T2 PLATFORM DEVELOPER'S WORKSHOP

LOGISTICS & WORKSHOP EXERCISES, TOM OINN, 16TH FEB 2009

This document contains the agenda for the workshop, a map of the important places (food, accommodation and the workshop itself!) and the exercises for the technical content of the first day and a half. The latter in particular are not stand-alone and require reference to the Platform Developer's Guide and the platform Javadoc, supplied in the 'Platform Docs' directory on the desktop of the virtual machine you'll be using.

CONTENTS

Workshop Programme 4
Day 1 - 18th Feb 2009 4
Day 2 - 19th Feb 2009 5
BoF Sessions 5
Map 6
Ex1. Introducing the Platform 7
ex1.1. Launching a Command Line Application from Eclipse 7
ex1.2. Building and Launching a Command Line Application from the Shell 7
ex1.3. Launching a Web Application from Eclipse 8
ex1.4. Building a WAR file. 9
ex1.5. Questions 10
ex1.6. Summary 10
ex2. Creating a new Platform Project in Eclipse 11
ex2.1. Creating the Project 11
ex2.2. Configuring and Running the Project 12
ex2.3. Summary 12
ex3. Working with Data 13
Ex3.1. Using the Reference Service 13
Ex3.2. Questions 13
ex4. Running a Pre-Defined Workflow 14
Ex4.1. Setup 14
Ex4.2. Loading a Workflow from XML 14
Ex4.3. Running the workflow 14
Ex4.4. Running with basic monitoring 15
Ex4.4. Questions 15
ex5. Creating a Web Application 16
Ex5.1. Setup 16
Ex5.2. Configuring the Web Application 16

Ex5.3. Implementing the Web Application 16
Ex5.4. Running the Web Application 16
Ex5.5. Questions 16
Ex6. Creating a new Activity 18
Ex6.1. Setup 18
Ex6.2. Implementing the Activity 18
Ex6.3. Building the Activity 18
Ex6.4. Describing the Activity 19
Ex6.4.1. Basic properties 19
Ex6.4.2. Authors 19
Ex6.4.3. Artifacts 19
Ex6.4.4. Saving the description 19
Ex7. Creating a Dataflow from the API - Using your new Plug-in 21
Ex7.1. Setup 21
Ex7.2. Configuration 21
Ex7.2.1. Default Plug-in Repository Configuration 21
Ex7.2.2. Default Artifact Repository Configuration 21
Ex7.2.3. Default Plug-in List 21
Ex7.3. Constructing and Running a new Dataflow 21
Ex8. Using the Monitor & MonitorReceiver 22
Ex8.1. Setup 22
Ex8.2. The default Monitor 22
Ex8.3. The invocation context 22
Ex8.4. Fixing the code 22
Ex8.5. Questions 23

WORKSHOP PROGRAMME

DAY 1 - 18TH FEB 2009

9.00-9.30

Registration, pick up packs, badges etc.

9.30-9.50

Welcome, Taverna and related projects roadmap briefing

9.50-10.10

Introductions & short intros from attendees

10.10-10.20

Summary of Workshop Contents, BoF Signup

10.20-11.00

Exercise 1 - Introduction to the Platform

11.00-11.15

Coffee in atrium

11.15-13.00

Exercise 2 - Creating a new Platform Application
Exercise 3 - Working with Data

13.00-13.45

Lunch in atrium

13.45-15.00

Exercise 4 - Running a Workflow
Exercise 5 - Running a Workflow From The Web

15.00-15.15

Coffee in atrium

15.15-17.00

Exercises 4 & 5 Continued

17.00-17.30

Q & A session, feedback and preparation for second day.

20.00

Workshop dinner, meet at Cocotoo (see map for location)

DAY 2 - 19TH FEB 2009

9.00-9.30

Feedback and Agenda re-adjustment

9.30-10.00

Plug-in management in the platform - structure, class-loading and potential pitfalls.

10.00-11.00

Exercise 6 - Creating, Describing and Deploying a New Activity Plug-in

11.00-11.15

Coffee, location TBC

11.15-12.00

Exercise 7 - Constructing a Workflow Programmatically; Using Your New Plug-in

13.00-13.45

Lunch, location TBC

13.45-15.45

Exercise 7 Continued
Advanced topics & **BoF sessions**

15.45-16.00

Coffee, location TBC

16.00-16.30

Advanced topics & **BoF sessions** Continued

16.30-17.30

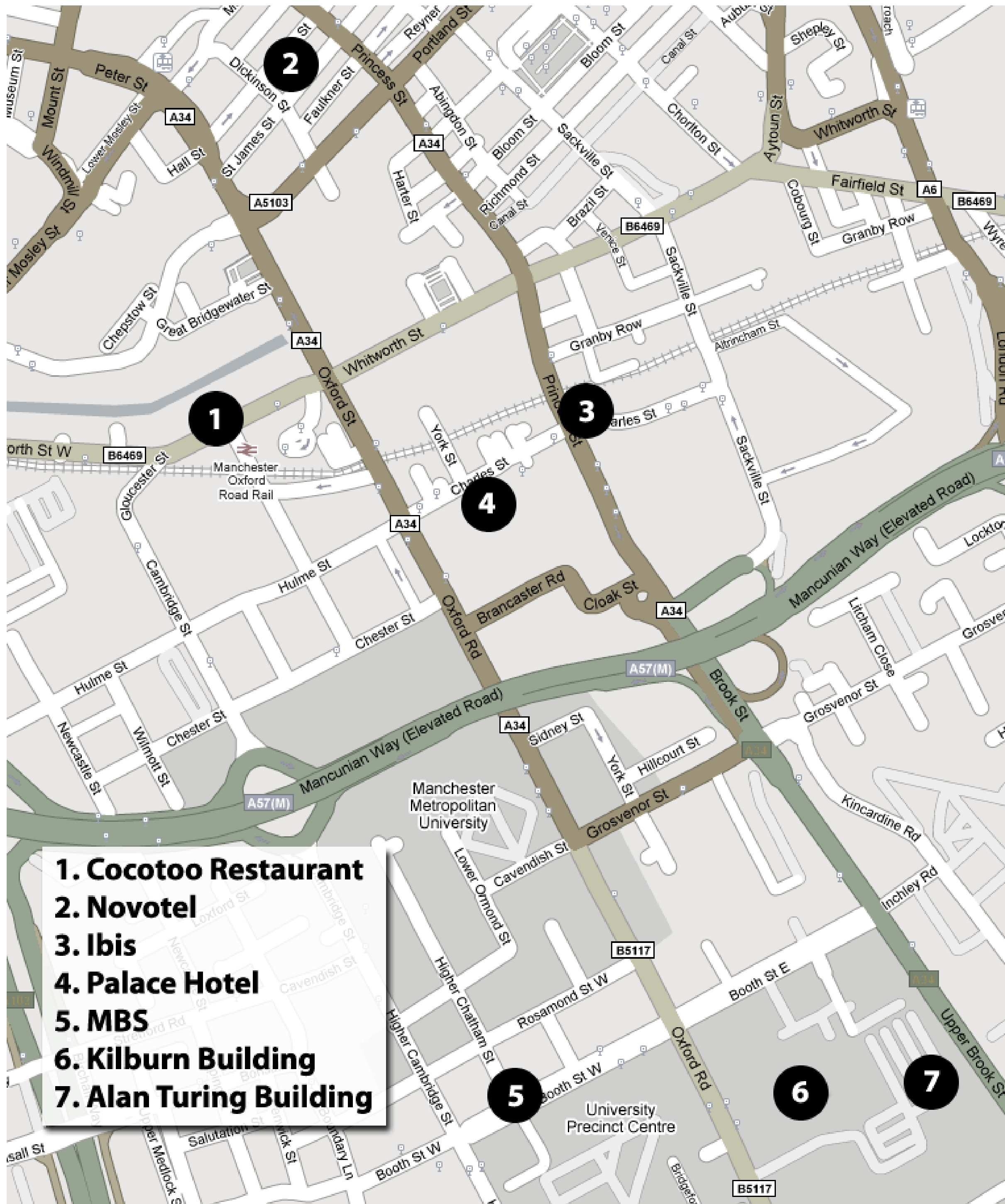
Wrap-up, discussion of future plans.

BOF SESSIONS

Feel free to propose more, three possible platform related sessions could be:

1. Monitoring and Steering
    - How to control and monitor a (long) running workflow
    - Federation of workflow and application level state
    - Extending your Activity to make it monitorable
2. Security and Resource Management in T2
    - Treatment of workflow scope shared resources
    - Use of the Invocation Context to handle single sign-on
    - How to avoid passing session identifiers, user names, passwords etc. in the data flow
3. Implicit handling of Reference Types
    - How do code plug-ins for 'data + compute' style grid systems

MAP



EX1. INTRODUCING THE PLATFORM

> This exercise uses a set of pre-constructed platform applications. The objective is to see them run and to explore the code and build structures and review the various configuration files and cache locations used.

This exercise uses two applications, one command line based and the second a simple web application using a Java Server Page (JSP) file. Both were generated from the platform archetypes and are therefore typical of the applications you'll be creating later in the tutorial.

EX1.1. LAUNCHING A COMMAND LINE APPLICATION FROM ECLIPSE

It's possible to run platform applications directly from within the Eclipse Integrated Development Environment (IDE). This is considerably easier than packaging your application when testing as it avoids a potentially time consuming rebuild.

1. Open the Eclipse IDE, there is a link on the desktop to do this
2. Locate the project 'ex1a' in the panel on the left of the IDE
3. Right click on this and select 'open project'
4. Expand the project to see the components.
5. There is a single java class in the src/main/java node, open this and navigate to the java source file.
6. To run the command line application from within Eclipse right click on the 'Application.java' node, locate the 'Run As' sub-menu and select the 'Java Application' option.
7. You will see the system output in the console within Eclipse, all being well this will tell you that the platform has been initialized, that a new workflow has been created and the print a list of workflow results.

Eclipse has managed the dependencies for you via its Maven plug-in. To see the dependencies for the application you can open the 'Maven Dependencies' node. This will show you a list of all the libraries that make up the platform.

The dependencies are defined in the 'pom.xml', a configuration file driving Maven's build system.

EX1.2. BUILDING AND LAUNCHING A COMMAND LINE APPLICATION FROM THE SHELL

The command line application archetype defines an assembly mechanism. The Maven assembly plug-in is used to create redistributable applications, bundling your code, its dependencies and any other resources. In the case of the assembly used by the platform this includes the Spring configuration files and both shell and batch scripts to run the generated application. This exercise shows how you can build and run the redistributable application from the command line.

1. Close (or minimize) Eclipse, and open a new Terminal using the shortcut on the desktop.
2. The projects in the Eclipse project list are located in the 'workspace' directory, change to it with:
3. cd workspace/ex1a
4. A directory listing (ls) will produce the following:

```
pom.xml   src   target
```

5. The target directory may not be present if you've not done exercise 1.1, as the name suggests it's where Maven builds to.

6. To build the application run:

```
> mvn assembly:assembly
```

7. There should be a large amount of information printed to the console, including a warning message which you can safely ignore.
8. Go into the target directory and list the contents:

```
> cd target
> ls
```

9. You should see several jar files and a couple of archives. The archives contain your redistributable application.
10. You could unzip the archive to run your app, but you don't have to as the build process produces an unzipped version.

```
> cd ex1a/ex1a
> ls
```

11. You should see a run.sh and run.bat file along with a couple of jar files and a conf directory. Run the application:

```
> ./run.sh
```

12. You should see the same output printed to the console as when you ran from Eclipse in the previous exercise.

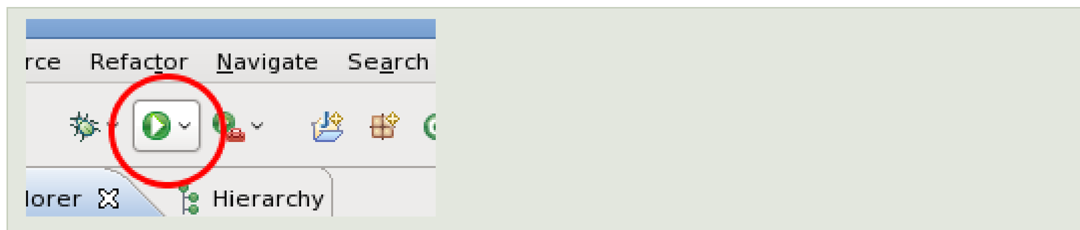
It's generally a good idea to test from the command line from time to time when developing. While Eclipse does a reasonable job of managing dependencies it's possible to confuse it, in particular by having multiple projects open simultaneously. Building and running from the shell acts as a good sanity check that your application isn't working only because of some peculiarity of the IDE.

EX1.3. LAUNCHING A WEB APPLICATION FROM ECLIPSE

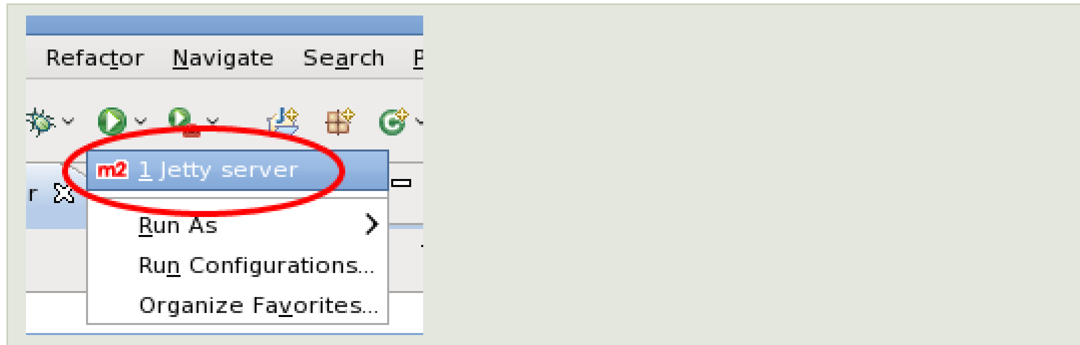
The project 'ex1b' contains a web application. This consists of a single JSP which will register a string as a workflow input, run the workflow (a trivial one using the same single service as the previous exercises) and print the results out. Because this is a web application the results are printed to a HTML table rather than to the console.

> Close ex1a if you haven't already - right click on it and select 'Close Project'. In general it's a good idea to close any projects you're not actively working with, not only does it lighten the load on the IDE but it also prevents any contamination between the libraries used by different projects. Theoretically there shouldn't be any, but there appear to be cases where this isn't quite true!

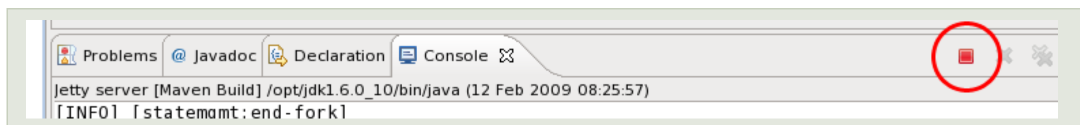
1. Open the 'ex1b' project.
2. Locate the 'run' button in the tool bar, it's a green circle with a right facing arrow
3. Click on the 'drop down' arrow just to the right of this



4. From the drop-down list, select the top one : 'Jetty server'



5. This launches an embedded application server which will listen on port 8080. It prints progress and debug information to the Eclipse console.
6. When the console displays the message '[INFO] Started Jetty Server' you can find your running web application by launching the browser and going to 'http://localhost:8080'. This will send you to a redirect page from where you can select your application (it'll be the only link on the page).
7. You should see a web form, enter the value it suggests and hit return.
8. The results, which should be the same as those from the previous exercises, will be displayed in a table below the input form.
9. To stop the server click on the stop button on the console:



EX1.4. BUILDING A WAR FILE.

As with the command line application, running from Eclipse is very convenient but you will eventually want to deploy your web application to an external server, or make it available for download. This can be done by running a packaging command, either from Eclipse or directly from the shell (which can also be used to launch the Jetty application server directly).

Web applications are packaged in a specialized form of jar file called a WAR (Web application ARchive, presumably). To generate such a file, which can then be deployed into any compliant application server, you need to open a shell, step into the project directory and run the appropriate maven goal:

```
> cd workspace/ex1b
> mvn package
```

This will produce a .war file within the target directory. Running the Jetty server is equally simple, rather than 'mvn package' you use 'mvn jetty:run'.

EX1.5. QUESTIONS

> To help you make sure you've got all the information out of each section every section will finish with a set of questions. These may require some digging around in the project structure or reference to the user manual!

1. Where are the platform configuration files held...
    a. ...in the command line application?
    b. ...in the web application?
    c. ...and where do they end up in the packaged applications in each case?
2. The platform needs to write to a directory somewhere on the file-system to store cached copies of downloaded files, persist the state of the plug-in manager etc.
    a. Where does the command line app store this information?
        i. ...and how do you change this?
    b. Where does the web application store this information?
        i. ...why don't you need to change this?
        ii. *(hint - the application server specification requires that each web-app be given its own working directory by the server)*
3. In Eclipse you can see the dependencies in the 'Maven Dependencies' node.
    a. Where are these dependencies assembled in the command line redistributable package?
    b. ...and in the war file?
4. Normally the platform would download the implementation jar files for the workflow engine and for the plug-ins. It doesn't in this case!
    a. How does the platform determine where to get plug-in definitions from?
    b. ...how about jar files used by plug-ins?
    c. ...and by core implementations?
    d. ...why doesn't it download anything in these examples?

EX1.6. SUMMARY

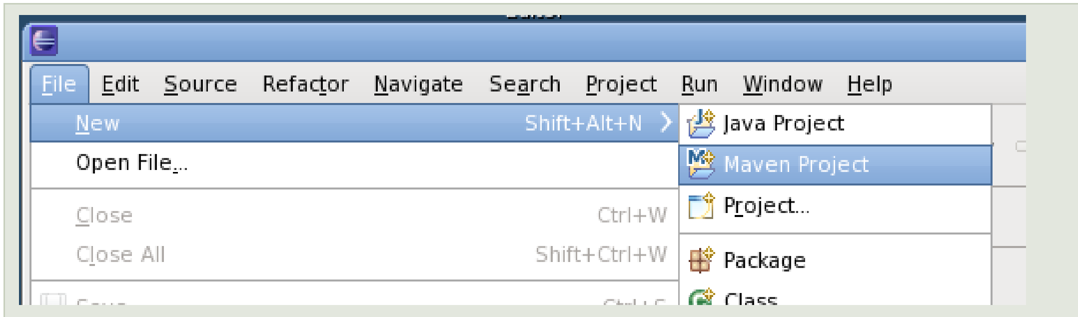
You've seen how to run both command line and web applications, from within Eclipse and from the shell. In addition you can create redistributable packages for both types of application.

EX2. CREATING A NEW PLATFORM PROJECT IN ECLIPSE

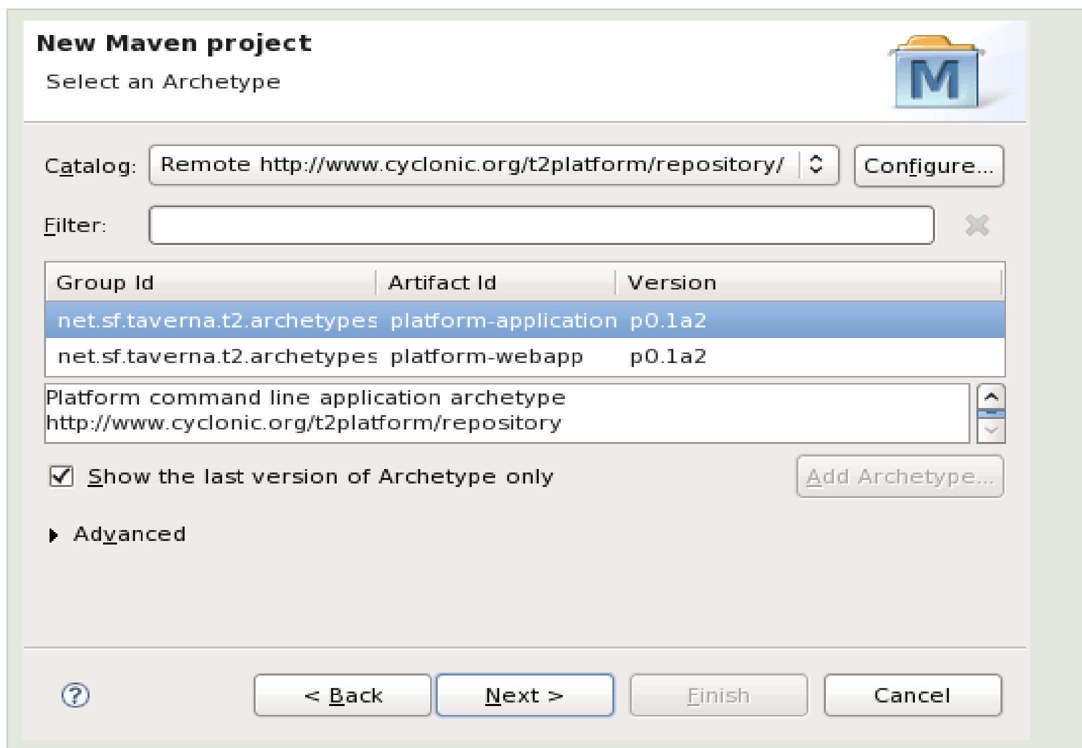
The later exercises will work from existing projects in the Eclipse workspace. First though you should be able to create such a project from scratch.

EX2.1. CREATING THE PROJECT

1. Close any open projects in Eclipse by right clicking on them and selecting 'Close Project'
2. Create a new project from 'File'-'New'-'Maven Project' :



3. This option is only present if your Eclipse installation includes the Maven plug-in, this is pre-installed for you on the workshop VM. You should see a panel entitled 'New Maven Project', click 'Next'.
4. This Eclipse installation is configured to be aware of the platform archetypes. The next screen will, by default, give a choice of two archetypes - one for command line and one for web based applications:



5. Select the 'platform-application' archetype and click 'Next'. This will then allow you to configure the new project - the options it presents refer to the Maven properties of the generated project. Fill in the values as shown below:

6. Click 'Finish' and your new project should appear after a short delay. The project you generate is, by default, named the same as the artifact ID you specify in this panel.
7. Explore the generated project, you should find a simple Application.java which initializes the platform (but nothing else) and the same set of configuration files as used in the command line application project in exercise 1.

You can also generate these projects from the command line, but it's slightly more awkward as you then have to persuade Eclipse to believe in their existence. It's possible, but outside the scope of this tutorial - see the developer manual for more details if you need to do this.

EX2.2. CONFIGURING AND RUNNING THE PROJECT

If you've completed exercise 1 and answered the questions you should know how to configure your new project to use the cached jars and plug-in descriptions, and to modify the location it writes to. Configure your new application to write to ${user.home}/exercises/ex2 and run it - check that there is a new 'ex2' directory inside the 'exercises' directory in your Home (accessible from the desktop).

EX2.3. SUMMARY

You can now create new projects using the platform archetypes from within the Eclipse IDE. You won't actually need to do this for the rest of the workshop but it's good to know where they come from!

EX3. WORKING WITH DATA

This set of exercises covers the Reference Service, initialization and configuration, registration of data from and rendition to objects. The enactor uses T2Reference objects, the reference service constructs these from values or references such as URLs, and converts the T2References coming back as results into values.

EX3.1. USING THE REFERENCE SERVICE

This exercise involves a broken project in the Eclipse workspace. Within this project there is a single Java file which contains comments and (commented out) some incomplete code. The comments describe the code that should be there, you will have to uncomment and fix the missing pieces.

In all these exercises there will also be a 'solution' project, containing the fully working end point you're trying to reach with the exercise.

1. Close any open projects in Eclipse
2. Open the project 'ex3'
3. Within this project navigate to and open the Application.java source file
4. Read through the source file to find the exercises
5. The application will run without any changes, but it won't do anything. Fix a section at a time and verify that the application runs at each point (run from within Eclipse with a right click on the Application.java, then 'run as' - 'java application').

EX3.2. QUESTIONS

1. The exercise prints the T2Reference identifiers to the console.
    a. What is the format of the identifier and what information does it contain in the case of a reference to a piece of data?
    b. ...a list?
    c. ...an error?
2. What's the interesting feature in the 'list to t2reference to list of string' example? What does this imply for workflows using this system?
3. What's the difference in the T2Reference for the list with an exception compared to the list without one?
4. What happens when you try to de-reference the list with an error in?
    a. Can you find a way to check for this before de-referencing?

EX4. RUNNING A PRE-DEFINED WORKFLOW

This exercise shows the parsing and basic invocation of an existing workflow created in the T2 workbench. You will use the coarse grained workflow level monitoring to detect completion of the workflow and print the results to the console.

The first part of this exercise is to load a pre-defined workflow from an XML serialization. The platform can only currently load Taverna 2 workflows, if you need to load a workflow from Taverna 1.x you'll have to load and export it from the T2 workbench.

There are three parts to this exercise:

1. Loading and examining a pre-defined workflow.
2. Invoking the workflow and waiting for results.
3. Invoking the workflow as in (2) but with basic monitoring through the WorkflowInstanceListener interface.

The full monitoring infrastructure is not part of this exercise!

EX4.1. SETUP

- Close any projects you have open in the Eclipse IDE
- Open the project 'ex4'
- Navigate to the src/main/java node and the package exercises.ex4, you should see three java files corresponding to the three parts of this exercise.

EX4.2. LOADING A WORKFLOW FROM XML

The first stage is to load a workflow definition from an XML file (in this case accessed as a resource on the classpath) into a Dataflow object.

- Open the 'Application1.java' file
- Fix the broken parts of the file, implementing the basic load and inspection functionality following the inline comments.
- When all the errors are dealt with you can run the application by right clicking on the Application1.java and selecting 'run as' then 'Java application'.
- Make a note of the output produced on the console, you'll need the port names and depths for the next part of the exercise.

EX4.3. RUNNING THE WORKFLOW

The second stage is to run the workflow you loaded in the previous step.

- Open the 'Application2.java' file
- You should see the code you fixed in Application1 already provided for you, and some new sections relating to getting data in and out to fix.
    - Refer to the exercises on the reference service if you have any problems with these sections.
- As before, when all errors are resolved run by right clicking on the 'Example2.java'
- You should see the results printed to the console

EX4.4. RUNNING WITH BASIC MONITORING

In the previous example you used the blocking method provided by the Enactor interface to wait for a result. This works perfectly well, but gives no feedback on the progress of the workflow. The first level of monitoring functionality is provided by the WorkflowInstanceListener, an interface you can implement and register to inform your application of changes in workflow state, completion, and failure (should this occur). In the case of more complex workflows which can stream their results it also gives you access to the 'raw' stream of events from each workflow output port.

- Open the 'Application3.java' file
- As before, the code from Application2 will be done for you, you just need to implement the listener and attach it to the workflow instance
    - Hint - if you hover the mouse over the 'new WorkflowInstanceListener' in the source file Eclipse will give you the option to automatically implement the missing methods. This is a good way to get started.
    - Implement the methods in the listener so each one prints an appropriate message to the console.
- Run as with the previous examples, note the extra messages printed to the console from your listener implementation.

EX4.4. QUESTIONS

- In ex4.2. what happens if you don't run the workflow validation step?
    - why does this happen? (see the user manual)
- What does the blocking method to get results do if the workflow fails?
- What are the possible states used in the 'workflow changed state' message in the workflow listener?
- Not covered in this example but...
    - ...how do you cancel a running workflow?
        - ...and what does this do in the monitor? (try adding the code to do this to your example just after the code to send data in)
    - ...how can you pause / resume a running workflow?
    - What happens if you register a workflow instance listener after the workflow has completed and produced its results? *(try this by adding the listener after the blocking result fetch method)*

EX5. CREATING A WEB APPLICATION

Creating a web application is very similar to creating a command line one in terms of the actual code. Where it differs is where this code is placed, and in how the initial application context is created. This exercise shows the same workflow as in Ex4 running from a JSP, in fact this is the same application as you ran in Ex1b

EX5.1. SETUP

- Close any open projects in the Eclipse IDE
- Open the project 'ex5'.

EX5.2. CONFIGURING THE WEB APPLICATION

Web applications are configured by a 'web.xml' file. This file is located (in the deployed servlet) under 'WEB-INF/web.xml'. This location is defined by the servlet specification. In a maven based project targeting a web application you can find this configuration file under the 'src/main/webapp' directory - this is the location the build system uses for files which will end up inside the generated web application.

- Navigate to the web.xml file and open it in the IDE
- All you need to do here is uncomment the various XML blocks configuring the Spring part of the platform. Note in particular:
    - How the web application locates the spring context file (called 'context.xml' in this case)
    - How the application context implementation class is defined

EX5.3. IMPLEMENTING THE WEB APPLICATION

The web application in this case is implemented as a simple JSP. This uses the same blocking method to fetch workflow results as the previous exercise, it's a fairly naive approach that works well enough for this simple case.

- The application root is the 'index.jsp' file. This file will end up in the root of the web application so is located in 'src/main/webapp'. Open this file in the IDE
- You will have to locate the missing sections in the code and implement them, while the format is slightly different the actual code required is extremely similar to that in the previous set of exercises!

EX5.4. RUNNING THE WEB APPLICATION

To run your application in the Jetty embedded server you should refer to the instructions in Ex1b.

EX5.5. QUESTIONS

- The application in this example is very simple, it doesn't handle errors, and it simply waits for the results.
    - What are the problems with this approach?
    - How might you fix them?
        - Hint - it is possible to put objects into a shared pool which is then accessible across different web requests, this is part of the servlet specification. You could, for example, put the workflow instance in such a container...

- This application has a workflow hard-coded into it.
    - How would you make it more generic?
    - Why might you not want to do this?
        - Making a completely generic enactor is somewhat dangerous, what could you add to your application to make it less risky to deploy as a public service?

EX6. CREATING A NEW ACTIVITY

In this exercise you will create a new Activity implementation. You'll then go through the steps required to package that new activity as a plug-in, use the plug-in description tool to describe it and publish it to a local plug-in repository in the form of a directory in your home on the local machine. This plug-in will then be used in exercise 7 where you'll construct a new workflow through the API using your new Activity.

> Unfortunately we're not able to show your new plug-in within the graphical workbench at the moment, neither can we implement those extension points relating to the UI integration such as search, custom editing etc. The current workbench lags behind the platform code and the APIs aren't compatible. The plug-in packaging process, however, will be the same in the future so to do a 'full' activity plug-in with UI support will just be a question of implementing additional UI related interfaces, all other steps in this exercise will still apply.

EX6.1. SETUP

If you were creating a new plug-in from scratch you'd use the regular maven quickstart archetype and customize the pom.xml file yourself. In this particular case this is already done for you.

- Close any open projects in the Eclipse IDE
- Open the project 'ex6'
- Look first at the pom.xml file - no editing is required here but the file is heavily commented and you would have to write this yourself if you were starting this project from scratch.
- The code you need to edit is in the src/main/java node as usual, open the MyActivity.java file

EX6.2. IMPLEMENTING THE ACTIVITY

The activity you'll implement is very simple, it doesn't actually do anything you couldn't do with a beanshell or similar. The point is not to create something useful, but to show the process required. With that in mind, you need to implement the three methods in the activity class. Refer to the manual for more details on how to do this!

EX6.3. BUILDING THE ACTIVITY

Once you've implemented the methods correctly you must build the activity using the 'mvn install' command. You can either run this from the command line (navigate to the project directory in the workspace and run 'mvn install') or, more easily, from within Eclipse. Right click on the pom.xml file and open the 'run as' sub-menu. Select 'Maven Install' from the menu, you should see a substantial amount of output on the console.

The install goal will compile your code, package it into a jar file and place it along with the pom.xml that describes it in your local maven repository. This is located in /home/user/.m2/repository.

> You must install the code before you can go onto the next stage!

EX6.4. DESCRIBING THE ACTIVITY

All plug-in packages are defined by a plug-in description file. This is an XML file containing information such as the name, identifier and description of the plug-in as well as a list of maven artifacts or arbitrary jars (identified by URL) containing the code required by the plug-in. To launch the description tool use the 'Plug-in Description Editor' launcher on the desktop, this will bring up a window with various tabs.

EX6.4.1. BASIC PROPERTIES

Under the 'Description' tab you need to define various mandatory properties. The most important of these is the plug-in identifier - this is used to specify the plug-in within the platform. It must be of the form 'plugin:<group>:<id>:version'. In this case you should use the following identifier:

```
plugin:exercises:ex6:0.1
```

The other fields on this page are up to you, you can add free text tags (which will be used to search for the plug-in in myExperiment and similar systems in the future) along with a URL to your project and free text name and description properties.

EX6.4.2. AUTHORS

You should add yourself as an author, you can do this from the 'Authors' tab.

EX6.4.3. ARTIFACTS

This is the most important section. Here you will specify the artifacts that your plug-in needs to function. In step 6.3. you built and installed your activity code. If you go back to the pom.xml file you can find the three part identifier for this artifact. The platform uses colon-separated strings to specify these, so your activity artifact will be:

```
exercises:ex6:0.1
```

In the 'Maven Artifacts' section press the 'New Item' button, enter this identifier into the text box that appears. The text will initially be red to indicate an invalid identifier, turning black when you finish entering the id. Once that's done press the 'Update Graph' button and you will see the dependency structure of your plug-in. In this particular case your plug-in is very simple, it really only has a single artifact. You will see, however, that this artifact has a few dependencies which it's inherited through the workflow model dependency. These dependencies are shown in white ovals in the diagram - this indicates that they will be provided by the platform and are not part of your plug-in.

EX6.4.4. SAVING THE DESCRIPTION

You are going to create a 'plug-in repository' in your home directory. For a plug-in with identifier plugin:GROUP:ID:VERSION the plug-in manager expects to find the XML file describing your plug-in within a directory 'GROUP' and named 'ID-VERSION.xml'. Create a new directory structure as follows:

```
home/
  plugins/
    exercises/
```

Use the 'File' menu in the plug-in description tool to save the description into a file 'ex6-0.1.xml' inside the 'exercises' directory you just created.

Close the plug-in description tool.

EX7. CREATING A DATAFLOW FROM THE API - USING YOUR NEW PLUG-IN

In this exercise you'll construct a simple dataflow from scratch using the API. This workflow will consist of a single processor using the new Activity plug-in you created in the previous exercise.

EX7.1. SETUP

- Close any open projects in the Eclipse IDE
- Open the project 'ex7'

EX7.2. CONFIGURATION

This project is going to use the plug-in you deployed in ex6. To do this you must configure the platform so it's aware of both the plug-in repository and your local maven repository (in a 'real' plug-in the plug-in description would actually contain the location of the code repository but in this case we're using the ~/.m2 default).

These configurations are done for you in this case, follow the steps below to see exactly how this is done.

EX7.2.1. DEFAULT PLUG-IN REPOSITORY CONFIGURATION

The platform configuration files are held in the 'src/main/resources' node. Open the 'platform.plugin.repositories.text' file. You will see your new repository is already included in the list using a file URL format. You should also see the inclusion, common to all examples up until this point, of the /opt/plugins/ location. This is a cache of the plug-in repository on www.cyclonic.org.

EX7.2.2. DEFAULT ARTIFACT REPOSITORY CONFIGURATION

In a similar fashion the platform.raven.repositories.text file contains the list of maven repositories. In this particular case the local /home/user/.m2/repository location has been added, again as a file URL.

EX7.2.3. DEFAULT PLUG-IN LIST

The platform can be configured with a list of plug-ins to load on startup. While plug-ins can be loaded on the fly through the PluginManager API we will use the default mechanism in this exercise to reduce the amount of code required. Open the platform.plugin.defaultplugins.text file, you will see the ID of the plug-in you deployed at the top of this along with the standard set of Taverna core extensions.

EX7.3. CONSTRUCTING AND RUNNING A NEW DATAFLOW

As with the previous exercises you will find an application to fix in the src/main/java node. The comments in the code are not enough - you will need to refer to section 8 of the user manual.

Once your code is fixed you should be able to run the application in the usual way and see it work - you have now deployed a new plug-in with a completely new Activity type and invoked it through the workflow engine. Of course, this is an extremely simple plug-in, it doesn't use the configuration mechanism, doesn't pipeline its outputs, doesn't interact with the monitor and doesn't use the shared resource mechanism needed for security and resource management but it's a start. (it also doesn't do anything useful, but that's not a problem here!)

EX8. USING THE MONITOR & MONITORRECEIVER

In exercise 4 you used the WorkflowInstanceListener to monitor high level workflow state. T2 provides a much more detailed monitor infrastructure through the MonitorReceiver interface. By associating a MonitorReceiver with a workflow instance you can be informed of fine grained events such as the creation of individual activity instances. In addition any node in the workflow can expose properties, which may be writable, through the monitoring system. The monitoring system can therefore also be a steering system - none of the current extensions expose mutable properties though.

> Note - while writing this exercise I found a couple of bugs with the monitor implementation. These are fixed in your virtual machine image but if you downloaded the p0.1a2 version of the workflow model implementation prior to the 16th Feb 2009 on your own machine you will need to clean your maven repository and re-fetch. This does not affect the workshop machines!

EX8.1. SETUP

- Close any open projects in the Eclipse IDE
- Open the project 'ex8'

EX8.2. THE DEFAULT MONITOR

The MonitorReceiver interface only deals with passing node creation and destruction events from the workflow enactor to your code, it does not handle the construction of the workflow state model. Because the latter is a common requirement, the platform provides a default implementation of the MonitorReceiver, Monitor, which also implements the TreeModel interface. This means that when you have an instance of Monitor you can visualize it by using it as the model for a JTree. This is exactly what you'll do in this exercise.

EX8.3. THE INVOCATION CONTEXT

In previous exercises you used a default invocation context when constructing your workflow instance. T2 uses an enrichment model for enactment - by default there are no facilities like monitoring, provenance capture, security handling and the like. To use these capabilities you must add the appropriate object to the invocation context. In the case of monitoring the workflow enactor will use any (and all) instances of MonitorReceiver it finds in the invocation context for a given workflow.

EX8.4. FIXING THE CODE

As with previous exercises you need to fix the 'Application1.java' code in the src/main/java node. Follow the comments in the code, and refer to the user manual as appropriate. In essence your code will be doing the following:

- Get an invocation context factory from the profile
- Use this to create an invocation context
- Get a monitor factory from the profile
- Use this to create a monitor (the default implementation of MonitorReceiver which presents its state as a TreeModel)
- Add the monitor to the context
- Create a new instance of your workflow using the Enactor as before, but this time explicitly specifying your invocation context.

- Running the workflow as normal (while displaying the monitor UI)

EX8.5. QUESTIONS

- What determines the hierarchical structure shown in the monitor?
    - What does it mean for a process to be a child of another process in this context?
- The monitor receiver interface can be used for many things, how might you...
    - ...profile the per-invocation performance of a service?
    - ...steer a long running computation? (*hard question, and one we'll cover in a BoF if there's enough interest*)
- Which workflow entities, in particular extension points, can inject properties into the monitor? (*hint - there are only two*)