Beyond the Basics

Table of Contents

Chapter 1. Beyond the Basics

This guide provides more advanced guidance on writing maintainable larger applications.

Later chapters discuss how to deploy your app, and discuss other ways in which you can extend or adapt the framework itself to your particular needs.

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
- DataNucleus object store
- Security
- Testing
- Beyond the Basics (this guide)

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. View Models

View models are a type of domain object (with state, behaviour etc) but where the state is *not* persisted into the JDO/DataNucleus-managed database, but is instead converted to/from a string memento, and held by the calling client. This opens up a number of more advanced use cases.

In this topic we'll explore those use cases, and learn the programming model and conventions to use view models in your application.

2.1. Use Cases

When developing an Apache Isis application you will most likely start off with the persistent domain entities: `Customer`, `Order`, `Product`, and so on. For some applications this may well suffice. However, if the application needs to integrate with other systems, or if the application needs to support reasonably complex business processes, then you may need to look beyond just domain entities. This section explores these use cases.

2.1.1. Externally-managed entities

Sometimes the entities that make up your application are persisted not in the local JDO/DataNucleus database but reside in some other system, for example accessible only through a SOAP web service. Logically that data might still be considered a domain entity and we might want to associate behaviour with it, however it cannot be modelled as a domain entity if only because JDO/DataNucleus doesn't know about the entity nor how to retrieve or update it.

There are a couple of ways around this: we could either replicate the data somehow from the external system into the Isis-managed database (in which case it is once again just another domain entity), or we could set up a stub/proxy for the externally managed entity. This proxy would hold the reference to the externally-managed domain entity (eg an external id), as well as the "smarts" to know how to interact with that entity (by making SOAP web service calls etc).

The stub/proxy is a type of view model: a view - if you like - onto the domain entity managed by the external system.

DataNucleus does in fact define its own Store Manager extension point, so an alternative architecture would be to implement this interface such that DataNucleus could make the calls to the external system; these externally-persisted domain entities would therefore be modelled as regular `@PersistenceCapable` entities after all. For entities not persisted externally the implementation would delegate down to the default RDBMS-specific `StoreManager` provided by DataNucleus itself.

An implementation that supported only reading from an external entity ought to be comparatively straight-forward, but implementing one that also supported updating external entities would need to carefully consider error conditions if the external system is unavailable; distributed transactions are most likely difficult/impossible to implement (and not desirable in any case).

2.1.2. In-memory entities

As a variation on the above, sometimes there are domain objects that are, conceptually at least entities, but whose state is not actually persisted anywhere, merely held in-memory (eg in a hash).

A simple example might be read-only configuration data that is read from a config file (eg log4j appender definitions) but thereafter is presented in the UI just like any other entity.

2.1.3. Application-layer view models

Domain entities (whether locally persisted using JDO/DataNucleus or managed externally) are the bread-and-butter of Apache Isis applications: the focus after all, should be on the business domain concepts and ensuring that they are solid. Generally those domain entities will make sense to the business domain experts: they form the *ubiquitous language* of the domain. These domain entities are part of the domain layer.

That said, it may not always be practical to expect end-users of the application to interact solely with those domain entities. For example, it may be useful to show a dashboard of the most significant data in the system to a user, often pulling in and aggregating information from multiple points of the app. Obtaining this information by hand (by querying the respective services/repositories) would be tedious and slow; far better to have a dashboard do the job for the end user.

A dashboard object is a model of the most relevant state to the end-user, in other words it is (quite literally) a view model. It is not a persisted entity, instead it belongs to the application layer.

A view model need not merely aggregate data; it could also provide actions of its own. Most likely these actions will be queries and will always ultimately just delegate down to the appropriate domain-layer service/repository. But in some cases such view model actions might also modify state of underlying domain entities.

Another common use for view models is to help co-ordinate complex business processes; for example to perform a quarterly invoicing run, or to upload annual interest rates from an Excel spreadsheet. In these cases the view model might have some state of its own, but in most cases that state does not need to be persisted per se.

Desire Lines

One way to think of application view models is as modelling the "desire line": the commonly-trod path that end-users must follow to get from point A to point B as quickly as possible.

To explain: there are documented examples that architects of university campus will only add in paths some while after the campus buildings are complete: let the pedestrians figure out the routes they want to take. The name we like best for this idea is "desire lines", though it has also been called a "desire path", "paving the path" or "paving the sidewalk".

What that means is you should add view models *after* having built up the domain layer, rather than before. These view models pave that commonly-trod path, automating the steps that the end-user would otherwise have to do by hand.

It takes a little practice though, because even when building the domain layer "first", you should still bear in mind what the use cases are that those domain entities are trying to support. You certainly *shouldn't* try to build out a domain layer that could support every conceivable use case before starting to think about view models.

Instead, you should iterate. Identify the use case/story/end-user objective that you will deliver value to the business. Then build out the minimum domain entities to support that use case (refining the ubiquitous language as you go). Then, identify if there any view models that could be introduced which would simplify the end-user interactions with the system (perhaps automating several related use cases together).

2.1.4. DTOs

DTOs (data transfer objects) are simple classes that (according to wikipedia) "carry data between processes".

If those two processes are parts of the same overall application (the same team builds and deploys both server and client) then there's generally no need to define a DTO; just access the entities using Apache Isis' RestfulObjects viewer.

On the other hand, if the client consuming the DTO is a different application — by which we mean developed/deployed by a different (possible third-party) team — then the DTOs act as a formal contract between the provider and the consumer. In such cases, exposing domain entities over RestfulObjects would be "A Bad Thing"™ because the consumer would in effect have access to implementation details that could then not be easily changed by the producer.

To support this use case, a view model can be defined such that it can act as a DTO. This is done by annotating the class using JAXB annotations; this allows the consumer to obtain the DTO in XML format along with a corresponding XSD schema describing the structure of that XML. A discussion of how that might be done using an ESB such as Apache Camel™ follows below.

In case it's not obvious, these DTOs are still usable as "regular" view models; they will render in the Wicket viewer just like any other. In fact (as the programming model section below makes clear), these JAXB-annotated view models are in many regards the most powerful of all the alternative

ways of writing view models.

It's also worth noting that it is also possible to download the XML (or XSD) straight from the UI, useful during development. The view model simply needs to implement the `Dto` marker interface; the framework has mixins that contribute the download actions to the view model.

DTO Consumers

The actual consumers of DTOs will generally obtain the XML of the view models either by requesting the XML directly, eg using the RestfulObjects viewer, or may have the XML sent to them asynchronously using an ESB such as Apache Camel.

In the former case, the consumer requests the DTO by calling the REST API with the appropriate HTTP `Accept` header. An appropriate implementation of `ContentMappingService` can then be used to return the appropriate DTO (as XML).

For the latter case, one design is simply for the application to instantiate the view model, then call the `JaxbService` to obtain its corresponding XML. This can then be published onto the ESB, for example using an Apache ActiveMQ ™ queue.

However, rather than try to push all the data that might be needed by any of these external systems in a single XML event (which would require anticipating all the requirements, likely a hopeless task), a better design is to publish only the fact that something of note has changed - ie, that an action on a domain object has been invoked - and then let the consumers call back to obtain other information if required. This can once again be done by calling the REST API with an appropriate HTTP `Accept` header.

This is an example of the VETRO pattern (validate, enrich, transform, route, operate). In our case we focus on the validation (to determine the nature of the inbound message, ie which action was invoked), and the enrich (callback to obtain a DTO with additional information required by the consumer).

The (non-ASF) Isis addons' publishmq module provides an out-of-the-box solution of this design. It provides an implementation of the `PublishingService`, but which simply publishes instances of `ActionInvocationMemento` to an ActiveMQ queue. Camel (or similar) can then be hooked up to consume these events from this queue, and use a processor to parse the action memento to determine what has changed on the source system. Thereafter, a subsequent Camel processor can then call back to the source - via the Restful Objects viewer - to enrich the message with additional details using a DTO.

2.2. Programming Model

So much for the theory; how should view models be implemented? Fundamentally all view models' state is serialized into a string memento; this memento is then held by the client (browser) in the form of a URL. As you might imagine, this URL can become quite long, but Apache Isis offers a mechanism (the `UrlEncodingService`) if it exceeds the maximum length for a URL (2083 characters). Also, of course, this string memento must only contain characters that it is valid for use within a URL.

While the underlying technique is the same irrespective of use case, the programming model provides various ways of defining a view model so that the original intent is not lost. They are:

Table 1. View model programming model

Use case	Code	Description
External entity	[source,java] ---- @DomainObject(nature=Nature.EXTERNAL_ENTITY) public class CustomerRecordOnSAP { … } ----	Annotated with `@DomainObject#nature()` and a nature of `EXTERNAL_ENTITY`, with memento derived automatically from the properties of the domain object. Collections are ignored, as are any properties annotated as not persisted.
In-memory entity	[source,java] ---- @DomainObject(nature=Nature.INMEMORY_ENTITY) public class Log4JAppender { … } ----	As preceding, but using a nature of `INMEMORY_ENTITY`.
Application view model	[source,java] ---- @DomainObject(nature=Nature.VIEW_MODEL) public class Dashboard { … } ----	As preceding, but using a nature of `VIEW_MODEL`.
Application view model	[source,java] ---- @ViewModel public class Dashboard { … } ----	Annotated with `@ViewModel` annotation (effectively just an alias)' memento is as preceding: from "persisted" properties, collections ignored
Application view model	[source,java] ---- public class ExcelUploadManager implements ViewModel { public String viewModelMemento() { … } public void viewModelInit(String memento) { … } }	Implement `ViewModel` interface. The memento is as defined by the interface's methods: the programmer has full control (but also full responsibility) for the string memento.

Use case	Code	Description
DTO	[source,java] ---- @XmlRootElement("customer") public class CustomerDto { … } ----	Annotate using JAXB <code>@XmlRootElement</code> annotation. Memento derived automatically by serializing the XML graph as implied by the JAXB annotations. Note that (unlike <code>@ViewModel</code> et al) this state _can include collections.

JAXB-annotated DTOs are discussed in more detail immediately below.

2.3. JAXB-annotated DTOs

As noted in the introduction, view models can also be defined using JAXB annotations. The serialized form of these view models is therefore XML, which also enables these view models to act as DTOs.

In case it's not obvious, these DTOs are still usable as "regular" view models; they will render in the Wicket viewer just like any other. In fact, these JAXB-annotated view models are in many regards the most powerful of all the various ways of writing view models:

- their entire state (collections as well as properties) is automatically managed from interaction to interaction.

  In contrast, using `@ViewModel` (or its `@DomainObject#nature()` equivalent) will only manage the state of properties, but not collections. And if using the `ViewModel` interface, then the programmer must write all the state management (lots of boilerplate).

- JAXB-annotated view models are editable.

The examples in this section uses the DTO for `ToDoItem`, taken from the (non-ASF) Isis addons' todoapp. This DTO is defined as follows:

```
package todoapp.app.viewmodels.todoitem.v1;                          ①
@XmlRootElement(name = "toDoItemDto")                                ②
@XmlType(
        propOrder = {                                                ③
            "majorVersion", "minorVersion",
            "description", "category", ...
            "toDoItem", "similarItems"
        }
)
@DomainObjectLayout(
        titleUiEvent = TitleUiEvent.Doop.class                       ④
)
public class ToDoItemV1_1 implements Dto {                           ⑤
    @XmlElement(required = true, defaultValue = "1")                 ⑥
    public final String getMajorVersion() { return "1"; }
    @XmlElement(required = true, defaultValue = "1")                 ⑦
    public String getMinorVersion() { return "1"; }

    @XmlElement(required = true)                                     ⑧
    @Getter @Setter
    protected String description;
    @XmlElement(required = true)
    @Getter @Setter
    protected String category;
    ...

    @Getter @Setter                                                  ⑨
    protected ToDoItem toDoItem;
    @XmlElementWrapper                                               ⑩
    @XmlElement(name = "todoItem")
    @Getter @Setter
    protected List<ToDoItem> similarItems = Lists.newArrayList();
}
```

① package name encodes major version; see discussion on versioning

② identifies this class as a view model and defines the root element for JAXB serialization

③ all properties in the class must be listed; (they can be ignored using `@XmlTransient`)

④ demonstrating use of UI events for a subscriber to provide the DTO's title; see `@DomainObjectLayout#titleUiEvent()`.

⑤ class name encodes (major and) minor version; see discussion on versioning

⑥ again, see discussion on versioning

⑦ again, see discussion on versioning

⑧ simple scalar properties

⑨ reference to a persistent entity; discussed below

⑩ reference to a collection of persistent entities; again discussed below

2.3.1. Referencing Domain Entities

It's quite common for view models to be "backed by" (be projections of) some underlying domain entity. The `ToDoItemDto` we've been using as the example in this section is an example: there is an underlying `ToDoItem` entity.

It wouldn't make sense to serialize out the state of a persistent entity: the point of a DTO is to act as a facade on top of the entity so that the implementation details (of the entity's structure) don't leak out to the consumer. However, the identity of the underlying entity can be well defined; Apache Isis defines the Common schema which defines the `<oid-dto>` element (and corresponding `OidDto` class): the object's type and its identifier. This is basically a formal XML equivalent to the `Bookmark` object obtained from the `BookmarkService`.

There is only one requirement to make this work: every referenced domain entity must be annotated with `@XmlJavaTypeAdapter`, specifying the framework-provided `PersistentEntityAdapter.class`. This class is similar to the `BookmarkService`: it knows how to create an `OidDto` from an object reference.

Thus, in our view model we can legitimately write:

```
package todoapp.app.viewmodels.todoitem.v1;
...
public class ToDoItemV1_1 implements Dto {
    ...
    @Getter @Setter
    protected ToDoItem toDoItem;
}
```

All we need to do is remember to add that `@XmlJavaTypeAdapter` annotation to the referenced entity:

```
@XmlJavaTypeAdapter(PersistentEntityAdapter.class)
public class ToDoItem ...  {
    ...
}
```

It's also possible for a DTO to hold collections of objects. These can be of any type, either simple properties, or references to other objects. The only bit of boilerplate that is required is the `@XmlElementWrapper` annotation. This instructs JAXB to create an XML element (based on the field name) to contain each of the elements. (If this is omitted then the contents of the collection are at the same level as the properties; almost certainly not what is required).

For example, the DTO also contains:

```
package todoapp.app.viewmodels.todoitem.v1;
...
public class ToDoItemV1_1 implements Dto {
    ...
    @XmlElementWrapper
    @XmlElement(name = "todoItem")
    @Getter @Setter
    protected List<ToDoItem> similarItems = Lists.newArrayList();
}
```

There's nothing to prevent a JAXB DTO from containing rich graphs of data, parent containing children containing children. Be aware though that all of this state will become the DTO's memento, ultimately converted into a URL-safe form, by way of the `UrlEncodingService`.

There are limits to the lengths of URLs, however. Therefore the DTO should not include state that can easily be derived from other information. If the URL does exceed limits, then provide a custom implementation of `UrlEncodingService` to handle the memento string in some other fashion (eg substituting it with a GUID, with the memento cached somehow on the server).

2.3.2. Versioning

The whole point of using DTOs (in Apache Isis, at least) is to define a formal contact between two inter-operating but independent applications. Since the only thing we can predicate about the future with any certainty is that it one or both of these applications will change, we should version DTOs from the get-go. This allows us to make changes going forward without unnecessarily breaking existing consumers of the data.

There are several ways that versioning might be accomplished; we base our guidelines on this article taken from Roger Costello's blog, well worth a read.

We can distinguish two types of changes:

- backwardly compatible changes
- breaking changes.

We can immediately say that the XSD namespace should change only when there is a major/breaking change, if following semantic versioning that means when we bump the major version number v1, v2, etc.

XML namespaces correspond (when using JAXB) to Java packages. We should therefore place our DTOs in a package that contains only the major number; this package will eventually contain a range of DTOs that are intended to be backwardly compatible with one another. The package should also have a `package-info.java`; it is this that declares the XSD namespace:

```
@javax.xml.bind.annotation.XmlSchema(
        namespace = "http://viewmodels.app.todoapp/todoitem/v1/Dto.xsd",        ①
        xmlns = {
                @javax.xml.bind.annotation.XmlNs(
                        namespaceURI = "http://isis.apache.org/schema/common",
                        prefix = "com"
                )
        },
        elementFormDefault = javax.xml.bind.annotation.XmlNsForm.QUALIFIED
)
package todoapp.app.viewmodels.todoitem.v1;                                     ②
```

① the namespace URI, used by the DTO residing in this package.

② the package in which the DTO resides. Note that this contains only the major version.

Although there is no requirement for the namespace URI to correspond to a physical URL, it should be unique. This usually means including a company domain name within the string.

As noted above, this package will contain multiple DTO classes all with the same namespace; these represent a set of minor versions of the DTO, each subsequent one intended to be backwardly compatible with the previous. Since these DTO classes will all be in the same package (as per the advice above), the class should therefore include the minor version name:

```
package todoapp.app.viewmodels.todoitem.v1;     ①
...
public class ToDoItemV1_1 implements Dto {      ②
    ...
}
```

① package contains the major version only

② DTO class contains the (major and) minor version

We also recommend that each DTO instance should also specify the version of the XSD schema that it is logically compatible with. Probably most consumers will not persist the DTOs; they will be processed and then discarded. However, it would be wrong to assume that is the case in all cases; some consumers might choose to persist the DTO (eg for replay at some later state).

Thus:

```
public class ToDoItemV1_1 implements Dto {
    @XmlElement(required = true, defaultValue = "1")
    public final String getMajorVersion() { return "1"; }   ①
    @XmlElement(required = true, defaultValue = "1")
    public String getMinorVersion() { return "1"; }         ②
    ...
}
```

① returns the major version (in sync with the package)

② returns the minor version (in sync with the class name)

These methods always return a hard-coded literal. Any instances serialized from these classes will implicitly "declare" the (major and) minor version of the schema with which they are compatible. If a consumer has a minimum version that it requires, it can therefore inspect the XML instance itself to determine if it is able to consume said XML.

If a new (minor) version of a DTO is required, then we recommend copying-and-pasting the previous version, eg:

```
public class ToDoItemV1_2 implements Dto {
    @XmlElement(required = true, defaultValue = "1")
    public final String getMajorVersion() { return "1"; }
    @XmlElement(required = true, defaultValue = "2")
    public String getMinorVersion() { return "2"; }
    ...
}
```

Obviously, only changes made must be backward compatible, eg new members must be optional.

Alternatively, you might also consider simply editing the source file, ie renaming the class and bumping up the value returned by `getMinorVersion()`.

We also *don't* recommend using inheritance (ie `ToDoItemV1_2` should not inherit from `ToDoItemV1_1`; this creates unnecessary complexity downstream if generating XSDs and DTOs for the downstream consumer.

2.3.3. Generating XSDs and DTOs

In the section above it was explained how a view model DTO can transparent reference any "backing" entities; these references are converted to internal object identifiers.

However, if the consumer of the XML is another Java process (eg running within an Apache Camel route), then you might be tempted/expect to be able to use the same DTO within that Java process. After a little thought though you'll realize that (duh!) of course you cannot; the consumer runs in a different process space, and will not have references to those containing entities.

There are therefore two options:

- either choose not to have the view model DTO reference any persistent entities, and simply limit the DTO to simple scalars.

  Such a DTO will then be usable in both the Apache Isis app (to generate the original XML) and in the consumer. The `BookmarkService` can be used to obtain the object identifiers

- alternatively, generate a different DTO for the consumer from the XSD of the view model DTO.

The (non-ASF) Isis addons' todoapp uses the second approach; generating the XSD and consumer's

DTO is mostly just boilerplate `pom.xml` file. In the todoapp this can be found in the `todoapp-xsd` Maven module, whose `pom.xml` is structured as two profiles:

```
<project ... >
    <artifactId>todoapp-xsd</artifactId>
    <dependencies>
        <dependency>
            <groupId>${project.groupId}</groupId>
            <artifactId>todoapp-app</artifactId>
        </dependency>
    </dependencies>
    <profiles>
        <profile>
            <id>isis-xsd</id>
            <activation>
                <property>
                    <name>!skip.isis-xsd</name>
                </property>
            </activation>
            ...
        </profile>
        <profile>
            <id>xjc</id>
            <activation>
                <property>
                    <name>!skip.xjc</name>
                </property>
            </activation>
            ...
        </profile>
    </profiles>
</project>
```

The `isis-xsd` profile generates the XSD using the `xsd` goal of Isis' maven plugin:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.isis.tool</groupId>
            <artifactId>isis-maven-plugin</artifactId>
            <version>${isis.version}</version>
            <configuration>
                <appManifest>todoapp.dom.ToDoAppDomManifest</appManifest>
                <jaxbClasses>
                    <jaxbClass>
todoapp.app.viewmodels.todoitem.v1.ToDoItemV1_1</jaxbClass>
                </jaxbClasses>
                <separate>false</separate>
                <commonSchemas>false</commonSchemas>
```

```
            </configuration>
            <dependencies>
                <dependency>
                    <groupId>${project.groupId}</groupId>
                    <artifactId>todoapp-dom</artifactId>
                    <version>${project.version}</version>
                </dependency>
                <dependency>
                    <groupId>com.google.guava</groupId>
                    <artifactId>guava</artifactId>
                    <version>16.0.1</version>
                </dependency>
            </dependencies>
            <executions>
                <execution>
                    <phase>generate-sources</phase>
                    <goals>
                        <goal>xsd</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
        <plugin>
            <artifactId>maven-assembly-plugin</artifactId>
            <version>2.5.3</version>
            <configuration>
                <descriptor>src/assembly/dep.xml</descriptor>
            </configuration>
            <executions>
                <execution>
                    <id>create-archive</id>
                    <phase>package</phase>
                    <goals>
                        <goal>single</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

The `todoapp.dom.ToDoAppDomManifest` is a cut-down version of the app manifest that identifies only the `dom` domain services.

The `xjc` profile, meanwhile, uses the `maven-jaxb2-plugin` (a wrapper around the `schemagen` JDK tool) to generate a DTO from the XSD generated by the preceding profile:

```
<build>
    <plugins>
        <plugin>
```

```
            <groupId>org.jvnet.jaxb2.maven2</groupId>
            <artifactId>maven-jaxb2-plugin</artifactId>
            <version>0.12.3</version>
            <executions>
                <execution>
                    <id>xjc-generate</id>
                    <phase>generate-sources</phase>
                    <goals>
                        <goal>generate</goal>
                    </goals>
                </execution>
            </executions>
            <configuration>
                <removeOldOutput>true</removeOldOutput>
                <schemaDirectory>
                    target/generated-resources/isis-xsd/viewmodels.app.todoapp
                </schemaDirectory>
                <schemaIncludes>
                    <schemaInclude>todoitem/v1/Dto.xsd</schemaInclude>
                </schemaIncludes>
                <bindingDirectory>src/main/resources</bindingDirectory>
                <bindingIncludes>
                    <bindingInclude>binding.xml</bindingInclude>
                </bindingIncludes>
                <catalog>src/main/resources/catalog.xml</catalog>
            </configuration>
        </plugin>
        <plugin>
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>build-helper-maven-plugin</artifactId>
            <version>1.9.1</version>
            <executions>
                <execution>
                    <id>add-source</id>
                    <phase>generate-sources</phase>
                    <goals>
                        <goal>add-source</goal>
                    </goals>
                    <configuration>
                    <sources>
                        <source>target/generated-sources/xjc</source>
                    </sources>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

Chapter 3. Decoupling

We use Maven modules as a way to group related domain objects together; we can then reason about all the classes in that module as a single unit. By convention there will be a single top-level package corresponding to the module.

This section describes how to use Apache Isis' features to ensure that your domain application remains decoupled. The techniques described here are also the ones that have been adopted by the various Isis Addons modules (not ASF) for security, commands, auditing etc.

The following sections describe how to re-assemble an application, in particular where some modules are in-house but others are potentially third-party (eg the Isis Addons modules).

There is some overlap with Java 9's Jigsaw concepts of "module"; in the future we expect to refactor Apache Isis to build on top of this module system.

3.1. Database Schemas

In the same way that Java packages act as a namespace for domain objects, it's good practice to map domain entities to their own (database) schemas. As of 1.9.0, all the Isis Addons (non-ASF) modules do this, for example:

```
@javax.jdo.annotations.PersistenceCapable( ...
        schema = "isissecurity",
        table = "ApplicationUser")
public class ApplicationUser ... { ... }
```

results in a `CREATE TABLE` statement of:

```
CREATE TABLE isissecurity."ApplicationUser" (
    ...
)
```

while:

```
@javax.jdo.annotations.PersistenceCapable( ...
        schema = "isisaudit",
        table="AuditEntry")
public class AuditEntry ... { ... }
```

similarly results in:

```
CREATE TABLE isisaudit."AuditEntry" (
    ...
)
```

If for some reason you don't want to use schemas (though we strongly recommend that you do), then note that you can override the `@PersistenceCapable` annotation by providing XML metadata (the `mappings.jdo` file); see the section on configuring DataNucleus Overriding Annotations for more details.

3.1.1. Listener to create schema

JDO/DataNucleus does not automatically create these schema objects, but it *does* provide a listener callback API on the initialization of each class into the JDO metamodel.

Actually, the above statement isn't quite true. In DN 3.2.x (as used by Apache Isis up to v1.8.0) there was no support for schemas. As of Apache Isis 1.9.0 and DN 4.0 there is now support. But we implemented this feature initially against DN 3.2.x, and it still works, so for now we've decided to leave it in.

Therefore Apache Isis attaches a listener, `CreateSchemaObjectFromClassMetadata`, that checks for the schema's existence, and creates the schema if required.

The guts of its implementation is:

```
public class CreateSchemaObjectFromClassMetadata
        implements MetaDataListener,
                   DataNucleusPropertiesAware {
    @Override
    public void loaded(final AbstractClassMetaData cmd) { ... }

    protected String buildSqlToCheck(final AbstractClassMetaData cmd) {
        final String schemaName = schemaNameFor(cmd);
        return String.format(
            "SELECT count(*) FROM INFORMATION_SCHEMA.SCHEMATA where SCHEMA_NAME =
'%s'", schemaName);
    }
    protected String buildSqlToExec(final AbstractClassMetaData cmd) {
        final String schemaName = schemaNameFor(cmd);
        return String.format("CREATE SCHEMA \"%s\"", schemaName);
    }
}
```

where `MetaDataListener` is the DataNucleus listener API:

```
public interface MetaDataListener {
    void loaded(AbstractClassMetaData cmd);
}
```

Although not formal API, the default `CreateSchemaObjectFromClassMetadata` has been designed to be easily overrideable if you need to tweak it to support other RDBMS'. Any implementation must implement `org.datanucleus.metadata.MetaDataListener`:

The implementation provided has has been tested for HSQLDB, PostgreSQL and MS SQL Server, and is used automatically unless an alternative implementation is specified (as described in the section below).

3.1.2. Alternative implementation

An alternative implementation can be registered and used through the following configuration property:

```
isis.persistor.datanucleus.classMetadataLoadedListener=\

org.apache.isis.objectstore.jdo.datanucleus.CreateSchemaObjectFromClassMetadata
```

Because this pertains to the JDO Objectstore we suggest you put this configuration property in `WEB-INF/persistor_datanucleus.properties`; but putting it in `isis.properties` will also work.

Any implementation must implement `org.datanucleus.metadata.MetaDataListener`. In many cases simply subclassing from `CreateSchemaObjectFromClassMetadata` and overriding `buildSqlToCheck(…)` and `buildSqlToExec(…)` should suffice.

If you *do* need more control, your implementation can also optionally implement `org.apache.isis.objectstore.jdo.datanucleus.DataNucleusPropertiesAware`:

```
public interface DataNucleusPropertiesAware {
    public void setDataNucleusProperties(final Map<String, String> properties);
}
```

This provides access to the properties passed through to JDO/DataNucleus.

If you do extend Apache Isis' `CreateSchemaObjectFromClassMetadata` class for some other database, please contribute back your improvements.

3.2. Mixins

A mixin object allows one class to contribute behaviour - actions, (derived) properties and (derived) collections - to another domain object, either a domain entity or view model.

Some programming languages use the term "trait" instead of mixin, and some languages (such as

AspectJ) define their own syntax for defining such constructs. In Apache Isis a mixin is very similar to a domain service, however it also defines a single 1-arg constructor that defines the type of the domain objects that it contributes to.

Why do this? Two reasons:

- The main reason is to allow the app to be decoupled, so that it doesn't degrade into the proverbial "big ball of mud". Mixins (and contributions) allow dependency to be inverted, so that the dependencies between modules can be kept acyclic and under control.
- However, there is another reason: mixins are also a convenient mechanism for grouping functionality even for a concrete type, helping to rationalize about the dependency between the data and the behaviour.

Both use cases are discussed below.

Syntactically, a mixin is defined using either the `@Mixin` annotation or using `@DomainObject#nature()` attribute (specifying a nature of `Nature.MIXIN`).

3.2.1. Contributed Collection

The example below shows how to contribute a collection:

```
@Mixin
public class DocumentHolderDocuments {

    private final DocumentHolder holder;
    public DocumentHolderDocuments(DocumentHolder holder) { this.holder = holder; }

    @Action(semantics=SemanticsOf.SAFE)                          ①
    @ActionLayout(contributed = Contributed.AS_ASSOCIATION)      ②
    @CollectionLayout(render = RenderType.EAGERLY)
    public List<Document> documents() {                          ③
        ...
    }
    public boolean hideDocuments() { ... }                       ④
}
```

① required; actions that have side-effects cannot be contributed as collections

② required; otherwise the mixin will default to being rendered as an action

③ must accept no arguments. The mixin is a collection rather than a property because the return type is a collection, not a scalar.

④ supporting methods follow the usual naming conventions. (That said, in the case of collections, because the collection is derived/read-only, the only supporting method that is relevant is `hideXxx()`).

The above will result in a contributed collection for all types that implement/extend from `DocumentHolder` (so is probably for a mixin across modules).

3.2.2. Contributed Property

Contributed properties are defined similarly, for example:

```
@Mixin
public class DocumentHolderMostRecentDocument {

    private final DocumentHolder holder;
    public DocumentHolderDocuments(DocumentHolder holder) { this.holder = holder; }

    @Action(semantics=SemanticsOf.SAFE)                          ①
    @ActionLayout(contributed = Contributed.AS_ASSOCIATION)      ②
    public Document> mostRecentDocument() {                      ③
        ...
    }
    public boolean hideMostRecentDocument() { ... }              ④
}
```

① required; actions that have side-effects cannot be contributed as collections

② required; otherwise the mixin will default to being rendered as an action

③ must accept no arguments. The mixin is a property rather than a collection because the return type is a scalar.

④ supporting methods follow the usual naming conventions. (That said, in the case of properties, because the property is derived/read-only, the only supporting method that is relevant is `hideXxx()`).

3.2.3. Contributed Action

Contributed properties are defined similarly, for example:

```
@Mixin
public class DocumentHolderAddDocument {

    private final DocumentHolder holder;
    public DocumentHolderDocuments(DocumentHolder holder) { this.holder = holder; }

    @Action()
    @ActionLayout(contributed = Contributed.AS_ACTION)           ①
    public Document> addDocument(Document doc) {
        ...
    }
    public boolean hideAddDocument() { ... }                     ②
}
```

① recommended

② supporting methods follow the usual naming conventions.

3.2.4. Inferred Name

Where the mixin follows the naming convention `SomeType_mixinName` then the method name can be abbreviated to "$$". The mixin name is everything after the last '_'.

For example:

```
@Mixin
public class DocumentHolder_documents {

    private final DocumentHolder holder;
    public DocumentHolder_documents(DocumentHolder holder) { this.holder = holder; }

    @Action(semantics=SemanticsOf.SAFE)
    @ActionLayout(contributed = Contributed.AS_ASSOCIATION)
    @CollectionLayout(render = RenderType.EAGERLY)
    public List<Document> $$() {                                    ①
        ...
    }
    public boolean hide$$() { ... }                                 ②
}
```

① using "$$" as the reserved method name

② supporting methods as usual

The character "$" is also recognized as a separator between the mixin type and mixin name. This is useful for mixins implemented as nested static types, discussed below.

3.2.5. As Nested Static Classes

As noted in the introduction, while mixins were originally introduced as a means of allowing contributions from one module to the types of another module, they are also a convenient mechanism for grouping functionality/behaviour against a concrete type. All the methods and supporting methods end up in a single construct, and the dependency between that functionality and the rest of the object is made more explicit.

When using mixins in this fashion, it is idiomatic to write the mixin as a nested static class, using the naming convention described above to reduce duplication.

For example:

```
public class Customer {

    @Mixin
    public static class placeOrder {                                    ①

        private final Customer customer;
        public documents(Customer customer) { this.customer = customer; }   ②

        @Action
        @ActionLayout(contributed = Contributed.AS_ACTION)
        public List<Order> $$(Product p, int quantity) {                ③
            ...
        }
        public boolean hide$$() { ... }                                 ④
        public String validate0$$(Product p) { ...  }
    }
}
```

① Prior to 1.13.2, had to be prefixed by an "_"; this is no longer required because "$" is also recognized as a way of parsing the class name in order to infer the mixin's name (eg `Customer$placeOrder`).

② typically contributed to concrete class

③ using the "$$" reserved name

④ supporting methods as usual

Moreover, the mixin class can be capitalized if desired. Thus:

```
public class Customer {

    @Mixin
    public static class PlaceOrder {                                    ①

        private final Customer customer;
        public documents(Customer customer) { this.customer = customer; }   ②

        @Action
        @ActionLayout(contributed = Contributed.AS_ACTION)
        public List<Order> $$(Product p, int quantity) {                ③
            ...
        }
        public boolean hide$$() { ... }                                 ④
        public String validate0$$(Product p) { ...  }
    }
}
```

In other words, all of the following are allowed:

- `public static class Documents { … }`
- `public static class documents { … }`
- `public static class _Documents { … }`
- `public static class _documents { … }`

The reserved method name "$$" can also be changed using `@Mixin#method()` or `@DomainObject#mixinMethod()`.

3.2.6. Programmatic usage

When a domain object is rendered, the framework will automatically instantiate all required mixins and delegate to them dynamically. If writing integration tests or fixtures, or (sometimes) just regular domain logic, then you may need to instantiate mixins directly.

For this you can use the xref:rgsvc.adoc#_rgsvc_api_DomainObjectContainer_object-creation-api[`DomainObjectContainer#mixin(…)` method. For example:

```
DocumentHolder_documents mixin = container.mixin(DocumentHolder_documents.class,
customer);
```

The `IntegrationTestAbstract` and `FixtureScript` classes both provide a `mixin(…)` convenience method.

3.2.7. Other reasons to use mixins

In the introduction to this topic we mentioned that mixins are most useful for ensuring that the domain app remains decoupled. This applies to the case where the contributee (eg `Customer`, being mixed into) is in one module, while the contributor mixin (`DocumentHolder_documents`) is in some other module. The `customer` module knows about the `document` module, but not vice versa.

However, you might also want to consider moving behaviour out of entities even within the same module, perhaps even within the same Java package. And the reason for this is to support hot-reloading of Java classes, so that you can modify and recompile your application without having to restart it. This can provide substantial productivity gains.

The Hotspot JVM has limited support for hot reloading; generally you can change method implementations but you cannot introduce new methods. However, the DCEVM open source project will patch the JVM to support much more complete hot reloading support. There are also, of course, commercial products such as JRebel.

The main snag in all this is the DataNucleus enhancer… any change to entities is going to require the entity to be re-enhanced, and the JDO metamodel recreated, which usually "stuffs things up". So hot-reloading of an app whose fundamental structure is changing is likely to remain a no-no.

However, chances are that the structure of your domain objects (the data) will change much less rapidly than the behaviour of those domain objects. Thus, it's the behaviour that you're most likely wanting to change while the app is still running. If you move that behaviour out into mixins (or contributed services), then these can be reloaded happily. (When running in prototype mode),

Apache Isis will automatically recreate the portion of the metamodel for any domain object as it is rendered.

3.2.8. Related reading

Mixins are an implementation of the DCI architecture architecture, as formulated and described by Trygve Reenskaug and Jim Coplien. Reenskaug was the inventor of the MVC pattern (and also the external examiner for Richard Pawson's PhD thesis), while Coplien has a long history in object-orientation, C++ and patterns.

DCI stands for Data-Context-Interaction and is presented as an evolution of object-oriented programming, but one where behaviour is bound to objects dynamically rather than statically in some context or other. The `@Mixin` pattern is Apache Isis' straightforward take on the same basic concept.

You might also wish to check out Apache Zest (formerly Qi4J), which implements a much more general purpose implementation of the same concepts.

3.3. Contributions

Contributed services provide many of the same benefits as mixins; indeed mixins are an evolution and refinement of the contributions concept.

It's possible that contributions may be deprecated and eventually removed in a future version of the framework, to be replaced entirely by mixins.

The main difference between contributed services and mixins is that the actions of a contributed service will contribute to *all* the parameters of its actions, whereas a mixin only contributes to the type accepted in its constructor. Also, contributed services are long-lived singletons, whereas mixins are instantiated as required (by the framework) and then discarded almost immediately.

There's further useful information on contributed services in the reference guide, discussing the @DomainService#nature() attribute, for the `NatureOfService.VIEW_CONTRIBUTIONS_ONLY` nature.

3.3.1. Syntax

Any n-parameter action provided by a service will automatically be contributed to the list of actions for each of its (entity) parameters. From the viewpoint of the entity the action is called a contributed action.

For example, given a service:

```
public interface Library {
    public Loan borrow(Loanable l, Borrower b);
}
```

and the entities:

```
public class Book implements Loanable { ... }
```

and

```
public class LibraryMember implements Borrower { ... }
```

then the `borrow(…)` action will be contributed to both `Book` and to `LibraryMember`.

This is an important capability because it helps to decouple the concrete classes from the services.

If necessary, though, this behaviour can be suppressed by annotating the service action with `@org.apache.isis.applib.annotations.NotContributed`.

For example:

```
public interface Library {
    @NotContributed
    public Loan borrow(Loanable l, Borrower b);
}
```

If annotated at the interface level, then the annotation will be inherited by every concrete class. Alternatively the annotation can be applied at the implementation class level and only apply to that particular implementation.

Note that an action annotated as being `@NotContributed` will still appear in the service menu for the service. If an action should neither be contributed nor appear in service menu items, then simply annotate it as `@Hidden`.

3.3.2. Contributed Action

TODO

3.3.3. Contributed Property

TODO

3.3.4. Contributed Collection

TODO

3.4. Vetoing Visibility

TODO - a write-up of the "vetoing subscriber" design pattern, eg as described in the `BookmarkService`

eg if included an addon such as auditing or security.

solution is to write a domain event subscriber that vetoes the visibility

All the addons actions inherit from common base classes so this can be as broad-brush or fine-grained as required

3.5. Event Bus

TODO - see `EventBusService`, `@Action#domainEvent()`, `@Property#domainEvent()`, `@Collection#domainEvent()`, `WrapperFactory`.

3.6. Pushing Changes

This technique is much less powerful than using the event bus . We present it mostly for completeness.

3.6.1. When a property is changed

If you want to invoke functionality whenever a property is changed by the user, then you should create a supporting `modifyXxx()` method and include the functionality within that. The syntax is:

```
public void modifyPropertyName(PropertyType param) { ... }
```

Why not just put this functionality in the setter? Well, the setter is used by the object store to recreate the state of an already persisted object. Putting additional behaviour in the setter would cause it to be triggered incorrectly.

For example:

```
public class Order() {
    public Integer getAmount() { ... }
    public void setAmount(Integer amount) { ... }
    public void modifyAmount(Integer amount) { ①
        setAmount(amount);  ③
        addToTotal(amount); ②
    }
    ...
}
```

① The `modifyAmount()` method calls ...

② ... the `addToTotal()` (not shown) to maintain some running total.

We don't want this `addToCall()` method to be called when pulling the object back from the object store, so we put it into the modify, not the setter.

You may optionally also specify a `clearXxx()` which works the same way as modify `modify Xxx()` but is called when the property is cleared by the user (i.e. the current value replaced by nothing). The syntax is:

```
public void clearPropertyName() { ... }
```

To extend the above example:

```
public class Order() {
    public Integer getAmount() { ... }
    public void setAmount(Integer amount) { ... }
    public void modifyAmount(Integer amount) { ... }
    public void clearAmount() {
        removeFromTotal(this.amount);
        setAmount(null);
    }
    ...
}
```

3.6.2. When a collection is modified

A collection may have a corresponding `addToXxx()` and/or `removeFromXxx()` method. If present, and direct manipulation of the contents of the connection has not been disabled (see ?), then they will be called (instead of adding/removing an object directly to the collection returned by the accessor).

The reason for this behaviour is to allow other behaviour to be triggered when the contents of the collection is altered. That is, it is directly equivalent to the supporting `modifyXxx()` and `clearXxx()` methods for properties (see ?).

The syntax is:

```
public void addTo<CollectionName>(EntityType param) { ... }
```

and

```
public void removeFromCollectionName(EntityType param) { ... }
```

where `EntityType` is the same type as the generic collection type.

For example:

```
public class Employee { ... }

public class Department {

    private int numMaleEmployees;                                ①
    private int numFemaleEmployees;                              ②

    private Set<Employee> employees = new TreeSet<Employee>();
    public Set<Employee> getEmployees() {
        return employees;
    }
    private void setEmployees(Set<Employee> employees) {
        this.employees = employees;
    }
    public void addToEmployees(Employee employee) {              ③
        numMaleEmployees += countOneMale(employee);
        numFemaleEmployees += countOneFemale(employee);
        employees.add(employee);
    }
    public void removeFromEmployees(Employee employee) {         ④
        numMaleEmployees -= countOneMale(employee);
        numFemaleEmployees -= countOneFemale(employee);
        employees.remove(employee);
    }
    private int countOneMale(Employee employee) { return employee.isMale()?1:0; }
    private int countOneFemale(Employee employee) { return employee.isFemale()?1:0; }

    ...
}
```

① maintain a count of the number of male ...

② ... and female employees (getters and setters omitted)

③ the `addTo···()` method increments the derived properties

④ the `removeFrom···()` method similarly decrements the derived properties

Chapter 4. i18n

Apache Isis' support for internationlization (i18n) allows every element of the domain model (the class names, property names, action names, parameter names and so forth) to be translated.

It also supports translations of messages raised imperatively, by which we mean as the result of a call to `title()` to obtain an object's title, or messages resulting from any business rule violations (eg `disable···()` or `validate···()`, and so on.

The Wicket viewer (that is, its labels and messages) is also internationalized using the same mechanism. If no translations are available, then the Wicket viewer falls back to using Wicket resource bundles.

Isis does not translate the values of your domain objects, though. So, if you have a domain concept such as `Country` whose name is intended to be localized according to the current user, you will need to model this yourself.

4.1. Implementation Approach

Most Java frameworks tackle i18n by using Java's own `ResourceBundle` API. However, there are some serious drawbacks in this approach, including:

- if a string appears more than once (eg "name" or "description") then it must be translated everywhere it appears in every resource bundle file
- there is no support for plural forms (see this SO answer)
- there is no tooling support for translators

Apache Isis therefore takes a different approach, drawing inspiration from GNU's gettext API and specifically its `.pot` and `.po` files. These are intended to be used as follows:

- the `.pot` (portable object template) file holds the message text to be translated
- this file is translated into multiple `.po` (portable object) files, one per supported locale
- these `.po` files are renamed according to their locale, and placed into the 'appropriate' location to be picked up by the runtime. The name of each `.po` resolved in a very similar way to resource bundles.

The format of the `.pot` and `.po` files is identical; the only difference is that the `.po` file has translations for each of the message strings. These message strings can also have singular and plural forms.

Although Apache Isis' implementation is modelled after GNU's API, it does *not* use any GNU software. This is for two reasons: (a) to simplify the toolchain/developer experience, and (b) because GNU software is usually GPL, which would be incompatible with the Apache license.

This design tackles all the issues of `ResourceBundle`s:

- the `.po` message format is such that any given message text to translate need only be translated once, even if it appears in multiple places in the application (eg "Name")
- the `.po` message format includes translations for (optional) plural form as well as singular form
- there are lots of freely available editors to be found, many summarized on this Drupal.org webpage.

  In fact, there are also online communities/platforms of translators to assist with translating files. One such is crowdin (nb: this link does not imply endorsement).

In Apache Isis' implementation, if the translation is missing from the `.po` file then the original message text from the `.pot` file will be returned. In fact, it isn't even necessary for there to be any `.po` files; `.po` translations can be added piecemeal as the need arises.

4.2. `TranslationService`

The cornerstone of Apache Isis' support for i18n is the `TranslationService` service. This is defined in the applib with the following API:

```
public interface TranslationService {
    public String translate(      ①
            final String context,
            final String text);
    public String translate(      ②
            final String context,
            final String singularText,
            final String pluralText,
            final int num);
    public enum Mode {
        READ,
        WRITE;
    }
    Mode getMode();               ③
}
```

① is to translate the singular form of the text

② is to translate the plural form of the text

③ indicates whether the translation service is in read or write mode.

The `translate(…)` methods are closely modelled on GNU's gettext API. The first version is used when no translation is required, the second is when both a singular and plural form will be required, with the `num` parameter being used to select which is returned. In both cases the `context` parameter provides some contextual information for the translator; this generally corresponds to the class member.

The mode meanwhile determines the behaviour of the service. More on this below.

4.2.1. `TranslationServicePo`

Isis provides a default implementation of `TranslationService`, namely `TranslationServicePo`.

If the service is running in the normal read mode, then it simply provides translations for the locale of the current user. This means locates the appropriate `.po` file (based on the requesting user's locale), finds the translation and returns it.

If however the service is configured to run in write mode, then it instead records the fact that the message was requested to be translated (a little like a spy/mock in unit testing mock), and returns the original message. The service can then be queried to discover which messages need to be translated. All requested translations are written into the `.pot` file.

To make the service as convenient as possible to use, the service configures itself as follows:

- if running in prototype mode deployment type or during integration tests, then the service runs in **write** mode, in which case it records all translations into the `.pot` file. The `.pot` file is written out when the system is shutdown.
- if running in server (production) mode deployment type, then the service runs in **read** mode. It is also possible to set a configuration setting in `isis.properties` to force read mode even if running in prototype mode (useful to manually test/demo the translations).

When running in write mode the original text is returned to the caller untranslated. If in read mode, then the translated `.po` files are read and translations provided as required.

4.3. Imperative messages

The `TranslationService` is used internally by Apache Isis when building up the metamodel; the name and description of every class, property, collection, action and action parameter is automatically translated. Thus the simple act of bootstrapping Apache Isis will cause most of the messages requiring translation (that is: those for the Apache Isis metamodel) to be captured by the `TranslationService`.

However, for an application to be fully internationalized, any validation messages (from either `disableXxx()` or `validateXxx()` supporting methods) and also possibly an object's title (from the `title()` method) will also require translation. Moreover, these messages must be captured in the `.pot` file such that they can be translated.

4.3.1. `TranslatableString`

The first part of the puzzle is tackled by an extension to Apache Isis' programming model. Whereas previously the `disableXxx()` / `validateXxx()` / `title()` methods could only return a `java.lang.String`, they may now optionally return a `TranslatableString` (defined in Isis applib) instead.

Here's a (silly) example from the SimpleApp archetype:

```
public TranslatableString validateUpdateName(final String name) {
    return name.contains("!")? TranslatableString.tr("Exclamation mark is not
allowed"): null;
}
```

This corresponds to the following entry in the `.pot` file:

```
#: dom.simple.SimpleObject#updateName()
msgid "Exclamation mark is not allowed"
msgstr ""
```

The full API of `TranslatableString` is modelled on the design of GNU gettext (in particular the gettext-commons library):

```
public final class TranslatableString {
    public static TranslatableString tr(       ①
            final String pattern,
            final Object... paramArgs) { ... }
    public static TranslatableString trn(      ②
            final String singularPattern,
            final String pluralPattern,
            final int number,
            final Object... paramArgs) { ... }
    public String translate(                   ③
            final TranslationService translationService,
            final String context) { ... }
}
```

① returns a translatable string with a single pattern for both singular and plural forms.

② returns a translatable string with different patterns for singular and plural forms; the one to use is determined by the 'number' argument

③ translates the string using the provided `TranslationService`, using the appropriate singular/regular or plural form, and interpolating any arguments.

The interpolation uses the format `{xxx}`, where the placeholder can occur multiple times.

For example:

```
final TranslatableString ts = TranslatableString.tr(
    "My name is {lastName}, {firstName} {lastName}.",
    "lastName", "Bond", "firstName", "James");
```

would interpolate (for the English locale) as "My name is Bond, James Bond".

For a German user, on the other hand, if the translation in the corresponding `.po` file was:

```
#: xxx.yyy.Whatever#context()
msgid "My name is {lastName}, {firstName} {lastName}."
msgstr "Ich heisse {firstName} {lastName}."
```

then the translation would be: "Ich heisse James Bond".

The same class is used in `DomainObjectContainer` so that you can raise translatable info, warning and error messages; each of the relevant methods are overloaded.

For example:

```
public interface DomainObjectContainer {
    void informUser(String message);
    void informUser(
        TranslatableMessage message,
        final Class<?> contextClass, final String contextMethod); ①
    ...
}
```

① are concatenated together to form the context for the `.pot` file.

4.3.2. `TranslatableException`

Another mechanism by which messages can be rendered to the user are as the result of exception messages thrown and recognized by an `ExceptionRecognizer`.

In this case, if the exception implements `TranslatableException`, then the message will automatically be translated before being rendered. The `TranslatableException` itself takes the form:

```
public interface TranslatableException {
    TranslatableString getTranslatableMessage(); ①
    String getTranslationContext();              ②
}
```

① the message to translate. If returns `null`, then the `Exception#getMessage()` is used as a fallback

② the context to use when translating the message

4.4. Wicket Viewer

The Wicket viewer (its labels and messages) is also internationalized using the `TranslationService`. This is done through an Isis-specific implementation of the Wicket framework's `org.apache.wicket.Localizer` class, namely `LocalizerForIsis`.

The Wicket `Localizer` defines the following API:

```
public String getString(
        final String key,                ①
        final Component component,       ②
        final IModel<?> model,
        final Locale locale,
        final String style,
        final String defaultValue)
    throws MissingResourceException { ... }
```

① The key to obtain the resource for

② The component to get the resource for (if any)

For example, `key` might be a value such as "okLabel", while `component` an internal class of the Wicket viewer, such as `EntityPropertiesForm`.

The `LocalizerForIsis` implementation uses the `key` as the `msgId`, while the fully qualified class name of the `component` is used as a context. There is one exception to this: if the component is the third-party select2 component (used for drop-downs), then that class name is used directly.

In the main, using Isis' i18n support means simply adding the appropriate translations to the `translation.po` file, for each locale that you require. If the translations are missing then the original translations from the Wicket resource bundles will be used instead.

4.4.1. Commonly used

Most of the translation requirements can be covered by adding in the following `msgId`s:

```
#: org.apache.isis.viewer.wicket.ui.pages.entity.EntityPage
msgid "CollectionContentsAsAjaxTablePanelFactory.Table"
msgstr "Table"

#: org.apache.isis.viewer.wicket.ui.pages.entity.EntityPage
msgid "CollectionContentsAsUnresolvedPanel.Hide"
msgstr "Hide"

#: org.apache.isis.viewer.wicket.ui.pages.entity.EntityPage
msgid "aboutLabel"
msgstr "About"

#: org.apache.isis.viewer.wicket.ui.pages.entity.EntityPage
msgid "cancelLabel"
msgstr "Cancel"

#: org.apache.isis.viewer.wicket.ui.pages.entity.EntityPage
msgid "datatable.no-records-found"
msgstr "No Records Found"

#: org.apache.isis.viewer.wicket.ui.pages.entity.EntityPage
msgid "editLabel"
```

```
msgstr "Edit"

#: org.wicketstuff.select2.Select2Choice
msgid "inputTooShortPlural"
msgstr "Please enter {number} more characters"

#: org.wicketstuff.select2.Select2Choice
msgid "inputTooShortSingular"
msgstr "Please enter 1 more character"

#: org.wicketstuff.select2.Select2Choice
msgid "loadMore"
msgstr "Load more"

#: org.apache.isis.viewer.wicket.ui.pages.entity.EntityPage
msgid "logoutLabel"
msgstr "Logout"

#: org.wicketstuff.select2.Select2Choice
msgid "noMatches"
msgstr "No matches"

#: org.apache.isis.viewer.wicket.ui.pages.entity.EntityPage
msgid "okLabel"
msgstr "OK"

#: org.wicketstuff.select2.Select2Choice
msgid "searching"
msgstr "Searching..."

#: org.wicketstuff.select2.Select2Choice
msgid "selectionTooBigPlural"
msgstr "You can only select {limit} items"

#: org.wicketstuff.select2.Select2Choice
msgid "selectionTooBigSingular"
msgstr "You can only select 1 item"
```

4.4.2. Login/self-sign-up

In addition, there are a reasonably large number of messages that are used for both login and the user registration (self sign-up) and password reset features.

These are:

```
#: org.apache.isis.viewer.wicket.ui.pages.login.WicketSignInPage
msgid "AutoLabel.CSS.required"
msgstr "Required"

#: org.apache.isis.viewer.wicket.ui.pages.accmngt.signup.RegistrationFormPage
```

```
#: org.apache.isis.viewer.wicket.ui.pages.accmngt.register.RegisterPage
#: org.apache.isis.viewer.wicket.ui.pages.accmngt.password_reset.PasswordResetPage
msgid "confirmPasswordLabel"
msgstr "Confirm password"

#: org.apache.isis.viewer.wicket.ui.pages.accmngt.signup.RegistrationFormPage
#: org.apache.isis.viewer.wicket.ui.pages.accmngt.register.RegisterPage
msgid "emailIsNotAvailable"
msgstr "The given email is already in use"

#: org.apache.isis.viewer.wicket.ui.pages.accmngt.password_reset.PasswordResetPage
msgid "emailPlaceholder"
msgstr "Enter your email"

#: org.apache.isis.viewer.wicket.ui.pages.accmngt.signup.RegistrationFormPage
#: org.apache.isis.viewer.wicket.ui.pages.accmngt.register.RegisterPage
msgid "emailPlaceholder"
msgstr "Enter an email for the new account"

#: org.apache.isis.viewer.wicket.ui.pages.accmngt.signup.RegistrationFormPage
#: org.apache.isis.viewer.wicket.ui.pages.accmngt.register.RegisterPage
#: org.apache.isis.viewer.wicket.ui.pages.accmngt.password_reset.PasswordResetPage
msgid "emailLabel"
msgstr "Email"

#: org.apache.isis.viewer.wicket.ui.pages.accmngt.signup.RegistrationFormPage
#: org.apache.isis.viewer.wicket.ui.pages.accmngt.register.RegisterPage
#: org.apache.isis.viewer.wicket.ui.pages.accmngt.password_reset.PasswordResetPage
msgid "emailSentMessage"
msgstr "An email has been sent to '${email}' for verification."

#: org.apache.isis.viewer.wicket.ui.pages.login.WicketSignInPage
msgid "forgotPasswordLinkLabel"
msgstr "Forgot your password?"

#: org.apache.isis.viewer.wicket.ui.pages.login.WicketSignInPage
msgid "loginHeader"
msgstr "Login"

#: org.apache.isis.viewer.wicket.ui.pages.accmngt.password_reset.PasswordResetPage
msgid "noSuchUserByEmail"
msgstr "There is no account with this email"

#: org.apache.isis.viewer.wicket.ui.pages.accmngt.password_reset.PasswordResetPage
msgid "noUserForAnEmailValidToken"
msgstr "The account seems to be either already deleted or has changed its email
address. Please try again."

#: org.apache.isis.viewer.wicket.ui.pages.accmngt.password_reset.PasswordResetPage
msgid "passwordChangeSuccessful"
msgstr "The password has been changed successfully. You can <a class=\"alert-success\"
```

```
style=\"text-decoration:underline;\" href=\"${signInUrl}\">login</a> now."

#: org.apache.isis.viewer.wicket.ui.pages.accmngt.password_reset.PasswordResetPage
msgid "passwordChangeUnsuccessful"
msgstr "There was a problem while updating the password. Please try again."

#: org.apache.isis.viewer.wicket.ui.pages.accmngt.signup.RegistrationFormPage
#: org.apache.isis.viewer.wicket.ui.pages.accmngt.register.RegisterPage
#: org.apache.isis.viewer.wicket.ui.pages.login.WicketSignInPage
#: org.apache.isis.viewer.wicket.ui.pages.accmngt.password_reset.PasswordResetPage
msgid "passwordLabel"
msgstr "Password"

#: org.apache.isis.viewer.wicket.ui.pages.accmngt.signup.RegistrationFormPage
#: org.apache.isis.viewer.wicket.ui.pages.accmngt.register.RegisterPage
#: org.apache.isis.viewer.wicket.ui.pages.login.WicketSignInPage
#: org.apache.isis.viewer.wicket.ui.pages.accmngt.password_reset.PasswordResetPage
msgid "passwordPlaceholder"
msgstr "Enter password"

#: org.apache.isis.viewer.wicket.ui.pages.accmngt.password_reset.PasswordResetPage
msgid "passwordResetExpiredOrInvalidToken"
msgstr "You are trying to reset the password for an expired or invalid token"

#: org.apache.isis.viewer.wicket.ui.pages.accmngt.password_reset.PasswordResetPage
msgid "passwordResetHeader"
msgstr "Forgot password"

#: org.apache.isis.viewer.wicket.ui.pages.accmngt.password_reset.PasswordResetPage
msgid "passwordResetSubmitLabel"
msgstr "Submit"

#: org.apache.isis.viewer.wicket.ui.pages.login.WicketSignInPage
msgid "registerButtonLabel"
msgstr "Register"

#: org.apache.isis.viewer.wicket.ui.pages.accmngt.register.RegisterPage
msgid "registerHeader"
msgstr "Register"

#: org.apache.isis.viewer.wicket.ui.pages.login.WicketSignInPage
msgid "rememberMeLabel"
msgstr "Remember Me"

#: org.apache.isis.viewer.wicket.ui.pages.login.WicketSignInPage
msgid "resetButtonLabel"
msgstr "Reset"

#: org.apache.isis.viewer.wicket.ui.pages.login.WicketSignInPage
msgid "signInButtonLabel"
msgstr "Sign in"
```

```
#: org.apache.isis.viewer.wicket.ui.pages.login.WicketSignInPage
msgid "signUpButtonLabel"
msgstr "Don't have an account? Sign up now."

#: org.apache.isis.viewer.wicket.ui.pages.accmngt.signup.RegistrationFormPage
#: org.apache.isis.viewer.wicket.ui.pages.accmngt.register.RegisterPage
#: org.apache.isis.viewer.wicket.ui.pages.accmngt.password_reset.PasswordResetPage
msgid "signUpButtonLabel"
msgstr "Verify email"

#: org.apache.isis.viewer.wicket.ui.pages.accmngt.signup.RegistrationFormPage
msgid "signUpHeader"
msgstr "Sign Up"

#: org.apache.isis.viewer.wicket.ui.pages.accmngt.signup.RegistrationFormPage
#: org.apache.isis.viewer.wicket.ui.pages.accmngt.register.RegisterPage
#: org.apache.isis.viewer.wicket.ui.pages.accmngt.password_reset.PasswordResetPage
msgid "usernameIsNotAvailable"
msgstr "The provided username is already in use"

#: org.apache.isis.viewer.wicket.ui.pages.accmngt.signup.RegistrationFormPage
#: org.apache.isis.viewer.wicket.ui.pages.accmngt.register.RegisterPage
#: org.apache.isis.viewer.wicket.ui.pages.login.WicketSignInPage
#: org.apache.isis.viewer.wicket.ui.pages.accmngt.password_reset.PasswordResetPage
msgid "usernameLabel"
msgstr "Username"

#: org.apache.isis.viewer.wicket.ui.pages.accmngt.signup.RegistrationFormPage
#: org.apache.isis.viewer.wicket.ui.pages.accmngt.register.RegisterPage
#: org.apache.isis.viewer.wicket.ui.pages.login.WicketSignInPage
#: org.apache.isis.viewer.wicket.ui.pages.accmngt.password_reset.PasswordResetPage
msgid "usernamePlaceholder"
msgstr "Username"
```

4.5. Integration Testing

So much for the API; but as noted, it is also necessary to ensure that the required translations are recorded (by the `TranslationService`) into the `.pot` file.

For this, we recommend that you ensure that all such methods are tested through an integration test (not unit test).

For example, here's the corresponding integration test for the "Exclamation mark" example from the simpleapp (above):

```
@Rule
public ExpectedException expectedException = ExpectedException.none();

@Inject
FixtureScripts fixtureScripts;

@Test
public void failsValidation() throws Exception {
    // given
    RecreateSimpleObjects fs = new RecreateSimpleObjects().setNumber(1);
    fixtureScripts.runFixtureScript(fs, null);
    SimpleObject simpleObjectWrapped = wrap(fs.getSimpleObjects().get(0));

    // expect
    expectedExceptions.expect(InvalidException.class);
    expectedExceptions.expectMessage("Exclamation mark is not allowed");

    // when
    simpleObjectWrapped.updateName("new name!");
}
```

Running this test will result in the framework calling the `validateUpdateName(…)` method, and thus to record that a translation is required in the `.pot` file.

When the integration tests are complete (that is, when Apache Isis is shutdown), the `TranslationServicePo` will write out all captured translations to its log (more on this below). This will include all the translations captured from the Apache Isis metamodel, along with all translations as exercised by the integration tests.

To ensure your app is fully internationalized app, you must therefore:

- use `TranslatableString` rather than `String` for all validation/disable and title methods.
- ensure that (at a minimum) all validation messages and title methods are integration tested.

We make no apologies for this requirement: one of the reasons that we decided to implement Apache Isis' i18n support in this way is because it encourages/requires the app to be properly tested.

Behind the scenes Apache Isis uses a JUnit rule (`ExceptionRecognizerTranslate`) to intercept any exceptions that are thrown. These are simply passed through to the registered `ExceptionRecognizer`s so that any messages are recorded as requiring translation.

4.6. Escaped strings

Translated messages can be escaped if required, eg to include embedded markup.

```
#: com.mycompany.myapp.OrderItem#quantity
msgid "<i>Quantity</i>"
msgstr "<i>Quantité</i>"
```

For this to work, the `namedEscaped()` attribute must be specified using either the dynamic layout json file, or using an annotation such as `@PropertyLayout` or `@ParameterLayout`.

For example:

```
@ParameterLayout(
    named="<i>Quantity</i>",      ①
    namedEscaped=false
)
public Integer getQuantity() { ... }
```

① required (even though it won't be used when a translation is read; otherwise the escaped flag is ignored)

4.7. Configuration

There are several different aspects of the translation service that can be configured.

4.7.1. Logging

To configure the `TranslationServicePo` to write to out the `translations.pot` file, add the following to the *integtests* `logging.properties` file:

```
log4j.appender.translations-po=org.apache.log4j.FileAppender
log4j.appender.translations-po.File=./translations.pot
log4j.appender.translations-po.Append=false
log4j.appender.translations-po.layout=org.apache.log4j.PatternLayout
log4j.appender.translations-po.layout.ConversionPattern=%m%n

log4j.logger.org.apache.isis.core.runtime.services.i18n.po.PoWriter=INFO,translations-
po
log4j.additivity.org.apache.isis.core.runtime.services.i18n.po.PotWriter=false
```

Just to repeat, this is *not* the `WEB-INF/logging.properties` file, it should instead be added to the `integtests/logging.properties` file.

4.7.2. Location of the `.po` files

The default location of the translated `.po` files is in the `WEB-INF` directory. These are named and searched for similarly to regular Java resource bundles.

For example, assuming these translations:

```
/WEB-INF/translations-en-US.po
        /translations-en.po
        /translations-fr-FR.po
        /translations.po
```

then:

- a user with `en-US` locale will use `translations-en-US.po`
- a user with `en-GB` locale will use `translations-en.po`
- a user with `fr-FR` locale will use `translations-fr-FR.po`
- a user with `fr-CA` locale will use `translations.po`

The basename for translation files is always `translations`; this cannot be altered.

4.7.3. Externalized translation files

Normally Apache Isis configuration files are read from the `WEB-INF` file. However, Apache Isis can be configured to read config files from an external directory; this is also supported for translations.

Thus, if in `web.xml` the external configuration directory has been set:

```
<context-param>
    <param-name>isis.config.dir</param-name>
    <param-value>location of external config directory</param-value>
</context-param>
```

Then this directory will be used as the base for searching for translations (rather than the default 'WEB-INF/' directory).

4.7.4. Force read mode

As noted above, if running in prototype mode then `TranslationServicePo` will be in write mode, if in production mode then will be in read mode. To force read mode (ie use translations) even if in prototype mode, add the following configuration property to `isis.properties`:

```
isis.services.translation.po.mode=read
```

4.8. Supporting services

The `TranslationServicePo` has a number of supporting/related services.

4.8.1. `LocaleProvider`

The `LocaleProvider` API is used by the `TranslationServicePo` implementation to obtain the locale of the "current user".

A default implementation is provided by the Wicket viewer.

Note that this default implementation does not support requests made through the Restful Objects viewer (there is no Wicket 'application' object available); the upshot is that requests through Restful Objects are never translated. Registering a different implementation of `LocaleProvider` that taps into appropriate REST (RestEasy?) APIs would be the way to address this.

4.8.2. `TranslationsResolver`

The `TranslationResolver` is used by the `TranslationService` implementation to lookup translations for a specified locale. It is this service that reads from the `WEB-INF/` (or externalized directory).

4.8.3. `TranslationServicePoMenu`

The `TranslationServicePoMenu` provides a couple of menu actions in the UI (prototype mode only) that interacts with the underlying `TranslationServicePo`:

- the `downloadTranslationsFile()` action - available only in write mode - allows the current `.pot` file to be downloaded.

  While this will contain all the translations from the metamodel, it will not necessarily contain all translations for all imperative methods returning `TranslatableString` instances; which are present and which are missing will depend on which imperative methods have been called (recorded by the service) prior to downloading.

- the `clearTranslationsCache()` action - available only in read mode - will clear the cache so that new translations can be loaded.

  This allows a translator to edit the appropriate `translations-xx-XX.po` file and check the translation is correct without having to restart the app.

Chapter 5. Headless access

This section tackles the topic of enabling access to an Isis application directly, or at least, not through either the Wicket or Restful viewers.

There are several main use-cases:

- enabling background execution, eg of a thread managed by Quartz scheduler and running within the webapp
- integration from other systems, eg for a subscriber on a pub/sub mechanism such as Camel, pushing changes through an Apache Isis domain model.
- leveraging an Isis application within a batch process

Note that the calling thread runs in the same process space as the Apache Isis domain object model (must be physically linked to the JAR files containing the domain classes). For use cases where the calling thread runs in some other process space (eg migrating data from a legacy system), then the Restful Objects viewer is usually the way to go.

The API described in this chapter is reasonably low-level, allowing code to interact very directly with the Apache Isis metamodel and runtime. Such callers should be considered trusted: they do not (by default) honour any business rules eg implicit in the Isis annotations or hide/disable/validate methods. However the `WrapperFactory` service could be used to enforce such business rules if required.

5.1. AbstractIsisSessionTemplate

The `AbstractIsisSessionTemplate` class (whose name is inspired by the Spring framework's naming convention for similar classes that query JDBC, JMS, JPA etc.) provides the mechanism to open up a 'session' within the Apache Isis framework, in order to resolve and interact with entities.

The class itself is intended to be subclassed:

```
public abstract class AbstractIsisSessionTemplate {

    public void execute(final AuthenticationSession authSession, final Object context)
{ ... } ①
    protected abstract void doExecute(Object context); ②

    protected ObjectAdapter adapterFor(final Object targetObject) { ... }
    protected ObjectAdapter adapterFor(final RootOid rootOid) { ... }

    protected PersistenceSession getPersistenceSession() { ... }
    protected IsisTransactionManager getTransactionManager() { ... }
    protected AdapterManager getAdapterManager() { ... }
}
```

① `execute(…)` sets up the `IsisSession` and delegates to ...

② `doExecute(…)`, the mandatory hook method for subclasses to implement. The passed object represents passes a context from the caller (eg the scheduler, cron job, JMS etc) that instantiated and executed the class.

The `protected` methods expose key internal APIs within Apache Isis, for the subclass to use as necessary.

One notable feature of `AbstractIsisSessionTemplate` is that it will automatically inject any domain services into itself. Thus, it is relatively easy for the subclass to "reach into" the domain, through injected repository services.

5.2. BackgroundCommandExecution

The `BackgroundCommandExecution` class (a subclass of AbstractIsisSessionTemplate) is intended to simplify the execution of background `Command`s persisted by way of the `CommandService` and the `BackgroundCommandService`.

Its signature is:

```
public abstract class BackgroundCommandExecution extends AbstractIsisSessionTemplate {
    protected void doExecute(Object context) { ... }
    protected abstract List<? extends Command> findBackgroundCommandsToExecute(); ①
}
```

① `findBackgroundCommandsToExecute()` is a mandatory hook method for subclasses to implement.

This allows for different implementations of the `CommandService` and `BackgroundCommandService` to persist to wherever.

The diagram below (yuml.me/363b335f) shows the dependencies between these various classes:

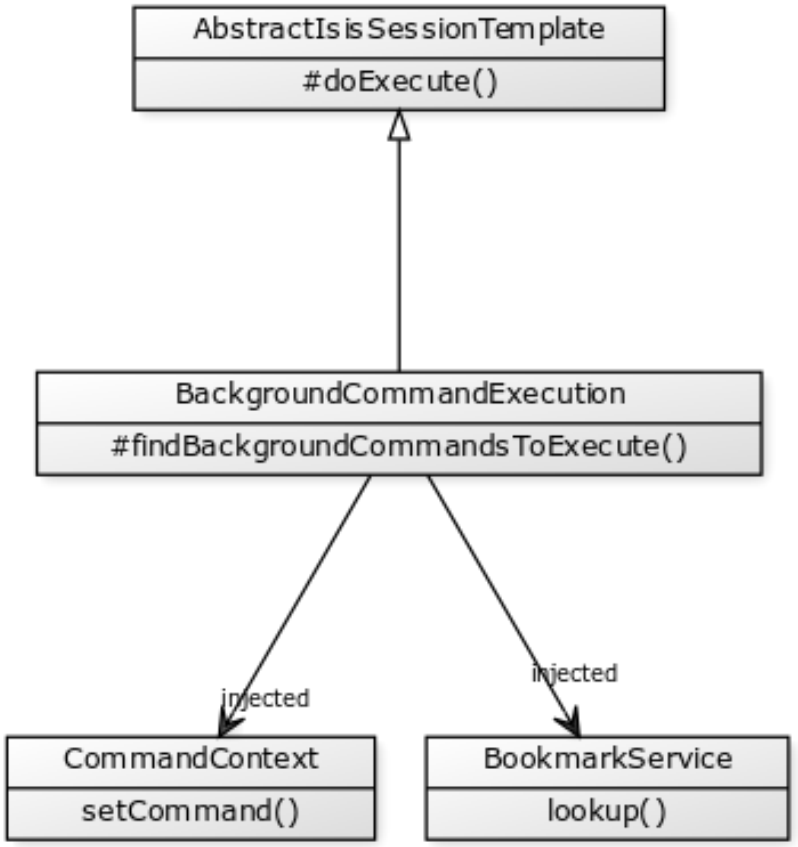


Figure 1. Inheritance Hierarchy for `BackgroundCommandExecution`

5.2.1. Background Execution

The `BackgroundCommandExecutionFromBackgroundCommandServiceJdo` is a concrete subclass of `BackgroundCommandExecution` (see the `BackgroundCommandService`), the intended use being for the class to be instantiated regularly (eg every 10 seconds) by a scheduler such as Quartz) to poll for `Command`s to be executed, and then execute them.

This implementation queries for `Command`s persisted by the Isis addons Command Module's implementations of `CommandService` and `BackgroundCommandService` using the `BackgroundCommandServiceJdoRepository`.

The diagram below (yuml.me/25343da1) shows the inheritance hierarchy for this class:

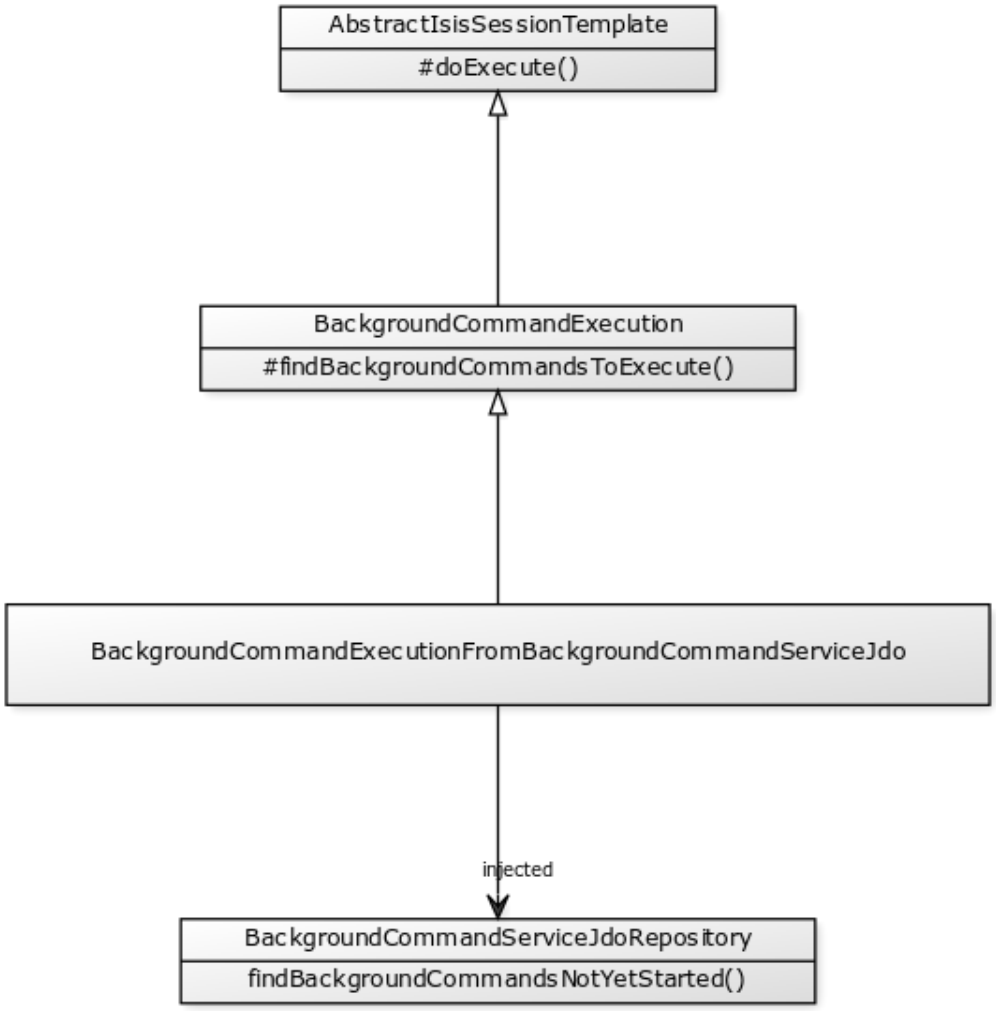


Figure 2. Inheritance Hierarchy for `BackgroundCommandExecutionFromBackgroundCommandServiceJdo`

Chapter 6. Other Techniques

This chapter pulls together a number of more advanced techniques that we've discovered and developed while building our own Isis applications.

6.1. Mapping RDBMS Views

TODO - as used in Estatio

6.2. Transactions and Errors

In Apache Isis, every interaction (action invocation or property edit) is automatically wrapped in a transaction, and any repository query automatically does a flush before hand.

What that means is that there's no need to explicitly start or commit transactions in Apache Isis; this will be done for you. Indeed, if you do try to manage transactions (eg by reaching into the JDO `PersistenceManager` exposed by the IsisJdoSupport domain service, then you are likely to confuse the framework and get a stack trace for your trouble.

However, you can complete a given transaction and start a new one. This is sometimes useful if writing a fixture script which is going to perform some sort of bulk migration of data from an old system. For this use case, use the `TransactionService`.

For example:

```
public class SomeLongRunningFixtureScript extends FixtureScript

    protected void execute(final ExecutionContext executionContext) {
        // do some work
        transactionService.nextTransaction();
        // do some work
        transactionService.nextTransaction();
        // do yet more work
    }

    @javax.inject.Inject
    TransactionService transactionService;
}
```

You get the idea.

6.2.1. Raise message/errors to users

The framework provides the `MessageService` as a means to return an out-of-band message to the end-user. In the Wicket viewer these are shown as "toast" pop-ups; the Restful Objects viewer returns an HTTP header.

The `UserService` provides three APIs, for different:

- `informUser()` - an informational message. In the Wicket viewer these are short-lived pop-ups that disappear after a short time.
- `warnUser()` - a warning. In the Wicket viewer these do not auto-close; they must be acknowledged.
- `raiseError()` - an error. In the Wicket viewer these do not auto-close; they must be acknowledged.

Each pop-up has a different background colour indicating its severity.

None of these messages/errors has any influence on the transaction; any changes to objects will be committed.

6.2.2. Aborting transactions

If you want to abort Apache Isis' transaction, this can be done by throwing an exception. The exception message is displayed to the user on the error page (if Wicket viewer) or a 500 status error code (if the Restful Objects viewer).

If the exception thrown is because of an unexpected error (eg a `NullPointerException` in the domain app itself), then the error page will include a stack trace. If however you want to indicate that the exception is in some sense "expected", then throw a `RecoverableException` (or any subclass, eg `ApplicationException`); the stack trace will then be suppressed from the error page.

Another way in which exceptions might be considered "expected" could be as the result of attempting to persist an object which then violates some type of database constraint. Even if the domain application checks beforehand, it could be that another user operating on the object at the same moment of time might result in the conflict.

To handle this the `ExceptionRecognizer` SPI can be used. The framework provides a number of implementations out-of-the-box; whenever an exception is thrown it is passed to each known recognizer implementation to see if it recognizes the exception and can return a user-meaningful error message. For example, `ExceptionRecognizerForSQLIntegrityConstraintViolationUniqueOrIndexException` checks if the exception inherits from `java.sql.SQLIntegrityConstraintViolationException`, and if so, constructs a suitable message.

6.3. Multi-tenancy

Of the various Isis Addons, the security module has the most features. One significant feature is the ability to associate users and objects with a "tenancy".

For more details, see the security module README.

6.4. Persisted Title

Normally the title of an object is not persisted to the database, rather it is recomputed automatically

from underlying properties. On occasion though you might want the title to also be persisted; either to make things easier for the DBA, or for an integration scenario, or some other purpose.

We can implement this feature by leveraging the JDO lifecycle. In the design we discuss here we make it a responsibility of the entities to persist the title as a property, by implementing a `ObjectWithPersistedTitle` interface:

```
public interface ObjectWithPersistedTitle {
    @PropertyLayout(hidden = Where.EVERYWHERE)  ①
    String getTitle();
    void setTitle(final String title);
}
```

① we don't want to expose this in the UI because the title is already prominently displayed.

We can then define a subscribing domain service that leverage this.

```
@DomainService(nature = NatureOfService.DOMAIN)
public class TitlingService extends AbstractSubscriber {
    @Subscribe
    public void on(final ObjectPersistingEvent ev) {
        handle(ev.getSource());
    }
    @Subscribe
    public void on(final ObjectUpdatingEvent ev) {
        handle(ev.getSource());
    }
    private void handle(final Object persistentInstance) {
        if(persistentInstance instanceof ObjectWithPersistedTitle) {
            final ObjectWithPersistedTitle objectWithPersistedTitle =
                (ObjectWithPersistedTitle) persistentInstance;
            objectWithPersistedTitle.setTitle(container.titleOf
(objectWithPersistedTitle));
        }
    }
    @Inject
    private DomainObjectContainer container;
}
```

Prior to 1.10.0 (when lifecycle events were introduced), this could also be be done by accessing the JDO API directly:

```
@RequestScoped
@DomainService(nature = NatureOfService.DOMAIN)
public class TitlingService {
    @PostConstruct
    public void init() {
        isisJdoSupport.getJdoPersistenceManager().addInstanceLifecycleListener(
            new StoreLifecycleListener() {
                @Override
                public void preStore(final InstanceLifecycleEvent event) {
                    final Object persistentInstance = event.getPersistentInstance();
                    if(persistentInstance instanceof ObjectWithPersistedTitle) {
                        final ObjectWithPersistedTitle objectWithPersistedTitle =
                            (ObjectWithPersistedTitle) persistentInstance;
                        objectWithPersistedTitle.setTitle(container.titleOf
(objectWithPersistedTitle));
                    }
                }
                @Override
                public void postStore(final InstanceLifecycleEvent event) {
                }
            }, null);
    }
    @Inject
    private IsisJdoSupport isisJdoSupport;
    @Inject
    private DomainObjectContainer container;
}
```

The above is probably the easiest and most straightforward design. One could imagine other designs where the persisted title is stored elsewhere. It could even be stored off into an Apache Lucene (or similar) database to allow for free-text searches.

6.5. Overriding Default Service Implns

The framework provides default implementations for many of the domain services. This is convenient, but sometimes you will want to replace the default implementation with your own service implementation.

The trick is to use the `@DomainServiceLayout#menuOrder()` attribute, specifying a low number (typically `"1"`).

For example, suppose you wanted to provide your own implementation of `LocaleProvider`. Here's how:

```
@DomainService(
        nature = NatureOfService.DOMAIN
)
@DomainServiceLayout(
        menuOrder = "1"                                 ①
)
public class MyLocaleProvider implements LocaleProvider {
    @Override
    public Locale getLocale() {
        return ...
    }
}
```

① takes precedence over the default implementation.

It's also quite common to want to decorate the existing implementation (ie have your own implementation delegate to the default); this is also possible and quite easy if using `1.10.0` or later:

```
@DomainService(
        nature = NatureOfService.DOMAIN
)
@DomainServiceLayout(
        menuOrder = "1"
①
)
public class MyLocaleProvider implements LocaleProvider {
    @Override
    public Locale getLocale() {
        return getDelegateLocaleProvider().getLocale();
②
    }
    Optional<LocaleProvider> delegateLocaleProvider;
③
    private LocaleProvider getDelegateLocaleProvider() {
        if(delegateLocaleProvider == null) {
            delegateLocaleProvider = Iterables.tryFind(localeProviders, input -> input
!= this);    ④
        }
        return delegateLocaleProvider.orNull();
    }
    @Inject
    List<LocaleProvider> localeProviders;
⑤
}
```

① takes precedence over the default implementation when injected elsewhere.

② this implementation merely delegates to the default implementation

③ lazily populated

④ delegate to the first implementation that isn't *this* implementation (else infinite loop!)

⑤ Injects all implementations, including this implemenation

Chapter 7. Customizing the Prog Model

This chapter explains the main APIs to extend or alter the programming conventions that Apache Isis understands to build up its metamodel.

7.1. Custom validator

Apache Isis' programming model includes a validator component that detects and prevents (by failing fast) a number of situations where the domain model is logically inconsistent.

For example, the validator will detect any orphaned supporting methods (eg `hideXxx()`) if the corresponding property or action has been renamed or deleted but the supporting method was not also updated. Another example is that a class cannot have a title specified both using `title()` method and also using `@Title` annotation.

The support for disallowing deprecated annotations is also implemented using the metamodel validator.

You can also impose your own application-specific rules by installing your own metamodel validator. To give just one example, you could impose naming standards such as ensuring that a domain-specific abbreviation such as "ISBN" is always consistently capitalized wherever it appears in a class member.

Isis' Maven plugin will also validate the domain object model during build time.

7.1.1. API and Implementation

There are several ways to go about implementing a validator.

`MetaModelValidator`

Any custom validator must implement Apache Isis' internal `MetaModelValidator` interface, so the simplest option is just to implement `MetaModelValidator` directly:

```
public interface MetaModelValidator
        implements SpecificationLoaderSpiAware { ①
    public void validate(
        ValidationFailures validationFailures);  ②
}
```

① the `SpecificationLoader` is the internal API providing access to the Apache Isis metamodel.

② add any metamodel violations to the `ValidationFailures` parameter (the collecting parameter pattern)

`Visitor`

More often than not, you'll want to visit every element in the metamodel, and so for this you can

instead subclass from `MetaModelValidatorVisiting.Visitor`:

```
public final class MetaModelValidatorVisiting ... {
    public static interface Visitor {
        public boolean visit(                       ①
            ObjectSpecification objectSpec,         ②
            ValidationFailures validationFailures); ③
    }
    ...
}
```

① return `true` continue visiting specs.

② `ObjectSpecification` is the internal API representing a class

③ add any metamodel violations to the `ValidationFailures` parameter

You can then create your custom validator by subclassing `MetaModelValidatorComposite` and adding the visiting validator:

```
public class MyMetaModelValidator extends MetaModelValidatorComposite {
    public MyMetaModelValidator() {
        add(new MetaModelValidatorVisiting(new MyVisitor()));
    }
}
```

If you have more than one rule then each can live in its own visitor.

`SummarizingVisitor`

As a slight refinement, you can also subclass from `MetaModelValidatorVisiting.SummarizingVisitor`:

```
public final class MetaModelValidatorVisiting ... {
    public static interface SummarizingVisitor extends Visitor {
        public void summarize(ValidationFailures validationFailures);
    }
    ...
}
```

A `SummarizingVisitor` will be called once after every element in the metamodel has been visited. This is great for performing checks on the metamodel as a whole. For example, Apache Isis uses this to check that there is at least one `@Persistable` domain entity defined.

7.1.2. Configuration

Once you have implemented your validator, you must register it with the framework by defining the appropriate configuration property:

```
isis.reflector.validator=com.mycompany.myapp.MyMetaModelValidator
```

7.2. Finetuning

The core metamodel defines APIs and implementations for building the Apache Isis metamodel: a description of the set of entities, domain services and values that make up the domain model.

The description of each domain class consists of a number of elements:

ObjectSpecification

Analogous to `java.lang.Class`; holds information about the class itself and holds collections of each of the three types of class' members (below);

OneToOneAssociation

Represents a class member that is a single-valued property of the class. The property's type is either a reference to another entity, or is a value type.

OneToManyAssociation

Represents a class member that is a collection of references to other entities. Note that Apache Isis does not currently support collections of values.

ObjectAction

Represents a class member that is an operation that can be performed on the action. Returns either a single value, a collection of entities, or is `void`.

The metamodel is built up through the ProgrammingModel, which defines an API for registering a set of FacetFactorys. Two special `FacetFactory` implementations - PropertyAccessorFacetFactory and CollectionAccessorFacetFactory - are used to identify the class members. Pretty much all the other `FacetFactory`s are responsible for installing Facets onto the metamodel elements.

There are many many such `Facet`s, and are used to do such things get values from properties or collections, modify properties or collections, invoke action, hide or disable class members, provide UI hints for class members, and so on. In short, the `FacetFactory`s registered defines the Apache Isis programming conventions.

7.2.1. Modifying the Prog. Model

The default implementation of `ProgrammingModel` is ProgrammingModelFacetsJava5, which registers a large number of `FacetFactory`s.

By editing `isis.properties` you can modify the programming conventions either by (a) using the default programming model, but tweaking it to include new `FacetFactory`s or exclude existing, or (b) by specifying a completely different programming model implementation.

Let's see how this is done.

Including or excluding facets

Suppose that you wanted to completely remove support for the (already deprecated) `@ActionOrder` annotation. This would be done using:

```
isis.reflector.facets.exclude=org.apache.isis.core.metamodel.facets.object.actionorder
.annotation.ActionOrderFacetAnnotationFactory
```

Or, suppose you wanted to use the example "namefile" `FacetFactory` as part of your programming conventions, use:

```
isis.reflector.facets.include=org.apache.isis.example.metamodel.namefile.facets.NameFi
leFacetFactory
```

To include/exclude more than one `FacetFactory`, specify as a comma-separated list.

This thread from the users mailing list (in Apr 2014) shows a typical customization (to enable per-instance security) (though note that Multi-Tenancy is now a better solution to that particular use-case.

Previously (prior to `1.13.0`) the framework also supported the "isis.reflector.facets" configuration property, to specify a completely new implementation of the (internal API) `ProgrammingModel`. This is no longer supported; use "isis.reflector.facets.include" and "isis.reflector.facet.exclude" to adjust the framework's default implementation (called `ProgrammingModelFacetsJava5`).

7.3. Layout Metadata Reader

The metadata for domain objects is obtained both statically and dynamically.

The default implementation for reading dynamic layout metadata is `org.apache.isis.core.metamodel.layoutmetadata.json.LayoutMetadataReaderFromJson`, which is responsible for reading from the `Xxx.layout.json` files on the classpath (for each domain entity `Xxx`).

You can also implement your own metadata readers and plug them into Apache Isis. These could read from a different file format, or they could, even, read data dynamically from a URL or database. (Indeed, one could imagine an implementation whereby users could share layouts, all stored in some central repository).

The use of dynamic layouts through the `.layout.json` - and therefore also the `LayoutMetadataReader` - is DEPRECATED. Instead, the dynamic XML layouts using `.layout.xml` enables much more sophisticated custom layouts than those afforded by `.layout.json`.

By default, custom XML layouts are read from the classpath. This behaviour can be customized by providing an alternative implementation of the `GridLoaderService`.

7.3.1. API and Implementation

Any reader must implement Apache Isis' internal `LayoutMetadataReader` interface:

```
public interface LayoutMetadataReader {
    public Properties asProperties(Class<?> domainClass) throws ReaderException;
}
```

The implementation "simply" returns a set of properties where the property key is a unique identifier to both the class member and also the facet of the class member to which the metadata relates.

See the implementation of the built-in `LayoutMetadataReaderFromJson` for more detail.

Returning either `null` or throwing an exception indicates that the reader was unable to load any metadata for the specified class.

Extended API

Optionally the reader can implement the extended `LayoutMetadaReader2` API:

```
public interface LayoutMetadataReader2 extends LayoutMetadataReader {
    public static class Support {
        public static Support entitiesOnly() {
            return new Support(false,false,false,false,false,false,false);
        }
        ...
        public boolean interfaces() { ... } ①
        public boolean anonymous() { ... }  ②
        public boolean synthetic() { ... }  ③
        public boolean array() { ... }      ④
        public boolean enums() { ... }      ⑤
        public boolean applibValueTypes() { ⑥
        public boolean services() { ... }   ⑦
    }
    Support support();
}
```

① whether this implementation can provide metadata for interface types.

② whether this implementation can provide metadata for anonymous classes.

③ whether this implementation can provide metadata for synthetic types.

④ whether this implementation can provide metadata for arrays.

⑤ whether this implementation can provide metadata for enums.

⑥ whether this implementation can provide metadata for applib value types.

⑦ whether this implementation can provide metadata for domain services.

The `support()` method returns a struct class that describes the types of classes are supported by this implementation.

The `LayoutMetadataReaderFromJson` implements this extended API.

7.3.2. Configuration

Once you have implemented your validator, you must register it with the framework by defining the appropriate configuration property:

```
isis.reflector.layoutMetadataReaders=\
           com.mycompany.myapp.MyMetaModelValidator,\

org.apache.isis.core.metamodel.layoutmetadata.json.LayoutMetadataReaderFromJson    ①
```

① the property replaces any existing metadata readers; if you want to preserve the ability to read from `Xxx.layout.json` then also register Apache Isis' built-in implementation.

Chapter 8. Deployment

This chapter provides guidance on some common deployment scenarios.

8.1. Command Line (`WebServer`)

As well as deploying an Apache Isis application into a servlet container, it is also possible to invoke from the command line using the `org.apache.isis.WebServer` utility class. This is especially useful while developing and testing, but may also suit some deployment scenarios (eg running as a standalone EXE within a Docker container, for example). Internally the `WebServer` spins up a Jetty servlet container.

The class also supports a number of command line arguments:

Table 2. Command line args for `org.apache.isis.Webserver`

Flag	Long format	Values (default)	Description
`-t`	`--type`	`server_prototype`, `server` (`server`)	Deployment type
`-m`	`--manifest`	FQCN	Fully qualified class name of the `AppManifest` to use to bootstrap the system. This flag sets/overrides the `isis.appManifest` configuration property to the specified class name.
`-f`	`--fixture`	FQCN	Fully qualified class name of the fixture (extending `FixtureScript`) to be run to setup data. This flag sets/overrides the `isis.fixtures` configuration property to the specified class name, and also sets the `isis.persistor.datanucleus.install-fixtures` configuration property to `true` to instruct the JDO/DataNucleus objectstore to actually load in the fixtures. It is also possible to specify the fixture class name using either the `$IsisFixture` or `$IsisFixtures` environment variable (case insensitive).
`-p`	`--port`	(8080)	The port number to listen on. This flag sets/overrides the `isis.embedded-web-server.port` configuration property.
`-c`	`--config`	filename	configuration file containing additional configuration properties

Flag	Long format	Values (default)	Description
`-D`		xxx=yyy	Specify additional arbitrary configuration properties. This can be specified multiple times. Further discussion below.
`-v`	`--version`		Prints the version, then quits
`-h`	`--help`		Prints a usage message, then quits

Note that the `-D` argument is **not** a system property, it is parsed by `WebServer` itself. That said, it is *also* possible to specify system properties, and these will also be processed in the exact same way.

Said another way, properties can be specified either as application arguments:

```
java org.apache.isis.WebServer -Dxxx=yyy -Daaa=bbb
```

or as system properties:

```
java -Dxxx=yyy -Daaa=bbb org.apache.isis.WebServer
```

The `Dummy` class

The framework also provides the `org.apache.isis.Dummy` class, which consists of a single empty `main(String[])` method. It was introduced as a convenience for developers using Eclipse in combination with the DataNucleus plugin; if used as a launch target then it allow the entities to be enhanced withouth the running an app.

8.2. Deploying to Tomcat

Some pointers when deploying to Tomcat (or any other servlet container).

8.2.1. Externalized Configuration

See the guidance below.

8.2.2. JVM Args

The `WrapperFactory` uses Javassist to create on-the-fly classes acting as a proxy. The cost of these proxies can be mitigated using:

```
-XX:+CMSClassUnloadingEnabled -XX:+UseConcMarkSweepGC
```

8.2.3. Using a JNDI Datasource

See the guidance in the configuring datanucleus section.

8.3. Externalized Configuration

As described here, by default Apache Isis itself bootstraps from the `isis.properties` configuration file. It will also read configuration from the (optional) component/implementation-specific configuration files (such as `persistor_datanucleus.properties` or `viewer_wicket.properties`), and also (optional) component-specific configuration files (such as `persistor.properties` or `viewer.properties`).

It's generally not good practice to have the same configuration property in more than one file, but if that does occur, then the subsequent configuration property will be ignored.

In addition the framework will also load the `overrides.properties` file. This is intended to support deployment through Docker, as discussed next.

All of these files are read from the `WEB-INF` directory. Having this configuration "baked into" the application is okay in a development environment, but when the app needs to be deployed to a test or production environment, this configuration should be read from an external location.

There are in fact several frameworks involved here, all of which need to be pointed to this external location:

- Apache Isis itself, which (as already discussed) reads `isis.properties` and optional component-specific config files.
- Apache Shiro, which reads the `shiro.ini` file (and may read other files referenced from that file)
- Apache log4j 1.2, for logging, which reads `logging.properties` file
- although not used by Apache Isis, there's a good chance you may be using the Spring framework (eg if using Apache Active MQ or Apache Camel.

Each of these frameworks has its own way of externalizing its configuration.

8.3.1. Apache Isis' Config

To tell Apache Isis to load configuration from an external directory, specify the `isis.config.dir` context parameter.

If the external configuration directory is fixed for all environments (systest, UAT, prod etc), then you can specify within the `web.xml` itself:

```
<context-param>
    <param-name>isis.config.dir</param-name>
    <param-value>location of external config directory</param-value>
</context-param>
```

If however the configuration directory varies by environment, then the context parameter will be specified to each installation of your servlet container. Most (if not all) servlet containers will provide a means to define context parameters through proprietary config files.

For example, if using Tomcat 7.0, you would typically copy the empty `$TOMCAT_HOME/webapps/conf/context.xml` to a file specific to the webapp, for example `$TOMCAT_HOME/webapps/conf/todo.xml`. The context parameter would then be specified by adding the following:

```
<Parameter name="isis.config.dir"
           value="/usr/local/tomcat/myapp/conf/"
           override="false"/>
```

Note that the `override` key should be set to "false", not "true". It indicates whether the application's own `web.xml` can override the setting. In most cases, you probably want to disallow that.

For more detail, see the Tomcat documentation on defining a context and on context parameters.

Note that running the app using Apache Isis' `org.apache.isis.WebServer` bootstrapper currently does not externalized Apache Isis configuration.

8.3.2. Shiro Config

If using Apache Isis' Shiro integration for authentication and/or authorization, note that it reads from the `shiro.ini` configuration file. By default this also resides in `WEB-INF`.

Similar to Apache Isis, Shiro lets this configuration directory be altered, by specifying the `shiroConfigLocations` context parameter.

You can therefore override the default location using the same technique as described above for Apache Isis' `isis.config.dir` context parameter. For example:

```
<Parameter name="shiroConfigLocations"
           value="file:/usr/local/myapp/conf/shiro.ini"
           override="false" />
```

Note that Shiro is more flexible than Apache Isis; it will read its configuration from any URL, not just a directory on the local filesystem.

8.3.3. Log4j Config

By default Apache Isis configures log4j to read the `logging.properties` file in the `WEB-INF` directory. This can be overridden by setting the `log4j.properties` system property to the URL of the log4j properties file.

For example, if deploying to Tomcat7, this amounts to adding the following to the `CATALINA_OPTS` flags:

```
export CATALINA_OPTS="-
Dlog4j.configuration=/usr/local/tomcat/myapp/conf/logging.properties"
```

`CATALINA_OPTS` was called `TOMCAT_OPTS` in earlier versions of Tomcat.

Further details an be found in the log4j documentation.

8.3.4. Spring Config

Although Apache Isis does not use Spring, it's possible that your app may use other components that do use Spring. For example, the (non-ASF) Isis addons' publishmq module uses ActiveMQ and Camel to support publishing; both of these leverage Spring.

There are several ways to externalized Spring config, but the mechanism described here is similar in nature to those that we use for externalizing Apache Isis' and Shiro's configuration. In your `web.xml`, you will probably load the Spring application context using code such as:

```
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-
class>
</listener>
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        classpath:my-application-context-config.xml
    </param-value>
</context-param>
```

Add a new application context `propertyPlaceholderConfigurer-config.xml` defining a `PropertyPlaceholderConfigurer` bean.

```
<beans
  xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
  http://activemq.apache.org/schema/core
http://activemq.apache.org/schema/core/activemq-core.xsd">
    <bean
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
        <property name="locations">
            <list>
                <value>${spring.config.file}</value>
            </list>
        </property>
    </bean>
</beans>
```

This reads the properties from a `spring.config.file`, defined as a context-param in the `web.xml`:

```
<context-param>
    <param-name>spring.config.file</param-name>
    <param-value>classpath:spring.properties</param-value>
</context-param>
```

Then update the bootstrapping in `web.xml` to use this new application context, eg:

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        classpath:my-application-context-config.xml,
        classpath:propertyPlaceholderConfigurer-config.xml
    </param-value>
</context-param>
```

To use some other externalized configuration, override the `spring.config.file` property, eg using Tomcat's config file:

```
<Parameter name="spring.config.dir"
           value="file:/usr/local/myapp/conf/spring.properties"
           override="false" />
```

An alternative approach

As mentioned, there are several other ways to externalize Spring's config; one approach is to use Spring's profile support.

For example, in the application context you could have:

```
<beans profile="default">
    <bean
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
        <property name="locations">
            <list>
                <value>classpath:dev.properties</value>
            </list>
        </property>
    </bean>
</beans>
<beans profile="externalized">
    <bean id="propertyPlaceHolder"
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
        <property name="locations">
            <list>
                <value>classpath:prod.properties</value>
            </list>
        </property>
    </bean>
</beans>
```

The idea being that specifying the "prod" profile rather than the "default" profile would cause a different set of configuration properties to be read.

The active profile can be overridden with a system property, eg:

```
-Dspring.active.profiles=prod
```

take a look at this SO answer on using Spring profiles.

8.3.5. See also

See JVM args for other JVM args and system properties that you might want to consider setting.

8.4. Docker

When running the application within a Docker container, the problem that must be solved is to override the configuration properties baked into the war file, eg to point to the app to a different JDBC URL.

There are several options.

All the options here rely on starting the Docker container with a set of arguments, some of which would very likely be passwords for database connections etc. As such these techniques are only suitable where the security of the Docker host can be assured.

8.4.1. Using an `overrides.properties` file

In addition to loading the regular configuration properties from `WEB-INF` directory (described here), Apache Isis will also load the `overrides.properties` file.

This file is treated slightly differently than the other configuration files; it is loaded last, and any configuration properties defined in it will *override* any configuration properties already read from other files (this includes any properties specified via the command line).

While the regular configuration files are "baked into" the application WAR file, the `overrides.properties` file is created dynamically as part of the Docker `ENTRYPOINT` script, eg as documented in the Dockerfile best practices.

Thus, Docker can be supported as follows:

- use `mvn` (as currently) to create a WAR file; set up with the `pom.xml` with the JDBC drivers of all DB servers that you might want to connect to (hsqldb, sql server, postgresql etc)
- in the `Dockerfile`, specify a base image containing Tomcat 8 + Java 8 (say)
- also in the `Dockerfile`, arrange it such that the WAR file is "exploded" (there is no need to copy over the WAR file itself)
- write a script that:
  - explodes the WAR file, copying it into the Tomcat's `webapp` directory. There is no need to copy over the WAR file itself.
  - creates the `overrides.properties` file from any input arguments, placing it into the `WEB-INF` directory
  - sets all files to read-only
- use `ENTRYPOINT` (and probably also `CMD`) to invoke above script.

8.4.2. Using system properties

The servlet context initializer will search for any system properties called `isis.xxx` and if present will use them as overrides.

Thus, an alternative option for a Docker image is to bootstrap the servlet container (Tomcat, Jetty) with appropriate system properties set up. For example, with Tomcat this can be done by writing into the `conf/catalina.properties` file (see for example this stackoverflow post).

The Docker's `ENTRYPOINT` therefore just needs to parse the Docker container's own command line arguments and use to create this file.

8.4.3. Using the ISIS_OPTS environment variable

The servlet context initializer will search for an environment variable called $ISIS_OPTS and if present will parse the content as a set of key/value pairs. Each key/value pair is separated by "||".

For example:

```
export ISIS_OPTS
="isis.appManifest=domainapp.app.DomainAppAppManifestWithFixtures||isis.objects.editin
g=false"
```

can be used to run with a different app manifest, and also to disable editing of properties.

To use a different separator, set the (optional) $ISIS_OPTS_SEPARATOR variable.

```
export ISIS_OPTS_SEPARATOR=";"
export ISIS_OPTS
="isis.appManifest=domainapp.app.DomainAppAppManifestWithFixtures;isis.objects.editing
=false"
```

The Docker's ENTRYPOINT therefore just needs to parse the Docker container's own command line arguments and use to set this environment variable.

8.5. Deploying to Google App Engine

The Google App Engine (GAE) provides a JDO API, meaning that you can deploy Apache Isis onto GAE using the JDO objectstore.

However, GAE is not an RDBMS, and so there are some limitations that it imposes. This page gathers together various hints, tips and workarounds.

8.5.1. Primary Keys and Owned/Unowned Relationships

All entities must have a @PrimaryKey. Within GAE, the type of this key matters.

For an entity to be an aggregate root, (ie a root of an GAE entity group), its key must be a Long, eg:

Any collection that holds this entity type (eg ToDoItem#dependencies holding a collection of ToDoItem`s) should then be annotated with `@Unowned (a GAE annotation).

If on the other hand you want the object to be owned (through a 1:m relationship somewhere) by some other root, then use a String:

Note: if you store a relationship with a String key it means that the parent object *owns* the child, any attempt to change the relationship raise and exception.

8.5.2. Custom Types

Currently Apache Isis' `Blob` and `Clob` types and the JODA types (`LocalDate` et al) are *not* supported in GAE.

Instead, GAE defines a fixed set of value types, including `BlobKey`. Members of the Apache Isis community have this working, though I haven't seen the code.

The above notwithstanding, Andy Jefferson at DataNucleus tells us:

GAE JDO/JPA does support some type conversion, because looking at StoreFieldManager.java for any field that is Object-based and not a relation nor Serialized it will call TypeConverstionUtils.java and that looks for a <code>TypeConverter</code> (specify <code>@Extension</code> with key of "type-converter-name" against a field and value as the <code>TypeConverter</code> class) and it should convert it. Similarly when getting the value from the datastore.

On further investigation, it seems that the GAE implementation performs a type check on a `SUPPORTED_TYPES` Java set, in `com.google.appengine.api.datastore.DataTypeUtils`:

```
if (!supportedTypes.contains(value.getClass())) {
    throw new IllegalArgumentException(prefix + value.getClass().getName() + " is not
a supported property type.");
}
```

We still need to try out Andy's recipe, above.

8.6. Neo4J

As of 1.8.0 Apache Isis has experimental support for Neo4J, courtesy of DataNucleus' Neo4J Datastore implementation.

The SimpleApp archetype has been updated so that they can be optionally run under Neo4J.

In addition, the Isis addons' neoapp (non-ASF) is configured to run with an embedded Neo4J server running alongside the Apache Isis webapp.

The steps below describe the configuration steps required to update an existing app.

8.6.1. ConnectionURL

In `persistor.properties`, update the JDO `ConnectionURL` property, eg:

```
isis.persistor.datanucleus.impl.javax.jdo.option.ConnectionURL=neo4j:neo4j_DB
```

The other connection properties (`ConnectionDriverName`, `ConnectionUserName` and `ConnectionPassword`) should be commented out.

8.6.2. Update pom.xml

Add the following dependency to the `webapp` project's `pom.xml`:

```
<dependencies>
    ...
    <dependency>
        <groupId>org.datanucleus</groupId>
        <artifactId>datanucleus-neo4j</artifactId>
        <version>4.0.5</version> ①
    </dependency>
    ...
</dependencies>
```

① for Isis v1.9.0, use the value shown. for Isis v1.8.0, use 3.2.3.

In the SimpleApp archetype this is defined under the "neo4j" profile so can be activated using `-P neo4j`.

8.6.3. Try it out!

If you want to quickly try out neo4j for yourself:

- run the SimpleApp archetype (v1.8.0)
- build the app:
- run the app:

If you visit the about page you should see the neo4j JAR files are linked in, and a `neo4j_DB` subdirectory within the `webapp` directory.

8.7. JVM Flags

TODO

The default JVM configuration will most likely not be appropriate for running Isis as a webapp. The table below suggests some JVM args that you will probably want to modify:

Table 3. JVM args

Flag	Description
-server	Run the JVM in server mode, meaning that the JVM should spend more time on the optimization of the fragments of codes that are most often used (hotspots). This leads to better performance at the price of a higher overhead at startup.
-Xms128m Minimum heap size	-Xmx768m
Maximum heap size	-XX:PermSize=64m
Minimum perm size (for class definitions)	-XX:MaxPermSize=256m
Maximum perm size (for class definitions)	-XX:+DisableExplicitGC

There are also a whole bunch of GC-related flags, that you might want to explore; see this detailed Hotspot JVM documentation and also this blog post.

8.7.1. Configuring in Tomcat

If using Tomcat, update the `CATALINA_OPTS` variable. (This variable is also updated if configuring logging to run externally).

Chapter 9. `web.xml`

Apache Isis provides two different viewers, the Wicket viewer and the RestfulObjects viewer. You can deploy both of these concurrently, or deploy just the Wicket viewer, or deploy just the Restful Objects viewer. The configuration in `web.xml` varies accordingly, both in terms of the servlet context listeners, filters and servlets.

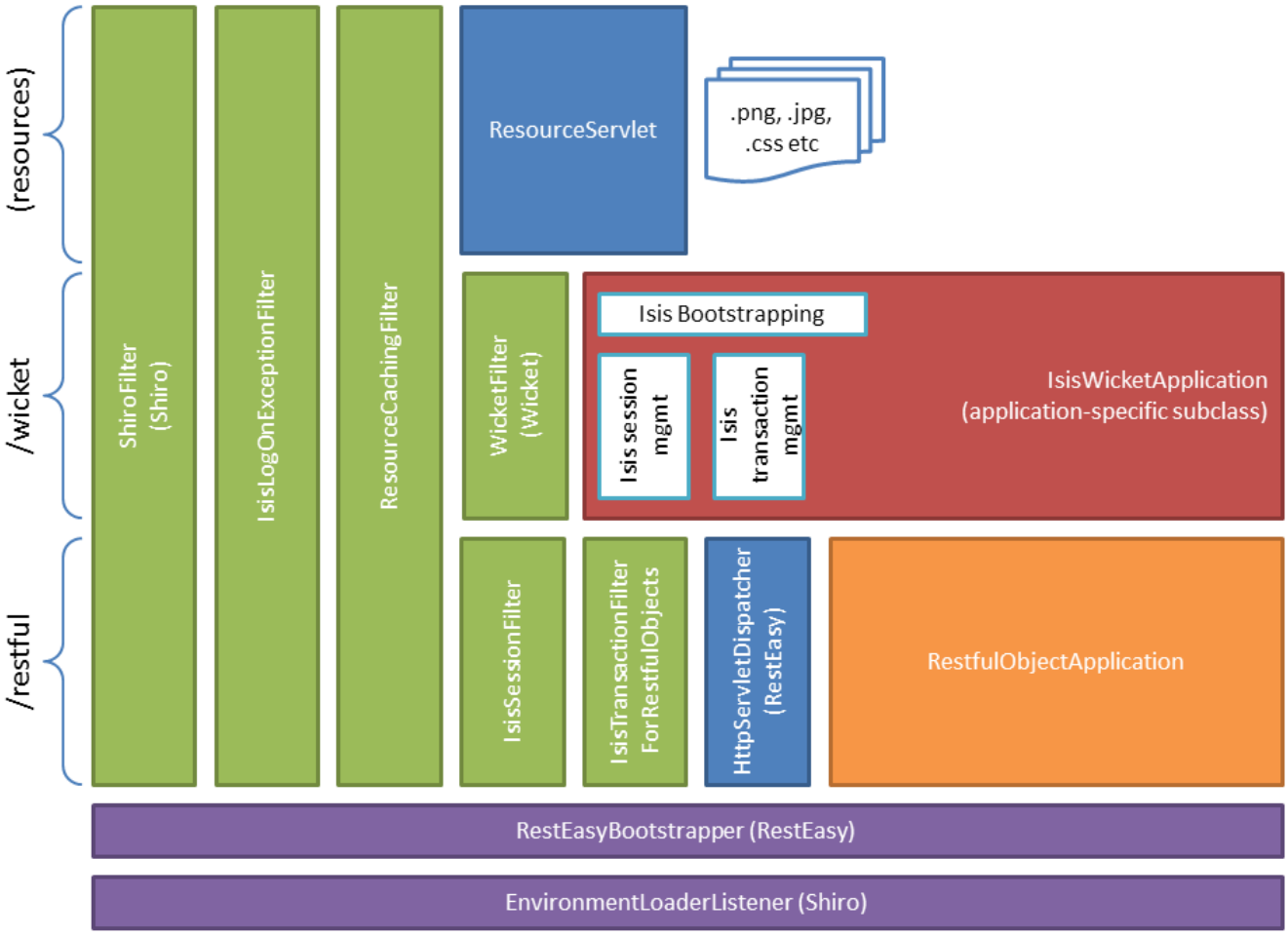
If you are using Apache Isis' integration with Apache Shiro (for security) then this also needs configuring in `web.xml`. See the security chapter for full details on this topic.

The servlets and filters are mapped to three main pipelines:

- `/wicket` - the Wicket viewer UI
- `/restful` - the Restful Objects resources (REST API)
- other paths, also static resources (such as `.png`, `.css`)

With the following key:

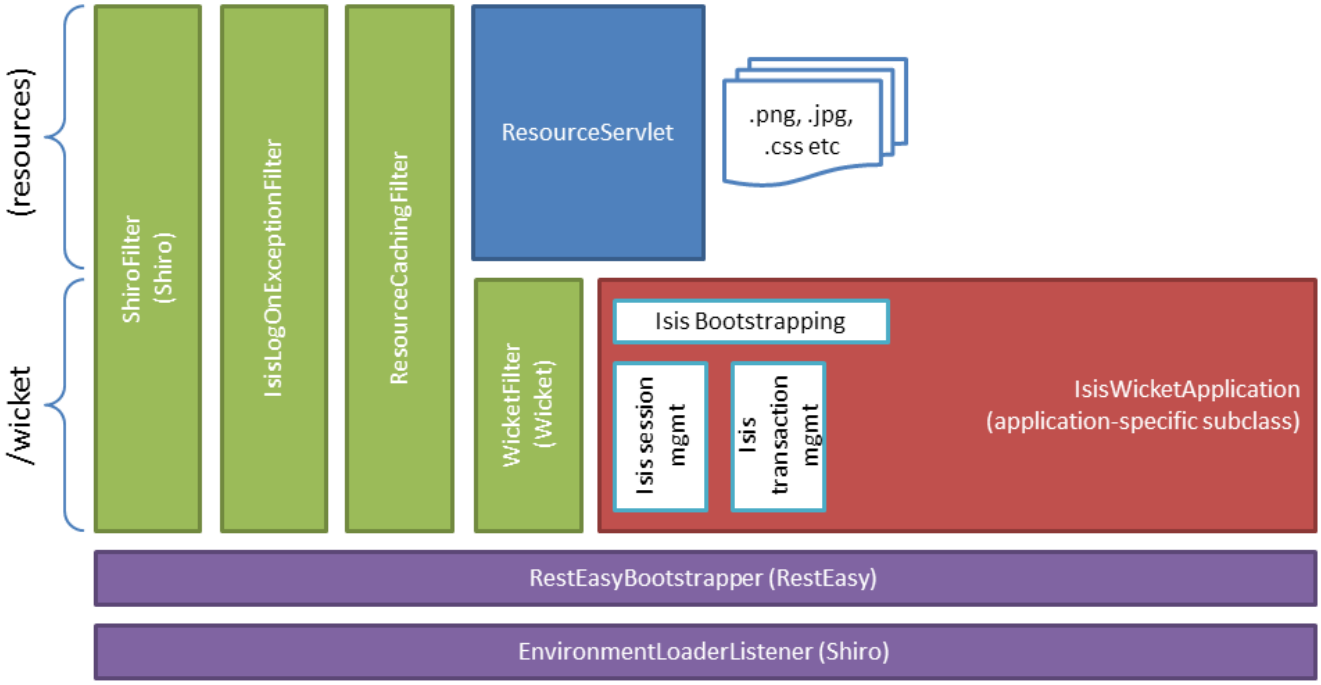
the diagram below shows the components to be configured if deploying both the Wicket viewer and Restful Objects viewer:



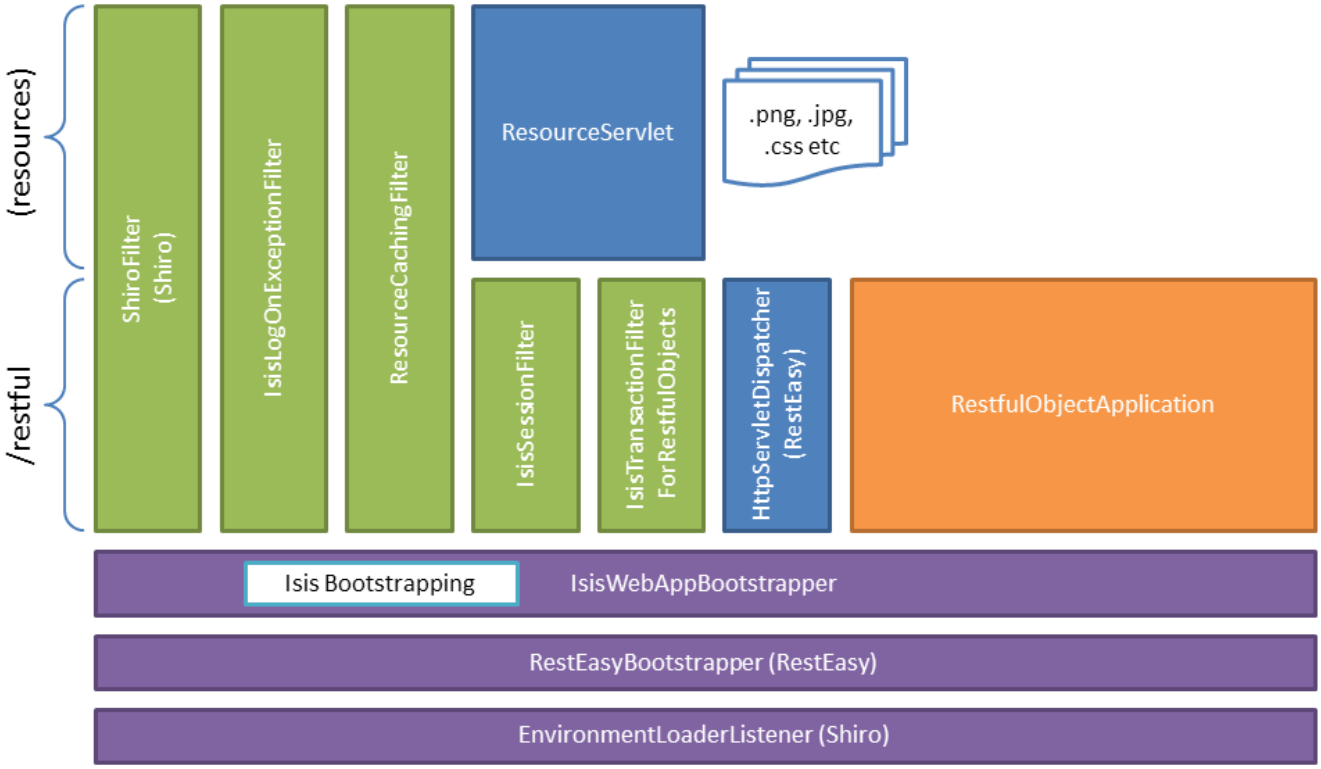
Here the Wicket viewer is responsible for the main bootstrapping of Apache Isis itself, in other words the shared (global) metadata; this is done by the `IsisWicketApplication` class (extending the `WicketApplication` Wicket API). This class is also responsible for Apache Isis' own session and transaction management.

The Restful Objects viewer - being a JAX-RS application implemented using the RestEasy framework - requires the `RestEasyBootstrapper` servlet context listener. It is this context listener that also sets up the `RestfulObjectsApplication`, which is then delegated to by the RestEasy `HttpServletDispatcher`. This pipeline uses the `IsisSessionFilter` and `IsisTransactionFilterForRestfulObjects` to perform the session and transaction management before it hits the RestEasy servlet.

If only the Wicket viewer is deployed, then the diagram is more or less the same: the RestEasy servlet, context listener and supporting filters are simply removed:



Finally, if only the Restful Objects viewer is deployed, then things change a little more subtly. Here, the Wicket filter is no longer needed. In its place, though the `IsisWebAppBootstrapper` context listener is required: this is responsible for seting up the shared (global) metadata.



The following sections detail these various listeners, filters and servlets in more detail.

9.1. Servlet Context Listeners

Servlet context listeners are used to perform initialization on application startup. Both Shiro (if configured as the security mechanism) and RestEasy (for the Restful Objects viewer) require their own context listener. In addition, if the Wicket viewer is *not* being used, then additional Apache Isis-specific listener is required for bootstrapping of the Apache Isis framework itself.

9.1.1. `EnvironmentLoaderListener` (Shiro)

Bootstrap listener to startup and shutdown the web application's Shiro `WebEnvironment` at startup and shutdown respectively.

Its definition is:

```
<listener>
    <listener-class>org.apache.shiro.web.env.EnvironmentLoaderListener</listener-
class>
</listener>
```

9.1.2. `IsisWebAppBootstrapper`

The `IsisWebAppBootstrapper` servlet context listener bootstraps the shared (global) metadata for the Apache Isis framework. This listener is not required (indeed must not be configured) if the Wicket viewer is in use.

Its definition is:

```
<listener>
    <listener-class>org.apache.isis.core.webapp.IsisWebAppBootstrapper</listener-
class>
</listener>
```

Its context parameters are:

```
<context-param>
    <param-name>deploymentType</param-name>
    <param-value>SERVER_PROTOTYPE</param-value>
</context-param>
<context-param>
    <param-name>isis.viewers</param-name>
    <param-value>restfulobjects</param-value>
</context-param>
```

9.1.3. `ResteasyBootstrap` (RestEasy)

The `ResteasyBootstrap` servlet context listener initializes the RestEasy runtime, specifying that classes (namely, those specified in Isis' `RestfulObjectsApplication`) to be exposed as REST resources. It is required if the Restful Objects viewer is to be deployed.

Its definition is:

```
<listener>
    <listener-class>
org.jboss.resteasy.plugins.server.servlet.ResteasyBootstrap</listener-class>
</listener>
```

There are two relevant context parameters:

```
<context-param>
    <param-name>javax.ws.rs.Application</param-name>        ①
    <param-
value>org.apache.isis.viewer.restfulobjects.server.RestfulObjectsApplication</param-
value>
</context-param>
<context-param>
    <param-name>resteasy.servlet.mapping.prefix</param-name>
    <param-value>/restful/</param-value>                    ②
</context-param>
```

① used by RestEasy to determine the JAX-RS resources and other related configuration

② should correspond to the filter mapping of the `HttpServletDispatcher` servlet

9.2. Servlets

Servlets process HTTP requests and return corresponding responses.

9.2.1. `HttpServletDispatcher` (RestEasy)

This servlet is provided by the RestEasy framework, and does the dispatching to the resources defined by Apache Isis' `RestfulObjectsApplication` (see above).

Its definition is:

```
<servlet>
    <servlet-name>RestfulObjectsRestEasyDispatcher</servlet-name>
    <servlet-class>
org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher</servlet-class>
</servlet>
```

Its mapping is:

```
<servlet-mapping>
    <servlet-name>RestfulObjectsRestEasyDispatcher</servlet-name>
    <url-pattern>/restful/*</url-pattern>
</servlet-mapping>
```

9.2.2. `ResourceServlet`

The `ResourceServlet` loads and services static content either from the filesystem or from the classpath, each with an appropriate mime type.

Static content here means request paths ending in `.js`, `.css`, `.html`, `.png`, `.jpg`, `.jpeg` and `gif`.

Its definition is:

```
<servlet>
    <servlet-name>Resource</servlet-name>
    <servlet-class>org.apache.isis.core.webapp.content.ResourceServlet</servlet-class>
</servlet>
```

Its mapping is:

```
<servlet-mapping>
    <servlet-name>Resource</servlet-name>
    <url-pattern>*.css</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>Resource</servlet-name>
    <url-pattern>*.png</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>Resource</servlet-name>
    <url-pattern>*.jpg</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>Resource</servlet-name>
    <url-pattern>*.jpeg</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>Resource</servlet-name>
    <url-pattern>*.gif</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>Resource</servlet-name>
    <url-pattern>*.svg</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>Resource</servlet-name>
    <url-pattern>*.js</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>Resource</servlet-name>
    <url-pattern>*.html</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>Resource</servlet-name>
    <url-pattern>*.swf</url-pattern>
</servlet-mapping>
```

9.3. Filters

The order in which filters appear in `web.xml` matters: first to last they define a pipeline. This is shown in the above diagrams, and the subsections also list the in the same order that they should appear in your `web.xml`.

9.3.1. `ShiroFilter` (Shiro)

Shiro filter that sets up a Shiro security manager for the request, obtained from the Shiro `WebEnvironment` set up by the Shiro `EnvironmentLoaderListener` (discussed above).

Its definition is:

```
<filter>
    <filter-name>ShiroFilter</filter-name>
    <filter-class>org.apache.shiro.web.servlet.ShiroFilter</filter-class>
</filter>
```

Its mapping is:

```
<filter-mapping>
    <filter-name>ShiroFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

9.3.2. IsisLogOnExceptionFilter

The `IsisLogOnExceptionFilter` filter simply logs the URL of any request that causes an exception to be thrown, then re-propagates the exception. The use case is simply to ensure that all exceptions are logged (against the `IsisLogOnExceptionFilter` slf4j appender).

Its definition is:

```
<filter>
    <filter-name>IsisLogOnExceptionFilter</filter-name>
    <filter-class>
org.apache.isis.core.webapp.diagnostics.IsisLogOnExceptionFilter</filter-class>
</filter>
```

Its mapping is:

```
<filter-mapping>
    <filter-name>IsisLogOnExceptionFilter</filter-name>
    <url-pattern>/wicket/*</url-pattern>
</filter-mapping>
<filter-mapping>
    <filter-name>IsisLogOnExceptionFilter</filter-name>
    <url-pattern>/restful/*</url-pattern>
</filter-mapping>
```

9.3.3. ResourceCachingFilter

The `ResourceCachingFilter` adds HTTP cache headers to specified resources, based on their pattern.

Its definition is:

```
<filter>
    <filter-name>ResourceCachingFilter</filter-name>
    <filter-class>org.apache.isis.core.webapp.content.ResourceCachingFilter</filter-
class>
    <init-param>
        <param-name>CacheTime</param-name>      ①
        <param-value>86400</param-value>
    </init-param>
</filter>
```

① cache time, in seconds

Its mapping is:

```
<filter-mapping>
    <filter-name>ResourceCachingFilter</filter-name>
    <url-pattern>*.css</url-pattern>
</filter-mapping>
<filter-mapping>
    <filter-name>ResourceCachingFilter</filter-name>
    <url-pattern>*.png</url-pattern>
</filter-mapping>
<filter-mapping>
    <filter-name>ResourceCachingFilter</filter-name>
    <url-pattern>*.jpg</url-pattern>
</filter-mapping>
<filter-mapping>
    <filter-name>ResourceCachingFilter</filter-name>
    <url-pattern>*.jpeg</url-pattern>
</filter-mapping>
<filter-mapping>
    <filter-name>ResourceCachingFilter</filter-name>
    <url-pattern>*.gif</url-pattern>
</filter-mapping>
<filter-mapping>
    <filter-name>ResourceCachingFilter</filter-name>
    <url-pattern>*.svg</url-pattern>
</filter-mapping>
<filter-mapping>
    <filter-name>ResourceCachingFilter</filter-name>
    <url-pattern>*.html</url-pattern>
</filter-mapping>
<filter-mapping>
    <filter-name>ResourceCachingFilter</filter-name>
    <url-pattern>*.js</url-pattern>
</filter-mapping>
<filter-mapping>
    <filter-name>ResourceCachingFilter</filter-name>
    <url-pattern>*.swf</url-pattern>
</filter-mapping>
```

9.3.4. `WicketFilter`

The `WicketFilter` is responsible for initiating the handling of Wicket requests.

Its definition is:

```
<filter>
    <filter-name>WicketFilter</filter-name>
    <filter-class>org.apache.wicket.protocol.http.WicketFilter</filter-class>
    <init-param>
        <param-name>applicationClassName</param-name>    ①
        <param-value>domainapp.webapp.SimpleApplication</param-value>
    </init-param>
</filter>
```

① specify the application (subclass of `IsisWicketApplication`) to use

Its mapping is:

```
<filter-mapping>
    <filter-name>WicketFilter</filter-name>
    <url-pattern>/wicket/*</url-pattern>
</filter-mapping>
```

This filter reads one context parameter:

```
<context-param>
    <param-name>configuration</param-name>
    <param-value>deployment</param-value>    ①
</context-param>
```

① alternatively set to "development"; see deployment types for further discussion.

9.3.5. `IsisSessionFilter`

The `IsisSessionFilter` is responsible for the (persistence) session management; in effect a wrapper around DataNucleus' `PersistenceManager` object. It is only required for the Restful Objects viewer.

```
<filter>
    <filter-name>IsisSessionFilterForRestfulObjects</filter-name>
    <filter-class>org.apache.isis.core.webapp.IsisSessionFilter</filter-class>
    <init-param>
        <param-name>authenticationSessionStrategy</param-name>  ①
        <param-value>

org.apache.isis.viewer.restfulobjects.server.authentication.AuthenticationSessionStrat
egyBasicAuth
        </param-value>
    </init-param>
    <init-param>
        <param-name>whenNoSession</param-name>                  ②
        <param-value>basicAuthChallenge</param-value>
    </init-param>
    <init-param>
        <param-name>passThru</param-name>                       ③
        <param-value>/restful/swagger</param-value>
    </init-param>
    <!--
    <init-param>
        <param-name>restricted</param-name>                     ④
        <param-value>...</param-value>
    </init-param>
    <init-param>
        <param-name>redirectToOnException</param-name>          ⑤
        <param-value>...</param-value>
    </init-param>
    -->
</filter>
```

① pluggable strategy for determining what the authentication session (credentials) are of the request

② what the servlet should do if no existing session was found. Usual values are either `unauthorized`, `basicAuthChallenge` or `auto`. Discussed in more detail below.

③ specify which URIs to ignore and simply passthru. Originally introduced to allow the `SwaggerSpec` resource (which does not require a session) to be invoked.

④ List of paths that are allowed through even if not authenticated. The servlets mapped to these paths are expected to be able to deal with there being no session. Typically they will be logon pages. See below for further details.

⑤ where to redirect to if an exception occurs.

The `whenNoSession` parameter determines what the behaviour should be if no existing session can be found. There are a number of predetermined values available:

- `unauthorized` will generates a 401 response
- `basicAuthChallenge` will also generate a 401 response, and also issues a Basic Auth challenge

using `WWW-Authenticate` response header

- `auto` combines the `unauthorized` and `basicAuthChallenge` strategies: it will generate a 401 response, but only issues a Basic Auth challenge if it detects that the request originates from a web browser (ie that the HTTP `Accept` header is set to `text/html`). This means that custom Javascript apps can perform their authentication correctly, while the REST API can still be explored using the web browser (relying upon the web browser's in-built support for HTTP Basic Auth).
- `continue`, in which case the request is allowed to continue but the destination expected to know that there will be no open session
- `restricted`, which allows access to a restricted list of URLs, otherwise will redirect to the first of that list of URLs

If accessing the REST API through a web browser, then normally `basicAuthChallenge` is appropriate; the browser will automatically display a simple prompt. If accessing the REST API through a custom Javascript app, then `unauthorized` is usually the one to use.

This filter should be mapped to the `servlet-name` for the RestEasy `HttpServletDispatcher`; for example:

```
<filter-mapping>
    <filter-name>IsisSessionFilterForRestfulObjects</filter-name>
    <servlet-name>RestfulObjectsRestEasyDispatcher</servlet-name>
</filter-mapping>
```

9.3.6. `IsisTransactionFilterForRestfulObjects`

The `IsisTransactionFilterForRestfulObjects` filter simply ensures that a transaction is in progress for all calls routed to the RestfulObjects viewer.

Its definition is:

```
<filter>
    <filter-name>IsisTransactionFilterForRestfulObjects</filter-name>
    <filter-
class>org.apache.isis.viewer.restfulobjects.server.webapp.IsisTransactionFilterForRest
fulObjects</filter-class>
</filter>
```

This filter should be mapped to the `servlet-name` for the RestEasy `HttpServletDispatcher`; for example:

```
<filter-mapping>
    <filter-name>IsisTransactionFilterForRestfulObjects</filter-name>
    <servlet-name>RestfulObjectsRestEasyDispatcher</servlet-name>
</filter-mapping>
```

9.4. Configuration Files

However Apache Isis is bootstrapped (using the `IsisWicketApplication` or using `IsisWebAppBootstrapper`), it will read a number of configuration files, such as `isis.properties`.

By default these are read from `WEB-INF` directory. This can be overriden using the `isis.config.dir` context parameter:

```
<context-param>
  <param-name>isis.config.dir</param-name>
  <param-value>location of your config directory if fixed</param-value>
</context-param>
```

Another context parameter, `isis.viewres` specifies which additional configuration files to search for (over and above the default ones of `isis.properties` et al):

```
<context-param>
    <param-name>isis.viewers</param-name>
    <param-value>wicket,restfulobjects</param-value>
</context-param>
```

For example, this will cause `viewer_wicket.properties` and `viewer_restfulobjects.properties` to also be loaded.