Apache Isis Lab: RRRADDD!!!

In this lab we shall run the Apache Isis quickstart archetype and familiarise ourselves with the app - the ubiquitous 'todo list' app. Then we'll strip away the main business rules and slowly add them back in so as to see how it fits all together. If we have time, we'll also look at Isis' ability to automatically create a RESTful API from a domain object model.

1. Prerequisites

The minimum you need installed is Java JDK 1.6 and mvn 3. To make your code edits you can optionally use an IDE with good Maven support; we tend to develop using Eclipse + m2e, but NetBeans and IntelliJ also have great Maven tooling.

For the later exercises, you can configure the app to run with a different RDBMS; the app is configured to use HSQLDB, but there are also (commented-out) configuration entries for PostgreSQL.

2. Run the "quickstart" app

Download and extract the app

There's been a lot of development on Apache Isis over the last few months, but we haven't yet got around to pushing out a new release. So, we'll be working with the latest snapshot.

Isis also provides a quickstart archetype, and this lab is based on that archetype. Since we don't need all the modules in that archetype, instead we'll be using the ZIP file provided by the instructor. This ZIP file also contains a number of code fragments for you to copy-n-paste later on in the lab.

Therefore:

- create a new working directory
- ask the instructor for the URL to the ZIP file (`myapp.zip`), and download it this directory.
- unzip the `myapp.zip`, creating a new `myapp` directory.

Explore the application

Locate the parent pom in `myapp/pom.xml`; the `<groupId>` and `<artifactId>` should be set to:

```
<groupId>com.mycompany</groupId>
<artifactId>myapp</artifactId>
<version>1.0.0-SNAPSHOT</version>
```

Also note the `<modules>` element:

```
<modules>
  <module>dom</module>
  <module>fixture</module>
  <module>objstore-jdo</module>
  <module>viewer-wicket</module>
  <module>viewer-restfulobjects</module>
</modules>
```

These modules correspond to the following files and key directories:

`myapp/`	`pom.xml`		parent module
	`dom/`	`pom.xml` `src/main/java/`	Domain object model entities and domain services
	`fixture/`	`pom.xml`	Fixtures (initial data)
	`objstore-jdo/`	`pom.xml` `src/main/java/`	Domain service implementations when using the JDO object store
	`viewer-wicket/`	`pom.xml`	Wicket viewer webapp
		`src/webapp/WEB-INF/`	web.xml and Isis config files
	`viewer-restfulobjects/`	`pom.xml`	Restful Objects viewer webapp
		`src/webapp/WEB-INF/`	web.xml and Isis config files

If you are using an IDE, this would be a good point to import the project so you can more easily explore the generated classes. However, do note that if you recompile the code in the IDE, be aware that the JDO enhancement process must be performed. There are plugins for major IDEs available[1]; but for the purposes of this lab we recommend that you always build from the mvn command-line.

Build the App

To build the app from the command line, just use:

```
mvn clean install
```

All being well, the application should compile. This may take a little while for the first time as modules are brought down from the remote snapshot repo.

Starting the (wicket viewer) webapp

Now that you've built the application, it's time to run it. For now we'll focus on the wicket webapp; later on we'll look at the restfulobjects webapp.

The easiest way to run the app is as a self-hosting version of the WAR file. This is a regular WAR, but includes an embedded jetty webserver (allowing the port number to be changed).

You can run it directly from Maven using:

```
mvn antrun:run
```

If you dig into the configuration (in the `viewer-wicket/pom.xml`) you'll see that this just runs
`java -jar viewer-wicket/target/myapp-viewer-1.0.0-SNAPSHOT-jetty-console.war.`

In the lab you'll be making various code edits, meaning you need to recompile and re-run. You can combine these steps into one:

```
mvn clean package antrun:run -o
```

The -o flag means to run offline, saving some time. You should then find the webapp at http://localhost:8080 (or whatever port number you specify in the self-hosting dialog box). To stop

[1] See http://www.datanucleus.org/products/accessplatform_3_1/jdo/guides/eclipse.html
http://www.datanucleus.org/products/accessplatform_3_1/guides/idea/index.html,
http://www.datanucleus.org/products/accessplatform_3_1/guides/netbeans/index.html

the app you'll need to use ctrl+C from the console to kill the Maven process, and also press the 'Exit' button on the splash screen to kill the self-hosted WAR process.

If you don't want to use the self-hosting WAR file, then an alternative is to run using Maven's Jetty plugin. To do this, you need to switch into the `viewer-wicket` directory:

```
cd viewer-wicket
mvn jetty:run
```

Run this way, you only need to ctrl+C from the console to kill the app.

Making changes and re-running is a little more involved if using the jetty plugin, because the `package` goal must be performed at the parent POM:

```
cd .. ; mvn clean package ; cd viewer-wicket ; mvn jetty:run
```

One thing to note if you use the `jetty:run` goal is that the URL will include the webapp context, ie will be http://localhost:8080/myapp-viewer-wicket.

Use the (wicket viewer) webapp

Now you have the webapp running, you can browse to the app and play around:

- navigate to http://localhost:8080 (or whichever port you specified)
- if required, enter user/password as *sven/pass*
- use the `Fixtures>Install` action to create some initial `ToDoItem`s.

  When you run this, you'll see "No results". This is fine… it just means that the action you've invoked is a `void` method.
- use the `ToDo` menu item to list existing `ToDoItem`s and to create new ones
- edit a `ToDoItem`'s properties, changing its description, its due by date, its category or adding notes

  mark a `ToDoItem` as done or not done using the action menu
- add and remove dependencies for a `ToDoItem`
- the `AuditServiceDemo` menu demonstrates a simple auditing capability provided by the JDO object store

As mentioned above, the app can be stopped by using ctrl+C in the console, along with pressing the Exit button on the splash screen if running the self-hosted WAR.

Reviewing the domain services

The menu items at the top of the web page are called domain services. These typically act as either a repository (find existing stuff) or as a factory (create new stuff).

The domain services are registered in `WEB-INF/isis.properties`, under the `isis.services` key. Try commenting out the fixtures and audit services; they should no longer appear in the top menu.

3. Strip back the app

Now you are somewhat familiar with the app, it's time to strip back things back. We'll then go through and start adding features back in, so that you can see how things fit together.

Strip back the ToDoItem class

Virtually all the functionality in the app is determined by the `ToDoItem` class, which resides in `dom/src/main/java/dom/todo/ToDoItem.java`. Replace this with the stripped back version provided for you in `dom/src/main/labfiles` directory:

```
cp dom/src/main/labfiles/ToDoItem.java dom/src/main/java/dom/todo/ToDoItem.java
```

When you've done this, you'll find that the `ToDoItem`'s class body now reads:

```
// do not alter the package, imports or annotations

public class ToDoItem {

    public static enum Category {
        Professional, Domestic, Other;
    }

    // {{ Description (property)
    private String description;
    public String getDescription() {
        return description;
    }
    public void setDescription(final String description) {
        this.description = description;
    }
    // }}

    // {{ Category (property)
    private Category category;
    public Category getCategory() {
        return category;
    }
    public void setCategory(final Category category) {
        this.category = category;
    }
    // }}

    // {{ DueBy (property)
    private LocalDate dueBy;
    @javax.jdo.annotations.Persistent   // DO NOT REMOVE
    public LocalDate getDueBy() {
        return dueBy;
    }
    public void setDueBy(final LocalDate dueBy) {
        this.dueBy = dueBy;
    }
    // }}

    // {{ Done (property)
    private boolean done;
    public boolean getDone() {
        return done;
    }
    public void setDone(final boolean done) {
        this.done = done;
    }
    // }}
    // {{ Notes (property)
    private String notes;
    public String getNotes() {
        return notes;
    }
    public void setNotes(final String notes) {
        this.notes = notes;
    }
    // }}

    // {{ OwnedBy (property, hidden)
    private String ownedBy;
    public String getOwnedBy() {
        return ownedBy;
    }
    public void setOwnedBy(final String ownedBy) {
        this.ownedBy = ownedBy;
    }
    // }}

    // {{ injected: DomainObjectContainer
    private DomainObjectContainer container;
    public void setDomainObjectContainer(final DomainObjectContainer container) {
        this.container = container;
    }
    // }}

    // {{ injected: ToDoItems
    private ToDoItems toDoItems;
    public void setToDoItems(final ToDoItems toDoItems) {
        this.toDoItems = toDoItems;
    }
    // }}
}
```

As might be obvious, this class is pretty much a simple POJO (albeit with some JDO annotations). We tend to group the instance variable, the getter and the setter together because collectively these three members constitute a single responsibility. The `{{` and `}}` brackets are used by a code-folding utility that some in the Isis community use (available to download from the Apache Isis website, http://incubator.apache.org/isis).

Reviewing the ToDoItem class

Even though the `ToDoItem` class has been stripped down, you can still run the application (as above). Some things will have disappeared, such as the actions and the collection, and you may find when you edit an object that some properties are now mandatory … but fundamentally the app still runs.

Some things to notice about the code:

- the `ToDoItem` entity does not need to inherit from a framework class:

```
public class ToDoItem {
...
}
```

- down the bottom of the class, Isis' `DomainObjectContainer` can optionally be injected:

```
    // {{ injected: DomainObjectContainer
    private DomainObjectContainer container;
    public void setDomainObjectContainer(final DomainObjectContainer container) {
        this.container = container;
    }
    // }}
```

  This is used to obtain the current user's identity, search for objects (though normally this is delegated to repositories) and raise warnings/errors to the user.

- Other domain services (eg repositories) can also be injected:

```
    // {{ injected: ToDoItems
    private ToDoItems toDoItems;
    public void setToDoItems(final ToDoItems toDoItems) {
        this.toDoItems = toDoItems;
    }
    // }}
```

Reviewing the JDO annotations

The `ToDoItem` type signature is already annotated for use with the JDO objectstore, meaning that it has a number of JDO-specific annotations:

```
@javax.jdo.annotations.PersistenceCapable(identityType=IdentityType.DATASTORE)
@javax.jdo.annotations.DatastoreIdentity(
          strategy=javax.jdo.annotations.IdGeneratorStrategy.IDENTITY)
@javax.jdo.annotations.Version(strategy=VersionStrategy.VERSION_NUMBER, column="VERSION")
public class ToDoItem {
```

Taken together, these specify that the entity is persistent, its unique id should be assigned by the JDO runtime, and that a version field should be introduced for optimistic locking

By and large JDO will automatically persist most datatypes, such as strings, ints, floats, enums etc:

```
    public String getDescription() {
    …
    public boolean getDone() {
    …
    public Category getCategory() {   // an enum
    …
```

JDO can also support third-party types such as Joda's `LocalDate`; however it requires that the property is annotated:

```
    …
    @javax.jdo.annotations.Persistent
    public LocalDate getDueBy() {
    …
```

For its part, Isis knows how to render these properties of different types appropriately (a date picker for dates, a checkbox for booleans, etc.). Entities can also have references to other entities (though the `ToDoItem` does not happen to have any as such).

4. Adding back some basic semantics

Although Isis can run the app with our very simple `ToDoItem`, the result isn't that pretty, and it no longer implements all the business rules that we might want to enforce. So let's start to add back in all the code we stripped out, understanding what it does as we go.

Adding a title

One of the first things we typically will do is to define a title for the entity. This is done with the `@Title` annotation on the getter for the property:

```
    @Title
    public String getDescription() {
```

This annotation (along with the others we discuss below), lives in `org.apache.isis.annotation`.

Run the application, and confirm that the `ToDoItem` instances (eg when returned from the `ToDoItem` > `NotYetDone` action) can now be distinguished from each other.

If we wanted to define a title from multiple properties, that too is supported. If the requirements are more complex than that, we can instead define a `title()` method; the following is equivalent:

```
    public String title() {
       return getDescription();
    }
```

While we're at it, you might ask: how does Isis know to render an icon for the entity? The answer is a bit of convention over configuration: Isis will look for an image file in the `images` package; in the case of `ToDoItem` it finds a resource `ToDoItem.gif` (in `src/main/resources` of the `dom` module).

Member Ordering

The next improvement we can make is to specify the order in which the class members (properties) should be rendered. This is done using the `@MemberOrder` annotation (again, on the getter):

```
    @MemberOrder(sequence = "1")
    public String getDescription() {
    …
    @MemberOrder(sequence = "2")
    public Category getCategory() {
    …
    @MemberOrder(sequence = "3")
    public LocalDate getDueBy() {
    …
    @MemberOrder(sequence = "4")
    public boolean getDone() {
    …
    @MemberOrder(sequence = "5")
    public String getNotes() {
```

Don't bother to annotate the `ownedBy` property.

Make these changes and run the app, and confirm that the `ToDoItem`'s properties are now displayed in the order specified. Change the sequence and check that the properties reorder in the web page. And as you should see, any properties without a `@MemberOrder` will be rendered at the end.

Member Groups

We can also put these properties into groups. Let's put the `dueBy` and `notes` properties into their own "Detail" group:

```
    @MemberOrder(name="Detail", sequence = "3")
    public LocalDate getDueBy() {
    …
    @MemberOrder(name="Detail", sequence = "5")
    public String getNotes() {
```

To specify the order of the member groups themselves, we use the `@MemberGroups` annotation on the `ToDoItem` class:

```
@MemberGroups({"General", "Detail"})
public class ToDoItem {
```

Run the app again to confirm the change.

Hidden properties

Sometimes we don't want properties to be visible at all. In the case of the `ToDoItem`, each instance has an `ownedBy` property, so we can distinguish between items created by one user and those created by another. But any given user probably wouldn't want to see this property, because it will always just show their user name. We therefore hide it using the `@Hidden` annotation:

```
    @Hidden
    public String getOwnedBy() {
```

This annotation has another trick up its sleeve. As you'll have seen, we can view our `ToDoItem`s either in list form (as the result of invoking one of the repository queries) or in a form. We'll usually want to see all the properties in the form, but showing them also in tables may be too much clutter.

We can therefore improve matters by once again using the `@Hidden` annotation, but this time with a `where` qualifier. Let's do that for the `notes` property:

```
    @Hidden(where=Where.ALL_TABLES)
    public String getNotes() {
```

Now would be a good time to run the application and confirm the last two changes.

Rendering hints

A common reason for hiding properties from the table views is for properties that are likely to contain a lot of content. The `notes` property is one such, but trying to write additional detail into a single line text box isn't particularly convenient. We can fix this using the `@MultiLine` annotation:

```
    @MultiLine(numberOfLines=5)
    public String getNotes() {
```

Optional properties

Not every property of an entity is likely to be mandatory; good examples are the `dueBy` and `notes` properties. We can indicate that these are not required using the `@Optional` annotation:

```
    @Optional
    public LocalDate getDueBy() {
    …
    @Optional
    public String getNotes() {
```

Isis' convention that properties are mandatory unless stated otherwise is deliberate: it takes business analysis to discover that a property is truly optional, so Isis makes the conservative assumption.

If you haven't run the app recently, do so now to confirm your changes.

Derived Properties

Thus far every property of the `ToDoItem` is persisted. But Isis also supports derived properties; these are implicitly read-only.

An example of this is the property to expose the optimistic locking version number. Because of the way that JDO works, the version property that it introduces to the class (see `@javax.jdo.annotation.Version` annotation shown earlier) is not a pojo property, and so it isn't visible in the UI.

If we wanted to make the value of this version property visible, we can use the following code (to save typing, you can copy-n-paste from `dom/src/main/labfiles/getVersionSequence.java`):

```
    // {{ Version (derived property)
    @Hidden(where=Where.ALL_TABLES)
    @MemberOrder(sequence = "99")
    public Long getVersionSequence() {
        if(!(this instanceof PersistenceCapable)) {
            return null;
        }
        PersistenceCapable persistenceCapable = (PersistenceCapable) this;
        final Long version = (Long) JDOHelper.getVersion(persistenceCapable);
        return version;
    }
    // }}
```

We've chosen to name the property `versionSequence` just to demonstrate that Isis can optionally rename properties in the UI; useful in those cases where the desired name is a reserved word in Java. This is done with the `@Named` annotation:

```
    @Named("Version")
    public Long getVersionSequence() {
```

Run the app once more to confirm that the property appears on object forms (but not in table views), is labelled "Version", and is read-only. Also check that each time you do edit the object, the value should increase.

Hiding Class Members Imperatively

The `version` property is only maintained if running with the JDO object store. We ought therefore to hide this property if the application is being run with something other than JDO.

This sort of conditional logic can't be handled using the declarative `@Hidden` annotation, and so Isis offers an imperative alternative, the `hideXxx()` method:

```
    public boolean hideVersionSequence() {
        return !(this instanceof PersistenceCapable);
    }
```

Isis uses the name of this method to match up with the corresponding property; in this case "VersionSequence". As you'll see later in the lab, the same convention also applies to collections and actions. If the result of the `hideXxx()` method is `true`, then the property/collection/action is hidden in the UI.

As a matter of style, these supporting methods are usually placed alongside the property that they pertain to (within the `{{` and `}}` brackets).

5. Adding some Richer Behaviour

Although Isis can be used to easily create a CRUD application, it was in fact designed to help build enterprise applications with complex business logic. Isis' is all about enabling you to rapidly develop domain-driven applications, and it does this by letting you iterate very rapidly in order to build up a ubiquitous language with the business/domain experts.

Of course, with our simple `ToDoItem` class, it's somewhat difficult to show this in practice, but we can at least demonstrate the principle. Let's start by adding some behaviour using actions.

Adding markAsDone / markAsNotDone actions

Rather than simply check/uncheck the `done` property, let's toggle the state using actions. First, we'll start off by make the `done` property read-only. This is done using the `@Disabled` annotation:

```
    @Disabled
    public boolean getDone() {
```

By the way, if we wanted to, we could have also done this imperatively using a `disableXxx()` method:

```
    public String disableDone() {
        return "Use the 'markAsDone' and 'markAsNotDone' actions";
    }
```

Any non-null string returned is taken to be the reason why the property cannot be modified. As for the `hideXxx()` method, there would normally be some sort of conditional check (otherwise just use a declaration annotation).

Back to the main story; next we'll add the actions `markAsDone()` and `markAsNotDone()` (to save typing, you can find the code in `dom/src/main/labfiles/actions.java`):

```
    // {{ markAsDone (action)
    @MemberOrder(sequence = "1")
    public ToDoItem markAsDone() {
        setDone(true);
        return this;
    }
    // }}

    // {{ markAsNotDone (action)
    @MemberOrder(sequence = "2")
    public ToDoItem markAsNotDone() {
        setDone(false);
        return this;
    }
    // }}
```

As you can see, these are just public methods that aren't recognised as properties or any of the supporting methods (such as `hideXxx()` and `disableXxx()`)

Try the application out; you should now have action items to invoke, with the `done` property no longer editable. Invoking the action should flip the `done` property appropriately; note that it also increments the `version` property.

Disabling actions

Just as we can disable properties, we can also disable (in other words, "grey out") actions.

In our case, it doesn't make sense to invoke `markAsDone()` if the `ToDoItem` is already checked as done; conversely we shouldn't be able to invoke `markAsNotDone()` if the `ToDoItem` is not checked as done.

We can enforce this using more `disableXxx()` imperative actions:

```
    public String disableMarkAsDone() {
        return done ? "Already done" : null;
    }
    …
    public String disableMarkAsNotDone() {
        return !done ? "Not yet done" : null;
    }
```

Remember to follow good style and to put these supporting methods alongside the actions to which they relate.

Try the application out and confirm that the actions are disabled appropriately.

6. Adding a Collection and supporting actions

The last bit of code that we had stripped out was a collection of dependencies: the idea being that one `ToDoItem` might depend on another `ToDoItem` that must be done first.

Adding a Collection

We introduce the collection as so (again, available at `dom/src/main/labfiles/collection.java`):

```
    // {{ Dependencies (Collection)
    private List<ToDoItem> dependencies = new ArrayList<ToDoItem>();
    @Disabled
    @MemberOrder(sequence = "1")
    public List<ToDoItem> getDependencies() {
        return dependencies;
    }
    public void setDependencies(final List<ToDoItem> dependencies) {
        this.dependencies = dependencies;
    }
    // }}
```

It's conventional to mark annotations as `@Disabled`.

You can run the application if you want to confirm that the collection appears; however you won't yet be able to populate the collection.

Adding actions for the collection

To populate the collection, we can use actions (available to copy-n-paste from `dom/src/main/labfiles/collection-actions.java`):

```
    // {{ add (action)
    @MemberOrder(sequence = "3")
    public ToDoItem add(final ToDoItem toDoItem) {
        getDependencies().add(toDoItem);
        return this;
    }
    // }}
```

and:

```
    // {{ remove (action)
    @MemberOrder(sequence = "4")
    public ToDoItem remove(final ToDoItem toDoItem) {
        getDependencies().remove(toDoItem);
        return this;
    }
    // }}
```

These are a little more interesting than the `markAsDone()` and `markAsNotDone()` actions we saw earlier, in that they take a `ToDoItem` parameter. Isis automatically renders a dialog for these, and provides an auto-complete list of items for us to select (more on this shortly).

Try running the application and check that the collection appears and that dependencies can be added and removed.

Grouping actions by collection

Another cosmetic improvement we can make is to have these actions render closer to the collection that they act upon. We can do that by updating the `@MemberOrder` annotation with a `name` attribute:

```
    @MemberOrder(name="dependencies", sequence = "1")
    public ToDoItem add(final ToDoItem toDoItem) {
    ….
    @MemberOrder(name="dependencies", sequence = "2")
    public ToDoItem remove(final ToDoItem toDoItem) {
```

If you wish, run the application again and confirm the change.

Eagerly resolving collections

Although we can now add dependencies between `ToDoItem`s, the table view does not show these when a `ToDoItem` object is initially displayed. Sometimes this is what we want, of course … there's a performance hit to pulling back additional data. But if we have a collection whose contents is often used, then we can indicate that it should be automatically "opened" using the `@Resolve` annotation:

```
@Resolve(Type.EAGERLY)
public List<ToDoItem> getDependencies() {
```

Run the application and check that the contents of the collection are shown by default.

Disabling an action

Although the actions above work, there are some business rules that we should enforce. For example, we shouldn't really be able to invoke the `remove()` action if the `dependencies` collection is empty. We can do this with another `disableXxx()` method:

```
    public String disableRemove() {
        return getDependencies().isEmpty()? "No dependencies to remove": null;
    }
```

Again, it's good style to put this supporting method between the `{{` and `}}` comments.

Validating action arguments

Not to be confused with disabling actions, we can also perform validation for those actions (such as `add(…)` and `remove(…)`) that accept arguments.

In the case of `add(…)`, for example, it probably doesn't make sense to add a dependency on a `ToDoItem` back to itself. Moreover, we also shouldn't be able to add set up a dependency more than once. We enforce this using a `validateXxx()` method:

```
    public String validateAdd(final ToDoItem toDoItem) {
        if(toDoItem == this) {
            return "Can't set up a dependency to self";
        }
        if(getDependencies().contains(toDoItem)) {
            return "Already a dependency";
        }
        return null;
    }
```

As for `disableXxx()`,a non-null string returned is taken as the reason why the action can't be invoked (with this argument).

Similarly, it doesn't make sense to try to remove an item that isn't yet a dependency:

```
    public String validateRemove(final ToDoItem toDoItem) {
        if(!getDependencies().contains(toDoItem)) {
            return "Not a dependency";
        }
        return null;
    }
```

Run the application and check that the above business rules are now applied.

Adding choices for action arguments

For some actions we can help the user by providing a list of choices of valid values; the `remove(…)` action is one such. We provide the list of choices using the `choicesNXxx()` method:

```
    public List<ToDoItem> choices0Remove() {
        return getDependencies();
    }
```

Note that the name of this action is `choicesNXxx()`, where N is the 0-based argument number, and Xxx is the action name as usual.

To test this last change, run the app, add a couple of dependencies, and then invoke the action to remove a dependency. Only the two dependencies that were added should be listed in the drop-down.

7. AutoComplete and Repositories

At this point we have added back in all the logic that we had originally stripped out. However, we did skip over the way in which Isis provides auto-complete behaviour for action arguments, so let's dig into that now.

The AutoComplete annotation

To recap, in the `add(…)` action Isis automatically provided a candidate list of objects. This happens because the `ToDoItem` class is annotated with the `@AutoComplete` annotation:

```
    @AutoComplete(repository=ToDoItems.class, action="autoComplete")
    public class ToDoItem {
```

In the `ToDoItems` repository, there is indeed an `autoComplete()` action. This accepts a single string (the value typed in by the user), and provides a candidate list of items:

```
    // {{ AutoComplete (hidden)
    @Hidden
    @MemberOrder(sequence = "1")
    public List<ToDoItem> autoComplete(final String description) {
        return allMatches(ToDoItem.class, new Filter<ToDoItem>() {
            @Override
            public boolean accept(final ToDoItem t) {
                return ownedByCurrentUser(t) &&
                        t.getDescription().contains(description);
            }
        });
    }
    // }}
```

The implementation shown here is 'naive', in that it uses the injected `DomainObjectContainer#allMatches(…)` method, passing in a `Filter<ToDoItem>`. The net effect is that all objects are returned, and then are filtered client side. This is fine for initial prototyping, but won't scale for larger volumes of data.

Alternative Domain Service Implementations

As an improvement on the above code, we can provide an alternative implementation that is more efficient… in other words, have the database do the filtering and return back only the matching objects.

This necessarily binds the domain service implementation to a specific object store implementation. For example, the `ToDoItemsJdo` implementation (a subclass of `ToDoItems`) uses the JDO object store's support for queries:

- the JDO named query is defined on the entity

```
@javax.jdo.annotations.Queries( {
    …
    @javax.jdo.annotations.Query(
        name="todo_autoComplete", language="JDOQL",
        value="SELECT FROM dom.todo.ToDoItem WHERE ownedBy == :ownedBy &&
description.startsWith(:description)")
})
public class ToDoItem  {
```

- the `ToDoItemsJdo` method uses this query using `DomainObjectContainer#allMatches(…)`, but passing in a `QueryDefault` rather than a `Filter`:

```
    // {{ autoComplete (action)
    @Override
    public List<ToDoItem> autoComplete(String description) {

        return allMatches(
                new QueryDefault<ToDoItem>(ToDoItem.class,
                        "todo_autoComplete",
                        "ownedBy", currentUserName(),
                        "description", description));
    }
    // }}
```

8. Running the Restful Objects Viewer

Thus far we have been working exclusively with the Wicket viewer. However, Isis also supports a number of other viewers; of these the Restful Objects viewer is interesting in that generates not an HTML website, but instead a set of JSON representations for well-defined resource URLs. This isn't therefore a UI that an end-user can use directly; rather it allows applications (either bespoke or generic) to be written – in whatever language you like – and it is *they* that provide a UI for use by end-users. It also provides a way to allow integrations between different systems.

You can read more about Restful Objects at http://restfulobjects.org.

Optional: Make changes persistent between runs

Before going any further, you will probably want to reconfigure both the wicket viewer and the restfulobjects viewer to use the same database. This will also obviate the need to re-run the fixtures each time.

- Locate the `viewer-wicket/src/webapp/WEB-INF/persistor_datanucleus.properties`
- Comment out the current JDBC properties (for HSQLDB in-memory)
- Uncomment the corresponding properties for HSQLDB configured to write to file
- Do the same for the `viewer-restfulobjects` module (in its `src/webapp/WEB-INF/persistor_datanucleus.properties` file)

Now run the wicket viewer one more time (using `mvn clean package antrun:run`) and populate the database. Make some changes, stop and restart the app, and confirm that those changes are persisted. If you wish, you can also manually verify the HSQLDB files (as per the JDBC properties settings) have been created. And remember that hereafter it won't be necessary to continually re-install the fixtures!

Optional: install jsonview and REST Console (or similar) into your browser

Because the restful objects viewer will be returning JSON rather than HTML, it's convenient (both for demo purposes now and in general development) to have a development tool that allows us to inspect and interact with that JSON. Both Chrome and Firefox have a *jsonview* extension (which displays JSON in a colour-coded form) and *REST Console* extension (for submitting HTTP requests).

Starting the restful objects viewer webapp

The restfulobjects webapp is run in the same way as the wicket webapp. However, to get the `mvn antrun:run` goal to run the restfulobjects WAR rather than the wicket WAR, reverse the order of the `<modules>` entries in the parent `pom.xml`, so that `viewer-restfulobjects` comes before `viewer-wicket`:

```
<modules>
  <module>dom</module>
  <module>fixture</module>
  <module>objstore-jdo</module>
  <module>viewer-restfulobjects</module>
  <module>viewer-wicket</module>
</modules>
```

You can now run the restfulobjects webapp in exactly the same way that you have been running the wicket viewer:

```
mvn clean package antrun:run
```

If you've been using Maven's jetty plugin, then (as you can probably guess) you can run the restfulobjects webapp using:

```
mvn clean package ; cd viewer-restfulobjects ; mvn jetty:run
```

"Using" the restful objects viewer webapp

With the webapp running, you can now navigate to the home resource, at http://localhost:8080.

- The user/password is *sven/pass* (as before)
- from there, click through to `~/services`, and then navigate to the `ToDoItems` service (`~/services/toDoItems`).
- from there, browse to the details of the `notYetDone()` action (`~/services/toDoItems/actions/notYetDone`);
- from there, invoke the action
- after that, you should be able to navigate to a `ToDoItem`, and inspect its contents.

Thus far we have been able to navigate between representations using HTTP GET requests. To invoke a non-safe operation (one that changes state) however, we must invoke either with HTTP PUT or HTTP POST. The links rendered by restful objects indicate which HTTP method to use (the former for idempotent operations, the latter for non-idempotent operations).

The *jsonview* extension does not support this, but the *REST Console* extension does. So, using this knowledge, see if you can use *REST Console* to:

- update a property (such as a description)
- mark a `ToDoItem` as done
- create a new `ToDoItem`.

9. What Next?

We hope you've enjoyed working your way through this lab. As you have now seen, Apache Isis is a powerful framework enabling very rapid development of your domain application. And with the Wicket viewer and the JDO object store, it is built upon well-established existing frameworks.

At the same time, because of the minimal coupling of your domain model to Isis (basically: programming conventions and annotations), you are also free to take your domain model and deploy it on some other platform if you wish. So Isis is also excellent for rapid prototyping.

Apache Isis entered the Apache incubator in Sept 2010 and was voted to graduate in Oct 2012 as a top-level project. You can find further information either at:

- http://incubator.apache.org/isis (the old website), or
- http://isis.apache.org (the new website – once we get it up and running)

From there you can find links to various resources; the standard way to get help with Apache projects is through the mailing lists linked from the above sites.