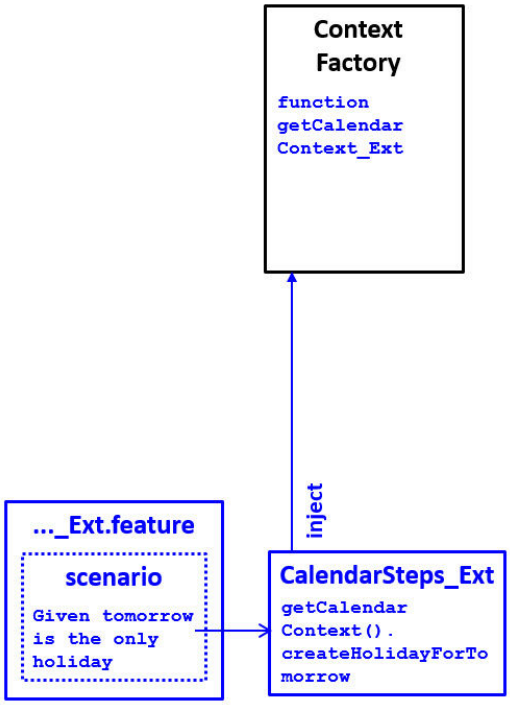
```
Feature: Calendar functionality

  As a supervisor,
  I want to administer the business calendar

  Background:
    Given I am a user with the "Underwriting Supervisor" role
    And today is a "Monday"
    And tomorrow is the only holiday

  Scenario: Adding a new holiday to the calendar
    When I create an activity using the "General reminder" activity pattern
    Then there should be a "General remainder" activity
    And the due date should be "2" days from today
```

For more information on creating scenarios, see “Writing scenarios” on page 31.

Step 2: Creating the step method

Create a new step class. The class must:

- Provide step methods for the new steps.
- Inject `ContextFactory`



The following text is an example of a new `CalendarSteps_Ext` class.

```
package gw.cucumber.customer.steps.admin

uses com.google.inject.Inject
uses cucumber.api.java.en.Given
uses cucumber.api.java.en.Then
uses cucumber.api.java.en.When
uses gw.cucumber.context.api.ContextFactory
// additional uses statements may be needed

class  CalendarSteps_Ext {

   @Inject
   var _contextFactory : ContextFactory

   @Given ("^today is a \"([^\"]*)\"$")
   public function todayIsDayOfWeek(dayOfWeek : String) {
     _contextFactory.getCalendarContext_Ext().setTodaysDate(dayOfWeek)
  }

   @Given ("^tomorrow is the only holiday$")
   public function createHolidayForTomorrow() {
     _contextFactory.getCalendarContext_Ext().createHolidayForTomorrow()
  }

   @When ("^I create an activity using the \"([^\"]*)\" activity pattern$")
   public function iCreateActivityFromActivityPattern(activityPattern : String) {
     _contextFactory.getCalendarContext_Ext().createActivityFromActivityPattern(activityPattern)
  }

   @Then ("^there should be a \"([^\"]*)\" activity$")
   public function thereShouldBeAnActivity(activityPattern : String) {
     _contextFactory.getCalendarContext_Ext().verifyActivity(activityPattern)
  }

   @Then ("^the due date is \"(\\d+)\" days from today$")
   public function theDueDateIsXDaysFromToday(dueDateOffset :  int ) {
```

```
      _contextFactory.getCalendarContext_Ext().verifyTimeUntilActivityIsDue(dueDateOffset)
  }

}
```

For more information on creating step methods, see “Creating step methods” on page 49.

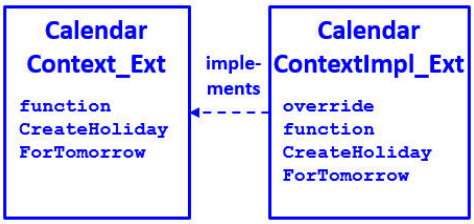
Step 3: Creating the context API interface

Create an interface for the new context API interface.

The following text is an example of a new `CalendarContext_Ext` interface.

```
package gw.cucumber.customer.context.api

// additional uses statements may be needed

interface CalendarContext_Ext {

   /**
   * Set the application date to the first date on or after the current application date that is the given day of
week.
   *
   * @param dayOfWeek
   */
   function setTodaysDate(dayOfWeek : String)

   /**
   * Create a holiday for the day after the application date.
   */
   function createHolidayForTomorrow()

   /**
   * Create an activity with default values using the given activity pattern.
   *
   * @param activityPattern
   */
   function createActivityFromActivityPattern(activityPattern : String)

   /**
   * Verify that an activity exists whose activity pattern is the current pattern.
   *
   * @param activityPattern
   */
   function verifyActivity(activityPattern : String)

   /**
   * Verify that the difference between the application date and the activity's due date is equal to the expected
offset
   *
   * @param dueDateOffset
   */
   function verifyTimeUntilActivityIsDue(dueDateOffset : int)

}
```

Step 4: Creating the context API impl



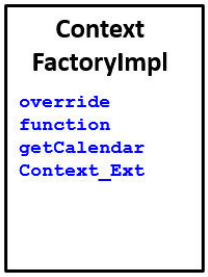
Create a new impl for the context API that implements the new interface.

For each method, Guidewire recommends writing initial implementations that simply throw an `UnsupportedOperationException`. When the initial work is complete, you can test the structure by running a test and verifying that this type of exception is thrown. Later, you can replace each implementation with actual code.

The following text is an example of a new `CalendarContextImpl_Ext` impl.

```
package  gw.cucumber.customer.context.impl.smoketest

// additional uses statements may be needed
uses gw.cucumber.customer.context.api.CalendarContext_Ext

class CalendarContextImpl_Ext implements CalendarContext_Ext {

   override function setTodaysDate(dayOfWeek : String) {
     throw new UnsupportedOperationException("Not yet implemented")
  }

   override function createHolidayForTomorrow() {
     throw new UnsupportedOperationException("Not yet implemented")
  }

   override function createActivityFromActivityPattern(activityPattern : String) {
     throw new UnsupportedOperationException("Not yet implemented")
  }

    override function verifyActivity(activityPattern : String) {
     throw new UnsupportedOperationException("Not yet implemented")
  }

   override function verifyTimeUntilActivityIsDue(dueDateOffset : int ) {
     throw new UnsupportedOperationException("Not yet implemented")
  }
}
```

Step 5: Modifying ContextFactory

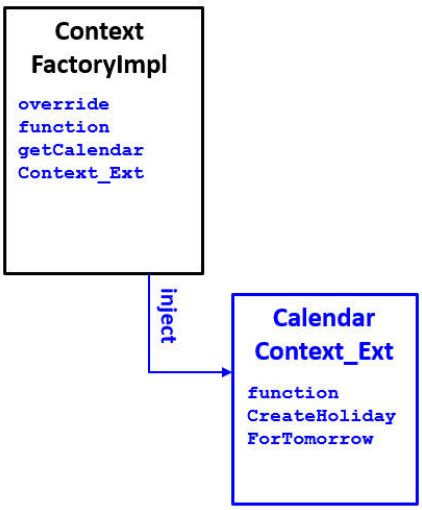


Modify `ContextFactory` so that it defines a method that returns the new context API.

The following text is an example of code that can be added to `ContextFactory.gs`.

```
/**
 * Returns the api implementation object casted to {@link
gw.cucumber.cucumber.customer.steps.context.api.CalendarContext_Ext}.
 * This method should be used on contexts that are LOB agnostic.
 */
function getCalendarContext_Ext() : CalendarContext_Ext
```

Step 6: Modifying ContextFactoryImpl

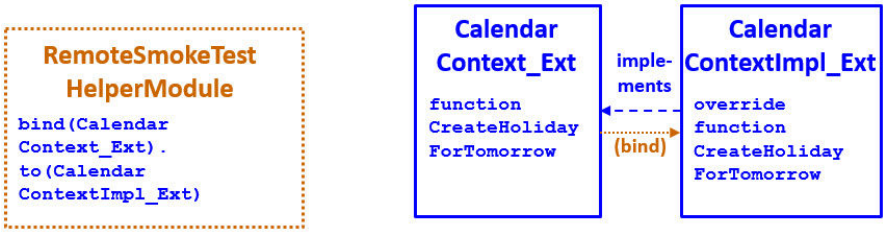


Modify `ContextFactoryImpl` so that it has an implementation for the new interface method and so that it injects the new context API.

The following text is an example of code that can be added to `ContextFactoryImpl`.

```
// Contexts
...
@Inject var _calendarContext_Ext : CalendarContext_Ext
...
override function getCalendarContext_Ext(): CalendarContext_Ext {  return _calendarContext_Ext  }
```

Step 7: Binding the new interface to the impl



Modify `PCRemoteSmokeTestHelperModule` so that it binds the extension interface and the extension impl.

The following is an example of code that can be added to `PCRemoteSmokeTestHelperModule.gs` to bind the new interface and impl.

```
...
bind(CalendarContext_Ext).to(CalendarContextImpl_Ext)
```

Running the first test

At this point, you can run the scenarios created in the first step. The scenario will fail because it will invoke a method whose implementation simply throws an `UnsupportedOperationException`. But the test will not result in an error. This ensures that you have the technical structure in place and you can begin to write the actual implementation code.

Summary of best practices

To minimize upgrade conflicts, Guidewire recommends that you avoid making modifications to the base configuration Behavior Testing Framework files. Instead, add your testing extensions to your own files. Exceptions

to this recommendation include the following files, which are singleton files and present minimal effort when resolving upgrade conflicts:

- `ContextFactory.gs`
- `ContextFactoryImpl.gs`
- `PCRemoteSmoketestHelperModule.gs`

To ensure that your customizations do not create any upgrade conflicts, Guidewire recommends the following:

- End all file names with an `_Ext` suffix.
- Create context API interfaces in `gw.cucumber.customer.context.api.<approach>`. The name of `<approach>` identifies the technology used in the implementations, such as smoketest or rest.
- Create context API impls in `gw.cucumber.customer.context.impl`.
- For you initial scenario, implement the interface methods so that they throw an `UnsupportedOperationException`. Once you have confirmed that the scenario fails and throw this exception, you can replace impl methods with actual code as needed.

chapter 9

The Line/Job Context API

The Line/Job Context API stores implementation information that is relevant to either a specific line of business, a specific job, or both. This topic discusses the structure of the Line/Job Context API.

- For information on supporting new lines of business for submission jobs, see “Testing submissions on a new line of business” on page 85.

Overview of LOB-specific business contexts

In PolicyCenter, some business contexts function differently based on the policy's line of business. One example of this is the context of creating a submission. Testing a personal auto submission has different requirements than testing a workers' compensation submission.

When a business context varies based on the lines of business:

- Some testing logic may be needed for all lines of business and is identical across all lines of business.
  - For example, creating an activity for a personal auto policy is identical to creating an activity for a workers' compensation policy.
- Some testing logic may be needed for all lines of business, but is different for each line of business.
  - For example, both personal auto and workers' compensation have costs. But, the logic to determine if a cost exists is different for each line of business.
- Some testing logic may be needed for only one line of business. For example:
  - Personal auto submissions require the ability to add a vehicle.
  - Workers' compensation submissions require the ability to add covered employees

When a business context has testing requirements that vary based on the line of business, they are referred to as *LOB-specific business contexts*. Behavior Testing Framework supports them using the LOB/Job Context API.

In the base configuration, the LOB/Job Context API provides support for two lines of business (personal auto and workers' compensation) and all of the jobs common to underwriting (submission, issuance, policy change, cancellation, reinstatement, rewrite, renewal, and audit). Testing requirements vary with each insurer, and therefore the LOB/Job Context API does not provide support for all possible combinations of these lines of business and jobs. Rather, the intention of the base configuration is to provide a wide breadth of examples. Based on these examples, insurers can extend the LOB/Job Context API to suit their specific needs.

Structure of the Line/Job Context API

When testing scenarios that are LOB-specific or job-specific, there is sometimes a need to share implementation code, and sometimes a need to isolate it. For example:

- A method to add line level coverages is needed for all policy-related tests, but:
  - A method to add drivers is needed only for personal auto policy tests.
  - A method to add covered workers is needed only for workers' compensation policy tests.
- A method to quote a job is needed for all job-related tests, but:
  - A method to start a renewal is needed only for renewal tests.
  - A method to apply preemption changes is needed only for policy change tests.

The Line/Job Context API has a multi-dimensional, multi-layer structure. This structure provides a way to share code as needed and isolate code as needed.

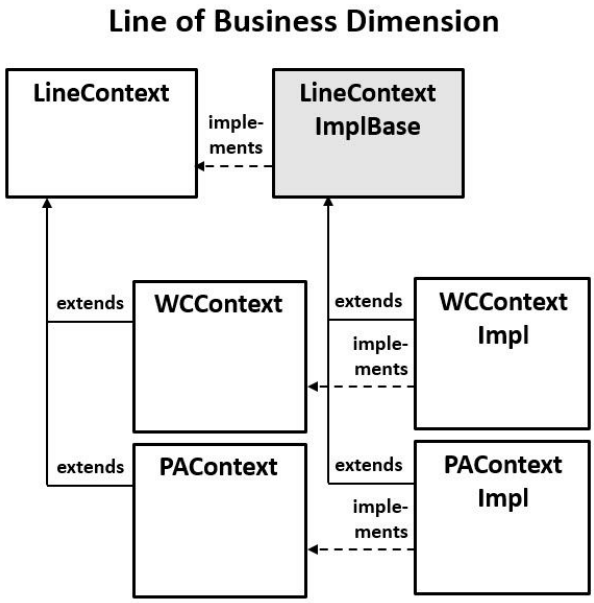
There are two dimensions to this context API: one for Line of Business and one for Job.

The line of business dimension

The line of business dimension has a top-level, generic LineContext layer and multiple line-specific child layers. Methods that are required for every line (such as `addLineLevelCoverage`) are declared at the top level. Methods that are specific to a given line are declared at the child levels. For example:

- `addCoveredEmployees` is declared at the `WCContext` level.
- `addDriver` is declared at the `PAContext` level.

The following diagram depicts the line of business dimension. Gray shading indicates the class is abstract.

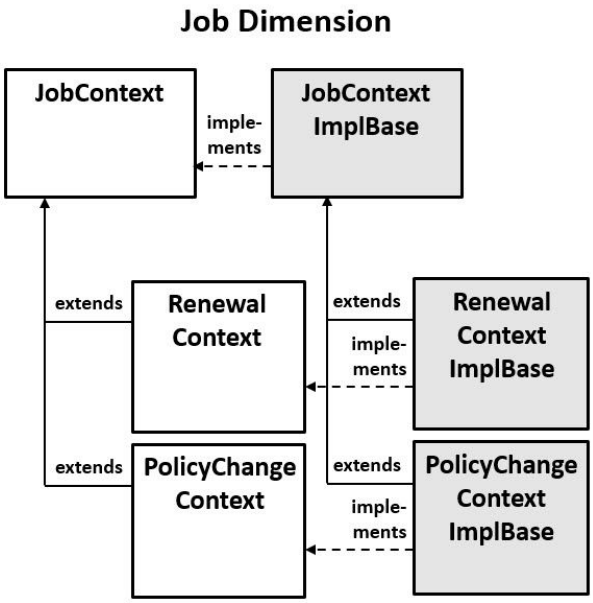


The job dimension

Similarly, the job dimension has a top-level, generic `JobContext` layer and multiple job-specific child layers. Methods that are required for every job (such as `quote`) are declared at the top level. Methods that are specific to a given job are declared at the child levels. For example:

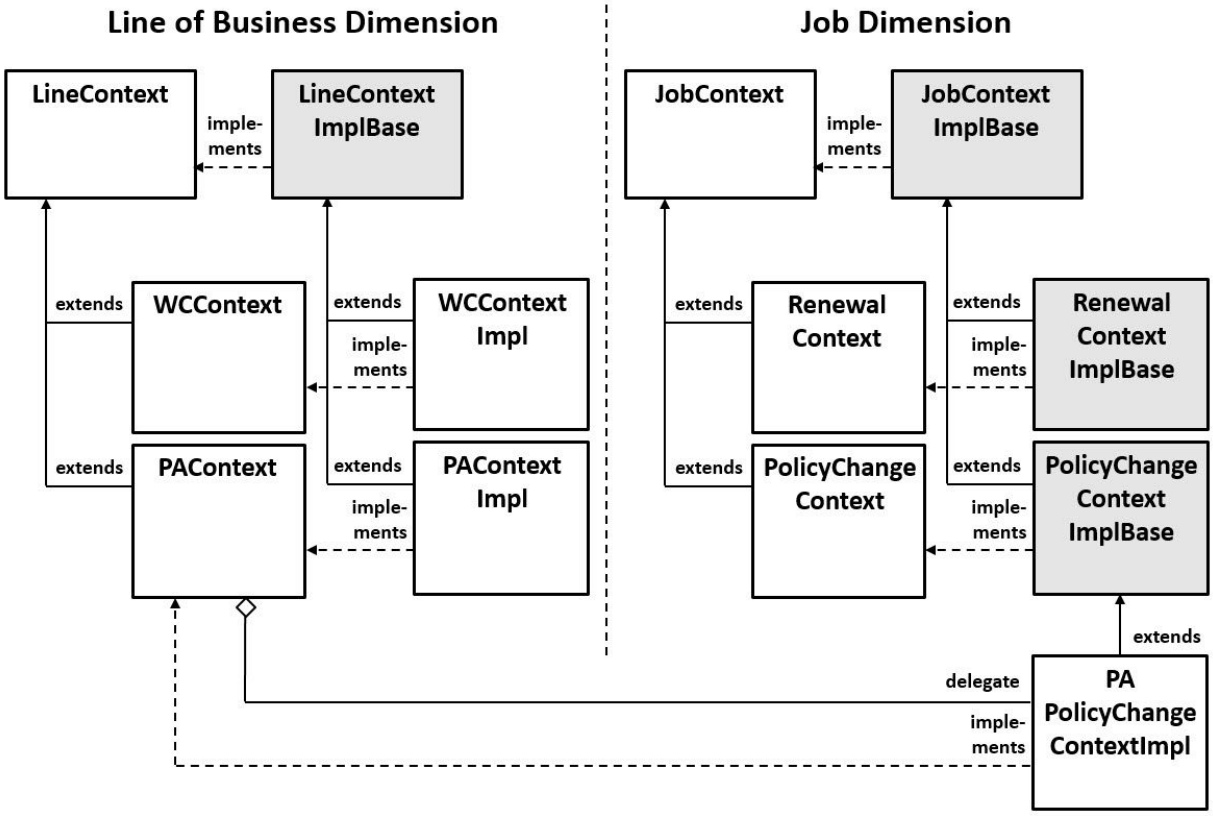
- `startRenewal` is declared at the `Renewal` level.
- `applyPreemptionChanges` is declared at the `PolicyChange` level.

The following diagram depicts the job dimension. Gray shading indicates the class is abstract.



The line/job impls

For every line/job combination that requires testing, there is an impl whose name is `<Line><Job>ContextImpl`. This impl combines methods from the appropriate line of business layer and the appropriate job layer. For example, the following diagram depicts an example of `PAPolicyChangeContextImpl`. This impl combines methods from the line-specific `PAContext` and the job-specific `PolicyChangeImpl`.



Like Java, Gosu does not support multiple inheritance. To create a `<Line><Job>ContextImpl` class (such as `PAPolicyChangeContextImpl`) that inherits information from multiple, non-related parent classes:

- The job dimension is considered the "primary" dimension. The `<Line><Job>ContextImpl` extends the appropriate `<Job>ContextImplBase` class and inherits its methods.
  - For example, `PAPolicyChangeContextImpl` extends `PolicyChangeContextImplBase`.
- The methods from the appropriate `<Line>ContextImpl` are made available to the `<Line><Job>ContextImpl` class through the use of a delegate named `_lineContextDelegate`.
  - For example, the methods from `PAContextImpl` are made available to `PAPolicyChangeContextImpl` through the use of a delegate named `_lineContextDelegate`.

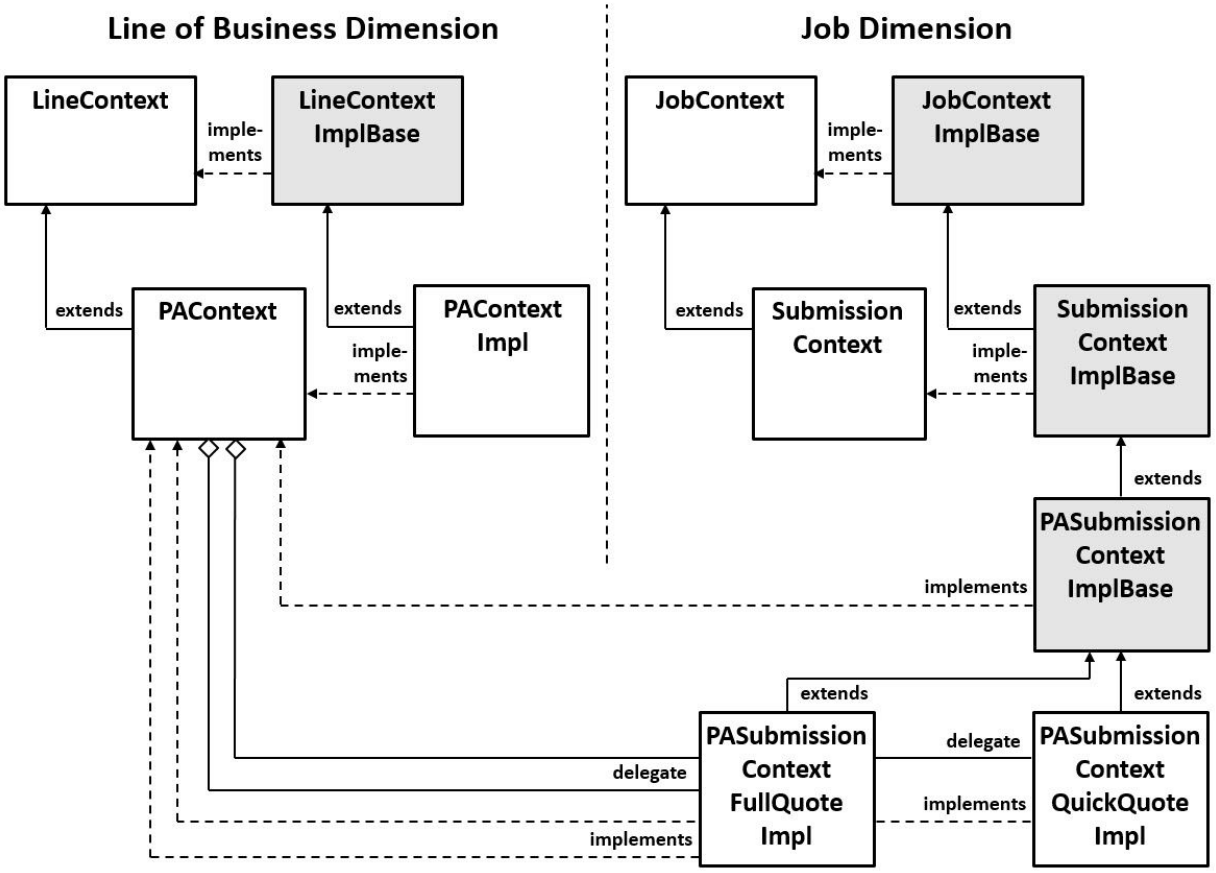
This multi-layered structure provides flexibility so that testing logic can be implemented in a way that matches the needs of each line of business and each job.

- Logic that is identical across all lines of business and jobs is implemented in either in `LineContext` and its impl or in `JobContext` and its impl. For example:
  - `addLineLevelCoverages` is implemented in `LineContextImplBase`.
  - `setEffectiveDate` is implemented in `JobContextImplBase`.
- Logic that is LOB-specific (but the same for all jobs) is declared in `<Line>Context` and its impl. For example:
  - `addDriver` is implemented in `PAContextImpl`.
- Logic that is job-specific (but the same for all LOBS) is declared in `<Job>Context` and its impl. For example:
  - `applyPreemptionChanges` is implemented in `PolicyChangeContextImpl`.
- Logic that is LOB-specific and job-specific is declared in `<Line><Job>Context` and its impl. For example:
  - `verifyVehicleExists` is implemented in `PAPolicyChangeContextImpl`.

The structure for submissions

All of the jobs follow the structure described above except for one - the submission job. PolicyCenter supports two styles of submission, quick quote and full quote. Some testing logic is needed for both styles, and some is needed only for quick quotes or for full quotes. To ensure that code is shared and isolated as appropriate, the submission job has an additional layer.

The following diagram depicts the structure for personal auto submissions. Gray shading indicates the class is abstract.



As is the case with other jobs, there is a `SubmissionContextImplBase` impl. However, the child of this class is not a concrete `<LOB>SubmissionContextImpl` impl. For example, there is no `PASubmissionContextImpl`.). Rather, it is a second abstract impl named `<LOB>SubmissionContextImplBase` (such as `PASubmissionContextImplBase`). This impl has two concrete children:

- `<LOB>SubmissionContextQuickQuoteImpl`
- `<LOB>SubmissoinContextFullQuoteImpl`

For example, `PASubmissionContextImplBase` has these two children:

- `PASubmissionContextQuickQuoteImpl`
- `PASubmissoinContextFullQuoteImpl`

Logic that is relevant to both types of submissions is declared in the parent `ContextImplBase` impl. Logic that is specific to either quick quotes only or full quotes only is declared in the relevant child impl.

Resources associated with the Line/Job Context API

Resources directly associated with the Line/Job Context API

There are two resources directly associated with the LOB/Job context API:

- Context Factory
- `PCRemoteSmokeTestHelper`

Context Factory

Context Factory is the name used to refer to two Gosu classes: `ContextFactory` (an interface) and `ContextFactoryImpl` (an implementation of the interface). All of the implementations needed for testing (such as

implementations for admin testing, activity testing, and so on) are injected into Context Factory. Context Factory is injected into every step file. In this way, Context Factory acts as a dependency object that provides every step definition with the ability to identify and call the appropriate implementation.

For the Line/Job context API, the only interfaces injected into `ContextFactoryImpl` are the generic job interfaces, such as `PolicyChangeContext`, `RenewalContext`, and so on. All of the impls that are specific to a single Line/Job combination (such as `WCPolicyChangeContextImpl` or `PARenewalContextImpl`) extend the generic job level, and therefore they are inherently available to `ContextFactoryImpl`. When you create a new line of business, the new line extends the generic job level. Therefore, you typically do not need to modify Context Factory.

PCRemoteSmokeTestHelper

`PCRemoteSmokeTestHelper` is a class that binds every context API implementation to one or more impl classes. This class contains the logic used during runtime to identify the correct impl to use for a given interface that has been injected into a step class through Context Factory.

For the Line/Job context API, the impls that are specific to a single Line/Job combination are bound to the `<Job>Context` interface using map binders. For example, the following code creates the `policyChangeContextBinder` map binder:

```
// Policy Change Contexts
var policyChangeContexts = MapBinder.newMapBinder(binder(), ProductCode, PolicyChangeContext)
policyChangeContexts.addBinding(ProductCode.PERSONAL_AUTO).to(PAPolicyChangeContextImpl)
policyChangeContexts.addBinding(ProductCode.WORKERS_COMPENSATION).to(WCPolicyChangeContextImpl)
```

When `PolicyChangeContext` is called in a scenario that uses a given policy type, the methods in the appropriate Line/Job-specific impl are used to execute the scenario.

Resources indirectly associated with the Line/Job Context API

There are four sets of resources indirectly associated with the Line/Job context API:

- Feature files
- Step classes
- `TagInfoProcessor`
- `ProductCode`

Feature files and step classes

Feature files and step classes contain step method code used for testing a specific line of business and job. But, neither of these files directly reference any context API. Feature files reference only step definitions in step classes, and step classes reference only Context Factory.

TagProcessorInfo

Every feature file that has LOB-specific scenarios starts with an LOB tag, such as `@personal_auto` or `@homeowners`. This tag identifies the LOB to be used for the scenarios.

Similarly, every feature file that has job-specific scenarios starts with a job tag, such as `@submission` or `@policy_change`. This tag identifies the job to be used for the scenarios.

`TagInfoProcessor` contains code that sets the product code and the job subtype to the appropriate values for each LOB-specific or job-specific scenario.

`TagInfoProcessor` is not directly associated with the Line/Job Context API. But, it contains code necessary for setting up pre-condition data for every line-specific or job-specific scenario. (`TagInfoProcessor` does not play a role in testing LOB-agnostic and job-agnostic scenarios.)

ProductCode

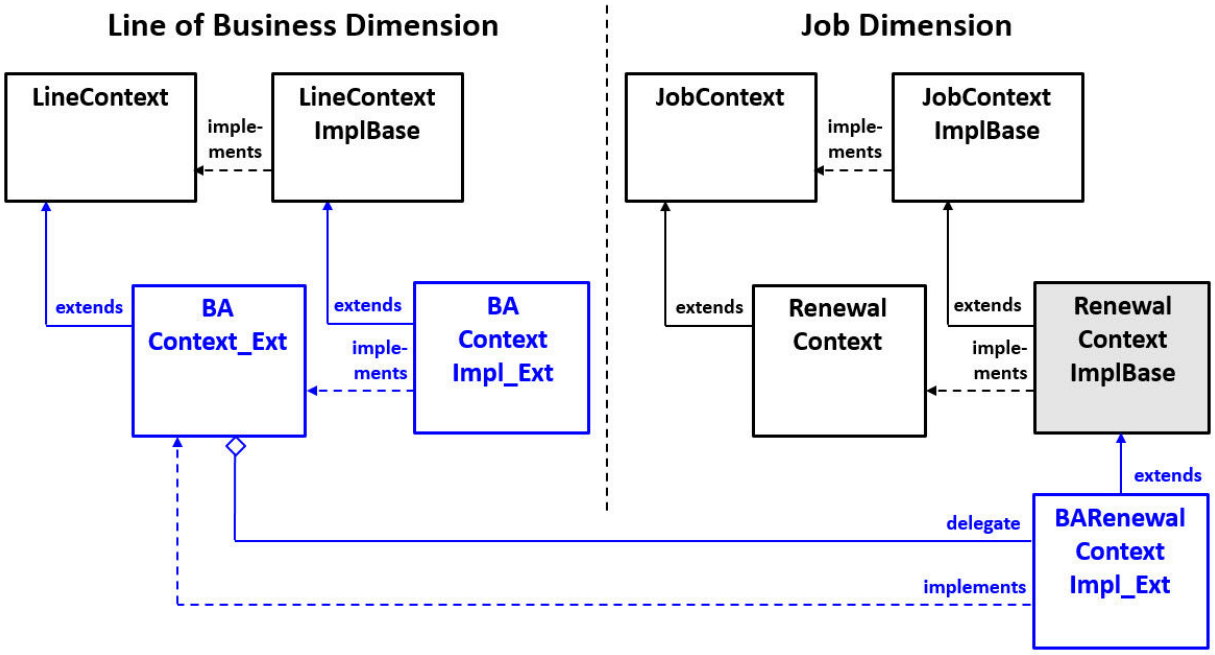
`ProductCode` is an enum that identifies the lines of business supported by Behavior Testing Framework.

For each line of business, there is an enum value (such as `HOMEOWNERS` or `WORKERSCOMPENSATION`) associated with the corresponding typecode from PolicyCenter (such as `HOPHomeowners` or `WorkersCompensation`).

Like `TagInfoProcessor`, `ProductCode` is not directly associated with the Line/Job Context API. But, it contains code necessary for setting up pre-condition data for every line-specific or job-specific scenario. (`ProductCode` does not play a role in testing LOB-agnostic and job-agnostic scenarios.)

Adding a new line of business

When you add a new line of business, you must create a new set of LOB/Job impls in the LOB/Job Context API and modify code in other resources to use those impls when appropriate. The following diagram illustrates an example of adding business auto to the renewal job. In the diagram, base configuration resources appear in black. New files and code added to existing files appears in blue.



For detailed information on adding a new line of business to submission, see “Testing submissions on a new line of business” on page 85.

chapter 10

Testing submissions on a new line of business

The process of adding testing support for a new line of business is slightly different for submissions than it is for the other non-submission jobs. This topic discusses the process for submissions.

- This topic assumes you are familiar with the structure of the Line/Job Context API. For more information, see "The Line/Job Context API" on page 77.

Summary of the Line/Job Context API

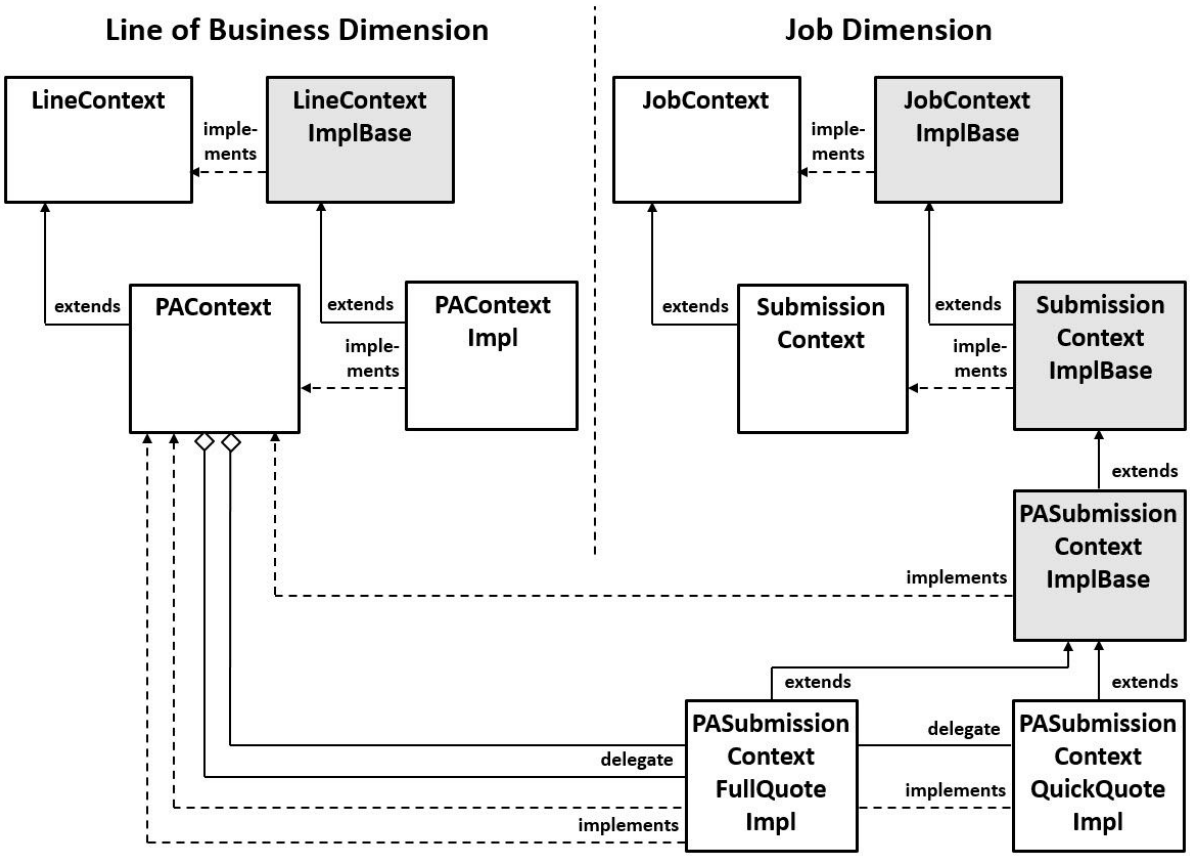
The Line/Job Context API is a context API that provides implementation code for all scenarios that are line-specific, job-specific, or both. It has two dimensions - a line of business dimension and a job dimension.

- The line of business dimension has a generic level and line-specific levels for each supported line of business. Generic implementation code is declared at the generic level. Line-specific implementation code is declared at the appropriate line-specific level.
- The job dimension has a generic level and job-specific levels for each supported job. Generic implementation code is declared at the generic level. Job-specific implementation code is declared at the appropriate job-specific level.

There is an additional set of impls for code that is specific to a single line of business and a single job.

PolicyCenter supports two styles of submission, quick quote and full quote. Therefore, there are typically two impls for logic specific to submissions for a given line of business: one for LOB-specific quick quote logic and one for LOB-specific full quote logic.

The following diagram depicts the structure for personal auto submissions. Gray shading indicates the class is abstract.



For more information on the Job/LOB Context API, see “The Line/Job Context API” on page 77.

Adding a new line of business for submission testing

The following is the high-level procedure for adding a new line of business for the Submission job. It is followed by an example that reiterates each step in a detailed manner.

The code in the examples can be copied and pasted into Studio to provide a working LOB extension for submission. Keep in mind that, in some cases, additional uses statements are required but are not present in the example code. If Studio states it "cannot resolve symbol" for a given type, you can add a uses statement by clicking inside the type name and pressing ALT + ENTER.

Add a line of business for submission

About this task

This procedure is followed by a detailed example with code that you can copy and paste into Studio to form working quick quote and full quote impls for business auto.

Procedure

1. Create new scenarios and feature files that specify new steps.
   - For more information on writing scenarios, see “Writing scenarios” on page 31.
2. If necessary, modify the `ProductCode` enum so that it includes the new line of business.
   - You only need to do this once for each new line of business. If you have already done this for another job type, you do not need to do it again for submissions.

3. If necessary, modify `TagProcessorInfo` so that it has a method to correctly set the product type for scenarios specific to the new line of business.
   - You only need to do this once for each new line of business. If you have already done this for another job type, you do not need to do it again for submissions.
4. Create a new step class that provides definitions for the new step methods.
   - For more information on creating step methods, see “Creating step methods” on page 49.
5. If necessary, create a new LOB-specific interface/impl pair.
   - You only need to do this once for each new line of business. If you have already done this for another job type, you do not need to do it again for submissions.
6. Create a `SubmissionContextImplBase` for the new line of business.
7. Create a new impl for quick quote submissions.
8. Create a new impl for full quote submissions.
9. Bind the new impls to the parent interface.

Step 1: Creating submission scenarios that use new steps

Write new submission scenarios that contain new step methods.

The following text is an example of a new `BA_Submission_Agent_Ext.feature` file. It contains several new step methods, including `"Given a business auto policy"`.

```
@business_auto @submission
Feature: Submission

  As an agent, I want to create a submission for a new policy, quote the submission so that I can give the
policyholder
  a quote for their new policy request.

  Background:
    Given I am a user with the "Producer" role
    And an account of type "Person":
      | First name | Jack  |
      | Last name  | Clark |

  Scenario: Issue a quoted Business Auto submission
    Given a Business Auto submission
    And the job status is "Quoted"
    When I issue the policy
    Then the submission status should be "Bound"
```

For more information on creating scenarios, see “Writing scenarios” on page 31.

Step 2: Modifying ProductCode

If this is the first job type to support the new line of business, you must add an additional member to the `ProductCode` enum. This step does not need to be repeated for additional job types that support the new line of business.

The following text is an example of code that could be added to the `ProductCode` enum to add business auto.

```
...
BUSINESS_AUTO("BusinessAuto"),
```

Step 3: Modifying TagInfoProcessor

If this is the first job type to support the new line of business, you must modify `TagProcessorInfo` so that it can assign the new line of business type to the policy data set that will be used for testing. This step does not need to be repeated for additional job types that support the new line of business.

The following is example of code that could be added to the `TagInfoProcessor` class to add business auto.

```
@Before(:value = {"@business_auto"})
function businessAuto() { setLine(ProductCode.BUSINESS_AUTO) }
```

Step 4: Creating new step methods

Create a new step file. The step file must:

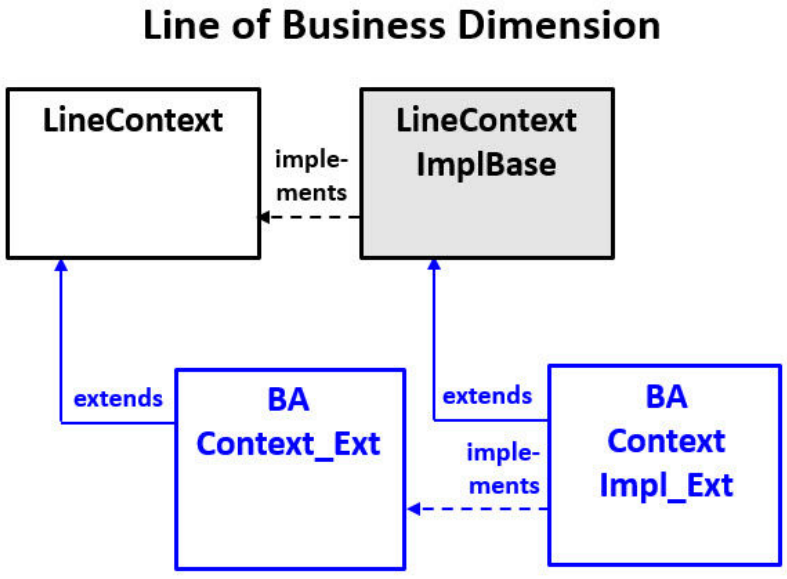
- Provide step definitions for the new step methods.
- Inject the Context Factory

The following text is an example of a new `BA_SubmissionSteps_Ext` class.

```
package gw.cucumber.customer.steps.lob.ba

uses com.google.inject.Inject
uses cucumber.api.java.en.Given
uses gw.cucumber.context.api.ContextFactory
// additional uses statements may be needed

/**
 * Business Auto specific {@link Submission} job steps
 */
class BA_SubmissionSteps_Ext {

  @Inject
  var _contextFactory : ContextFactory

  @Given("^a Business Auto submission$")
  function aBusinessAutoSubmission() {
    _contextFactory.getSubmissionContext().createSubmissionSetup()
  }

}
```

Step 5: Creating an interface/impl pair for the new LOB

If you have not already done so, create an interface and an impl for the new LOB.



The interface for the new LOB

The interface must extend `LineContext`.

The following test is an example of a new `BAContext_Ext` interface. Provide signatures for the methods needed for all tests of this line of business, regardless of the job being tested.

```
package gw.cucumber.customer.context.api.lob.ba

// additional uses statements may be needed

/**
 * Methods common to {@link BusinessAutoLine}.
 * These methods are common to {@link Job} subtypes, for example Submission, PolicyChange, etc.
 */
interface BAContext_Ext extends LineContext {

}
```

The impl for the new LOB

The impl must extend `LineContextImplBase` and implement the new interface.

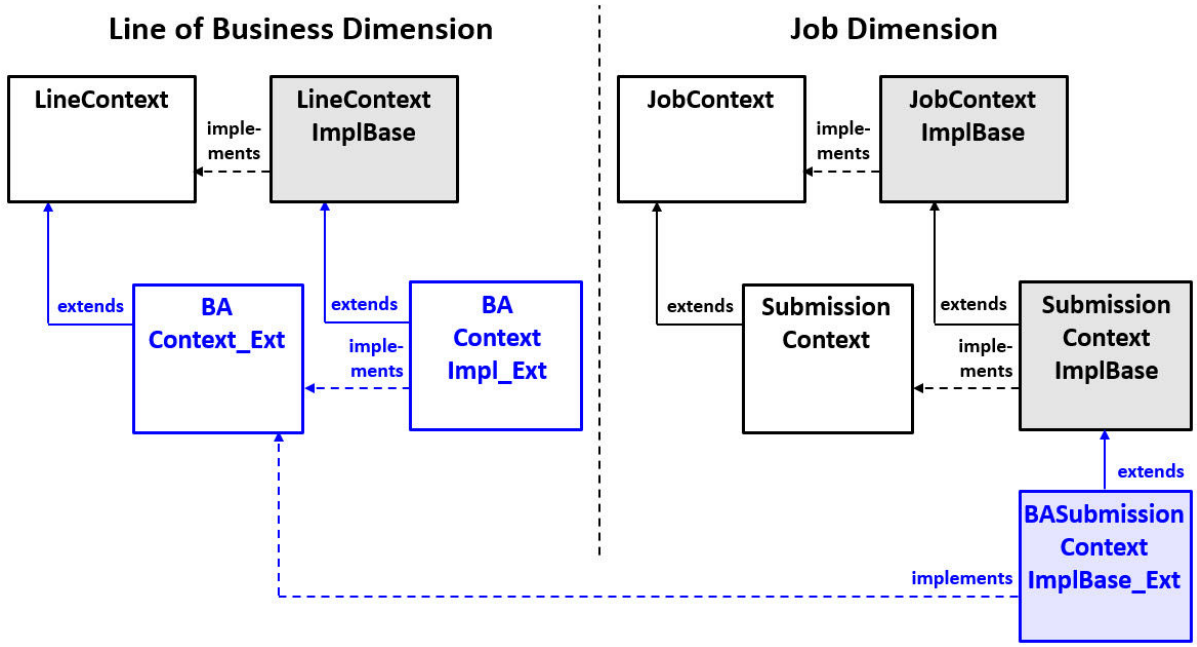
For each method, Guidewire recommends writing initial implementations that simply throw an `UnsupportedOperationException`. When the initial work is complete, you can test the structure by running a test and verifying that this type of exception is thrown. Later, you can replace each implementation with actual code.

The following text is an example of a new `BAContextImpl_Ext` impl.

```
package gw.cucumber.customer.context.impl.smoketest.lob.ba

uses gw.cucumber.context.impl.smoketest.lob.LineContextImplBase
uses gw.cucumber.customer.context.api.lob.ba.BAContext_Ext
// additional uses statements may be needed

class BAContextImpl_Ext extends LineContextImplBase implements BAContext_Ext {

  override function removeLineLevelCoverage(covName : String) {
    throw new UnsupportedOperationException("Not yet implemented")
  }
}
```

Step 6: Creating a SubmissionContextImplBase for the new LOB

The submission job uses a structure with a parent ImplBase class and two child impls - one for quick quotes and one for full quotes. This structure exists to appropriate share and isolate impl logic as needed. Logic that is relevant to both types of submissions is declared in the parent `ContextImplBase` impl. Logic that is specific to either quick quotes only or full quotes only is declared in the relevant child impl.



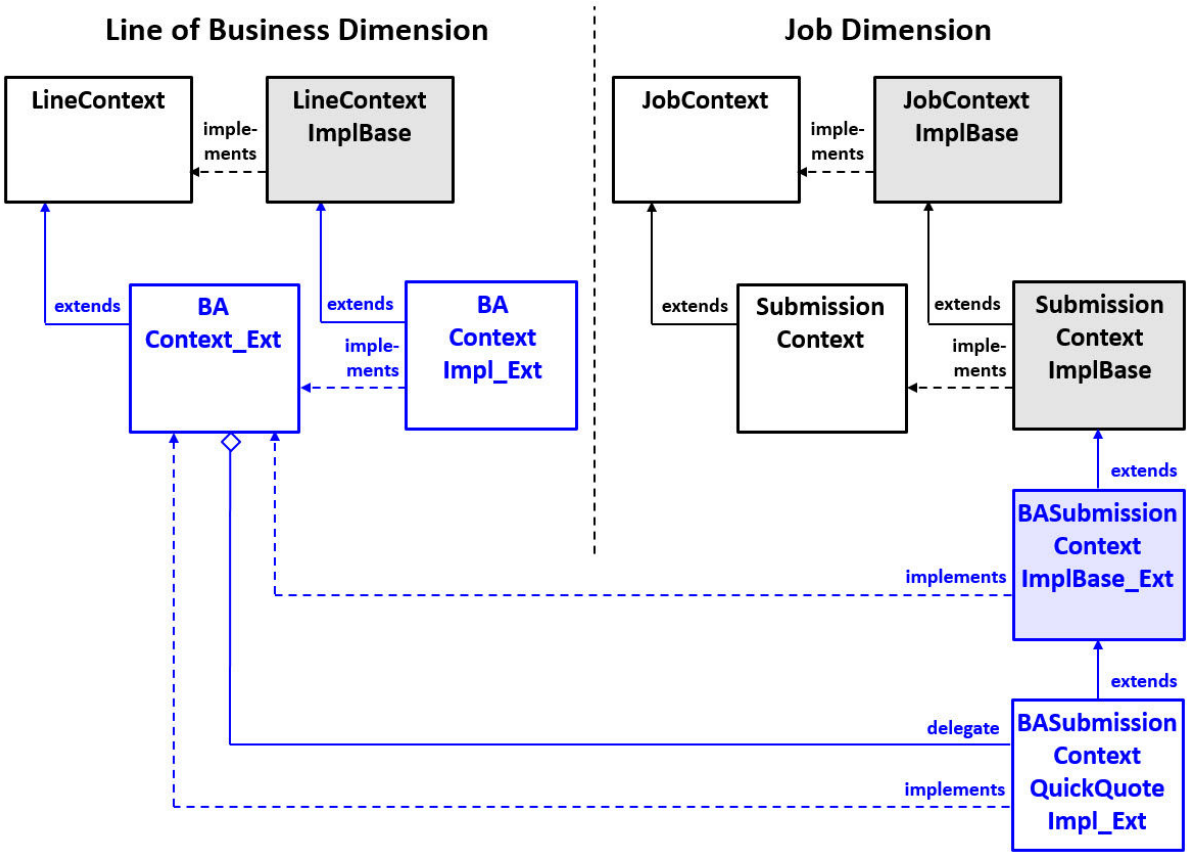
The new SubmissionContextImplBase impl must:

- Be abstract
- Extend `SubmissionContextImplBase`
- Implement the new LOB interface

The following text is an example of a new `BASubmissionContextImplBase_Ext` impl.

```
package gw.cucumber.customer.context.impl.smoketest.lob.ba

uses gw.cucumber.context.impl.smoketest.job.submission.SubmissionContextImplBase
uses gw.cucumber.customer.context.api.lob.ba.BAContext_Ext
// additional uses statements may be needed

abstract class BASubmissionContextImplBase_Ext extends SubmissionContextImplBase implements BAContext_Ext {

  function createSubmissionSetup() {
    throw new UnsupportedOperationException("Not yet implemented")
  }
}
```

Step 7: Creating the new line/job impl for quick quotes



The new impl must:

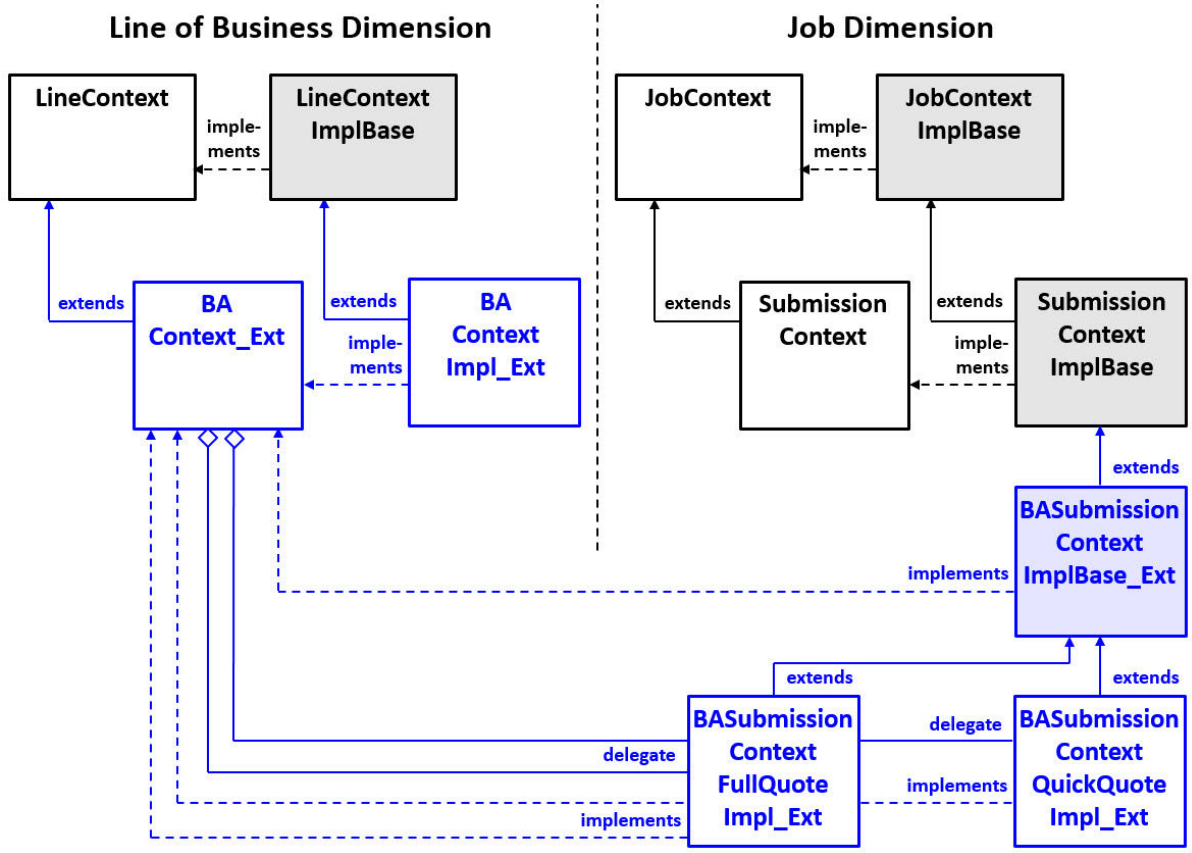
- Extend the SubmissionContextImplBase for the new LOB
- Implement the new LOB interface
- Have a delegate object that represents the new LOB interface. In the base configuration impls, this object is named `_lineContextDelegate`.

The following text is an example of a new `BASubmissionContextQuickQuoteImpl_Ext` impl.

```
package gw.cucumber.customer.context.impl.smoketest.lob.ba

uses com.google.inject.Inject
```

```
uses gw.cucumber.customer.context.api.lob.ba.BAContext_Ext
// additional uses statements may be needed

class BASubmissionContextQuickQuoteImpl_Ext extends BASubmissionContextImplBase_Ext implements BAContext_Ext {

  @Inject delegate _lineContextDelegate represents BAContext_Ext

  protected override function createLOBSpecificDataSetup() {
    throw new UnsupportedOperationException("Not yet implemented")
  }

  override function createABoundPolicy() {
    throw new UnsupportedOperationException("Not yet implemented")
  }

  override function createABoundPolicy(skipIssuance : boolean) {
    throw new UnsupportedOperationException("Not yet implemented")
  }
}
```

Step 8: Creating the new line/job impl for full quotes



The new impl must:

- Extend the SubmissionContextImplBase for the new LOB
- Implement the new LOB interface
- Have a delegate object that represents the new LOB interface. In the base configuration, this object is named `_lineContextDelegate`.

The following text is an example of a new `BASubmissionContextFullQuoteImpl_Ext` impl.

```
package gw.cucumber.customer.context.impl.smoketest.lob.ba

uses com.google.inject.Inject
uses gw.cucumber.customer.context.api.lob.ba.BAContext_Ext
// additional uses statements may be needed
```

```
class BASubmissionContextFullQuoteImpl_Ext extends BASubmissionContextImplBase_Ext implements BAContext_Ext {

  @Inject delegate _lineContextDelegate represents BAContext_Ext

  protected override function createLOBSpecificDataSetup() {
    throw new UnsupportedOperationException("Not yet implemented")
  }

  override function createABoundPolicy() {
    throw new UnsupportedOperationException("Not yet implemented")
  }

  override function createABoundPolicy(skipIssuance : boolean) {
    throw new UnsupportedOperationException("Not yet implemented")
  }
}
```

Step 9: Binding the new impls to the parent interface

Modify `PCRemoteSmokeTestHelperModule` so that it binds the new impls to the parent interface using the relevant product codes. This must be done at the line level as well as at the submission context levels.

The following text contains code that can be added to `PCRemoteSmokeTestHelperModule` to bind the line level impl.

```
// Line specific contexts
...
bind(BAContext_Ext).to(BAContextImpl_Ext)
```

The following text contains code that can be added to `PCRemoteSmokeTestHelperModule` to bind the submission context levels.

```
// Quick Quote Submission Contexts
...
quickQuoteSubmissionContexts.addBinding(ProductCode.BUSINESS_AUTO).to(BASubmissionContextQuickQuoteImpl_Ext)

// Full Quote Submission Contexts
...
fullQuoteSubmissionContexts.addBinding(ProductCode.BUSINESS_AUTO).to(BASubmissionContextFullQuoteImpl_Ext)
```

Running the first test

At this point, you can run the scenarios created in the first step. The scenario will fail because it will invoke a method whose implementation simply throws an `UnsupportedOperationException`. But the test will not result in an error. This ensures that you have the technical structure in place and you can begin to write the actual implementation code.

Summary of best practices

To minimize upgrade conflicts, Guidewire recommends that you avoid making modifications to the base configuration Behavior Testing Framework files. Instead, add your testing extensions to your own files. Exceptions to this recommendation include the following files, which are singleton files and present minimal effort when resolving upgrade conflicts:

- `ContextFactory.gs`
- `ContextFactoryImpl.gs`
- `PCRemoteSmoketestHelperModule.gs`

To ensure that your customizations do not create any upgrade conflicts, Guidewire recommends the following:

- End all file names with an `_Ext` suffix.
- Create context API interfaces in `gw.cucumber.customer.context.api.<approach>`. The name of `<approach>` identifies the technology used in the implementations, such as smoketest or rest.
- Create context API impls in `gw.cucumber.customer.context.impl`.
- For you initial scenario, implement the interface methods so that they throw an `UnsupportedOperationException`. Once you have confirmed that the scenario fails and throw this exception, you can replace impl methods with actual code as needed.

Part 3

Writing implementation code

The implementation code that you write for behavior tests uses the following types of classes:

- Data builders
- Data wrappers
- Transformers

chapter 11

Working with test data

The following topic identifies features used in Behavior Testing Framework to create and manipulate test data.

Test data overview

There are several features used in Behavior Testing Framework to create and manipulate test data. Some of these features are part of Gosu and are not exclusive to Behavior Testing Framework. Other features are specific to Behavior Testing Framework.

General testing functionality

Gosu has several features that are used to create and manipulate test data. The features are used by the Behavior Testing Framework base configuration, but they are not exclusive to Behavior Testing Framework. They are discussed in detail in other areas of the documentation.

Developers who intended to write implementation code for Behavior Testing Framework need to have a good understanding of these features.

Test data builders

A *test data builder* is a Gosu class that can be used to build entity objects for testing purposes. Test data builders simplify the work of creating and committing data for test purposes. For more information on using test data builders, see “Using test data builders” on page 123. For more information on extending or creating test data builders, see “Creating an entity builder” on page 132.

Database transactions and Gosu bundles

A *database transaction* is a set of one or more SQL steps that are processed as a unit. All statements in a transaction must succeed before the transaction can be committed. If any one statement fails, the entire transaction is rolled back.

A *bundle* is a Gosu object that is used to manage a database transaction. Within Gosu, you can manually create bundles, add objects to bundles, and commit bundles. Gosu internally creates the appropriate SQL statements to create and commit database transactions as needed.

For more information on Gosu bundles, refer to the *Gosu Reference Guide* topic on database transactions and bundles.

Testing functionality specific to Behavior Testing Framework

There are also features used to create and manipulate test data that are specific to Behavior Testing Framework.

Developers who intended to write implementation code for Behavior Testing Framework need to have a good understanding of these features as well.

Test data sets

A *test data set* is a set of data that is loaded into the application prior to the execution of any feature file of scenario. A test data set typically provides enough administration and business data to make the application sufficient robust for behavior testing.

For example, suppose that you want to test assignment behavior for an activity. To test this behavior, you may need a certain number of groups and users to ensure that each activity is assigned to an appropriate group and user.

In the base configuration of Behavior Testing Framework, the test data set is the PolicyCenter "small" sample data. Customers can configure Behavior Testing Framework to use a custom test data set as needed.

Data wrappers

A *data wrapper* is an object that stores information needed by multiple step methods or impl methods. Data wrapper objects are injected into various step classes and impl classes. Whenever a data wrapper is injected into a class, the methods of that class can:

- Retrieve information from the data wrapper object, even though the data wrapper object is not explicitly passed to the method as an input parameter.
- Put information into the data wrapper, even though the data wrapper object is not explicitly returned by the method as a return value.

Data wrappers simplify code by providing a mechanism for methods to exchange information within a single scenario without explicitly passing parameter or calling methods on other objects.

For more information, see “Data wrappers” on page 98.

The DataSetup class

The DataSetup classes provide methods that simplify the ability to use a single builder across multiple impl methods. They also provide methods to enhance some of the platform test data builders.

For more information, see “The DataSetup class” on page 100.

Transformers

Behavior Testing Framework provides a set of Transformer classes. Each class has a transform method that simplifies and standardizes the transformation of String values into some other type of value, such as typelist values or monetary values.

For more information, see “Transformers” on page 102.

Data wrappers

Data wrappers overview

Within object-oriented programming, a method is a procedure that exists on an object and that can be called by other objects and processes to execute a particular action. In traditional object-oriented programming, a method is not universally aware of all of the application data. It typically has access only to data stored in its object, data that it gets by calling methods on other objects, or data that is passed to it as an input parameter.

This style of strict information exchange is problematic for Behavior Testing Framework scenarios. A scenario can consist of several steps. Each step maps to a step method which, in turn, calls an impl method in one of the context APIs. Steps are intended to be usable in any combination that makes sense from a business perspective, regardless of which classes contain the impl methods that execute the code. Whenever a given method is called, there is no guarantee which methods were executed before it, and which methods will be executed after it. If Behavior Testing Framework followed a strict data exchange paradigm, every method would need to send and received a large number of values to enable all the possible permutations of methods.

To simplify the exchange of information, Behavior Testing Framework uses a set of injected data wrapper objects. A *data wrapper* is an object that stores information needed by multiple step methods or impl methods. Data wrapper objects are injected into various step classes and impl classes. Whenever a data wrapper is injected into a class, the methods of that class can:

- Retrieve information from the data wrapper object, even though the data wrapper object is not explicitly passed to the method as an input parameter.
- Put information into the data wrapper, even though the data wrapper object is not explicitly returned by the method as a return value.

For example, one of the data wrapper objects is the `_currentUser` object. This object is a data wrapper that wraps a String value. The String value stores the name of the user that will log onto PolicyCenter and interact with the user interface. When the object is injected into a class, every method in the class can access the object. The `_currentUser` object makes it easier to share the name of the active user across every method in any class that needs this information.

In the base configuration, the scope of each data wrapper is a single scenario. When the execution of a given scenario is complete, all data wrappers are discarded. The next scenario will use a new set of data wrappers.

Data wrappers are used extensively in Behavior Testing Framework for BillingCenter and ClaimCenter. They are used minimally in Behavior Testing Framework for PolicyCenter and ContactManager.

The datawrapper class

All data wrapper objects are created as instances of `gw.cucumber.testdata.datawrapper`.

The `datawrapper` class includes the `@ScopedScenario` annotation. This is a Guice annotation that tells Behavior Testing Framework to create exactly one instance of the class when the scenario is started, to make the instance available throughout the running of the scenario, and then to discard the instance when the scenario is complete. This annotation both ensures the object is available throughout the entire scenario and prevents the object from interfering with subsequent scenarios.

Using data wrapper objects

Injecting a data wrapper object

Data wrapper objects must be injected into a class in order to make the object available to the class's methods.

Data wrapper objects are injected into classes using the following syntax:

```
@Inject var objName : DataWrapper<objType>
```

where:

- `objName` is the object name.
- `objType` is the object type.

For example:

```
@Inject
private var _currentUser : DataWrapper<String>
```

Interacting with a data wrapper object

Once a data wrapper object has been injected into a class, all of the methods can interact with it.

For example, the following method comes from the `CucumberStepBase` class:

```
01 protected var _defaultUser : String = "aapplegate"
02
03 property set CurrentUser(username : String) {
04   _currentUser.set(username)
05 }
06
07 property get CurrentUser() : String {
08   if (_currentUser.get() == null) {
09     CurrentUser = _defaultUser
```

```
10   }
11   return _currentUser.get() as String
12 }
```

Line 1 defines a _defaultUser String, which is set to `"aapplegate"`.

Lines 3 to 5 define a setter for the `CurrentUser` property, which simply sets the `_currentUser` object.

Lines 5 through 12 define a getter for the `CurrentUser` property. The intention of this getter is to always provide a non-null value. If `_currentUser` has been set to a non-null value by some other method, this method returns that value. Otherwise, this method sets `_currentObject` to the default user's name (which, based on line 1, is `aapplegate`) and then returns that name.

Data wrapper objects in the base configuration

The base configuration makes use of the following data wrapper objects:

- `_currentUser : DataWrapper<String>`

ContactManager uses:

- `_StringWrapper : DataWrapper<String>`

Behavior Testing Framework also includes two objects that are not implemented using the DataWrapper class, but otherwise function as data wrapping objects. They both contain information that may be relevant to multiple impl methods. They are both injected into CucumberStepBase , and are therefore available to all methods in CucumberStepBase and its child class.

- `_scenarioInfoProvider : ScenarioInfoProvider`
- `_dataTableRepository : DataTableRepository`

The DataSetup class

Scenario-specific test data

When a test data object is created for a specific scenario, the scenario may specify values that the object must have. For example, consider the following scenario:

```
When I create an activity
And the activity priority is "Urgent"
And the activity is due in "7" days
And I assign the activity using automated assignment
Then the activity is assigned to my supervisor
```

An activity must be created specifically for this scenario. The activity's priority must be Urgent, and the activity's due date must be 7 days from the current date.

Scenario-specific test data is typically created using test data builders. For more information on using test data builders, see “Using test data builders” on page 123.

Mapping impl code to modular steps

Suppose the activity in the previous scenario is created using the following code:

```
01 var testActivity = ActivityBuilder()
02     .withPriority(Priority.TC_URGENT)
03     .withDueInDaysFromNow(7)
04     .create(bundleObj)
```

The impl code appears as a single block. However, the impl code may need to spread across several methods. This is because scenario steps are designed to be usable in a variety of combinations. For example, some activity scenarios may specify the activity priority, but others may not. Therefore, each line of code must be assigned to a different method. The following table lists the scenario steps for the previous example and the impl methods that could be created to execute each step.

Scenario step	Impl methods

When I create an activity	`function createActivity() {`
	`  ...`
	`  var testActivity = ActivityBuilder()`
	`  ...`
	`}`

And the activity priority is "Urgent"	`function setActivityPriority() {`
	`  ...`
	`  .withPriority(Priority.TC_URGENT)`
	`  ...`
	`}`

And the activity is due in "7" days	`function setActivityDueDate() {`
	`  ...`
	`  .withDueInDaysFromNow(7)`
	`  ...`
	`}`

	`  ...`
	`  .create(bundleObj)`
	`  ...`

And I assign the activity using automated assignment	`function assignActivityUsingRules() {`
	`  ...`
	`  testActivity.autoAssign()`
	`  ...`
	`}`

Then the activity is assigned to my supervisor	(not relevant to the creation of test data)

The modularity of scenario steps can impose a requirement on impl methods. In some cases, multiple impl methods must share a common builder. Furthermore, the object must not be created from the builder until after all of the set-up values have been specified.

The DataSetup hierarchy

The DataSetup hierarchy consists of a series of classes. There is a parent DataSetup class in gw.cucumber.setup . This has a set of child classes, as shown below:

- `AccountDataSetup`
- `JobDataSetup`
  - `SubmissionDataSetup`
    - `HOPSubmissionDataSetup`
    - `PASubmissionDataSetup`
    - `WCSubmissionDataSetup`
  - `NonSubmissionDataSetup`
    - `CancellationDataSetup`
    - `IssuanceDataSetup`
    - `PolicyChangeDataSetup`
    - `ReinstatementDataSetup`
    - `RenewalDataSetup`
    - `RewriteDataSetup`

These classes provide methods to simplify the sharing of builders across different impl methods. For example, `PASubmissionDataSetup` has the following provideDefaults method:

```
01 override function provideDefaults() : PASubmissionBuilder {
02   provideDefaultDriver()
```

```
03   provideDefaultVehicle()
04   provideDefaultCoverages()
05   return Builder
06 }
```

Line 1 declares the method. Note that there are no input parameters. Also note that it returns a value of type `PASubmissionBuilder`.

Lines 2 through 4 execute methods that provide default objects for the submission.

Line 5 returns the `PASubmissionBuilder`. The builder now has values for a default driver, vehicle, and coverages. But, the actual `PASubmission` object has not yet been created.

Enhancements to the platform builders

Some of the `DataSetup` classes also provide additional methods to enhance some of platform test data builders. For example, `PASubmissionDataSetup` is a `DataSetup` class that enhances `PASubmissionBuilder` by providing the following methods:

- `provideDefaultCoverages()` - Sets specific deductibles to prevent the creation of underwriting issues.
- `provideDefaultDriver()` - If not already provided, provide a default driver.
- `provideDefaultVehicle()` - If not already provided, provide a default vehicle with an existing driver.
- `withDriverAge(age)` - Add a driver having the given age to the builder. A driver under the age of 25 will cause an underwriting rule to execute.
- `withDriver(PersonBuilder)` - Add a driver with the given `PersonBuilder` to the submission.
- `withVehicleGarageState(State)` - Add a vehicle having the given garage state.
- `withVehicle(price, state, PolicyDriverBuilder)` - Add a vehicle with the given price, given state, and given driver to the submission.

Transformers

Behavior Testing Framework provides a set of Transformer classes. Each class has a transform method that simplifies and standardizes the transformation of String values into some other type of value. They are all declared in the `gtest.gw.transformer` package.

Base configuration transformers

The base configuration provides the following classes:

CancelOptionTransformer

Transforms one of the following Strings into a corresponding display key. This is used to simplify code that must specify a specific drop-down menu option in the user interface.

- The method expects the input String to be one of the following:
  - `"Cancel Now"`
  - `"Cancel Policy"`

CloseOptionTransformer

Transforms one of the following Strings into a corresponding display key. This is used to simplify code that must specify a specific drop-down menu option in the user interface.

- The method expects the input String to be one of the following:
  - `"Not-Taken"`
  - `"Decline"`
  - `"Withdraw Transaction"`

ConflictOverideTransformer

Transforms one of the following Strings into a corresponding display key. This is used to simplify code that must specify a specific drop-down menu option in the user interface.

- The method expects the input String to be one of the following:
  - `"Override None"`
  - `"Override All"`

DateTransformer

Transforms a String into a date value.

- You can optionally pass a second parameter, a date value pattern, to the method. If there is no second parameter, the date pattern value defaults to `"MM/dd/yyyy"`.
- The method does not parse the value leniently. In other words, the method uses strict parsing, which requires a data format value to be an exact match to a know data format.

MonetaryAmountTransformer

Transforms a String into a monetary amount. The String is expected to have the format `<amount><space><currency>`. For example, `"100 USD"` or `"50 EUR"`. The transformer returns a monetary amount whose amount is the given numeric value and whose currency is the given currency value.

TypelistTransformer

Transforms a String into a typekey. For example, it can convert the String `"Urgent"` into the typekey `Priority.TC_URGENT`.

- The method returns the first typekey whose display name matches the input String, ignoring case.

Best practices for transformers

If you need to transform a String value and the base configuration provides a corresponding transformer, Guidewire recommends you use the transformer. This ensures that similar values are being transformed in a consistent way.

If you need to transform a String value and the base configuration does not provide a corresponding transformer, you may want to consider creating a transformer using an approach similar to the ones used in the base configuration.

Part 4

Additional topics

As well as testing PolicyCenter, you can use Behavior Testing Framework to test ContactManager. The Behavior Testing Framework in PolicyCenter does not support ContactManager. You install and use a separate instance of Behavior Testing Framework for ContactManager.

chapter 12

Behavior Testing Framework for ContactManager

Installing Behavior Testing Framework for ContactManager

Guidewire provides the Behavior Testing Framework extension pack in a compressed file. You must extract the contents of this compressed file. The following directions refer to the extracted contents as the "extracted directory".

Note: Each release of Behavior Testing Framework supports one or more specific versions of the core application. You must ensure that the version of Behavior Testing Framework is compatible with your version of the core application. To determine the versions of the core application that are compatible with a given version of Behavior Testing Framework, refer to the Behavior Testing Framework release notes.

Install Behavior Testing Framework for ContactManager

Procedure

1. Extract the Behavior Testing Framework distribution file.
   - The distribution file is named `cm-isbtf.zip`.
   - The distribution file is located in the core product's `isbtf` subdirectory.
2. Install the framework files.
   a. In the extracted directory, navigate to the `framework` directory. This directory contains a `modules` directory.
   b. Merge the `modules` directory into the ContactManager directory.
   c. If messages appear that ask whether you want to merge `modules` and any other directories, select the option that merges the directory.
3. Enable code generation of the native page model. This lets Behavior Testing Framework interact with PCF elements and screens.
   a. In the ContactManager directory, open the `gradle.properties` file in a text editor.
   b. Locate the `isPcfTestCodegenEnabled` property and set its value to `true`.
   c. Save and close the file.

4. Add a command for running a Cucumber test suite to the `build.gradle` file.
   a. In the ContactManager directory, navigate to the `modules/configuration` directory.
   b. Open both the `build.gradle` file and the `build.gradle_isbtf` file in a text editor.
   c. In the `build.gradle_isbtf` file, locate the line that contains the following text:

      ```
      /** ISBTF: ONLY MERGE THE LINE BELOW */
      ```

   d. Copy the line between the `/** ISBTF: ONLY MERGE THE LINE BELOW */` and `/** ISBTF: ONLY MERGE THE LINE ABOVE */` lines.
   e. Paste the copied line at the end of the `build.gradle` file.
   f. Save and close the `build.gradle` file.
   g. Close the `build.gradle_isbtf` file.
5. Some files in the PolicyCenter refer to generated PCFs. If the generated files do not exist, the application will not start. To ensure the files exist:
   a. Open Guidewire Studio.
   b. Click **Codegen→Generate Page Configuration Classes**. When the class generation is complete, Studio shows a "Page configuration codegen done..." message in the lower left corner.

Install the behavior testing example

Procedure

1. In the extracted directory that you created in “Install Behavior Testing Framework for ContactManager” on page 107, navigate to the `example` directory. This directory contains a `modules` directory.
2. Merge the `modules` directory to the ContactManager directory.
3. If messages appear that ask whether you want to merge the `modules` and any other directories, select the option that merges the directory.
4. If other messages that ask whether you want to replace existing files appear, select the options that replace the existing files.

Verify the Behavior Testing Framework for ContactManager installation

About this task

Check that you installed the framework correctly. You can do this by: (1) verifying that the appropriate directories and files have been added, (2) running a base configuration feature file, and (3) running a base configuration suite.

The scenarios provided in Behavior Testing Framework will pass when executed against a stand-alone base configuration instance of ContactManager. If the instance has been integrated with other Guidewire applications or configured, some tests may not pass.

Procedure

1. Open Guidewire Studio.
2. Optionally verify that the following directories and files exist:
   - In `configuration/gsrc/gw/api/test`, an `ABSmokeTestClassBase`.
   - A `configuration/gsrc/gw/smoketest.pl` directory.
   - A `configuration/gtest/gw/cucumber` directory.
   - A `configuration/gtest/gw/enhancement` directory.
   - In `configuration/gtest/gw/suites`, an `ABExampleSmokeSuite` file.
   - A `configuration/gtest/gw/smoketest` directory.
   - In `configuration/res/cucumber`, a `DuplicateContactsBatchExample.feature` file.

3. Test a base configuration feature file.

   The feature file assumes that ContactManager has not been configured. When run on a base configuration of ContactManager, all scenarios in the feature file will pass. If ContactManager has been configured, one or more scenarios may fail.

   a. Navigate to `configuration/res/cucumber`.

   b. Right-click the `DuplicateContactsBatchExample.feature` file, and then click **Run 'Feature: DuplicateContactsBatchExample'**.

   Studio opens a **Run Feature <FeatureName>** tab. The left pane contains icons that identify the tests that pass or fail. The right pane contains a console with the server output during testing.

4. Test a base configuration suite.

   A suite is a collection of feature files that are executed together. The suite assumes that ContactManager has not been configured. When run on a base configuration of ContactManager, all scenarios in the suite will pass. If ContactManager has been configured, one or more scenarios may fail.

   There is a known issue with Behavior Testing Framework that prevents a test suite from running if there is a running test server. As a workaround, if there is a running test server, stop it before executing the following steps.

   a. Navigate to `configuration/gtest/gw/suites`.

   b. Right-click the `ABExampleCucumberSuite` file, and then click **Run 'ABExampleCucumberSuite'**.

   Studio opens a **Run <SuiteName>** tab. The left pane contains icons that identify the tests that pass or fail. The right pane contains a console with the server output during testing.

Resolving errors that occur during verification

Errors that occur when executing a test suite

The following table lists errors that you might see when executing a suite, as well as possible causes and recommend fixes.

Error message	Possible cause	Suggested fix
Studio throws an `ArrayIndexOutOfBounds` exception	The test server might already be running.	Stop the test server before executing the file.

Part 5

Platform-level testing functionality

Platform-level testing functionality includes the GUnit test framework and classes that are available to that framework and to the Behavior Testing Framework. These classes support building data that the testing classes use. Other Gosu classes and interfaces support testing PCF files.

chapter 13

GUnit Test Framework

PolicyCenter provides many integration points for you to write Gosu configuration code as new classes, extensions, interface implementations, expressions in PCF files, and so on. To ensure that your Gosu configuration code performs the functionality that you require, you can write a set of test classes to exercise that code.

The GUnit Test Framework provides an environment that supports writing and running the tests for your Gosu configuration code.

Tests are written in Gosu. You can execute these tests from the command prompt to include the tests in an automated continuous integration work flow. You can also execute these tests from Guidewire Studio.

Creating a framework test

The GUnit Test Framework provides the following base classes to support different types of tests.

`PCServerTestClassBase`
: Base class for tests that require a running server

`PCUnitTestClassBase`
: Base class for tests that do not need the services that by a running server provides

A test class extends one of the base classes. The test class name must end with the suffix `Test`, such as `MySampleTest`. The base class implementation of the framework's `setUp` method tests for this condition and throws an `IllegalStateException` if this test fails.

Store all test class files in the Studio `modules/configuration/gtest` directory hierarchy. You can create new subdirectories in the hierarchy to organize test classes for your convenience.

Optionally, the `@Suites` annotation can be specified on the test class. The `@Suites` annotation accepts a unique `String` argument. All the test classes in a particular suite must specify the same `@Suites` annotation. An associated suite class also references the suite's `String` value.

```
@Suites("UniqueSuiteName")
class MyTest extends PCUnitTestClassBase {
  ...
}
```

The following sample source classes demonstrate the use of the `@Suites` annotation.

- The `MySuiteNames` class defines a string constant called `SAMPLE_SUITE_NAME`.

```
// ========== File: MySuiteNames.gs ==========
// Define suite name strings.
// Test and suite classes reference the same string to group the tests into the suite.
// A test class references the string in a @Suites annotation.

package doc.example
```

```
class MySuiteNames {

  public static final var SAMPLE_SUITE_NAME : String = "AnyUniqueString_1"
  // ... Define other suite-name strings here
}
```

- The `MyTest` class definition uses a `@Suites` annotation that specifies the `SAMPLE_SUITE_NAME` constant. If other test classes exist in the suite, each class must specify the same `@Suites` annotation.

```
// ========== File: MyTest.gs ==========
// Implement test class

package doc.example

uses gw.testharness.v3.Suites
uses gw.api.test.PCUnitTestClassBase

@Suites(MySuiteNames.SAMPLE_SUITE_NAME)
class MyTest extends PCUnitTestClassBase {
  ...
}
```

- The `MySuiteClass` class is a suite class that references `SAMPLE_SUITE_NAME`.

```
// ========== File: MySuiteClass.gs ==========
// Implement suite class

package doc.example

class MySuiteClass {
  // References MySuiteNames.SAMPLE_SUITE_NAME.
  // All test classes with a @Suites annotation that references SAMPLE_SUITE_NAME are included in the suite.
  // Details of implementing the suite class are described in another topic.

  static function suite() : Test {
    return new SuiteBuilder(PCUnitTestClassBase)
          .withSuiteName(MySuiteNames.SAMPLE_SUITE_NAME)
          .build()
  }
}
```

The `@Suites` annotation is optional. Similarly, a suite class does not need to reference a unique suite name. A suite class that does not reference a suite name includes all the test classes in the package that do not specify a `@Suites` annotation. This catch-all style of suite grouping can be useful early in the development cycle when only a few tests exist and logical criteria for grouping them into suites are not yet evident.

Each test class provides the following Gosu constructors.

```
construct()
construct(name : String)
```

The name of a framework test method must begin with the string `test` as in `testAddNumbers`. When running the individual tests in a test class, the framework executes each method that begins with the string `test`.

The following code illustrates the minimal skeletal structure of a test class that extends the `PCServerTestClassBase` class.

```
class MyTest extends PCServerTestClassBase {

  construct(testName : String) {
    super(testName)
    ...
  }

  function testBasicSystemCheck() {
    ...
  }

  function testAnotherServerTest() {
    ...
  }

  // Other class methods
```

```
  ...
}
```

Core functions of the test framework base classes

The base classes of the GUnit Test Framework provide core functions like `setUp`, `beforeMethod`, `afterMethod`, and `tearDown` that define the basic skeletal structure of a test. The core functions are called by the framework as it processes the individual tests in a test class and test suite.

This section presents the core functions in the order in which the framework executes them when running a series of tests in a test class and test suite. The following pseudocode illustrates the execution order of the core functions.

```
for each test class in a test suite {
  construct()
  beforeClass()

  for each test in a test class {
    setUp()
    beforeMethod()

    // Actual test method, such as testCheckSystem()

    afterMethod()
    tearDown()
  }

  afterClass()
}
```

Constructors

```
construct()
construct(name : String)
```

The `name` property is optional and not used by the framework. A test or test suite can use the name to identify a particular test object. If the constructor does not initialize the name, you can set its value by calling the object's `setName` method.

Method: beforeClass

```
function beforeClass() : void
```

The `beforeClass` method is executed once, before running any of the test methods that the test class defines. In a test suite with multiple test classes, each test class's `beforeClass` method is executed immediately before running the tests in the class.

Do not create test instance variables in the `beforeClass` method because the variables would exist only for the first test that runs. Instead, create instance variables in the `beforeMethod` method.

```
override function beforeClass() {
  super.beforeClass()
  ...
}
```

Method: setUp

```
final function setUp() : void
```

The `setUp` method is executed before running each test method that the test class defines. The method typically configures the test context and creates any data objects that the test requires.

A test class can overload the `setUp` method, but cannot override it.

In the base class implementation, if the test object's class name does not end with the string `Test`, an `IllegalStateException` is thrown.

```
function setUp(someArg : int) {
  super.setUp()
```

```
  ...
}
```

Method: beforeMethod

```
function beforeMethod() : void
```

The `beforeMethod` method is executed before running each test method that the test class defines.

```
override function beforeMethod() {
  super.beforeMethod()
  ...
}
```

Method: afterMethod

```
function afterMethod(possibleException : Throwable) : void
```

The `afterMethod` method is called after each test method completes its execution.

The base class implementation clears all instance variables of the completed test.

```
override function afterMethod(possibleException : Throwable) {
  super.afterMethod(possibleException)
  ...
}
```

Method: tearDown

```
final function tearDown() : void
```

The `tearDown` method is called after each test method completes its execution. The method typically performs clean-up operations.

A test class can overload the `tearDown` method, but cannot override it.

```
function tearDown(someArg : int) {
  super.tearDown()
  ...
}
```

Method: afterClass

```
function afterClass() : void
```

The `afterClass` method is called after the completion of all tests in a test class.

```
override function afterClass() {
  super.afterClass()
  ...
}
```

Support functions for the test framework

The GUnit Test Framework provides several categories of support functions that tests can call to implement a test operation.

Environment configuration methods for the test framework

The framework provides methods to configure the testing environment and return information about it.

Method: assert...

The framework provides static methods that assert whether a particular condition exists. Each method's name begins with the prefix `assert` followed by the condition tested, as in `assertEquals` and `assertNotZero`.

The framework supports all the methods defined by the `Assert` class in the Java JUnit unit testing framework. The `Assert` class provides commonly used methods like `assertEquals`, `assertTrue`, and `assertFalse`. For a complete list of supported methods, refer to the JUnit `Assert` class documentation.

In addition, the framework extends the `Assert` class methods with methods applicable to testing InsuranceSuite configuration code. Example methods include `assertBigDecimalEquals` and `assertDateEquals`. The framework assert methods are defined in the `PLAssertions` class, which is included in the `gw.testharness.v3` package.

The following table groups the methods in the `PLAssertions` class into general categories. For complete details, refer to the `PLAssertions` class documentation.

Category	Methods
BigDecimal	`assertBigDecimalEquals`, `assertBigDecimalNotEquals`, `assertBigDecimalIsZero`, `assertBigDecimalIsNotZero`
Collection	`assertEmpty`, `assertSize`, `assertCollectionContains`, `assertCollectionDoesNotContain`, `CollectionEquals`, `CollectionSame`
Equals	`assertEquals`, `assertEqualsIgnoreCase`, `assertEqualsIgnoreLineEnding`, `assertEqualsIgnoreWhiteSpace`, `assertEqualsReplaceAll`, `assertEqualsUnordered`, `assertComparesEqual`
General purpose	`assertDateEquals`, `assertFalseFor`, `assertTrueWithin`, `assertGreaterThan`, `assertZero`, `assertNotZero`
Hashtable	`assertHashtableContains`, `assertHashtableContainsKey`
Helper	`assertCollection`, `assertList`, `assertSet`, `assertThat`
Iterator	`assertIteratorEquals`, `assertIteratorSame`
List	`assertListEquals`, `assertListSame`, `findObjectInListUsingComparator`
Miscellaneous	`assertAssignable`, `assertExceptionThrown`, `assertExceptionThrownWithMessage`
Object	`assertArrayContains`, `assertArrayDoesNotContain`, `assertLength`

Method: registerPlugin

```
registerPlugin<T extends InternalPlugin>(pluginInterface : Class<T>, implementation : T) : void
```

Call the `registerPlugin` method to register a replacement plugin implementation to use during the class's testing operations. The method is available only in test classes that extend `PCServerTestClassBase`.

The `pluginInterface` argument specifies the plugin to be replaced. The `implementation` argument references the plugin implementation instance to use during the testing operations.

The original plugin implementation is restored at the completion of the class's tests by the base class implementation of the `afterClass` method.

```
class MySampleTest extends PCServerTestClassBase {

  // ... Constructors, other methods, and so on.

  // *** Replacement plugin implementation for testing operations
  class MyReplacementPlugin implements IGenericPlugin {

    // ... Constructors, overrides of plugin methods, and so on.
  }

  // *** Function to test my configuration code
  // The replaced plugin is subsequently restored automatically by the afterClass() method
  function testMyConfigCode() {

    registerPlugin(IGenericPlugin, new MyReplacementPlugin())    // Register replacement plugin for testing operations

    // ... Perform tests here.
  }
```

```
  // *** Another test function
  // At the function's completion, restore the original plugin; don't wait until afterClass()
  function testMyConfigCode02() {
    var savedPlugin : IGenericPlugin

    savedPlugin = Plugins.get(IGenericPlugin.class)              // Save original plugin implementation
    registerPlugin(IGenericPlugin, new MyReplacementPlugin())    // Register replacement plugin

    // ... Perform tests here.

    registerPlugin(IGenericPlugin, savedPlugin)                  // Restore original plugin
  }
}
```

Method: setUpMutableSystemClock

```
setUpMutableSystemClock() : void
```

The `setUpMutableSystemClock` method establishes a temporary system clock available for testing purposes. Test code can adjust the time of the temporary clock without affecting the actual system clock. The method is available only in test classes that extend `PCServerTestClassBase`.

Call the `setUpMutableSystemClock` method in the test class's `beforeClass` method. The temporary clock is initialized to the current system day and time. The original system clock is automatically restored by the base class implementation's `afterClass` method.

The `ChangesCurrentTimeUtil` class provides static methods to set and advance the temporary clock. In general, time is advanced to the future. Reversing time to the past can result in unexpected application behavior and is strongly discouraged.

```
setCurrentTime(test : TestCase, timeInMillis : long) : void
incCurrentTime(test : TestCase, deltaInMillis : long) : void
```

The `setCurrentTime` method sets the temporary system clock to a specified time. The `incCurrentTime` method increments the clock by a specified number of milliseconds.

The `test` argument references the test class. The `timeInMillis` argument specifies the time in milliseconds to assign to the clock. The `deltaInMillis` argument specifies the number of milliseconds to advance the clock.

```
// Set the temporary system clock to a future date
ChangesCurrentTimeUtil.setCurrentTime(this, mySampleCalendar.getSampleEventDateTime())

// Advance the clock by one day
var ONE_DAY_IN_MILLISECONDS = 1000L*60L*60L*24L
ChangesCurrentTimeUtil.incCurrentTime(this, ONE_DAY_IN_MILLISECONDS)
```

Property: TestResultsDir

```
TestResultsDir : File
```

The `TestResultsDir` property is a read-only property. This `java.io.File` object references the directory that stores the test results.

Logging functions for the test framework

The framework provides functions that support logging. The logging functions are based on the `org.slf4j.Logger` object. Refer to the SLF4J (Simple Logging Facade for Java) documentation for details.

Method: addLoggingAppender

```
addLoggingAppender(logger : Logger, appender : LogAppender) : void
```

The `addLoggingAppender` method adds a logging output destination to a `Logger` object.

The `logger` argument references the `Logger` object to receive the new output destination. The `appender` argument specifies the new logging output destination. The `LogAppender` object is based on Log4j. Refer to the Log4J documentation for details on using an appender to define a new output destination.

Method: getLogger

```
getLogger() : Logger
```

The `getLogger` method returns the `org.slf4j.Logger` object used by the test framework for logging.

Method: setLoggerLevel

```
setLoggerLevel(logger : Logger, level : LogLevel) : void
```

The `setLoggerLevel` method sets the types of messages to log.

The `logger` argument specifies the `Logger` object used by the test framework. The object can be retrieved by the `getLogger` method.

The `level` argument specifies the log level. The following hierarchical levels are supported. Each level includes the levels below it. For example, the `INFO` level will log messages of the types `INFO`, `WARN`, and `ERROR`.

- `ALL` - Log all message types.
- `TRACE` - Verbose logging mode.
- `DEBUG` - Log debugging messages.
- `INFO` - Log informational messages about executing operations. INFO is the default log level.
- `WARN` - Log potential problems.
- `ERROR` - Log error conditions.
- `OFF` - Disable logging.

```
uses gw.logging.LogLevel

setLoggerLevel(getLogger(), LogLevel.DEBUG)
```

A test can log a message by calling the `Logger` method related to the relevant log level. Refer to the `Logger` documentation of the SLF4J documentation for details.

```
class MySampleServerTest extends PCServerTestClassBase {

  function testMySampleServerTest() {
    Logger.info("Running sample server test")       // Log a message at the INFO level

    // ... Perform test

    // ... Log messages at other levels
    Logger.debug("Debug information")               // Log a message at the DEBUG level
    Logger.error("Error information")               // Log a message at the ERROR level
    Logger.trace("Details about the test")          // Log a message at the TRACE level
    Logger.warn("Potential issue in the test")      // Log a message at the WARN level
  }
}
```

Method: startLogCapture

```
startLogCapture(logger : Logger) : CapturingLogger
```

The `startLogCapture` method creates a new logging object that captures subsequent logging events.

The method returns a `CapturingLogger` object.

```
uses gw.testharness.CapturingLogger

class MySampleServerTest extends PCServerTestClassBase {

  function testMySampleServerTest() {
    var logger : CapturingLogger

    try {
      logger = startLogCapture(getLogger())         // Start capturing the test logger

      assertThat(logger.getCapturedEvents())        // Verify that log is initially empty
            .as("Log should be initially empty")
            .isEmpty()
```

```
      // ... Perform test here

      assertThat(logger.getCapturedEvents())        // Verify that log is no longer empty
            .as("Should have logged messages")
            .isNotEmpty()
  }
}
```

General-purpose methods for the test framework

The framework provides several general-purpose methods.

Method: getName

```
getName() : String
```

The `getName` method retrieves the value of the object's `name` property.

The `name` property can be assigned in the object's constructor or by calling the `setName` method.

Method: getUniqueSuffixForTest

```
getUniqueSuffixForTest() : String
```

The `getUniqueSuffixForTest` method generates a globally unique string. Possible uses of the string include avoidance of duplicate keys and as a suffix to the test object's `name` property.

```
mySampleTest01.setName("MySampleTest" + mySampleTest01.getUniqueSuffixForTest())
mySampleTest02.setName("MySampleTest" + mySampleTest02.getUniqueSuffixForTest())
```

Method: setName

```
setName(name : String) : void
```

The `setName` method assigns the value of the `name` argument to the object's `name` property.

Method: toString

```
toString() : String
```

The `toString` method returns a string representation of the test object.

In the base class implementation, the object's string representation is constructed by concatenating the following values.

- The object's name property
- The object's package name
- The object's class name

The format of the concatenated string is `name(package.class)`.

Creating a framework test suite

Multiple test classes can be combined to form a single test suite. Each test class in a suite must be based on the same class type. For example, `MyServerTestSuite` might consist of the test classes `MyServerTestOne`, `MyServerTestTwo`, and `MyServerTestThree`, where each class is based on `PCServerTestClassBase`.

A test suite is created by defining a suite class. The suite class must define a static method called `suite`. The `suite` method creates the test suite by using the `SuiteBuilder` class.

The `SuiteBuilder` class defines a constructor and the following methods.

`withSuiteName`
: Accepts a `String` suite name argument. The generated suite will include all the test classes defined with a @Suites annotation value that matches the suite name argument. To create a suite that includes all the test

classes that were defined without a `@Suites` annotation, do not call the `withSuiteName` method when building the `SuiteBuilder` instance.

`build`
Accepts no arguments. Creates the test suite object.

The following suite class builds a suite that includes each test class that specified a `@Suites` annotation with the `SAMPLE_SUITE_NAME` argument. Also, each test class in the suite extends the `PCServerTestClassBase` class.

```
package doc.example

uses gw.suites.MySuiteNames
uses gw.api.test.PCUnitTestClassBase
uses gw.api.test.SuiteBuilder
uses junit.framework.Test

class MySuiteClass {

  static function suite() : Test {
    return new SuiteBuilder(PCUnitTestClassBase)
          .withSuiteName(MySuiteNames.SAMPLE_SUITE_NAME)
          .build()
  }
}
```

The framework calls the `suite` method during normal framework processing. The method returns a `JUnit`-based `Test` object. The `Test` object is managed by the framework and does not need to be manipulated by test code.

Running a test suite at the command prompt

To run a test suite, execute the following operations at the command prompt.

```
gwb compile
gwb runSuite -Dsuite=TestSuiteClassName {-Doption...}
```

The `gwb compile` command compiles the GUnit test and test suite classes.

The `gwb runSuite` command runs the tests in a specified test suite. The command accepts the following options. Each option is prefixed by `-D`.

Option	Description
`suite`	Required. Fully-qualified suite class name, such as `com.acme.MyTestSuite`. Example: `-Dsuite=com.acme.ux.tests.UxTestSuite`
`dir.results`	Optional. Directory in which to store results. If a relative directory is specified, the location is relative to the application's root directory. The results file contents are in a format that can be processed by continuous integration systems. Default: The `/tmp` directory. Example: `-Ddir.results=build/test-results`
`dir.temp`	Optional. Directory in which to store temporary test files. If a relative directory is specified, the location is relative to the application's root directory. Default: The `/tmp` directory. Example: `-Ddir.temp=build/test-temp`
`memory`	Optional. Maximum memory in MB to allocate to run tests. Default: 768 Example: `-Dmemory=2048`

chapter 14

Using test data builders

Test data builders are Gosu classes that simplify the creation of test data objects. Test data builders are useful for both Behavior Testing Framework scenarios and GUnit tests.

Overview of test data builders

Tests that verify application functionality often require objects to test against. For example, suppose you want to write a test that verifies the logic for assigning an activity to a user in the correct group. In order to test this logic, you must have at least one activity, one user, and one group. Furthermore, the state of these objects must be appropriate for the test. For example, to test activity assignment, you might need to start with an activity that is unassigned.

Theoretically, you could execute these tests using the approach taken to create objects in a production environment. Within a test framework, this approach tends to be cumbersome for several reasons:

- Objects often have fields that are required at the database level. In a production environment, values must be specified for these fields. However, within a test environment, many of these values may be irrelevant to a given test. Setting these values increases the code length and can make the code harder to read.
- Objects may have values that are cumbersome to set, such as:
  - A given object may require a parent object.
  - A given field may need to have a value that is unique
- Objects in a production environment are tied to automatic application behaviors, such as pre-update rules or messaging events. These behaviors can be irrelevant to testing and may throw errors for reasons irrelevant to testing.

To simplify object creation for testing purposes, Guidewire provides test data builder. A *test data builder* is a class that can be used to build objects for testing purposes. Test data builders provide the following benefits:

- All required fields have associated default values. When you create an object from a test data builder, you only need to specify values for the fields relevant to the test. The class provides any required values not specified by your code.
- Helper methods exist to simply the setting of more cumbersome values, such as:
  - Required associated objects
  - Fields whose values must be unique
- Objects created from test data builders do not trigger automatic application behaviors, such as pre-update rules.

For example, the following code uses a test data builder to create a test user object and then print values from that object. Note that the FirstName value is specified in the code, but the LastName field is not. When the test user object is created, the FirstName field is set as specified, and the LastName field is given a default value.

```
var testUser = new UserContactBuilder()
    .withFirstName("Tom")
    .create()
print(testUser.FirstName)
print(testUser.LastName)
```

OUTPUT:

```
Tom
House 0006
```

Test data builder classes are also referred to as *builders*, *data builders*, and *entity builders*.

The base configuration provides test data builders for most frequently-used entities. They are declared in the `gw.api.databuilder` package. If necessary, insurers can extend these test data builders. They can create their own test data builders. For more information, see "Creating and extending test data builders" on page 131.

Test data builders focus only on fields required by the database

Test data builders are designed to create objects that have enough data to be saved to the database. There may be circumstances where this is insufficient to meet the needs of a given test.

For example, suppose there is a test that requires an `Account` object to be shown in a specific user interface screen. The screen has the **Email** field marked as required, even though this field is not required in the database. If you create an `Account` object using a test data builder and do not provide an `Email` value, an exception may be thrown when the object is tested in the user interface.

Similarly, suppose there is a test for `Note` objects that requires the **Topic** field to have a value. This field is not required at the database level and is normally set by preupdate rules. Test data builders do not trigger preupdate rules. Therefore, if you create a **Note** object using a test data builder and do not manually provide a value for the `Topic` field , the test will fail.

When writing code that uses test data builders, keep in mind that each use case may require data beyond what is strictly needed to save the object to the database. In these cases, you must write code the provides values for these additional requirements.

Test data builders are not supported in production environments

Test data builders are available only in non-production environments. Objects creates from test data builder classes bypass many application constraints and are therefore not appropriate for a production environment.

> **Note:** Test data builder classes are intended for use in testing environments only. Guidewire does not support their use in a production environments.

To enable test data builders, you must set the Enable Internal Debug Tools configuration parameter in `config.xml` to true.

Creating objects using test data builders

The runWithNewBundle method

Within InsuranceSuite, a *bundle* is a set of objects that are committed to the database as a group. Bundles are analogous to database transactions. Every object is created within the context of a bundle. The object is saved to the database when the bundle is committed.

One of the most common methods used to interact with bundle is the `gw.transaction.Transaction` class's `runWithNewTransaction`. This method has a signature that specifies a Gosu block and a specific user. The Gosu block names a bundle and typically executes code that creates objects within that bundle. At the end of the block, the bundle is inherently committed to the database. The commit is executed as the specified user.

The syntax for this usage of `runWithNewTransaction` is:

```
gw.transaction.Transaction.runWithNewBundle( \ bundleObject -> {
      // code that creates objects in bundleObj
} , "userName" )
```

For more information on `runWithNewTransaction`, refer to the section on bundles and database transactions in the *Gosu Reference Guide*.

Creating test objects in runWithNewBundle

Code that creates test objects typically appears within the block of a `runWithNewBundle` call. The code typically has the following structure:

- One line that declares a variable for the new object and calls the appropriate builder class
- Optionally, one or more lines that set values for the new object
- One line that creates the object

The following line of code demonstrates this three-part structure:

```
gw.transaction.Transaction.runWithNewBundle( \ bundleObj -> {
    var testUser = new UserContactBuilder()
        .withFirstName("Tom")
        .create(bundleObj)
} , "su" )
```

- The first line calls `runWithNewBundle` and names the bundle `bundleObj`.
- The second line declares a new object whose name is `testUser` and whose type is `UserContactBuilder`.
- The third line sets the `FirstName` field on the new object.
- The fourth line creates the object in the `bundleObj` bundle.
- The final line ends the block, which inherently commits the bundle as the super user `su`.

The following sections provide more detail on how to declare the test object variable, set its values, and create the object using those values.

Declaring the test object variable

The base configuration test data builder classes are declared in the `gw.api.databuilder` package. You can refer to this package for a complete list of builders.

To declare a new test object variable from a test data builder, use the following syntax:

```
var testObj = new dataBuilderClass()
```

The `gw.api.databuilder` package must be either imported using a `uses` statement or included as part of the builder class name. Both approaches are demonstrated in the following blocks of code:

```
uses gw.api.databuilder
var testUser = new UserContactBuilder()
...
var testUser = new gw.api.databuilder.UserContactBuilder()
```

Setting object values

Objects created from test data builders do not need to have any field values set. Every test data builder class provides default values for all required fields, and they create any related and required objects as needed.

For example, the following line of code creates a test user, even though no values have been provided for the test user:

```
var testUser = new UserContactBuilder()
    .create(bundleObj)
```

Specifying field values

You can specify values using methods within the test data builder class. These methods typically start with one of the following words:

- `with`*`FieldName`* - These methods let you specify a value for a given field.
  - For example, `addressBuilder.withCountry(Country.TC_FR)`
- `as`*`Value`* - These methods let you specify that the object has a given state. They can simplify code by not requiring you to specify the correct field or value to set the state.
  - For example, `addressBuilder.asBusinessAddress()`
- `on`*`ForeignKeyName`* - These methods let you specify a parent object that is the owner of the new object.
  - For example, `policyBuilder.onAccount(testAccount)`

When writing code that sets field values, the standard convention is to list each method on a separate, indented line that starts with the dot and method name. This makes it easier to use the code completion feature within Studio, improves the code's readability, and makes it easier to add additional lines.

The following code is an example of a test object that sets multiple field values and follows the standard convention for code format.

```
var testBusinessAddress = new AddressBuilder()
    .withAddressLine1("123 Main Street")
    .withCity("San Francisco")
    .withState(State.TC_CA)
    .withPostalCode("94110")
    .withCountry(Country.TC_US)
    .asBusinessAddress()
    .create(bundleObj)
```

Specifying unique values

Some fields require a field value that is unique before the object can be committed to the database. For example, person contacts have a `TaxID` field. This field must be a unique string value in the form XXX-XX-XXXX, where X is a digit 0 through 9 and - is a literal hyphen (such as "123-45-6789").

The `gw.api.databuilder.UniqueKeyGenerator` class can be used to generate unique values. It has its own sequence of integers and a get method. The get method has several child methods that return the next integer in the sequence in various formats. The get method never uses the same integer twice. So, within the context of testing, any two values generated from this method ought to be unique:

- `nextID` - returns a 13-character alphanumeric string that consists of 6 random letters, a hyphen, and the next integer in the sequence as a 6-digit string with padded zeroes as needed.
- `nextInteger` - returns the next integer in the sequence
- `nextKey` - returns an 8-character alphanumeric string whose final characters are the next integer in the sequence

The following code is an example of executing each of these methods multiple times. Before this code was executed, the integers up through 36 have already been used.

```
uses gw.api.databuilder.UniqueKeyGenerator

print ("Next ID:  " + UniqueKeyGenerator.get().nextID())
print ("Next ID:  " + UniqueKeyGenerator.get().nextID())
print ("Next int: " + UniqueKeyGenerator.get().nextInteger())
print ("Next int: " + UniqueKeyGenerator.get().nextInteger())
print ("Next key: " + UniqueKeyGenerator.get().nextKey())
print ("Next key: " + UniqueKeyGenerator.get().nextKey())
```

OUTPUT:

```
Next ID:  QGJAJX-000037
Next ID:  FUGZFL-000038
Next int: 39
Next int: 40
Next key: WBFMHN41
Next key: MVJCVB42
```

The following code is an example of creating a user with a tax ID in the format XXX-XX-XXXX. The code appends the last four digits of the next ID to the string "000-00-". To do this, it uses the `substring` method, starting with the 9th character (where 0 is the first character and 9 is the first of the last four characters).

```
gw.transaction.Transaction.runWithNewBundle( \ bundleObj -> {
    var testUser = new UserContactBuilder()
        .withFirstName("Tom")
        .withLastName("Thompson")
        .withTaxID("000-00-" + UniqueKeyGenerator.get().nextID().
            substring(9))
        .create(bundleObj)
    print(testUser.FirstName)
    print(testUser.LastName)
    print(testUser.TaxID)
} , "su" )
```

OUTPUT:

```
Tom
Thompson
000-00-0043
```

Fields that cannot be set using methods

For every builder class, there may be fields whose values cannot be set using a `with`*X*, `as`*X*, or `on`*X* method. This includes:

- Base configuration fields for which there is no related `with`*X*, `as`*X*, or `on`*X* method
- Fields added through configuration

You can extend a base configuration test data builder to add methods to it so that a given field's value can be set. For more information, see “Creating and extending test data builders” on page 131.

Alternately, you can set the value after the object has been created. For example, suppose you extended the UserContact entity by adding a `RemoteEmployee_Ext` Boolean field. There is no method in the base configuration to set this field value. The following code shows an example of setting the field after the object has been created.

```
gw.transaction.Transaction.runWithNewBundle( \ bundleObj -> {
      var testUser = new UserContactBuilder()
        .withFirstName("Tom")
        .create(bundleObj)
    testUser.RemoteEmployee_Ext = true
} , "su" )
```

Creating the object

Once you have specified the field values that are important to the test, you must create the object. Typically, this is done using the test data builder's `create` method. For example, in the following code, the fourth line creates the object.

```
gw.transaction.Transaction.runWithNewBundle( \ bundleObj -> {
    var testUser = new UserContactBuilder()
        .withFirstName("Tom")
        .create(bundleObj)
} , "su" )
```

Which bundle is the object created in

The create method has multiple signatures. One of them is:

```
create(bundleObj)
```

This signature creates the object in the specified bundle. For test objects, Guidewire recommends using this signature, as it gives you the ability to create the object in the bundle managed by `runWithNewBundle`. This bundle is inherently committed at the end of the block.

There is another signature:

```
create()
```

This signature creates the object in the default bundle. The *default bundle* is a bundle that typically exists by default. When objects are created without a specified bundle, they are created in the default bundle.

For test objects, Guidewire does not recommend using this signature. The default bundle is not inherently committed at the end of the `runWithNewBundle` block. If you create objects in the default bundle, you must also manually commit that bundle. Also, if other processes have added objects to the default bundle, committing the bundle can lean to unexpected behaviors.

Creating multiple test objects from a template object

In some situations, a test may require a set of objects that have nearly identical settings. For example, there could be a test that requires a set of activities that have the same type and priority.

You can create multiple test objects with the same values using a template object. A *template object* is an object created from a test data builder class whose `create` method is never called. Instead, multiple objects are declared and created from it.

To use a template object:

1. Declare the template object.
2. On the template object, set the values that need to be identical on each of the test objects.
   - Do not execute the `create` method on the template object.
3. Create the test objects from the template object.
   - Execute the `create` method on the test objects only.

The following code is an example of creating multiple activities from a template object. The type and priority fields are set in the template object. Therefore, both activities have the same type and priority. However, each activity has a distinct subject.

```
gw.transaction.Transaction.runWithNewBundle( \ bundleObj -> {
    var templateActivityBuilder = new gw.api.databuilder.ActivityBuilder()
        .withType(ActivityType.TC_GENERAL)
        .withPriority(Priority.TC_HIGH)

    var activity1 = templateActivityBuilder
                    .withSubject("Test activity 1")
                    .create(bundleObj)
    var activity2 = templateActivityBuilder
                    .withSubject("Test activity 2")
                    .create(bundleObj)
} , "su" )
```

Creating related objects

For some tests, you may need an object that is related to a given object, such as an `Address` that is related to a `UserContact`. There are two different ways the related object can be created and associated with the primary object.

Creating related objects using nested builders

Some test data builders have `withX` methods that take a nested builder as a parameter. For example, on the `UserContactBuilder` class, the `withAddress` builder method has a signature that takes an `AddressBuilder`. This is demonstrated in the following code example.

```
gw.transaction.Transaction.runWithNewBundle( \ bundleObj -> {
    var testUser = new UserContactBuilder()
        .withFirstName("Tom")
        .withLastName("Thompson")
        .withPrimaryAddress(new AddressBuilder()
            .withAddressLine1("123 Main Street")
            .withCity("San Francisco")
            .withState(State.TC_CA)
            .withPostalCode("94110")
            .withCountry(Country.TC_US)
            .asBusinessAddress()
            .create(bundleObj))
        .create(bundleObj)
} , "su" )
```

Note that the primary address is specified using a nested `AddressBuilder`. To make this clear, the `AddressBuilder` code uses a second level of indentation.

Creating related objects manually

Some builders have `withX` methods that take an object as a parameter. For example, on the `UserContactBuilder` class, the `withAddress` builder method takes has a signature that takes an `Address` object. This is demonstrated in the following code example.

```
gw.transaction.Transaction.runWithNewBundle( \ bundleObj -> {
  var testBusinessAddress = new AddressBuilder()
      .withAddressLine1("123 Main Street")
      .withCity("San Francisco")
      .withState(State.TC_CA)
      .withPostalCode("94110")
      .withCountry(Country.TC_US)
      .asBusinessAddress()
      .create(bundleObj)
  var testUser = new UserContactBuilder()
      .withFirstName("Tom")
      .withLastName("Thompson")
      .withPrimaryAddress(testBusinessAddress)
      .create(bundleObj)
} , "su" )
```

chapter 15

Creating and extending test data builders

Note: Guidewire does not recommend or support the use of classes that extend `gw.api.databuilder.DataBuilder` or classes that reside in the `gw.api.databuilder.*` package in a production environment. Guidewire provides GUnit as a development test facility only.

As you run tests against code, you need to run these test in the context of a known set of data objects. This set of objects is generally known as a *test fixture*. You use Gosu entity builders to create the set of data objects to use in testing.

Guidewire provides a number of entity "builders" as utility classes to quickly and concisely create objects (entities) to use as test data. The PolicyCenter base configuration provides builders for the base entities (such as `PolicyBuilder`, for example). However, if desired, you can extend the base `DataBuilder` class to create new or extended entities. You can commit any test data that you create using builders to the test database using the `bundle.commit` method.

For example, the following builder creates a new `Person` object with a `FirstName` property set to "Sean" and a `LastName` property set to "Daniels". It also adds the new object to the default test bundle.

```
var myPerson = new PersonBuilder()
  .withFirstName("Sean")
  .withLastName("Daniels")
  .create()
```

For readability, Guidewire recommends that you place each configuration method call on an indented separate line starting with the dot. This makes code completion easier. It also makes it simpler to alter a line or paste a new line into the middle of the chain or to comment out a line.

Gosu builders extend from the base class `gw.api.databuilder.DataBuilder`. To view a list of valid builder types in Guidewire PolicyCenter, use the Studio code completion feature. Type `gw.api.databuilder.` in the Gosu editor and Studio displays the list of available builders.

Package completion

As you create an entity builder, you must either use the full package path, or add a `uses` statement at the beginning of the test file. However, in general, Guidewire recommends that you place the package path in a `uses` statement at the beginning of the file.

```
uses gw.api.databuilder.AccountBuilder

@gw.testharness.ServerTest
class MyTest extends TestBase {
```

```
  construct(testname : String) {
    super(testname)
  }
  ...
  function testSomething() {
    //perform some test
    var account = new AccountBuilder().create()
  }
  ...
}
```

Guidewire provides certain of the `Builder` classes in `gw.api.builder.*` and others in `gw.api.databuilder`. Verify the package path as you create new builders.

Creating an entity builder

To create a new entity builder of a particular type, use the following syntax:

```
new TypeOfBuilder()
```

This creates a new builder of the specified type, with the Builder class setting various default properties on the builder entity. Each entity builder provides different default property values depending on its particular implementation. For example, to create (or build) a default address, use the following:

```
var address = new AddressBuilder()
```

To set specific properties to specific values, use the property configuration methods. The following are the types of property configuration methods, each which serves a different purpose as indicated by the method's initial word:

Initial word	Indicates
on	A link to a parent. For example, `PolicyPeriod` is on an `Account`, so the method is `onAccount(Account account)`.
as	A property that holds only a single state. For example, `asSmallBusiness` or `asAgencyBill`.
with	The single element or property to be set. For example, the following sets a `FirstName` property: `withFirstName("Joe")`

Use a `DataBuilder.with(...)` configuration method to add a single property or value to a builder object. For example, the following Gosu code creates a new `Address` object and uses a number of `with(...)` methods to initialize properties on the new object. It then uses an `asType(...)` method to set the address type.

```
var address = new AddressBuilder()
  .withAddressLine1( streetNumber + " Main St." )
  .withAddressLine2( "Suite " + suiteNumber)
  .withCity( "San Mateo" )
  .withState( "CA" )
  .withPostalCode( postalCode )
  .asType(typeStr)
  ...
```

After you create a builder entity, you are responsible for writing that entity to the database as part of a transaction bundle. In most cases, you must use one of the builder `create` methods to add the entity to a bundle. Which `create` method one you choose depends on your purpose.

To complete the previous example, add a `create` method at the end.

```
var address = new AddressBuilder()
  .withAddressLine1( streetNumber + " Main St." )
  ...
  .create()
```

Builder create methods

The `DataBuilder` class provides the following `create` methods:

```
builderObject.create( bundle )
builderObject.create()
builderObject.createAndCommit()
```

The following list describes these `create` methods.

Method	Description
`create()`	Creates an instance of this builder's entity type in the default bundle. This method does not commit the bundle. Studio resets the default bundle before every test class and method.
`createAndCommit()`	Creates an instance of this builder's entity type in the default bundle, and performs a commit of that default bundle.
`create(bundle)`	Creates an instance of this builder's entity type, with values determined by prior calls to the entity. The *bundle* parameter sets the bundle to use while creating this builder instance.

The no-argument create method

The no-argument `create` method uses a default bundle that all the builders share. This is adequate for most test purposes. However, as all objects created this way share the same bundle, committing the bundle on just one of the created objects commits all of the objects to the database. This also makes them available to the PolicyCenter interface portion of a test. For example:

```
var address = new AddressBuilder()
  .withCity( "Springfield" )
  .asHomeAddress()
  .create()

 new PersonBuilder()
  .withFirstName("Sean")
  .withLastName("Daniels")
  .withPrimaryAddress(address)
  .create()

 address.Bundle.commit()
```

In this example, `Address` and `Person` share a bundle, so committing `address.Bundle` also stores `Person` in the database. If you do not need a reference to the `Person`, then you do not need to store it in a variable.

GUnit resets the default bundle before every test class and method.

The create and commit method

The `createAndCommit` method is similar to the `create` method. However, after adding the entity to the default bundle, this method commits that bundle to the database.

The create with bundle method

If you need to work with a specific bundle, use the `create(bundle)` method. Guidewire recommends that you use this method inside of a transaction block. A transaction block provides the following:

- It creates the bundle at the same time as it creates the new builder.
- It automatically commits the bundle as it exits.

The following example illustrates the use of a data builder inside a transaction block.

```
uses gw.transaction.Transaction

 function myTest() {
  var person : Person

  Transaction.runWithNewBundle( \ bundle -> {
    person = new PersonBuilder()
            .withFirstName( "John" )
            .withLastName( "Doe" )
            .withPrimaryAddress( new AddressBuilder()
                    .withCity( "Springfield" )
                    .asHomeAddress() )
            .create( bundle )
```

```
  } )

  assertEquals( "Doe", person.LastName )

}
```

Notice the following about this example:

- The example declares the `person` variable outside the transaction block, making it accessible elsewhere in the method.
- The data builder uses an `AddressBuilder` object nested inside `PersonBuilder` to build the address.
- The `Transaction.runWithNewBundle` statement creates the bundle and automatically commits it after Gosu Runtime executes the supplied code block.

In summary, the `create(`*`bundle`*`)` method does not create a bundle. Rather, it uses the bundle passed into it. Guidewire recommends that you use this method inside a transaction block that both creates the bundle and commits it automatically.

If you do not use this method inside a transaction block that automatically commits a bundle, then you must commit the bundle yourself. To do so, add `bundle.commit` to your code.

Creating new builders

If you need additional builder functionality than that provided by the PolicyCenter base configuration builders, you can do either of the following:

- Extend an existing builder class and add new builder methods to that class.
- Extend the base `DataBuilder` class and create a new builder class with its own set of builder methods.

You can also create a builder for a custom entity that you created by extending the `DataBuilder` class.

Extending an existing builder class

To extend an existing builder class, use the following syntax:

```
class MyExtendedBuilder extends SomeExistingBuilder {
  construct() {
    ...
  }
  ...
  function someNewFunction() : MyExtendedBuilder {
    ...
    return this
  }
  ...
}
```

The following `MyPersonBuilder` class extends the existing `PersonBuilder` class. The existing `PersonBuilder` class contains methods to set a home phone number and mobile phone number. The new extended class contains a method to set the person's alternative phone number. As there is no static field for the properties on a type, you must look up the property by name. Note that you must first define the new property in the data model.

```
uses gw.api.databuilder.PersonBuilder

class MyPersonBuilder extends PersonBuilder {

  construct() {
    super( true )
  }

function withAltPhone( testname : String ) : MyPersonBuilder {
    set(Person#AltPhone, testname)
    return this
  }

 }
```

The `PersonBuilder` class has two constructors. This code sample uses the one that takes a Boolean that means create this class `withDefaultOfficialID`.

Another more slightly complex example would be if you extended the `Person` object and added a new `PreferredName` property. In this case, you might want to extend the `PersonBuilder` class also and add a `withPreferredName` method to populate that field through a builder.

Extending the DataBuilder class

To extend the `DatabBuilder` class, use the following syntax:

```
class MyNewBuilder extends DataBuilder<BuilderEntity, BuilderType> {
   ...
}
```

The `DataBuilder` class takes the following parameters:

Parameter	Description
`BuilderEntity`	Type of entity created by the builder. The `create` method requires this parameter so that it can return a strongly-typed value and, so that other builder methods can declare strongly-typed parameters.
`BuilderType`	Type of the builder itself. The `with` methods require this on the `DataBuilder` class so that it can return a strongly-typed builder value (to facilitate the chaining of `with` methods).

If you choose to extend the `DataBuilder` class (`gw.api.databuilder.DataBuilder`), place your newly created builder class in the `gw.api.databuilder` package in the Studio **Tests** folder. Start any method that you define in your new builder with one of the following recommended words:

Initial word	Indicates
on	A link to a parent. For example, `PolicyPeriod` is on an `Account`, so the method is `onAccount(Account account)`.
as	A property that holds only a single state. For example, `asSmallBusiness` or `asAgencyBill`.
with	The single element or property to be set. For example, the following sets a `FirstName` property: `withFirstName("Joe")`

Your configuration methods can set properties by calling `DataBuilder.set` and `DataBuilder.addArrayElement`. You can provide property values as any of the following:

- Simple values.
- Beans to be used as subobjects.
- Other builders, which PolicyCenter uses to create subobjects if it calls your builder's `create` method.
- Instances of `gw.api.databuilder.ValueGenerator`. For example, an instance that generates a different value to satisfy uniqueness constraints for each instance constructed.

`DataBuilder.set` and `DataBuilder.addArrayElement` optionally accept an integer order argument that determines how PolicyCenter configures that property on the target object. (PolicyCenter processes properties in ascending order.) If you do not provide an order for a property, Studio uses `DataBuilder.DEFAULT_ORDER` as the order for that property. PolicyCenter processes properties with the same order value (for example, all those that do not have an order) in the order in which they are set on the builder.

In most cases, Guidewire recommends that you omit the order value as you are implement builder configuration methods. This enables callers of your builder to select the execution order through the order of the configuration method calls.

Constructors for builders can call `set`, and similar methods to set up default values. These are useful to satisfy `null` constraints so it is possible to commit built objects to the database. However, Guidewire generally recommends that you limit the number of defaults. This is so that you have the maximum control over the target object.

Other DataBuilder classes

The `gw.api.databuilder` package also includes `gw.api.databuilder.ValueGenerator`. You can use this class to generate a different value for each instance constructed to satisfy uniqueness constraints. The `databuilder` package includes `ValueGenerator` class variants for generating unique integers, strings, and typekeys.

- `gw.api.databuilder.SequentialIntegerGenerator`
- `gw.api.databuilder.SequentialStringGenerator`
- `gw.api.databuilder.SequentialTypeKeyGenerator`

Custom builder populators

Ideally, all building can be done through simple property setters, using the `DataBuilder.set` or `DataBuilder.addArrayElements` methods. However, you may want to define more complex logic, if these methods do not suffice. To achieve this, you can define a custom implementation of `gw.api.databuilder.populator.BeanPopulator` and pass it to `DataBuilder.addPopulator`. Guidewire provides an abstract implementation, `AbstractBeanPopulator`, to support short anonymous `BeanPopulator` objects.

The following example uses an anonymous subclass of `AbstractBeanPopulator` to call the `withCustomSetting` method. This code passes the group to the constructor, and the code inside of `execute` only accesses it through the `vals` argument. This allows the super-class to handle packaging details.

```
public MyEntityBuilder withCustomSetting(group : Group) {

  addPopulator(new AbstractBeanPopulator<MyEntity>(group) {

    function execute(e : MyEntity, vals : Object[]) {
      e.customGroupSet(vals[0] as Group)
    }
  })
  return this
}
```

The `AbstractBeanPopulator` class automatically converts builders to beans. That is, if you pass a builder to the constructor of `AbstractBeanPopulator`, it returns the bean that it builds in the `execute` method. The following example illustrates this.

```
public MyEntityBuilder withCustomSetting(groupBuilder : DataBuilder<Group, ?>) : MyEntityBuilder {

  addPopulator(new AbstractBeanPopulator<MyEntity>(groupBuilder) {

     function execute(e : MyEntity, vals : Object[]) {
      e.customGroupSet(vals[0] as Group)
     }
  })
  return this
}
```

Create and test a builder for a custom entity

About this task

It is possible to create a builder for a custom entity. For example, suppose that you want each PolicyCenter user to have an array of external credentials for automatic sign-on to linked external systems. To implement this requirement, you can create an array of `ExtCredential` on `User`, with each `ExtCredential` having the following parameters.

Parameter	Type
`ExtSystem`	Typekey
`UserName`	String
`Password`	String

After creating your custom entity and its builder class, you need to test it. To accomplish this, you need to do the following:

Task	Affected files
1. Create a custom `ExtCredential` array entity and extend the `User` entity to include it.	`ExtCredential.eti` `User.etx`
2. Create an `ExtCrendentialBuilder` by extending the `DataBuilder` class and adding `withXXX` methods to it.	`ExtCredentialBuilder.gs`
3. Create a test class to exercise and test your new builder.	`ExtCredentialBuilderTest.gs`

To create a new array `ExtCredential` custom entity, do the following tasks, as shown in the procedure.

- Add the `ExtSystem` typelist by using the **Typelist** editor in Guidewire Studio.
- Define the `ExtCredential` array entity in the `ExtCredential`, by using the editor in Guidewire Studio.
- Modify the array entity definition to include a foreign key to `User` in `ExtCredential`.
- Add an array field to the `User` entity in `User.etx`.

Procedure

1. Add the `ExtSystem` typelist.
   a. In Guidewire Studio, navigate to **configuration→config→Extensions→Typelist**.
   b. Right-click **Typelist**, and then click **New→Typelist**.
   c. In **Name**, type `ExtSystem`. Then, click **OK**.
   d. Add a few *external system* typecodes.
      Add `SystemOne`, `SystemTwo`, or similar items.
2. Create the `ExtCredential` entity type.
   a. In Guidewire Studio, navigate to **configuration→config→Extensions→Entity**.
   b. Right-click **Entity**, and then click **New→Entity**.
   c. In **Entity**, type `ExtCredential`. Then, click **OK**.
   d. Set the `exportable` attribute to `true`.
   e. Set the `platform` attribute to `true`.
   f. Add a typekey with the following attributes:
      name
      `ExtSystem`
      typelist
      `ExtSystem`
      nullok
      `false`
      desc
      `Type of external system`
   g. Add a column with the following attributes:
      name
      `UserName`
      type
      `shorttext`
      nullok
      `false`

h. Add a column with the following attributes:

name

`Password`

type

`shorttext`

nullok

`false`

i. Add a `foreignkey` with the following attributes:

name

`UserID`

fkentity

`User`

nullok

`false`

desc

`FK back to User`

3. Modify the `User` entity.
   a. In Guidewire Studio, in **configuration→config→Extensions→Entity**, double-click `User.etx`.
   b. Add an array with the following attributes:

name

`ExtCredentialRetirable`

type

`ExtCredential`

desc

`An array of ExtCredential objects`

arrayfield

`UserID`

exportable

`false`

4. Create an `ExtCredentialBuilder` class that extends the base `DataBuilder` class. Place this class in its own package under **configuration** and in the `gsrc` folder.
   a. In Guidewire Studio, navigate to **configuration→config→gsrc**.
   b. Right-click **gsrc**, and then click **New→Package**.
   c. In **Enter new package name**, type a name for the new package. Then, click **OK**.
   Type `AllMyClasses`.
   d. Right-click **AllMyClasses**, and then click **New→Gosu Class**.
   e. In **Name**, type `ExtCredentialBuilder`. Then, click **OK**.
   f. In the editor, enter the following code:

```
package AllMyClasses

uses gw.api.databuilder.DataBuilder

class ExtCredentialBuilder extends DataBuilder<ExtCredential, ExtCredentialBuilder>  {

  construct() {
    super(ExtCredential)
  }

  function withType (  type: typekey.ExtSystem ) : ExtCredentialBuilder {
```

```
    set(ExtCredential.TypeInfo.getProperty( "ExtSystem" ), type)
    return this
  }

  function withUserName( somename : String ) : ExtCredentialBuilder {
    set(ExtCredential.TypeInfo.getProperty( "UserName" ), somename)
    return this
  }

  function withPassword( password : String ) : ExtCredentialBuilder {
     set(ExtCredential.TypeInfo.getProperty( "Password" ), password)
     return this
  }
}
```

Notice the following about this code sample.

- It includes a `uses ... DataBuilder` statement.
- It extends the `Databuilder` class, setting the *`BuilderType`* parameter to `ExtCredential` and the *`BuilderEntity`* parameter to `ExtCredentialBuilder`.
- It uses a constructor for the super class—`DataBuilder`—that requires the entity type to create.
- It implements multiple `withXXX` methods that populate an `ExtCredential` array object with the passed in values.

5. Create an `ExtCredentialBuilderTest` class that is a GUnit test that uses the `ExtCredentialBuilder` class to create test data. Place this class in its own package in the **Tests** folder.

```
package MyTests

uses AllMyClasses.ExtCredentialBuilder
uses gw.transaction.Transaction

@gw.testharness.ServerTest
class ExtCredentialBuilderTest extends gw.testharness.TestBase {

  static var credential : ExtCredential
  construct() {
  }

  function beforeClass () {
    Transaction.runWithNewBundle( \ bundle -> {
      credential = new ExtCredentialBuilder()
              .withType(  "SystemOne" )
              .withUserName( "Peter Rabbit" )
              .withPassword( "carrots" )
              .create( bundle )
      }
    )
  }

  function testUsername() {
    assertEquals("User names do not match.", credential.UserName, "Peter Rabbit")
  }

  function testPassword() {
    assertEquals("Passwords do not match.", credential.Password, "carrots")
  }
}
```

Notice the following about this code sample:

- It includes the `uses` statements for both `ExtCredentialBuilder` and `gw.transaction.Transaction`.
- It creates a static `credential` variable. As the code declares this variable outside of a method—as a class variable—it is available to all methods within the class. (GUnit maintains a single copy of this variable.) As you run a test, GUnit creates a single instance of the test class that each test method uses. Therefore, to preserve a variable value across multiple test methods, you must declare it as a static variable.
- It uses a `beforeClass` method to create the `ExtCredential` test data. This method calls `ExtCredentialBuilder` as part of a transaction block, which creates and commits the bundle automatically. GUnit calls the `beforeClass` method before it instantiates the test class for the first time. Thereafter, the test class uses the test data created by the `beforeClass` method. It is important to understand that GUnit does not drop the database between execution of each test method within a test class. However, if you run

multiple test classes together (for example, by running all the test classes in a package), GUnit resets the database between execution of each test class.

- It defines several test methods, each of which starts with `test`, with each method including an `assertXXX` method to test the data.

Result

If you run the `ExtCredentialBuilderTest` class as defined, the GUnit tester displays green icons, indicating that the tests were successful.

chapter 16

Gosu PCF tests

This topic details information about PCF testing with Gosu. These can be used as part of GUnit tests or in conjunction with Behavior Testing Framework.

Support methods for user interface tests

The framework provides methods to test a user interface defined by PCF (Page Configuration Format) files. The methods are available in test classes that extend `pcSmokeTestClassBase`.

Note: The PolicyCenter use of the term smoke test is specific to this framework. A framework smoke test verifies the user interface, such as the existence and operation of elements on a PCF page or a smoke test verifies the general reliability of a build.

Method: dismissAndGetAlert

```
dismissAndGetAlert() : String
```

The `dismissAndGetAlert` method closes an alert dialog. The method returns the alert message.

```
class MyTestClass extends pcSmokeTestClassBase {

  function testMySampleTest() {
    ...
    // Click the Page.Screen.Edit button
    myScreenPage.MyScreen.MyEditButton.click()
    dismissAndGetAlert()
    ...
  }
}
```

Method: dismissAndGetConfirmation

```
dismissAndGetConfirmation() : String
```

The `dismissAndGetConfirmation` method closes a confirmation dialog. The method returns the confirmation message.

```
class MyTestClass extends pcSmokeTestClassBase {

  function testMySampleTest() {
    ...
    // Click the Page.Screen.Close button. Not expecting a confirmation dialog.
    myScreenPage.MyScreen.MyCloseButton.click()
    if(isConfirmationShowing()) {
      PLAssertions.fail("Unable to close screen because:\n" + dismissAndGetConfirmation())
```

```
    }
    ...
  }
}
```

Method: getCurrentPage

```
getCurrentPage() : Object
```

The `getCurrentPage` method retrieves the Gosu object that represents the current PCF page.

For every PCF file, the framework provides a corresponding class that includes a strongly typed property for each of the page's elements. Retrieving the page object of a PCF file enables test code to access the object's properties to evaluate and modify the page's fields, buttons, and other items, including sub-objects, like detail views.

```
// Retrieve the page object for the current PCF page
var page = getCurrentPage()

// Retrieve the detail view sub-object on this page
var detailView = page.MyDetailScreen.MyDetailView

// The detail view has an input with an ID value of "Code." Set its value.
detailView.Code.Value = 'X'

// The page has an Update button. Click it.
page.Update.click()

// Verify the detail view's input value still contains the modified value.
if (detailView.Code.Value != 'X') {
    // ... Handle the issue
}
```

Method: getCurrentWorksheet

```
getCurrentWorksheet() : Object
```

The `getCurrentWorksheet` method retrieves the Gosu object that represents the current worksheet. If there is no current worksheet, the method returns `null`.

A PCF worksheet is similar to a PCF page. The framework provides a corresponding class for each worksheet. The class includes a strongly-typed property for each of the worksheet's elements. Retrieving the worksheet object enables test code to access the object's properties to evaluate and modify the worksheet's fields.

```
// Click the page's Show button to show a worksheet.
page.Show.click()
assertThat(getCurrentWorksheet()).isNotNull()

// Retrieve the worksheet and verify its expected contents exist
var myWorksheet = getCurrentWorksheet() as pcftest.MyAwesomeWorksheet
assertThat(myWorksheet.itemsLV._Entries.Count).isEqualTo(10)
assertThat(myWorksheet.itemsLV._Entries[0].Name.Text).isEqualTo("Awesome")

// Click worksheet item's Delete button and verify the expected behavior
myWorksheet.itemsLV._Entries[0].Delete.click()
assertThat(getCurrentWorksheet().itemsLV._Entries.Count).isEqualTo(9)
```

Method: goToEntryPoint

```
goToEntryPoint(entryPoint : String) : Object
goToEntryPoint(entryPoint : String, parameters : Map<String, String>) : Object
```

The `goToEntryPoint` method transfers the user interface focus to a specified page. Multiple method signatures are provided.

The `entryPoint` argument references the name of the page to transfer to. The `parameters` argument is a list of key-value pairs that are passed as arguments to the loaded page.

The method returns the page object of the page transferred to.

```
class MyTestClass extends PCSmokeTestClassBase {
  // Any point in the application can be accessed from the tab bar
```

```
  var _tabBar : pcftest.TabBar as TabBar

  // ... Constructors and other overridden methods

  /**
   * Log in a user with the given name and password. Initialize the TabBar property.
   */
  function login(username : String, password : String) {
    var startPage = (this.goToEntryPoint("Login") as pcftest.Login).login(username, password)
    _tabBar = (startPage as pcftest.DesktopActivities)._parent.TabBar
  }
}
```

Method: isConfirmationShowing

```
isConfirmationShowing() : boolean
```

The `isConfirmationShowing` method determines whether a confirmation dialog is showing.

The method returns `true` if a confirmation dialog is showing. Otherwise, returns `false`.

For example code, see the `dismissAndGetConfirmation` method.

Method: refreshPage

```
refreshPage() : Object
```

The `refreshPage` method posts the current page to the server. The operation is equivalent to refreshing the browser window.

The method returns the page object for the refreshed page.