Testing

Table of Contents

Chapter 1. Testing

If you are going to use Apache Isis for developing complex business-critical applications, then being able to write automated tests for those applications becomes massively important. As such Apache Isis treats the topic of testing very seriously. (Though we say it ourselves), the framework has support that goes way above what is provided by other application frameworks.

This guide describes those features available to you for testing your Apache Isis application.

1.1. Other Guides

Apache Isis documentation is broken out into a number of user, reference and "supporting procedures" guides.

The user guides available are:

- Fundamentals
- Wicket viewer
- Restful Objects viewer
- Security
- Testing (this guide)
- Beyond the Basics

The reference guides are:

- Annotations
- Domain Services
- Configuration Properties
- Classes, Methods and Schema
- Apache Isis Maven plugin
- Framework Internal Services

The remaining guides are:

- Developers' Guide (how to set up a development environment for Apache Isis and contribute back to the project)
- Committers' Guide (release procedures and related practices)

Chapter 2. Overview

2.1. Unit tests vs Integ tests

We divide automated tests into two broad categories:

- unit tests exercise a single unit (usually a method) of a domain object, in isolation.

  Dependencies of that object are mocked out. These are written by a developer and for a developer; they are to ensure that a particular "cog in the machine" works correctly

- integration tests exercise the application as a whole, usually focusing on one particular business operation (action).

  These are tests that represent the acceptance criteria of some business story; their intent should make sense to the domain expert (even if the domain expert is "non-technical")

To put it another way:

> Integration tests help ensure that you are ***building the right system***
>
> Unit tests help ensure that you are ***building the system right***.

2.2. Integ tests vs BDD Specs

We further sub-divide integration tests into:

- those that are implemented in Java and JUnit (we call these simply *"integration tests"*)

  Even if a domain expert understands the intent of these tests, the actual implementation will be opaque to them. Also, the only output from the tests is a (hopefully) green CI job

- tests (or rather, specifications) that are implemented in a *behaviour-driven design* (BDD) language such as Cucumber (we call these *"BDD specs"*)

  The natural language specification then maps down onto some glue code that is used to drive the application. But the benefits of taking a BDD approach include the fact that your domain expert will be able to read the tests/specifications, and that when you run the specs, you also get documentation of the application's behaviour ("living documentation").

It's up to you whether you use BDD specs for your apps; it will depend on your development process and company culture. But you if don't then you certainly should write integration tests: acceptance criteria for user stories should be automated!

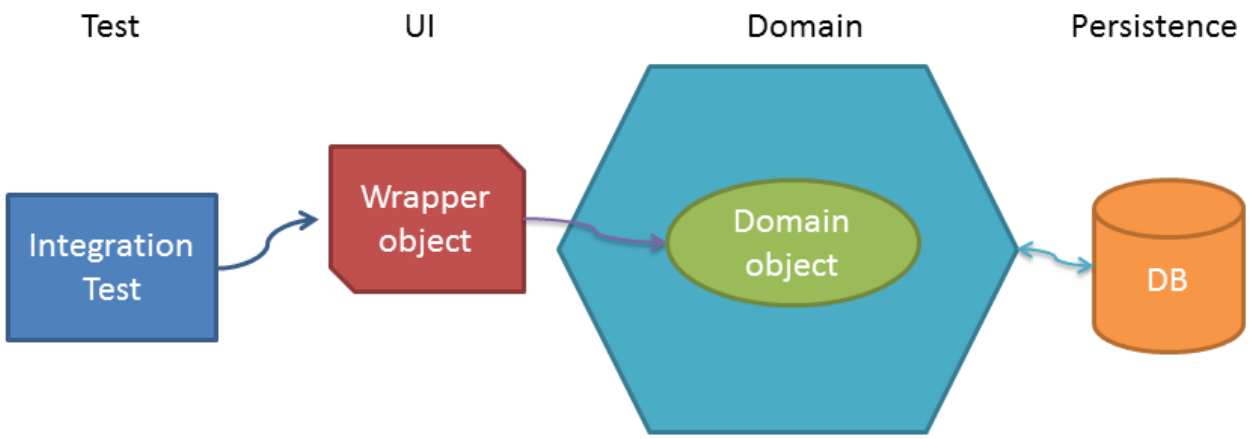
2.3. Simulated UI (`WrapperFactory`)

When we talk about integration tests/specs here, we mean tests that exercise the domain object

logic, through to the actual database. But we also want the tests to exercise the app from the users's perspective, which means including the user interface.

For most other frameworks that would require having to test the application in a very heavy weight/fragile fashion using a tool such as Selenium, driving a web browser to navigate . In this regard though, Apache Isis has a significant trick up its sleeve. Because Apache Isis implements the naked objects pattern, it means that the UI is generated automatically from the UI. This therefore allows for other implementations of the UI.

The `WrapperFactory` domain service allows a test to wrap domain objects and thus to interact with said objects "as if" through the UI:



If the test invokes an action that is disabled, then the wrapper will throw an appropriate exception. If the action is ok to invoke, it delegates through.

What this means is that an Isis application can be tested end-to-end without having to deploy it onto a webserver; the whole app can be tested while running in-memory. Although integration tests re (necessarily) slower than unit tests, they are not any harder to write (in fact, in some respects they are easier).

2.4. Dependency Injection

Isis provides autowiring dependency injection into every domain object. This is most useful when writing unit tests; simply mock out the service and inject into the domain object.

There are a number of syntaxes supported, but the simplest is to use `@javax.inject.Inject` annotation; for example:

```
@javax.inject.Inject
CustomerRepository customers;
```

Isis can inject into this even if the field has package-level (or even `private`) visibility. We recommend that you use package-level visibility, though, so that your unit tests (in the same package as the class under test) are able to inject mocks.

Isis does also support a couple of other syntaxes:

```
public void setCustomerRepository(CustomerRepository customers) { ... }
```

or

```
public void injectCustomerRepository(CustomerRepository customers) { ... }
```

Apache Isis also supports automatic dependency injection into integration tests; just declare the service dependency in the usual fashion and it will be automatically injected.

2.5. Given/When/Then

Whatever type of test/spec you are writing, we recommend you follow the given/when/then idiom:

- **given** the system is in this state (preconditions)
- **when** I poke it with a stick
- **then** it looks like this (postconditions)

A good test should be 5 to 10 lines long; the test should be there to help you reason about the behaviour of the system. Certainly if the test becomes more than 20 lines it'll be too difficult to understand.

The "when" part is usually a one-liner, and in the "then" part there will often be only two or three assertions that you want to make that the system has changed as it should.

For unit test the "given" part shouldn't be too difficult either: just instantiate the class under test, wire in the appropriate mocks and set up the expectations. And if there are too many mock expectations to set up, then "listen to the tests" ... they are telling you your design needs some work.

Where things get difficult though is the "given" for integration tests; which is the topic of the next section...

2.6. Fixture Management

In the previous section we discussed using given/when/then as a form of organizing tests, and why you should keep your tests small.

For integration tests though it can be difficult to keep the "given" short; there could be a lot of prerequisite data that needs to exist before you can actually exercise your system. Moreover, however we do set up that data, but we also want to do so in a way that is resilient to the system changing over time.

The solution that Apache Isis provides is a domain service called Fixture Scripts, that defines a pattern and supporting classes to help ensure that the "data setup" for your tests are reusable and

maintainable over time.

2.7. Fake data

In any given test there are often quite a few variables involved, to initialize the state of the objects, or to act as arguments for invoking a method, or when asserting on post-conditions. Sometimes those values are important (eg verifying that an `Order's state went from PENDING to SHIPPED, say), but often they aren't (a customer's name, for example) but nevertheless need to be set up (especially in integration tests).

We want our tests to be easily understood, and we want the reader's eye to be drawn to the values that are significant and ignore those that are not.

One way to do this is to use random (or fake) values for any insignificant data. This in effect tells the reader that "any value will do". Moreover, if it turns out that any data won't do, and that there's some behaviour that is sensitive to the value, then the test will start to flicker, passing and then failing depending on inputs. This is A Good Thing™.

Apache Isis does not, itself, ship with a fake data library. However, the Isis addons' fakedata module (non-ASF) does provide exactly this capability.

Using fake data works very well with fixture scripts; the fixture script can invoke the business action with sensible (fake/random) defaults, and only require that the essential information is passed into it by the test.

2.8. Feature Toggles

Writing automated tests is just good development practice. Also good practice is developing on the mainline (master, trunk); so that your continuous integration system really is integrating all code. Said another way: don't use branches!

Sometimes, though, a feature will take longer to implement than your iteration cycle. In such a case, how do you use continuous integration to keep everyone working on the mainline without revealing a half-implemented feature on your releases? One option is to use feature toggles.

Apache Isis does not, itself, ship with a feature toggle library. However, the Isis addons' togglz module (non-ASF) does provide exactly this capability.

With all that said, let's look in detail at the testing features provided by Apache Isis.

Chapter 3. Unit Test Support

Isis Core provides a number of unit test helpers for you to use (if you wish) to unit test your domain objects.

3.1. Contract Tests

Contract tests ensure that all instances of a particular idiom/pattern that occur within your codebase are implemented correctly. You could think of them as being a way to enforce a certain type of coding standard. Implementation-wise they use Reflections library to scan for classes.

3.1.1. SortedSets

This contract test automatically checks that all fields of type `java.util.Collection` are declared as `java.util.SortedSet`. In other words, it precludes either `java.util.List` or `java.util.Set` from being used as a collection.

For example, the following passes the contract test:

```
public class Department {
    private SortedSet<Employee> employees = new TreeSet<Employee>();
    ...
}
```

whereas this would not:

```
public class SomeDomainObject {
    private List<Employee> employees = new ArrayList<Employee>();
    ...
}
```

If using DataNucleus against an RDBMS (as you probably are) then we strongly recommend that you implement this test, for several reasons:

- first, `Set`s align more closely to the relational model than do `List`s. A `List` must have an additional index to specify order.
- second, `SortedSet` is preferable to `Set` because then the order is well-defined and predictable (to an end user, to the programmer).

  The `ObjectContracts` utility class substantially simplifies the task of implementing `Comparable` in your domain classes.

- third, if the relationship is bidirectional then JDO/Objectstore will automatically maintain the relationship.

To use the contract test, subclass `SortedSetsContractTestAbstract`, specifying the root package to

search for domain classes.

For example:

```
public class SortedSetsContractTestAll extends SortedSetsContractTestAbstract {

    public SortedSetsContractTestAll() {
        super("org.estatio.dom");
        withLoggingTo(System.out);
    }
}
```

3.1.2. Bidirectional

This contract test automatically checks that bidirectional 1:m or 1:1 associations are being maintained correctly (assuming that they follow the mutual registration pattern

(If using the JDO objectstore, then) there is generally no need to programmatically maintain 1:m relationships (indeed it may introduce subtle errors). For more details, see here. Also check out the templates in the developers' guide (live templates for IntelliJ / editor templates for Eclipse) for further guidance.

For example, suppose that `ParentDomainObject` and `ChildDomainObject` have a 1:m relationship (`ParentDomainObject#children` / `ChildDomainObject#parent`), and also `PeerDomainObject` has a 1:1 relationship with itself (`PeerDomainObject#next` / `PeerDomainObject#previous`).

The following will exercise these relationships:

```
public class BidirectionalRelationshipContractTestAll
        extends BidirectionalRelationshipContractTestAbstract {

    public BidirectionalRelationshipContractTestAll() {
        super("org.apache.isis.core.unittestsupport.bidir",
                ImmutableMap.<Class<?>,Instantiator>of(
                    ChildDomainObject.class, new InstantiatorForChildDomainObject(),
                    PeerDomainObject.class, new InstantiatorSimple
(PeerDomainObjectForTesting.class)
                ));
        withLoggingTo(System.out);
    }
}
```

The first argument to the constructor scopes the search for domain objects; usually this would be something like `"com.mycompany.dom"`.

The second argument provides a map of `Instantiator` for certain of the domain object types. This has two main purposes. First, for abstract classes, it nominates an alternative concrete class to be instantiated. Second, for classes (such as `ChildDomainObject`) that are `Comparable` and are held in a

SortedSet), it provides the ability to ensure that different instances are unique when compared against each other. If no Instantiator is provided, then the contract test simply attempts to instantiates the class.

If any of the supporting methods (addToXxx(), removeFromXxx(), modifyXxx() or clearXxx()) are missing, the relationship is skipped.

To see what's going on (and to identify any skipped relationships), use the withLoggingTo() method call. If any assertion fails then the error should be descriptive enough to figure out the problem (without enabling logging).

The example tests can be found here.

3.1.3. Injected Services Method

It is quite common for some basic services to be injected in a project-specific domain object superclass; for example a ClockService might generally be injected everywhere:

```
public abstract class EstatioDomainObject {
    @javax.inject.Inject
    protected ClockService clockService;
}
```

If a subclass inadvertantly overrides this method and provides its own ClockService field, then the field in the superclass will never initialized. As you might imagine, NullPointerExceptions could then arise.

This contract test automatically checks that the injectXxx(…) method, to allow for injected services, is not overridable, ie final.

This contract test is semi-obsolete; most of the time you will want to use @javax.inject.Inject on fields rather than the injectXxx() method. The feature dates from a time before Apache Isis supported the @Inject annotation.

To use the contract test, , subclass SortedSetsContractTestAbstract, specifying the root package to search for domain classes.

For example:

```
public class InjectServiceMethodMustBeFinalContractTestAll extends
InjectServiceMethodMustBeFinalContractTestAbstract {

    public InjectServiceMethodMustBeFinalContractTestAll() {
        super("org.estatio.dom");
        withLoggingTo(System.out);
    }
}
```

3.1.4. Value Objects

The `ValueTypeContractTestAbstract` automatically tests that a custom value type implements `equals()` and `hashCode()` correctly.

For example, testing JDK's own `java.math.BigInteger` can be done as follows:

```
public class ValueTypeContractTestAbstract_BigIntegerTest extends
ValueTypeContractTestAbstract<BigInteger> {

    @Override
    protected List<BigInteger> getObjectsWithSameValue() {
        return Arrays.asList(new BigInteger("1"), new BigInteger("1"));
    }

    @Override
    protected List<BigInteger> getObjectsWithDifferentValue() {
        return Arrays.asList(new BigInteger("2"));
    }
}
```

The example unit tests can be found here.

3.2. JMock Extensions

As noted earlier, for unit tests we tend to use JMock as our mocking library. The usual given/when/then format gets an extra step:

- **given** the system is in this state
- **expecting** these interactions (set up the mock expectations here)
- **when** I poke it with a stick
- **then** these state changes and interactions with Mocks should have occurred.

If using JMock then the interactions (in the "then") are checked automatically by a JUnit rule. However, you probably will still have some state changes to assert upon.

Distinguish between queries vs mutators

For mock interactions that simply retrieve some data, your test should not need to verify that it occurred. If the system were to be refactored and starts caching some data, you don't really want your tests to start breaking because they are no longer performing a query that once they did. If using JMock API this means using the `allowing(..)` method to set up the expectation.

On the other hand mocks that mutate the state of the system you probably should assert have occurred. If using JMock this typically means using the `oneOf(…)` method.

For more tips on using JMock and mocking in general, check out the GOOS book, written by JMock's authors, Steve Freeman and Nat Pryce and also Nat's blog.

Apache Isis' unit test support provides `JUnitRuleMockery2` which is an extension to the JMock's `JunitRuleMockery`. It provides a simpler API and also providing support for autowiring.

For example, here we see that the class under test, an instance of `CollaboratingUsingSetterInjection`, is automatically wired up with its `Collaborator`:

```
public class JUnitRuleMockery2Test_autoWiring_setterInjection_happyCase {

    @Rule
    public JUnitRuleMockery2 context = JUnitRuleMockery2.createFor(Mode
.INTERFACES_AND_CLASSES);

    @Mock
    private Collaborator collaborator;

    @ClassUnderTest
    private CollaboratingUsingSetterInjection collaborating;

    @Test
    public void wiring() {
        assertThat(collaborating.collaborator, is(not(nullValue())));
    }
}
```

Isis also includes (and automatically uses) a Javassist-based implementation of JMock's `ClassImposteriser` interface, so that you can mock out concrete classes as well as interfaces. We've provided this rather than JMock's own cglib-based implementation (which is problematic for us given its own dependencies on asm).

The example tests can be found here

3.3. SOAP Fake Endpoints

No man is an island, and neither are most applications. Chances are that at some point you may need to integrate your Apache Isis application to other external systems, possibly using old-style SOAP web services. The SOAP client in this case could be a domain service within your app, or it might be externalized, eg invoked through a scheduler or using Apache Camel.

While you will want to (of course) perform manual system testing/UAT with a test instance of that external system, it's also useful to be able to perform unit testing of your SOAP client component.

The `SoapEndpointPublishingRule` is a simple JUnit rule that allows you to run a fake SOAP endpoint within an unit test.

The (non-ASF) Isis addons' publishmq module provides a full example of how to integrate and test an Apache Isis application with a (faked out) external system.

3.3.1. `SoapEndpointPublishingRule`

The idea behind this rule is that you write a fake server endpoint that implements the same WSDL contract as the "real" external system does, but which also exposes additional API to specify responses (or throw exceptions) from SOAP calls. It also typically records the requests and allows these to be queried.

In its setup your unit test and gets the rule to instantiate and publish that fake server endpoint, and then obtains a reference to that server endpoint. It also instantiates the SOAP client, pointing it at the address (that is, a URL) that the fake server endpoint is running on. This way the unit test has control of both the SOAP client and server: the software under test and its collaborator.

In the test methods your unit test sets up expectations on your fake server, and then exercises the SOAP client. The SOAP client calls the fake server, which then responds accordingly. The test can then assert that all expected interactions have occurred.

So that tests don't take too long to run, the rule puts the fake server endpoints onto a thread-local. Therefore the unit tests should clear up any state on the fake server endpoints.

Your unit test uses the rule by specifying the endpoint class (must have a no-arg constructor):

```
public class FakeExternalSystemEndpointRuleTest {
    @Rule
    public SoapEndpointPublishingRule serverRule =
        new SoapEndpointPublishingRule(FakeExternalSystemEndpoint.class);          ①
    private FakeExternalSystemEndpoint fakeServerEndpoint;
    private DemoObject externalSystemContract;                                     ②
    @Before
    public void setUp() throws Exception {
        fakeServerEndpoint =
            serverRule.getEndpointImplementor(FakeExternalSystemEndpoint.class);   ③
        final String endpointAddress =
            serverRule.getEndpointAddress(FakeExternalSystemEndpoint.class);       ④
        final DemoObjectService externalSystemService =                            ⑤
                new DemoObjectService(ExternalSystemWsdl.getWsdl());               ⑥
        externalSystemContract = externalSystemService.getDemoObjectOverSOAP();
        BindingProvider provider = (BindingProvider) externalSystemContract;
        provider.getRequestContext().put(
                BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
                endpointAddress)
        );
    }
    @Test
    public void happy_case() throws Exception {
        // given
        final Update update = new Update();                                        ⑦
        ...
        // expect
        final UpdateResponse response = new UpdateResponse();                      ⑧
        ...
        fakeServerEndpoint.control().setResponse(updateResponse);
        // when
        PostResponse response = externalSystemContract.post(update);               ⑨
        // then
        final List<Update> updates =                                               ⑩
            fakeServerEndpoint.control().getUpdates();
        ...
    }
}
```

① specify the class that implements the endpoint (must have a no-arg constructor)

② the SOAP contract as defined in WSDL and generated by wsdl2java

③ get hold of the fake server-side endpoint from the rule…

④ … and its endpoint address

⑤ use factory (also generated by wsdl2java) to create client-side endpoint

⑥ `getWsdl()` is a utility method to return a URL for the WSDL (eg from the classpath)

⑦ create a request object in order to invoke the SOAP web service

⑧ instruct the fake server endpoint how to respond

⑨ invoke the web service

⑩ check the fake server endpoint was correctly invoked etc.

The rule can also host multiple endpoints; just provide multiple classes in the constructor:

```
@Rule
public SoapEndpointPublishingRule serverRule =
                new SoapEndpointPublishingRule(
                    FakeCustomersEndpoint.class,
                    FakeOrdersEndpoint.class,
                    FakeProductsEndpoint.class);
```

To lookup a particular endpoint, specify its type:

```
FakeProductsEndpoint fakeProductsServerEndpoint =
            serverRule.getPublishedEndpoint(FakeProductsEndpoint.class);
```

The endpoint addresses that the server endpoints run on are determined automatically. If you want more control, then you can call one of `SoapEndpointPublishingRule`'s overloaded constructors, passing in one or more `SoapEndpointSpec` instances.

3.3.2. XML Marshalling Support

Apache Isis' unit testing support also provides helper `JaxbUtil` and `JaxbMatchers` classes. These are useful if you have exampler XML-serialized representations of the SOAP requests and response payloads and want to use these within your tests.

3.4. Maven Configuration

Apache Isis' unit test support is automatically configured if you use the SimpleApp archetype. To set it up manually, update the `pom.xml` of your domain object model module:

```
<dependency>
    <groupId>org.apache.isis.core</groupId>
    <artifactId>isis-core-unittestsupport</artifactId>
    <scope>test</scope> ①
</dependency>
```

① Normally `test`; usual Maven scoping rules apply.

We also recommend that you configure the `maven-surefire-plugin` to pick up the following class patterns:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.10</version>
    <configuration>
        <includes>
            <include>**/*Test.java</include>
            <include>**/*Test_*.java</include>
            <include>**/*Spec*.java</include>
        </includes>
        <excludes>
            <exclude>**/Test*.java</exclude>
            <exclude>**/*ForTesting.java</exclude>
            <exclude>**/*Abstract*.java</exclude>
        </excludes>
        <useFile>true</useFile>
        <printSummary>true</printSummary>
        <outputDirectory>${project.build.directory}/surefire-reports</outputDirectory>
    </configuration>
</plugin>
```

Chapter 4. Integration Test Support

As discussed in the introductory overview of this chapter, Apache Isis allows you to integration test your domain objects from within JUnit. There are several parts to this:

- configuring the Apache Isis runtime so it can be bootstrapped (mostly boilerplate)
- defining a base class to perform said bootstrapping
- using fixture scripts to set up the app
- using `WrapperFactory` so that the UI can be simulated.

We'll get to all that shortly, but let's start by taking a look at what a typical integration test looks like.

4.1. Typical Usage

This example adapted from the Isis addons' todoapp (non-ASF). The code we want to test (simplified) is:

```
public class ToDoItem ... {

    private boolean complete;
    @Property( editing = Editing.DISABLED )
    public boolean isComplete() {
        return complete;
    }
    public void setComplete(final boolean complete) {
        this.complete = complete;
    }

    @Action()
    public ToDoItem completed() {
        setComplete(true);
        ...
        return this;
    }
    public String disableCompleted() {
        return isComplete() ? "Already completed" : null;
    }
    ...
}
```

We typically put the bootstrapping of Apache Isis into a superclass (`AbstractToDoIntegTest` below), then subclass as required.

For this test (of the "completed()" action) we need an instance of a `ToDoItem` that is not yet complete. Here's the setup:

```
public class ToDoItemIntegTest extends AbstractToDoIntegTest {

    @Inject
    FixtureScripts fixtureScripts;                                  ①
    @Inject
    ToDoItems toDoItems;                                            ②
    ToDoItem toDoItem;                                              ③

    @Before
    public void setUp() throws Exception {
        RecreateToDoItemsForCurrentUser fixtureScript =             ④
            new RecreateToDoItemsForCurrentUser();
        fixtureScripts.runFixtureScript(fixtureScript, null);
        final List<ToDoItem> all = toDoItems.notYetComplete();      ⑤
        toDoItem = wrap(all.get(0));                                ⑥
    }
    ...
}
```

① the `FixtureScripts` domain service is injected, providing us with the ability to run fixture scripts

② likewise, an instance of the `ToDoItems` domain service is injected. We'll use this to lookup...

③ the object under test, held as a field

④ the fixture script for this test; it deletes all existing todo items (for the current user only) and then recreates them

⑤ we lookup one of the just-created todo items...

⑥ and then wrap it so that our interactions with it are as if through the UI

The following code tests the happy case, that a not-yet-completed `ToDoItem` can be completed by invoking the `completed()` action:

```
public class ToDoItemIntegTest ... {
    ...
    public static class Completed extends ToDoItemIntegTest { ①
        @Test
        public void happyCase() throws Exception {
            // given
            assertThat(toDoItem.isComplete()).isFalse();        ②
            // when
            toDoItem.completed();
            // then
            assertThat(toDoItem.isComplete()).isTrue();
        }
        ...
    }
}
```

① the idiom we follow is to use nested static classes to identify the class responsibility being tested

② the todoapp uses AssertJ.

What about when a todo item is already completed? The `disableCompleted()` method in the class says that it shouldn't be allowed (the action would be greyed out in the UI with an appropriate tooltip). The following test verifies this:

```
        @Test
        public void cannotCompleteIfAlreadyCompleted() throws Exception {
            // given
            unwrap(toDoItem).setComplete(true);                    ①
            // expect
            expectedExceptions.expectMessage("Already completed");  ②
            // when
            toDoItem.completed();
        }
```

① we unwrap the domain object in order to set its state directly

② the expectedExceptions JUnit rule (defined by a superclass) verifies that the appropiate exception is indeed thrown (in the "when")

And what about the fact that the underlying "complete" property is disabled (the `@Disabled` annotation?). If the `ToDoItem` is put into edit mode in the UI, the complete checkbox should remain read-only. Here's a verify similar test that verifies this also:

```
        @Test
        public void cannotSetPropertyDirectly() throws Exception {
            // expect
            expectedExceptions.expectMessage("Always disabled");  ①
            // when
            toDoItem.setComplete(true);
        }
```

① again we use the exceptedExceptions rule.

4.2. Bootstrapping

Integration tests instantiate an Apache Isis "runtime" (as a singleton) within a JUnit test. Because (depending on the size of your app) it takes a little time to bootstrap Apache Isis, the framework caches the runtime on a thread-local from one test to the next.

Nevertheless, we do need to bootstrap the runtime for the very first test.

As of 1.9.0 the bootstrapping of integration tests and webapps has been simplified through the `AppManifest` class. Since this isn't mandatory, for now we present both techniques.

The example code in this section is taken from the app generated by the SimpleApp archetype.

4.2.1. System Initializer

The bootstrapping itself is performed by a "system initializer" class. This is responsible for instantiating the Apache Isis runtime, and binding it to a thread-local.

1.13.0 (Improved intent)

As of `1.13.0`, the `IsisConfigurationForJdoIntegTests` (which provides a number of configuration settings specifically for running integration tests) can be removed; instead all configuration properties can be defined through the `AppManifest`:

For example:

```
public class DomainAppSystemInitializer {
    public static void initIsft() {
        IsisSystemForTest isft = IsisSystemForTest.getElseNull();
        if(isft == null) {
            isft = new IsisSystemForTest.Builder()
                    .withLoggingAt(org.apache.log4j.Level.INFO)
                    .with(new DomainAppAppManifest() {
                        @Override
                        public Map<String, String> getConfigurationProperties() {
                            final Map<String, String> map = Maps.newHashMap();
                            Util.withJavaxJdoRunInMemoryProperties(map);
                            Util.withDataNucleusProperties(map);
                            Util.withIsisIntegTestProperties(map);
                            return map;
                        }
                    })
                    .build();
            isft.setUpSystem();
            IsisSystemForTest.set(isft);
        }
    }
}
```

While the code is slightly longer than previously, the responsibilities for returning the configuration properties to use for the test now reside in a single location. The new `AppManifest.Util` class provides the helper methods to actually add in the appropriate config properties.

1.9.0 (`AppManifest`)

As of 1.9.0, the code (using `AppManifest`) is:

```
public class DomainAppSystemInitializer {
    public static void initIsft() {
        IsisSystemForTest isft = IsisSystemForTest.getElseNull();
        if(isft == null) {
            isft = new IsisSystemForTest.Builder()
                    .withLoggingAt(org.apache.log4j.Level.INFO)
                    .with(new DomainAppAppManifest())
                    .with(new IsisConfigurationForJdoIntegTests())
                    .build()
                    .setUpSystem();
            IsisSystemForTest.set(isft);
        }
    }
}
```

where `DomainAppAppManifest` in turn is defined as:

```
public class DomainAppAppManifest implements AppManifest {
    @Override
    public List<Class<?>> getModules() {
        return Arrays.asList(
                domainapp.dom.DomainAppDomainModule.class,
                domainapp.fixture.DomainAppFixtureModule.class,
                domainapp.app.DomainAppAppModule.class
        );
    }
    ...
}
```

Further details on bootstrapping with the `AppManifest` can be found in the reference guide.

1.8.0 and earlier

Prior to 1.9.0, the services and entities had to be specified in two separate locations. The suggested way to do this was to introduce a subclass of the `IsisSystemForTest.Builder` class:

```
private static class DomainAppSystemBuilder extends IsisSystemForTest.Builder {
①
    public DomainAppSystemBuilder() {
        withLoggingAt(org.apache.log4j.Level.INFO);
        with(testConfiguration());
        with(new DataNucleusPersistenceMechanismInstaller());
②
        withServicesIn( "domainapp" );
③
    }
    private static IsisConfiguration testConfiguration() {
        final IsisConfigurationForJdoIntegTests testConfiguration =
            new IsisConfigurationForJdoIntegTests();
④
        testConfiguration.addRegisterEntitiesPackagePrefix("domainapp.dom.modules");
⑤
        return testConfiguration;
    }
}
```

① subclass the framework-provided `IsisSystemForTest.Builder`.

② equivalent to `isis.persistor=datanucleus` in `isis.properties`

③ specify the `isis.services` key in `isis.properties` (where "domainapp" is the base package for all classes within the app)

④ `IsisConfigurationForJdoIntegTests` has pre-canned configuration for using an in-memory HSQLDB and other standard settings; more on this below.

⑤ equivalent to `isis.persistor.datanucleus.RegisterEntities.packagePrefix` key (typically in `persistor_datanucleus.properties`)

This builder could then be used within the system initializer:

```
public class DomainAppSystemInitializer {
    public static void initIsft() {
        IsisSystemForTest isft = IsisSystemForTest.getElseNull();
        if(isft == null) {
            isft = new DomainAppSystemBuilder()    ①
                            .build()
                            .setUpSystem();
            IsisSystemForTest.set(isft);           ②
        }
    }
    private static class DomainAppSystemBuilder
        extends IsisSystemForTest.Builder { ... }
}
```

① instantiates and initializes the Apache Isis runtime (the `IsisSystemForTest` class)

② binds the runtime to a thread-local.

IsisConfigurationForJdoIntegTests

Integration tests are configured programmatically, with a default set of properties to bootstrap the JDO/DataNucleus objectstore using an HSQLDB in-memory database.

To remove a little bit of boilerplate, the `IsisConfigurationForJdoIntegTests` class (in the `org.apache.isis.objectstore.jdo.datanucleus` package) can be used to bootstrap the application. If necessary, this class can be subclassed to override these defaults.

Table 1. Default Configuration Properties for Integration Tests

Property	Value	Description
`isis.persistor.datanucleus.impl.javax.jdo.option.ConnectionURL`	jdbc:hsqldb:mem:test	JDBC URL
`isis.persistor.datanucleus.impl.javax.jdo.option.ConnectionDriverName`	org.hsqldb.jdbcDriver	JDBC Driver
`isis.persistor.datanucleus.impl.javax.jdo.option.ConnectionUserName`	sa	Username
`isis.persistor.datanucleus.impl.javax.jdo.option.ConnectionPassword`	<empty string>	Password
`isis.persistor.datanucleus.impl.datanucleus.schema.autoCreateAll`	true	Recreate DB for each test run (an in-memory database)
`isis.persistor.datanucleus.impl.datanucleus.schema.validateAll`	false	Disable validations (minimize bootstrap time)
`isis.persistor.datanucleus.impl.datanucleus.persistenceByReachabilityAtCommit`	false	As per WEB-INF/persistor_datanucleus.properties
`isis.persistor.datanucleus.impl.datanucleus.identifier.case`	MixedCase	As per WEB-INF/persistor_datanucleus.properties
`isis.persistor.datanucleus.impl.datanucleus.cache.level2.type`	none	As per WEB-INF/persistor_datanucleus.properties
`isis.persistor.datanucleus.impl.datanucleus.cache.level2.mode`	ENABLE_SELECTIVE	As per WEB-INF/persistor_datanucleus.properties

Property	Value	Description
`isis.persistor.datanucleus.install-fixtures`	true	Automatically install any fixtures that might have been registered
`isis.persistor.enforceSafeSemantics`	false	
`isis.deploymentType`	server_prototype	

4.2.2. Abstract Class

We recommend defining a base class for all your other classes to integration classes to inherit from. The main responsibility of this class is tocall the system initializer, described earlier. We only need the initialization to be performed once, so this call is performed in a `@BeforeClass` hook.

The code below shows the general form:

```
public abstract class DomainAppIntegTest {
    @BeforeClass
    public static void initClass() {
        org.apache.log4j.PropertyConfigurator.configure("logging.properties");   ①
        DomainAppSystemInitializer.initIsft();                                    ②
        new ScenarioExecutionForIntegration();                                    ③
    }
}
```

① ensure that logging messages don't get swallowed

② initialize the Apache Isis runtime

③ primarily exists to support the writing of BDD specifications, but also enables finer-grained management of sessions/transactions (discussed below).

`IntegrationTestAbstract`

In fact, we recommend that your base class inherit from Apache Isis' `IntegrationTestAbstract` class:

```
public abstract class DomainAppIntegTest extends IntegrationTestAbstract {
    ...
}
```

Although not mandatory, this provides a number of helper/convenience methods and JUnit rules:

```
@Rule
public IsisTransactionRule isisTransactionRule =                          ①
    new IsisTransactionRule();
@Rule
public JUnitRuleMockery2 context =                                        ②
    JUnitRuleMockery2.createFor(Mode.INTERFACES_AND_CLASSES);
@Rule
public ExpectedException expectedExceptions =                             ③
    ExpectedException.none();
@Rule
public ExceptionRecognizerTranslate exceptionRecognizerTranslations =     ④
    ExceptionRecognizerTranslate.create();
```

① ensures an Apache Isis session/transaction running for each test

② sets up a JMock context (using Apache Isis' extension to JMock as described in JMock Extensions.

③ standard JUnit rule for writing tests that throw exceptions

④ to capture messages that require translation, as described in i18 support.

All of these rules could be inlined in your own base class; as we say, they are a convenience.

The `IntegrationTestAbstract` also provides a number of helper/convenience methods, though most of these have been deprecated because the functionality they expose is now readily accessible through various domain services; most notably these are:

- `WrapperFactory`

  to wrap objects simulating interaction through the user interface)

- `TransactionService`

  most commonly used to commit changes after the fixture setup) and,

- `SessionManagementService`

  for tests that check interactions over multiple separate sessions.

4.3. Wrapper Factory

The `WrapperFactory` service is responsible for wrapping a domain object in a dynamic proxy, of the same type as the object being proxied. And the role of this wrapper is to simulate the UI.

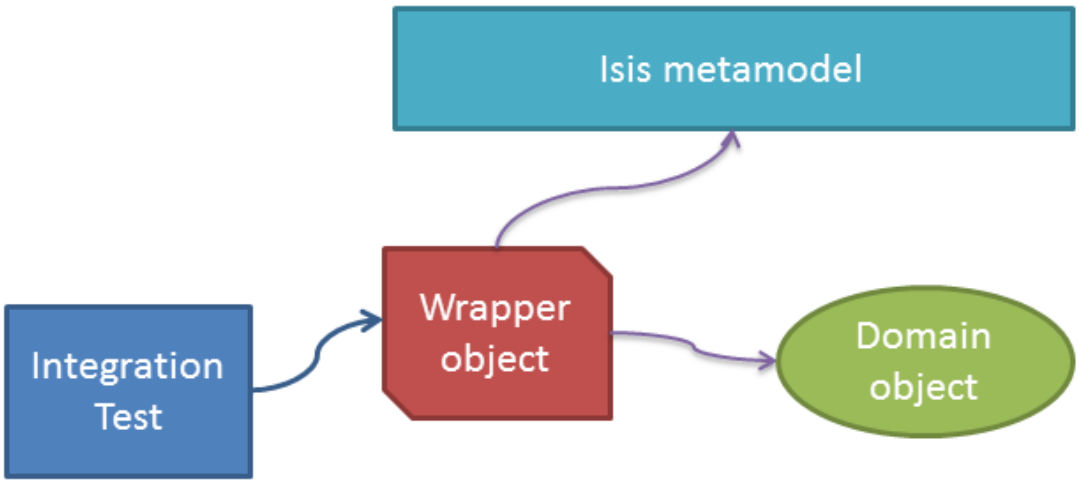
`WrapperFactory` uses javassist to perform its magic; this is the same technology that ORMs such as Hibernate use to manage lazy loading/dirty tracking (DataNucleus uses a different mechanism).

It does this by allowing through method invocations that would be allowed if the user were interacting with the domain object through one of the viewers, but throwing an exception if the user attempts to interact with the domain object in a way that would not be possible if using the UI.

The mechanics are as follows:

1. the integration test calls the `WrapperFactory` to obtain a wrapper for the domain object under test. This is usually done in the test's `setUp()` method.
2. the test calls the methods on the wrapper rather than the domain object itself
3. the wrapper performs a reverse lookup from the method invoked (a regular `java.lang.reflect.Method` instance) into the Apache Isis metamodel
4. (like a viewer), the wrapper then performs the "see it/use it/do it" checks, checking that the member is visible, that it is enabled and (if there are arguments) that the arguments are valid
5. if the business rule checks pass, then the underlying member is invoked. Otherwise an exception is thrown.

The type of exception depends upon what sort of check failed. It's straightforward enough: if the member is invisible then a `HiddenException` is thrown; if it's not usable then you'll get a `DisabledException`, if the args are not valid then catch an `InvalidException`.



Let's look in a bit more detail at what the test can do with the wrapper.

4.3.1. Wrapping and Unwrapping

Wrapping a domain object is very straightforward; simply call `WrapperFactory#wrap(…)`.

For example:

```
Customer customer = ...;
Customer wrappedCustomer = wrapperFactory.wrap(wrappedCustomer);
```

Going the other way — getting hold of the underlying (wrapped) domain object — is just as easy; just call `WrapperFactory#unwrap(…)`.

For example:

```
Customer wrappedCustomer = ...;
Customer customer = wrapperFactory.unwrap(wrappedCustomer);
```

If you prefer, you also can get the underlying object from the wrapper itself, by downcasting to `WrappingObject` and calling `__isis_wrapped()` method:

```
Customer wrappedCustomer = ...;
Customer customer = (Customer)((WrappingObject)wrappedCustomer).__isis_wrapped());
```

We're not sure that's any easier (in fact we're certain it looks rather obscure). Stick with calling `unwrap(…)`!

4.3.2. Using the wrapper

As the wrapper is intended to simulate the UI, only those methods that correspond to the "primary" methods of the domain object's members are allowed to be called. That means:

- for ***object properties*** the test can call the getter or setter method
- for ***object collections*** the test can call the getter.

  If there is a supporting `addTo…()` or `removeFrom…()` method, then these can be called. It can also call `add(…)` or `remove(…)` on the collection (returned by the gettter) itself.

  In this respect the wrapper is more functional than the Wicket viewer (which does not expose the ability to mutate collections directly).

- for ***object actions*** the test can call the action method itself.

As a convenience, we also allow the test to call any `default…(),choices…()` or `autoComplete…()` method. These are often useful for obtaining a valid value to use.

What the test can't call is any of the remaining supporting methods, such as `hide…()`, `disable…()` or `validate…()`. That's because their value is implied by the exception being thrown.

The wrapper *does* also allow the object's `title()` method or its `toString()` , however this is little use for objects whose title is built up using the `@Title` annotation. Instead, we recommend that your test verifies an object's title by calling `DomainObjectContainer#titleOf(…)` method.

4.3.3. Firing Domain Events

As well as enforcing business rules, the wrapper has another important feature, namely that it will cause domain events to be fired.

For example, if we have an action annotated with `@Action(domainEvent=…)`:

```
public class ToDoItem ... {
    @Action(
            domainEvent =CompletedEvent.class
    )
    public ToDoItem completed() { ... }
    ...
}
```

then invoking the action through the proxy will cause the event (`CompletedEvent` above) to be fired to any subscribers. A test might therefore look like:

```
@Inject
private EventBusService eventBusService;                                    ①

@Test
public void subscriberReceivesEvents() throws Exception {

    // given
    final ToDoItem.CompletedEvent[] evHolder = new ToDoItem.CompletedEvent[1];    ②
    eventBusService.register(new Object() {
        @Subscribe
        public void on(final ToDoItem.CompletedEvent ev) {                   ③
            evHolder[0] = ev;
        }
    });

    // when
    toDoItem.completed();                                                    ④

    // then
    then(evHolder[0].getSource()).isEqualTo(unwrap(toDoItem));               ⑤
    then(evHolder[0].getIdentifier().getMemberName()).isEqualTo("completed");
}
```

① inject `EventBusService` into this test

② holder for subscriber to capture event to

③ subscriber's callback, using the guava subscriber syntax

④ invoking the domain object using the wrapper

⑤ assert that the event was populated

The wrapper will also fire domain events for properties (if annotated with `@Property(domainEvent=…)`) or collections (if annotated with `@Collection(domainEvent=…)`).

It isn't possible to use the `WrapperFactory` in a unit test, because there needs to be a running instance of Apache Isis that holds the metamodel.

4.4. Maven Configuration

Apache Isis' integration test support is automatically configured if you use the SimpleApp archetype. To set it up manually, update the `pom.xml` of your domain object model module:

```
<dependency>
    <groupId>org.apache.isis.core</groupId>
    <artifactId>isis-core-integtestsupport</artifactId>
</dependency>
<dependency>
    <groupId>org.apache.isis.core</groupId>
    <artifactId>isis-core-wrapper</artifactId>
</dependency>
```

We also recommend that you configure the `maven-surefire-plugin` to pick up the following class patterns:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.10</version>
    <configuration>
        <includes>
            <include>**/*Test.java</include>
            <include>**/*Test_*.java</include>
            <include>**/*Spec*.java</include>
        </includes>
        <excludes>
            <exclude>**/Test*.java</exclude>
            <exclude>**/*ForTesting.java</exclude>
            <exclude>**/*Abstract*.java</exclude>
        </excludes>
        <useFile>true</useFile>
        <printSummary>true</printSummary>
        <outputDirectory>${project.build.directory}/surefire-reports</outputDirectory>
    </configuration>
</plugin>
```

Chapter 5. BDD Spec Support

Behaviour-driven design (BDD) redefines testing not as an after-the-fact "let's check the system works", but rather as a means to work with the domain expert to (a) specify the behaviour of the feature *before* starting implementation, and (b) provide a means that the domain expert can verify the feature after it has been implemented.

Since domain experts are usually non-technical (at least, they are unlikely to be able to read or want to learn how to read JUnit/Java code), then applying BDD typically requires writing specifications in using structured English text and (ASCII) tables. The BDD tooling parses this text and uses it to actually interact with the system under test. As a byproduct the BDD frameworks generate readable output of some form; this is often an annotated version of the original specification, marked up to indicate which specifications passed, which have failed. This readable output is a form of "living documentation"; it captures the actual behaviour of the system, and so is guaranteed to be accurate.

There are many BDD tools out there; Apache Isis provides an integration with Cucumber JVM (see also the github site):

5.1. How it works

At a high level, here's how Cucumber works

- specifications are written in the Gherkin DSL, following the "given/when/then" format.
- Cucumber-JVM itself is a JUnit runner, and searches for feature files on the classpath.
- These in turn are matched to step definitions through regular expressions.

  It is the step definitions (the "glue") that exercise the system.

The code that goes in step definitions is broadly the same as the code that goes in an integration test method. One benefit of using step definitions (rather than integration tests) is that the step definitions are reusable across scenarios, so there may be less code overall to maintain. For example, if you have a step definition that maps to "given an uncompleted todo item", then this can be used for all the scenarios that start with that as their precondition.

5.2. Key classes

There are some key framework classes that make up the spec support; these are discussed below.

some of these are also used by Apache Isis' Integration Test support.

5.2.1. `IsisSystemForTest`

The `IsisSystemForTest` class allows a complete running instance of Apache Isis to be bootstrapped (with the JDO objectstore); this is then held on a a `ThreadLocal` from one test to another.

Typically bootstrapping code is used to lazily instantiate the `IsisSystemForTest` once and once only.

The mechanism for doing this is line-for-line identical in both BDD step defs and integration tests.

5.2.2. `ScenarioExecution`

The `ScenarioExecution` provides a context for a scenario that is being executed. It is Cucumber that determines which step definitions are run, and in which order, and so state cannot be passed between step definitions using local variables or instance variables. Instead the `ScenarioExecution` acts like a hashmap, allowing each step to put data (eg "given an uncompleted todoitem") into the map or get data ("when I complete the todoitem") from the map. This is done using the `putVar(…)` and `getVar(…)` methods.

This corresponds broadly to the "World" object in Ruby-flavoured Cucumber.

The `ScenarioExecution` also provids access to the configured domain services (using the `service(…)` method) and the `DomainObjectContainer` (through the `container()` method).

This could probably be refactored; Cucumber JVM provides automatic dependency injection into setp definitions, but Apache Isis does not currently leverage or exploit this capability.

Like the `IsisSystemForTest` class, the `ScenarioExecution` class binds an instance of itself onto a `ThreadLocal`. It can then be accessed in BDD step definitions using `ScenarioExecution.current()` static method.

5.2.3. `WrapperFactory`

As with integration tests, the UI can be simulated by "wrapping" each domain object in a proxy using the `WrapperFactory`.

5.2.4. `CukeGlueAbstract`

The `CukeGlueAbstract` acts as a convenience superclass for writing BDD step definitions (analogous to the `IntegrationTestAbstract` for integation tests). Underneath the covers it delegates to an underlying `ScenarioExecution`.

5.3. Writing a BDD spec

BDD specifications contain:

- a `XxxSpec.feature` file, describing the feature and the scenarios (given/when/then)s that constitute its acceptance criteria
- a `RunSpecs.java` class file to run the specification (all boilerplate). This will run all `.feature` files in the same package or subpackages.
- one or several `XxxGlue` constituting the step definitions to be matched against.

  The "glue" (step definitions) are intended to be reused across features. We therefore recommend that they reside in a separate package, and are organized by the entity type upon which they

act.

For example, given a feature that involves `Customer` and `Order`, have the step definitions pertaining to `Customer` reside in `CustomerGlue`, and the step definitions pertaining to `Order` reside in `OrderGlue`.

The `glue` attribute of the Cucumber-JVM JUnit runner eallows you to indicate which package(s) should be recursively searched to find any glue.

- a system initializer class. You can reuse the system initializer from any integration tests (as described in Integration Test Support, bootstrapping section).

Here's an example of a feature from the SimpleApp archetype:

```
@SimpleObjectsFixture
Feature: List and Create New Simple Objects

  @integration
  Scenario: Existing simple objects can be listed and new ones created
    Given there are initially 3 simple objects
    When  I create a new simple object
    Then  there are 4 simple objects
```

The `@SimpleObjectsFixture` is a custom tag we've specified to indicate the prerequisite fixtures to be loaded; more on this in a moment. The `@integration` tag, meanwhile, says that this feature should be run with integration-level scope.

Although BDD specs are most commonly used for end-to-end tests (ie at the same scope as an integration test), the two concerns (expressability of a test to a business person vs granularity of the test) should not be conflated. There are a couple of good blog posts discussing this. The basic idea is to avoid the overhead of a heavy-duty integration test if possible.

Apache Isis does also support running BDD specs in unit test mode; by annotating the scenario with the `@unit` (rather than `@integration` tag). When running under unit-level scope, the Apache Isis system is *not* instantiated. Instead, the `ScenarioExecution` class returns JMock mocks (except for the `WrapperFactory`, if configured).

To support unit testing scope Apache Isis provides the `InMemoryDB` class; a glorified hashmap of "persisted" objects. Use of this utility class is optional.

Writing a BDD spec that supports both modes of operation therefore takes more effort and we expect most users interested in BDD will use integration-testing scope; for these reasons we have chosen *not* to include unit-testing support in the SimpleApp archetype. For those who do require faster-executing test suite, it's worthwhile knowing that Apache Isis can support this.

The `RunSpecs` class to run this feature (and any other features in this package or subpackages) is just

boilerplate

```
@RunWith(Cucumber.class)
@CucumberOptions(
        format = {
                "html:target/cucumber-html-report"
                ,"json:target/cucumber.json"
        },
        glue={"classpath:domainapp.integtests.specglue"},
        strict = true,
        tags = { "~@backlog", "~@ignore" })
public class RunSpecs {
    // intentionally empty
}
```

The JSON formatter allows integration with enhanced reports, for example as provided by Masterthought.net (screenshots at end of page). (Commented out) configuration for this is provided in the SimpleApp archetype.

The bootstrapping of Apache Isis can be moved into a `BootstrappingGlue` step definition:

```
public class BootstrappingGlue extends CukeGlueAbstract {
    @Before(value={"@integration"}, order=100)
    public void beforeScenarioIntegrationScope() {
        org.apache.log4j.PropertyConfigurator.configure("logging.properties");
        SimpleAppSystemInitializer.initIsft();

        before(ScenarioExecutionScope.INTEGRATION);
    }
    @After
    public void afterScenario(cucumber.api.Scenario sc) {
        assertMocksSatisfied();
        after(sc);
    }
}
```

The fixture to run also lives in its own step definition, `CatalogOfFixturesGlue`:

```
public class CatalogOfFixturesGlue extends CukeGlueAbstract {
    @Before(value={"@integration", "@SimpleObjectsFixture"}, order=20000)
    public void integrationFixtures() throws Throwable {
        scenarioExecution().install(new RecreateSimpleObjects());
    }
}
```

Note that this is annotated with a tag (`@SimpleObjectsFixture`) so that the correct fixture runs. (We might have a whole variety of these).

The step definitions pertaining to `SimpleObject` domain entity then reside in the `SimpleObjectGlue` class. This is where the heavy lifting gets done:

```
public class SimpleObjectGlue extends CukeGlueAbstract {
    @Given("^there are.* (\\d+) simple objects$")
    public void there_are_N_simple_objects(int n) throws Throwable {
        try {
            final List<SimpleObject> findAll = service(SimpleObjects.class).listAll();
            assertThat(findAll.size(), is(n));
            putVar("list", "all", findAll);

        } finally {
            assertMocksSatisfied();
        }
    }
    @When("^I create a new simple object$")
    public void I_create_a_new_simple_object() throws Throwable {
        service(SimpleObjects.class).create(UUID.randomUUID().toString());
    }
}
```

If using Java 8, note that Cucumber JVM supports a simplified syntax using lambdas.

5.4. BDD Tooling

To help write feature files and generate step definitions, we recommend Roberto Lo Giacco's Eclipse plugin. For more information, see Dan's short blog post. It works very well. Of interest: this is implemented using XText.

5.5. Maven Configuration

Apache Isis' BDD spec support is automatically configured if you use the SimpleApp archetype. To set it up manually, update the `pom.xml` of your domain object model module:

```
<dependency>
    <groupId>org.apache.isis.core</groupId>
    <artifactId>isis-core-specsupport</artifactId>
    <scope>test</scope> ①
</dependency>
```

① Normally `test`; usual Maven scoping rules apply.

We also recommend that you configure the `maven-surefire-plugin` to pick up the following class patterns:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.10</version>
    <configuration>
        <includes>
            <include>**/*Test.java</include>
            <include>**/*Test_*.java</include>
            <include>**/*Spec*.java</include>
        </includes>
        <excludes>
            <exclude>**/Test*.java</exclude>
            <exclude>**/*ForTesting.java</exclude>
            <exclude>**/*Abstract*.java</exclude>
        </excludes>
        <useFile>true</useFile>
        <printSummary>true</printSummary>
        <outputDirectory>${project.build.directory}/surefire-reports</outputDirectory>
    </configuration>
</plugin>
```

You may also find it more convenient to place the `.feature` files in `src/test/java`, rather than `src/test/resources`. If you wish to do this, then your integtest module's `pom.xml` must contain:

```
<build>
    <testResources>
        <testResource>
            <filtering>false</filtering>
            <directory>src/test/resources</directory>
        </testResource>
        <testResource>
            <filtering>false</filtering>
            <directory>src/test/java</directory>
            <includes>
                <include>**</include>
            </includes>
            <excludes>
                <exclude>**/*.java</exclude>
            </excludes>
        </testResource>
    </testResources>
</build>
```

Chapter 6. Fixture Scripts

When writing integration tests (and implementing the glue for BDD specs) it can be difficult to keep the "given" short; there could be a lot of prerequisite data that needs to exist before you can actually exercise your system. Moreover, however we do set up that data, but we also want to do so in a way that is resilient to the system changing over time.

On a very simple system you could probably get away with using SQL to insert directly into the database, or you could use a toolkit such as dbunit to upload data from flat files. Such approaches aren't particularly maintainable though. If in the future the domain entities (and therefore corresponding database tables) change their structure, then all of those data sets will need updating.

Even more significantly, there's no way to guarantee that the data that's being loaded is logically consistent with the business behaviour of the domain objects themselves. That is, there's nothing to stop your test from putting data into the database that would be invalid if one attempted to add it through the app.

The solution that Apache Isis provides is a small library called ***fixture scripts***. A fixture script is basically a wrapper for executing arbitrary work, but that work almost always invoking a business action.

If you want to learn more on this topic (with live coding!), check out this presentation given at BDD Exchange 2014.

There is another benefit to Apache Isis' fixture script approach; the fixtures can be (in prototyping mode) run from your application. This means that fixture scripts can actually help all the way through the development lifecycle:

- when specifying a new feature, you can write a fixture script to get the system into the "given" state, and then start exploring the required functionality with the domain expert actually *within* the application

  And if you can't write a fixture script for the story, it probably means that there's some prerequisite feature that needs implementing that you hadn't previously recognized

- when the developer implements the story, s/he has a precanned script to run when they manually verify the functionality works
- when the developer automates the story's acceptance test as an integration test, they already have the "given" part of the test implemented
- when you want to pass the feature over to the QA/tester for additional manual exploratory testing, they have a fixture script to get them to a jumping off point for their explorations
- when you want to demonstrate the implemented feature to your domain expert, your demo can use the fixture script so you don't bore your audience in performing lots of boring setup before getting to the actual feature
- when you want to roll out training to your users, you can write fixture scripts as part of their training exercises

The following sections explain the API and how to go about using the API.

6.1. API and Usage

There are two parts to using fixture scripts: the `FixtureScripts` domain service class, and the `FixtureScript` view model class:

- The role of the `FixtureScripts` domain service is to locate all fixture scripts from the classpath and to let them be invoked, either from an integration test/BDD spec or from the UI of your Isis app.
- The role of `FixtureScript` meanwhile is to subclass for each of the scenarios that you want to define. You can also subclass from `FixtureScript` to create helpers; more on this below.

Let's look at `FixtureScripts` domain service in more detail first.

6.1.1. `FixtureScripts`

There are two ways in which you can provide a `FixtureScripts` service.

The original (pre-1.9.0) approach is to subclass subclass `FixtureScripts` domain service, with your subclass specifying which package to search for. Various other settings can also be provided, and - being a custom class - you can also add in additional actions. A common example is to provide a "one-shot" action to recreate a standard demo set of objects.

As of 1.9.0 there is an alternative design. Instead of subclassing `FixtureScripts` you instead implement the `FixtureScriptsSpecificationProvider` SPI. (As its name suggests), this provides a `FixtureScriptsSpecification` object that contains the same information as would otherwise have been in the `FixtureScripts` subclass.

The actual implementation of the `FixtureScripts` service is then provided by the framework itself, namely the `FixtureScriptsDefault` domain service, annotated to be rendered on the secondary "Prototyping" menu. This uses the `FixtureScriptsSpecificationProvider` to adjust itself accordingly.

For example, here's the `FixtureScriptsSpecificationProvider` service that's generated by the SimpleApp archetype:

```
@DomainService(nature = NatureOfService.DOMAIN)
public class DomainAppFixturesProvider implements FixtureScriptsSpecificationProvider
{
    @Override
    public FixtureScriptsSpecification getSpecification() {
        return FixtureScriptsSpecification
                .builder(DomainAppFixturesProvider.class)
①
                .with(FixtureScripts.MultipleExecutionStrategy.EXECUTE)
②
                .withRunScriptDefault(RecreateSimpleObjects.class)
③
                .withRunScriptDropDown(FixtureScriptsSpecification.DropDownPolicy
.CHOICES)       ④
                .withRecreate(RecreateSimpleObjects.class)
⑤
                .build();
    }
}
```

① search for all fixture scripts under the package containing this class

② if the same fixture script (class) is encountered more than once, then run anyway; more on this in Organizing Fixture scripts], below.

③ specify the fixture script class to provide as the default for the service's "run fixture script" action

④ whether the service's "run fixture script" action should display other fixture scripts using a choices drop down or (if there are very many of them) using an auto-complete

⑤ if present, enables a "recreate objects and return first" action to be displayed in the UI

The benefit of this design - decoupling the responsibilities of the superclass and the subclass - is that it ensures that there is always an instance of `FixtureScripts` available, even if the application itself doesn't provide one. Some of the (non-ASF) Isis Addons modules (eg Isis addons' security module) expect there to be a `FixtureScripts` service for seeding data, which has caused some confusion.

By way of comparison, if using the older pre-1.9.0 approach then you would create a subclass:

```
@DomainService
@DomainServiceLayout(
        menuBar = DomainServiceLayout.MenuBar.SECONDARY,          ①
        named="Prototyping",                                      ②
        menuOrder = "500"
)
public class DomainAppFixturesService extends FixtureScripts {   ③
    public DomainAppFixturesService() {
        super(
            "domainapp",                                          ④
            MultipleExecutionStrategy.EXECUTE);                   ⑤
    }
    @Override
    public FixtureScript default0RunFixtureScript() {
        return findFixtureScriptFor(RecreateSimpleObjects.class); ⑥
    }
    @Override
    public List<FixtureScript> choices0RunFixtureScript() {       ⑦
        return super.choices0RunFixtureScript();
    }
}
```

① in the UI, render the on the right hand side (secondary) menu bar…

② … under the "Prototyping" menu

③ inherit from `org.apache.isis.applib.fixturescripts.FixtureScripts`

④ search for all fixture scripts under the "domainapp" package; in your code this would probably be "com.mycompany.myapp" or similar

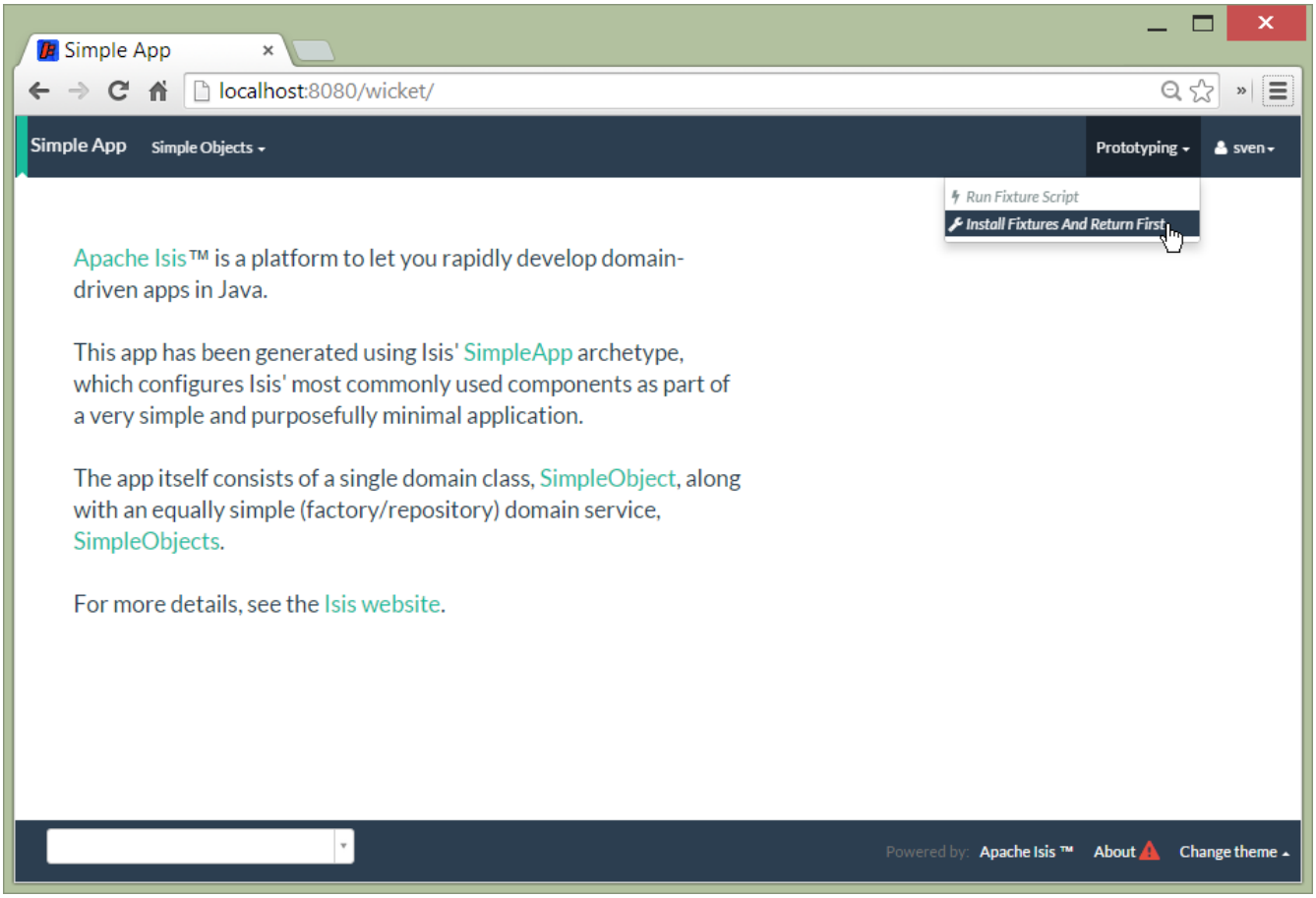
⑤ if the same fixture script (class) is encountered more than once, then run anyway; more on this in Organizing Fixture scripts], below.

⑥ (optional) specify a default fixture script to run, in this case `RecreateSimpleObjects` fixture script (see below)

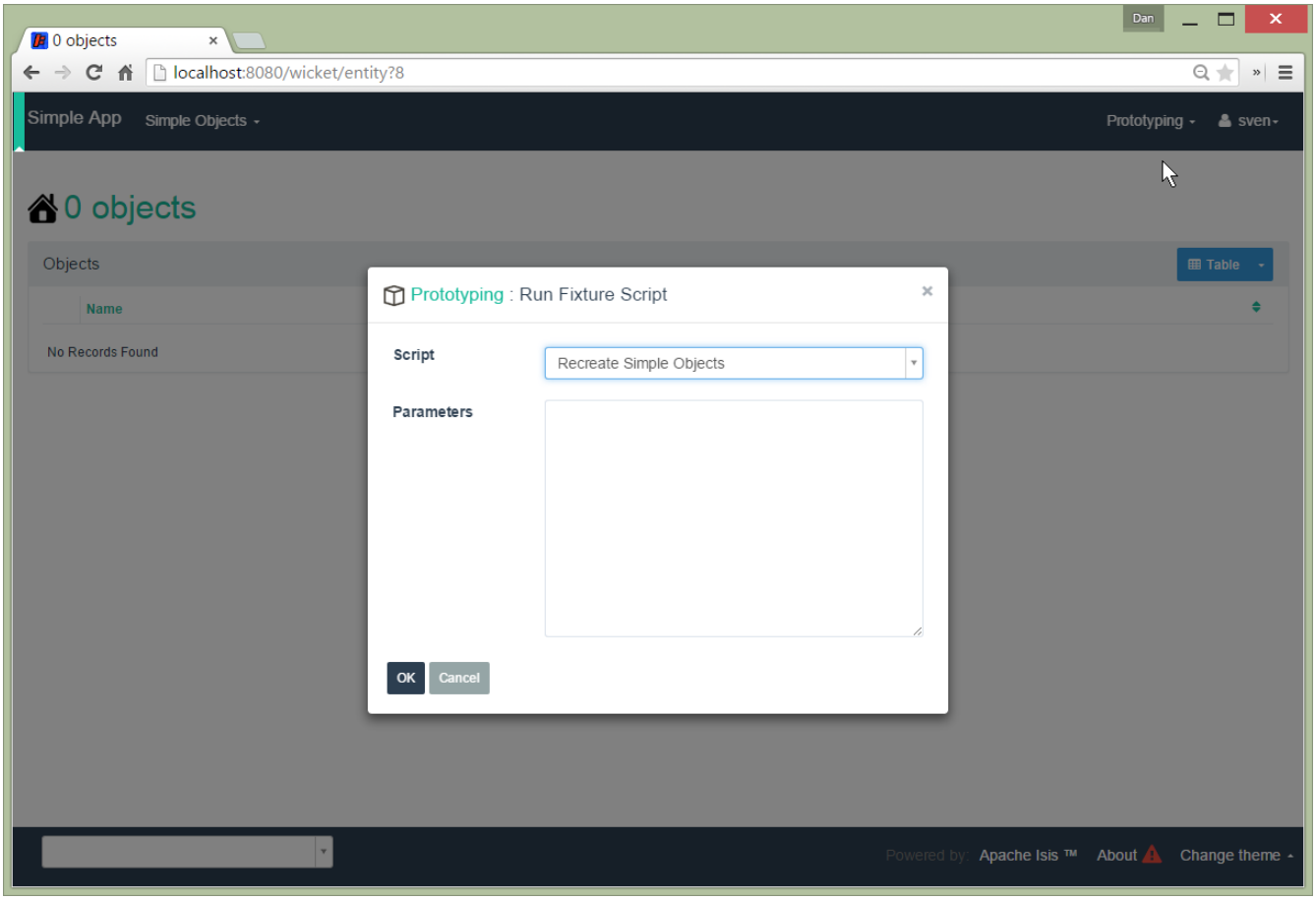
⑦ make all fixture scripts found available as a drop-down (by increasing the visibility of this method to `public`)

This isn't quite equivalent; you would need to write additional code to support the "recreate objects and return first" action, for example.

Either way, here's how the domain service looks like in the UI:

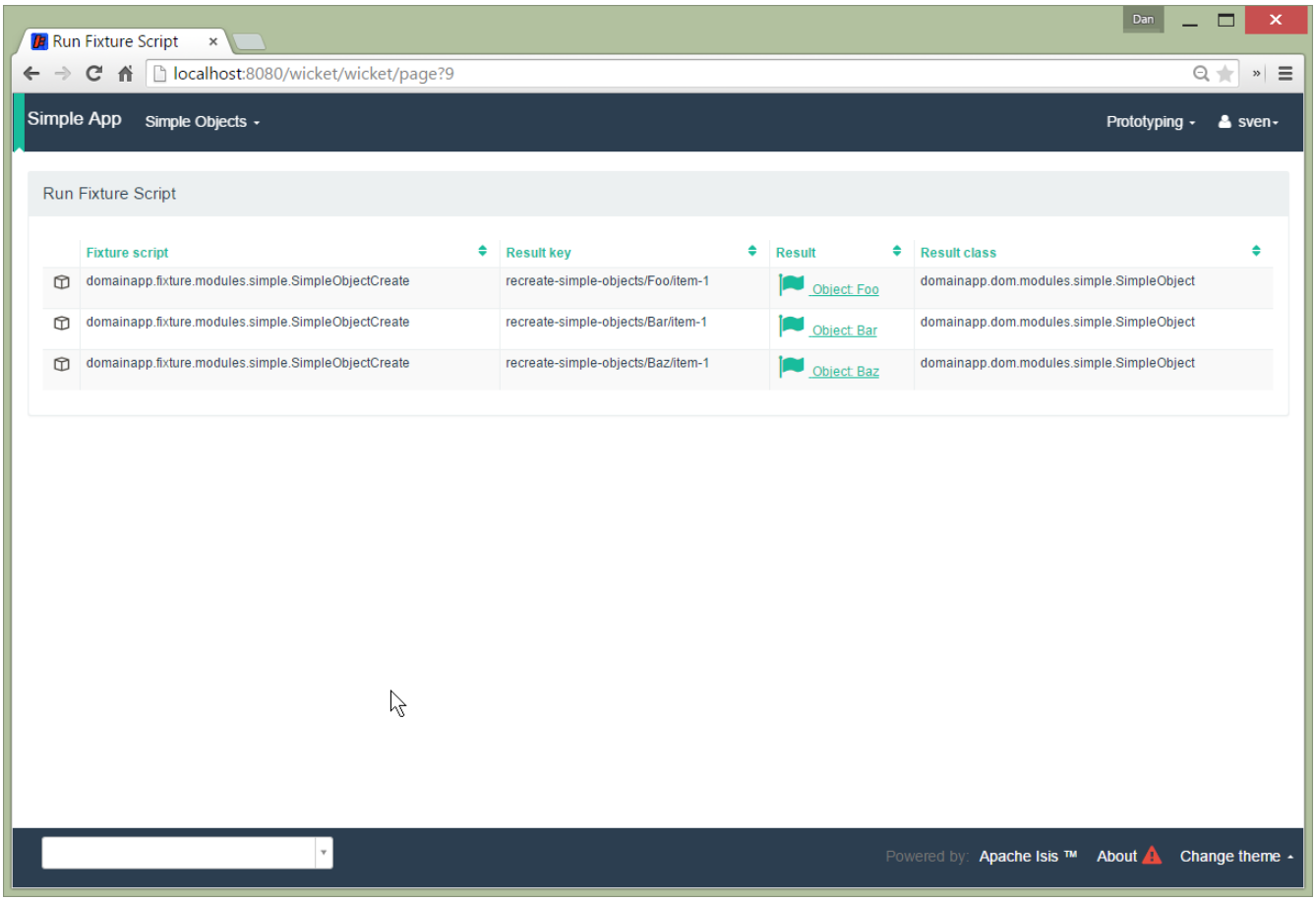


and here's what the `runFixtureScript` action prompt looks like:



when this is executed, the resultant objects (actually, instances of FixtureResult`) are shown in the

UI:



If you had defined many fixture scripts then a drop-down might become unwieldy, in which case your code would probably override the `autoComplete…()` instead:

```
    @Override
    public List<FixtureScript> autoComplete0RunFixtureScript(final @MinLength(1)
String searchArg) {
        return super.autoComplete0RunFixtureScript(searchArg);
    }
```

You are free, of course, to add additional "convenience" actions into it if you wish for the most commonly used/demo'd setups ; you'll find that the SimpleApp archetype adds this additional action:

```
    @Action(
            restrictTo = RestrictTo.PROTOTYPING
    )
    @ActionLayout(
            cssClassFa="fa fa-refresh"
    )
    @MemberOrder(sequence="20")
    public Object recreateObjectsAndReturnFirst() {
        final List<FixtureResult> run = findFixtureScriptFor(RecreateSimpleObjects
.class).run(null);
        return run.get(0).getObject();
    }
```

Let's now look at the `FixtureScript` class, where there's a bit more going on.

6.1.2. `FixtureScript`

A fixture script is ultimately just a block of code that can be executed, so it's up to you how you implement it to set up the system. However, we strongly recommend that you use it to invoke actions on business objects, in essence to replay what a real-life user would have done. That way, the fixture script will remain valid even if the underlying implementation of the system changes in the future.

Here's the `RecreateSimpleObjects` fixture script from the SimpleApp archetype:

```
public class RecreateSimpleObjects extends FixtureScript {                    ①

    public final List<String> NAMES = Collections.unmodifiableList(Arrays.asList(
            "Foo", "Bar", "Baz", "Frodo", "Froyo",
            "Fizz", "Bip", "Bop", "Bang", "Boo"));                             ②
    public RecreateSimpleObjects() {
        withDiscoverability(Discoverability.DISCOVERABLE);                     ③
    }
    private Integer number;                                                    ④
    public Integer getNumber() { return number; }
    public RecreateSimpleObjects setNumber(final Integer number) {
        this.number = number;
        return this;
    }
    private final List<SimpleObject> simpleObjects = Lists.newArrayList();    ⑤
    public List<SimpleObject> getSimpleObjects() {
        return simpleObjects;
    }
    @Override
    protected void execute(final ExecutionContext ec) {                        ⑥
        // defaults
        final int number = defaultParam("number", ec, 3);                      ⑦
        // validate
        if(number < 0 || number > NAMES.size()) {
            throw new IllegalArgumentException(
                String.format("number must be in range [0,%d)", NAMES.size()));
        }
        // execute
        ec.executeChild(this, new SimpleObjectsTearDown());                    ⑧
        for (int i = 0; i < number; i++) {
            final SimpleObjectCreate fs =
                new SimpleObjectCreate().setName(NAMES.get(i));
            ec.executeChild(this, fs.getName(), fs);                           ⑨
            simpleObjects.add(fs.getSimpleObject());                           ⑩
        }
    }
}
```

① inherit from `org.apache.isis.applib.fixturescripts.FixtureScript`

② a hard-coded list of values for the names. Note that the Isis addons' fakedata module (non-ASF) could also have been used

③ whether the script is "discoverable"; in other words whether it should be rendered in the drop-down by the `FixtureScripts` service

④ input property: the number of objects to create, up to 10; for the calling test to specify, but note this is optional and has a default (see below). It's important that a wrapper class is used (ie `java.lang.Integer`, not `int`)

⑤ output property: the generated list of objects, for the calling test to grab

⑥ the mandatory execute(...) API

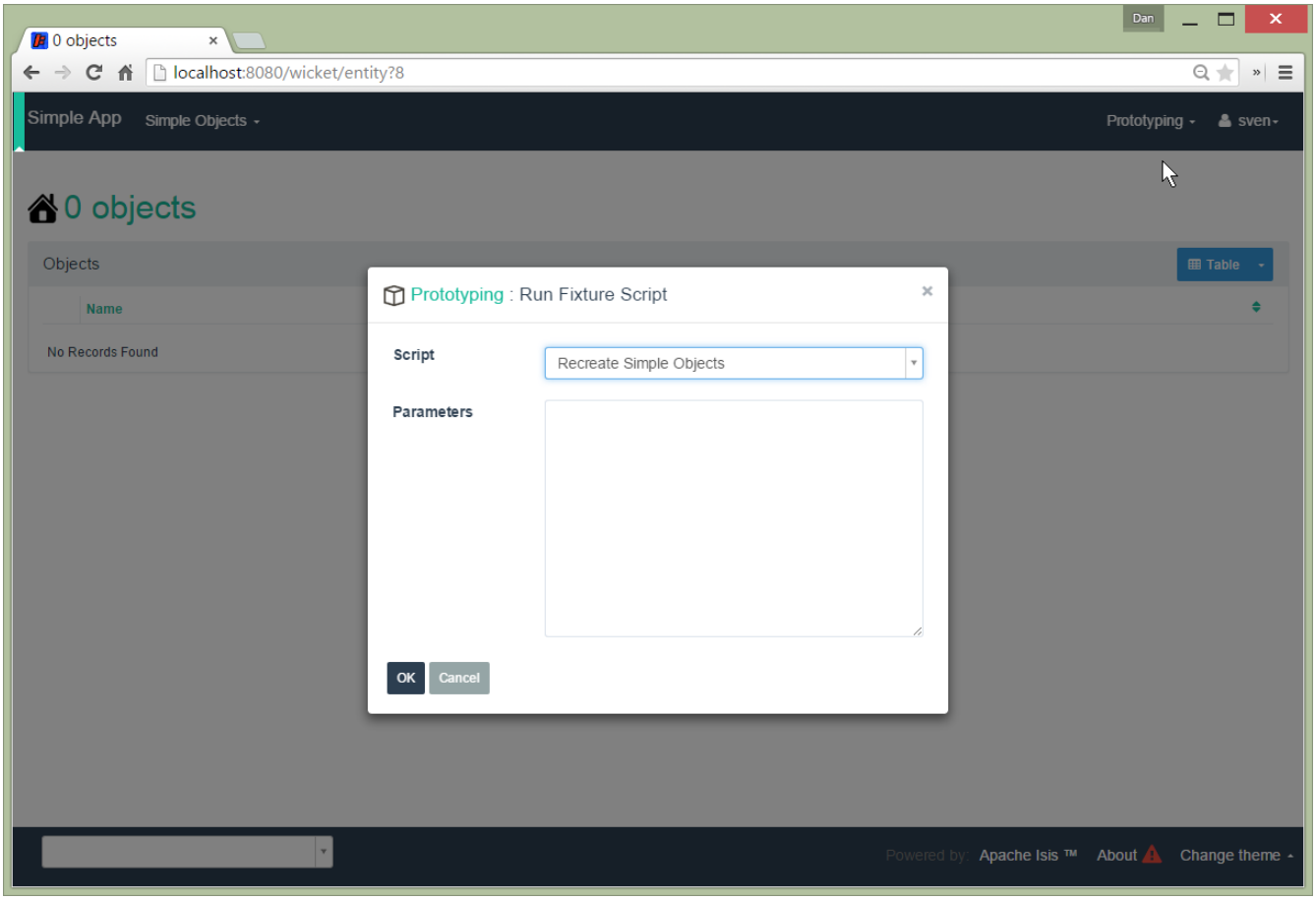
⑦ the `defaultParam(…)` (inherited from `FixtureScript`) will default the `number` property (using Java's Reflection API) if none was specified

⑧ call another fixture script (`SimpleObjectsTearDown`) using the provided `ExecutionContext`. Note that although the fixture script is a view model, it's fine to simply instantiate it (rather than using `DomainObjectContainer#newTransientInstance(…)`).

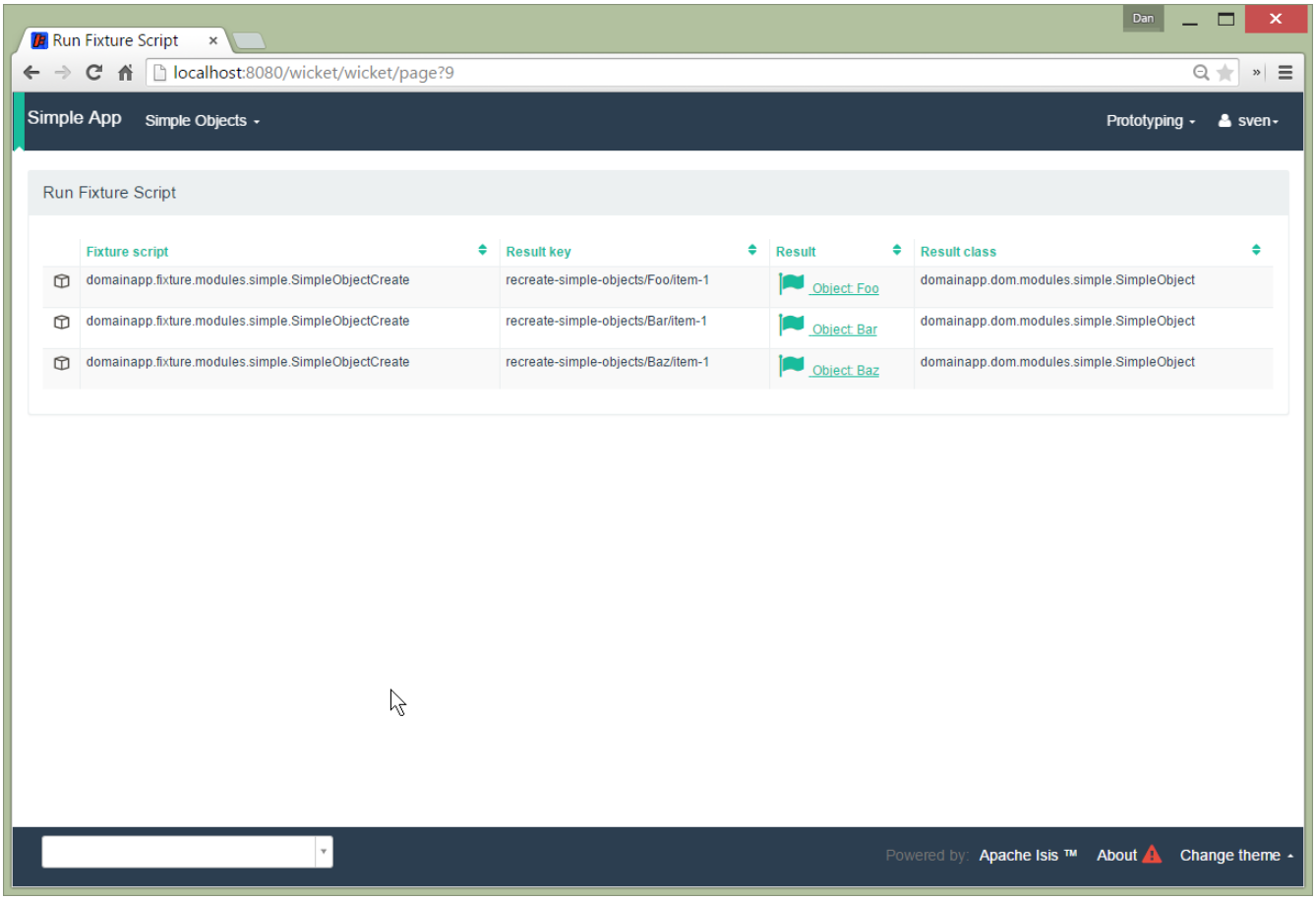
⑨ calling another fixture script (`SimpleObjectCreate`) using the provided `ExecutionContext`

⑩ adding the created object to the list, for the calling object to use.

Because this script has exposed a "number" property, it's possible to set this from within the UI. For example:



When this is executed, the framework will parse the text and attempt to reflectively set the corresponding properties on the fixture result. So, in this case, when the fixture script is executed we actually get 6 objects created:



It's commonplace for one fixture script to call another. In the above example this script called `SimpleObjectsTearDown` and `SimpleObjectCreate`. Let's take a quick look at `SimpleObjectCreate`:

```
public class SimpleObjectCreate extends FixtureScript {        ①

    private String name;                                        ②
    public String getName() { return name; }
    public SimpleObjectCreate setName(final String name) {
        this.name = name;
        return this;
    }
    private SimpleObject simpleObject;                          ③
    public SimpleObject getSimpleObject() {
        return simpleObject;
    }
    @Override
    protected void execute(final ExecutionContext ec) {         ④
        String name = checkParam("name", ec, String.class);     ⑤
        this.simpleObject = wrap(simpleObjects)                 ⑥
                                .create(name);                  ⑦
        ec.addResult(this, simpleObject);                       ⑧
    }
    @javax.inject.Inject
    private SimpleObjects simpleObjects;                        ⑨
}
```

① inherit from `org.apache.isis.applib.fixturescripts.FixtureScript`, as above

② input property: name of the object; this time it is required

③ output property: the created simple object

④ the mandatory execute(...) API

⑤ the `checkParam(…)` (inherited from `FixtureScript`) will check that the "name" property has been populated (using Java's Reflection API) and throw an exception if not

⑥ wrap the injected `SimpleObjects` domain service (using the `WrapperFactory`) to simulate interaction through the UI...

⑦ .. and actually create the object using the "create" business action of that service

⑧ add the resulting object to the execution context; this makes the object available to access if run from the UI

⑨ inject the domain service into the fixture script

6.1.3. Using within Tests

Fixture scripts can be called from integration tests just the same way that fixture scripts can call one another.

For example, here's an integration test from the SimpleApp archetype:

```
public class SimpleObjectIntegTest extends SimpleAppIntegTest {
    @Inject
    FixtureScripts fixtureScripts;                          ①
    SimpleObject simpleObjectWrapped;
    @Before
    public void setUp() throws Exception {
        // given
        RecreateSimpleObjects fs =
             new RecreateSimpleObjects().setNumber(1);      ②
        fixtureScripts.runFixtureScript(fs, null);          ③

        SimpleObject simpleObjectPojo =
            fs.getSimpleObjects().get(0);                   ④
        assertThat(simpleObjectPojo).isNotNull();

        simpleObjectWrapped = wrap(simpleObjectPojo);       ⑤
    }
    @Test
    public void accessible() throws Exception {
        // when
        final String name = simpleObjectWrapped.getName();
        // then
        assertThat(name).isEqualTo(fs.NAMES.get(0));
    }
    ...
}
```

① inject the `FixtureScripts` domain service (just like any other domain service)

② instantiate the fixture script for this test, and configure

③ execute the fixture script

④ obtain the object under test from the fixture

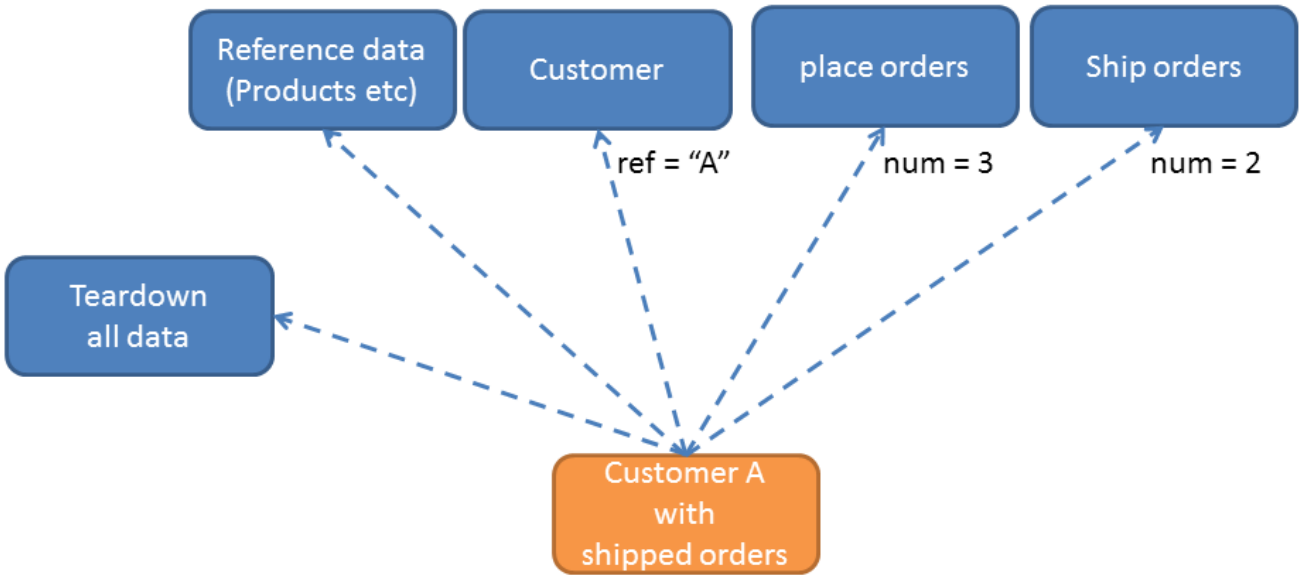
⑤ wrap the object (to simulate being interacted with through the UI)

6.1.4. Organizing Fixture scripts

There are lots of ways to organize fixture scripts, but we've used them as either:

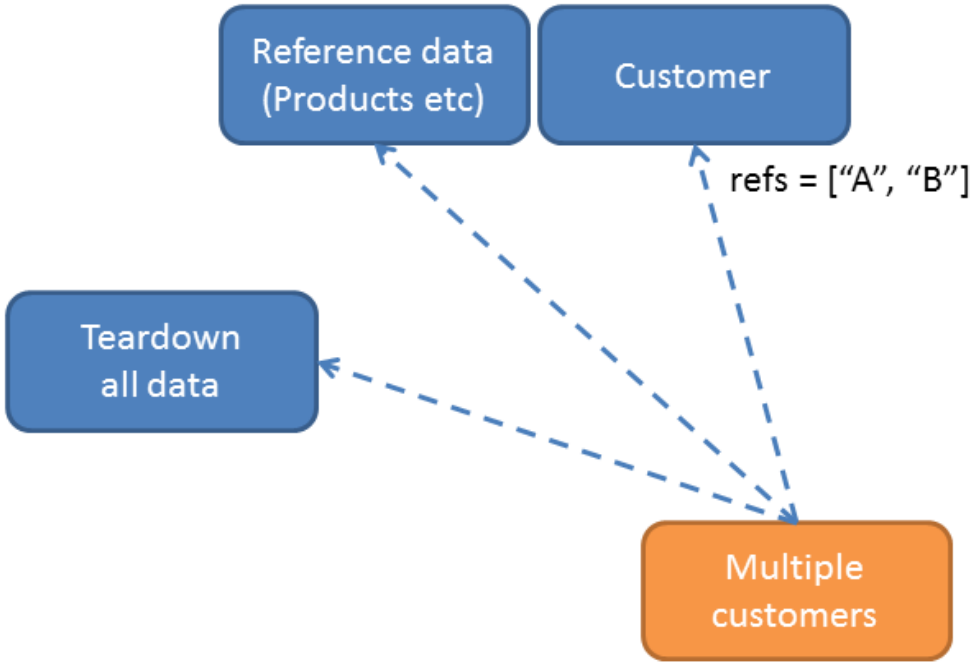
- a fairly flat style, eg as in the SimpleApp archetype, also as in the Isis addons' todoapp;
- in a "composite pattern" style, eg as in the Estatio open source app.

These two styles are probably best illustrated with, well, some illustrations. Here's a fixture script in the "flat" style for setting up a customer with some orders, a number of which has been placed:



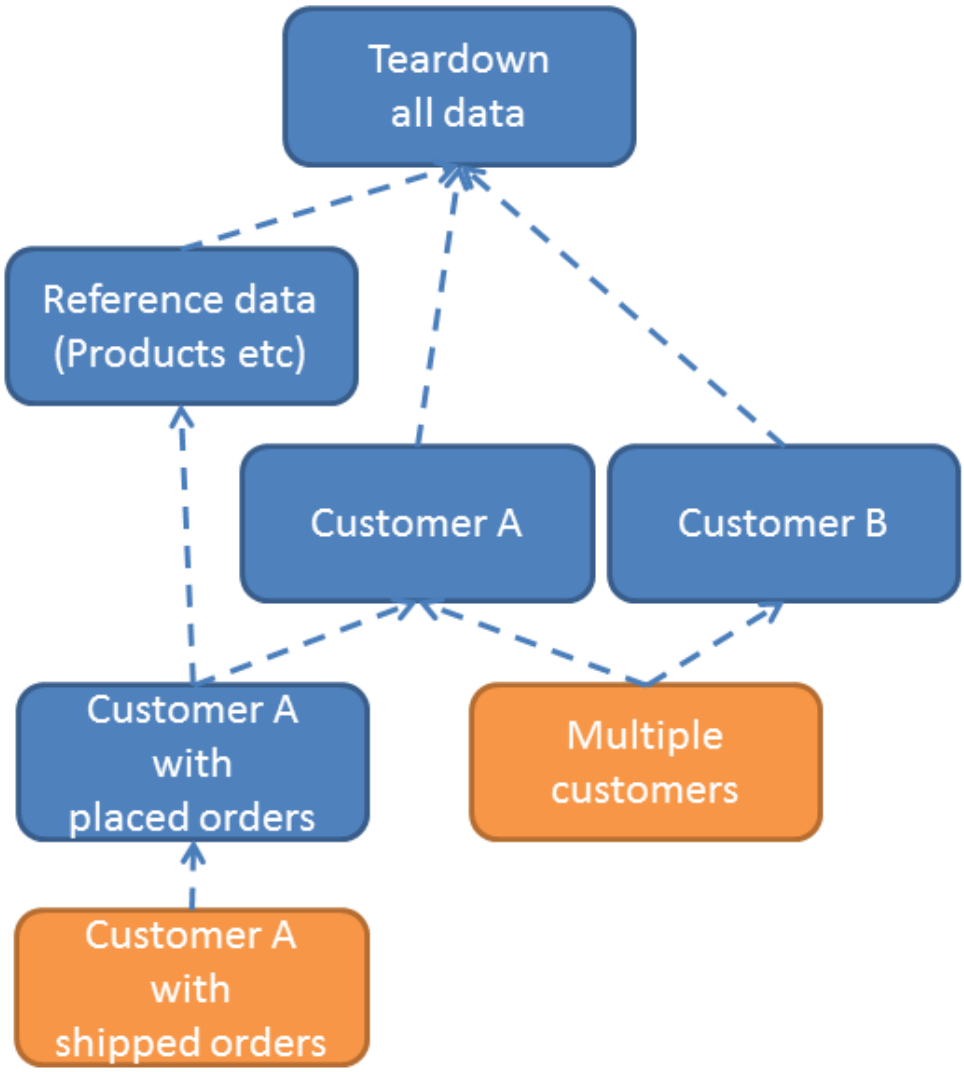
Notice how we have a single script that’s in control of the overall process, and takes responsibility for passing data from one fixture script to the next.

Here’s a similar, simpler script, from the same fictional app, to set up two customers:



We can reuse the same fixture "customer" script, either calling it twice or (perhaps better) passing it an array of customer details to set up.

With the composite style, we rely on each fixture script to set up its own prerequisites. Thus:



Back in the earlier section we noted the `MultipleExecutionStrategy` setting. We can now explain the meaning of this: the enum value of `EXECUTE` is designed for the flat style (where every fixture script will be called), whereas the enum value of `IGNORE` is designed for the composite style, and ensures that any fixture scripts visited more than once (eg TearDown) are only every executed the first time.

As already noted , the app generated by the SimpleApp archetype uses the flat structure, and we feel that it's a better at separating out the "how" (how we set up some graph of domain objects into a known state, eg a customer with shipped placed orders and a newly placed order) from the "what" (what data should we actually use for the customer's name, say).

The composite style tends to combine these, which one could argue does not separate responsibilities well enough. On the other hand, one could also make an argument that the composite style is a good way to implement precanned personas, eg "Joe", the customer who has a newly placed order, from "Mary" customer who has none.

Further approaches

As of there are two other approaches.

The first is to take advantage of a new `MultipleExecutionStrategy`, namely `EXECUTE_ONCE_BY_VALUE`.

Under this strategy the determination as to whether to run a given fixture script is by comparing the fixture script against all others that have run. If all fixture scripts implement value semantics, then they can effectively determine whether they need to run or not.

This strategy was introduced in order to better support the `ExcelFixture` fixture script (provided by the (non-ASF) Isis addons' excel module. The `ExcelFixture` takes an Excel spreadsheet and loads up each row. For those cases where we wish to ensure that the same spreadsheet is not loaded more than once, the `IGNORE` strategy would have required that a trivial subclass of `ExcelFixture` is created for each and every spreadsheet. The `EXECUTE_ONCE_BY_VALUE` on the other hand delegates the determination to the value semantics of the `ExcelFixture`, which is based on the contents of the spreadsheet.

Note that as of `1.10.0` the `IGNORE` enum has been deprecated, replaced by `EXECUTE_ONCE_BY_CLASS`

The second approach is in recognition that there is, in fact, something of a design flaw with the concept of `MultipleExecutionStrategy`: it requires that all fixture scripts being run follow the same conventions. There's a good argument that this shouldn't be a global "setting": the responsibility for determining whether a given fixture script should be executed should reside not in the `FixtureScripts` service but in the `FixtureScript` itself.

Thus, the `FixtureScripts.ExecutionContext` exposes the `getPreviouslyExecuted()` method that allows the fixture script to check for itself which fixture scripts have already been run, and act accordingly. For this approach, the `MultipleExecutionStrategy` should be left as simply `EXECUTE`.

6.2. SudoService

Sometimes in our fixture scripts we want to perform a business action running as a particular user. This might be for the usual reason - so that our fixtures accurately reflect the reality of the system with all business constraints enforced using the `WrapperFactory` - or more straightforwardly it might be simply that the action depends on the identity of the user invoking the action.

An example of the latter case is in the (non-ASF) Isis addons' todoapp's `ToDoItem` class:

Production code that depends on current user

```
public ToDoItem newToDo(
        @Parameter(regexPattern = "\\w[@&:\\-\\,\\.\\+ \\w]*") @ParameterLayout(named
="Description")
        final String description,
        @ParameterLayout(named="Category")
        final Category category,
        @Parameter(optionality = Optionality.OPTIONAL) @ParameterLayout(named
="Subcategory")
        final Subcategory subcategory,
        @Parameter(optionality = Optionality.OPTIONAL) @ParameterLayout(named="Due
by")
        final LocalDate dueBy,
        @Parameter(optionality = Optionality.OPTIONAL) @ParameterLayout(named="Cost")
        final BigDecimal cost) {
    return newToDo(description, category, subcategory, currentUserName(), dueBy,
cost);
}
private String currentUserName() {
    return container.getUser().getName(); ①
}
```

① is the current user.

The fixture for this can use the `SudoService` to run a block of code as a specified user:

Fixture Script

```
final String description = ...
final Category category = ...
final Subcategory subcategory = ...
final LocalDate dueBy = ...
final BigDecimal cost = ...
final Location location = ...

toDoItem = sudoService.sudo(username,
        new Callable<ToDoItem>() {
            @Override
            public ToDoItem call() {
                final ToDoItem toDoItem = wrap(toDoItems).newToDo(
                        description, category, subcategory, dueBy, cost);
                wrap(toDoItem).setLocation(location);
                return toDoItem;
            }
        });
```

Behind the scenes the `SudoService` simply talks to the `DomainObjectContainer` to override the user returned by the `getUser()` API. It is possible to override both users and roles.