

## Fornax-Platform : 5.2 Rich Client CRUD GUI Tutorial (CSC)

---

This page last changed on Jan 17, 2011 by [patrik\\_nordwall](#).

# Sculptor Rich Client CRUD GUI

This is a tutorial for the Sculptor CRUD GUI using Eclipse Rich Client Platform (RCP). It describes how to generate, customize and use the CRUD GUI, and it will use the [Helloworld example application](#) as base.

Table of content:

- [Sculptor Rich Client CRUD GUI](#)
- - [Business Tier](#)
  - [Rich Client Project](#)
    - [Plugin Installation](#)
    - [Create Project](#)
    - [Generate](#)
    - [Run](#)
  - [Evolution](#)
  - [JUnit Tests](#)
  - [Explanations](#)
    - [Separated Presentation](#)
    - [Model](#)
    - [View](#)
    - [Controller](#)
    - [Lazy loading of associations](#)
    - [Internationalization](#)
    - [Error Handling](#)
    - [Multi User Features](#)
  - [Customization](#)
    - [GUI Model](#)
    - [Not Persistent Value Objects](#)
    - [Data Transfer Objects](#)
    - [Hand Written Subclasses](#)
      - [Change layout](#)
      - [Replace widget](#)
      - [Change labels in the tree](#)
      - [Login](#)
      - [Preferences](#)
    - [Replace Runtime Framework](#)
    - [Adopt Generation with Aspect-Oriented Features](#)
      - [Skip re-generation of MANIFEST.MF](#)
      - [Adopt generation of plugin.xml](#)
      - [Replace communication](#)
  - [Source](#)

## Business Tier

The first step is to set up the project structure for the business tier, import to eclipse, apply an initial model, generate code, etc, all according to [Archetype tutorial](#). You can run with Jetty, Tomcat, JBoss or some other servlet container. The web CRUD GUI project is not necessary, but you need to start the Spring container with something, such as an web application, so you can start with the generated web application.

Add `rcp` to the project nature in `sculptor-generator.properties`. This results in that the scaffold methods includes a method that is needed for lazy loading of associations.

```
project.nature=business-tier, rcp
```

The problem with the Hello world model (see [Hello World Tutorial](#)) is that the model isn't going to be very interesting to run the CRUD GUI with. We need at least one more entity and we need some services to do operations on them (for more info about what's generated, see the Explanations part).

Use a model like this:

```
Application Universe {
  basePackage=org.helloworld

  Module milkyway {
    Entity Planet {
      scaffold
      String name key;
      String message;
      Integer diameter nullable;
      Integer population nullable;
      - Set<@Moon> moons opposite planet;
    }
    Entity Moon {
      not aggregateRoot // belongs to Planet Aggregate
      String name key;
      Integer diameter nullable;
      - @Planet planet opposite moons;
    }
  }
}
```

Complete the JUnit tests for the business tier project, see part 3 of the [Hello World Tutorial](#).



You can run `Fornax mvn\--install skip test`, i.e. `mvn` with `-Dmaven.test.skip=true` if you don't want to bother with JUnit tests now.



If you use the **scaffold** key word on an entity, you automatically get CRUD services for that entity

Select the `helloworld-parent` project and run the launcher `Fornax mvn-install`.

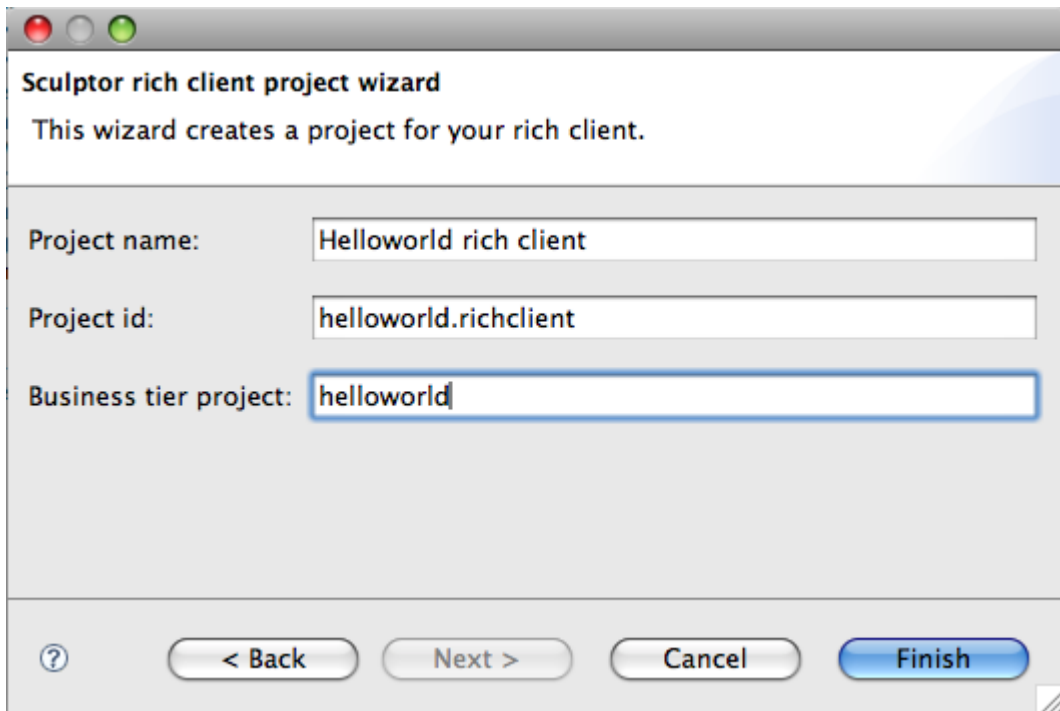
## Rich Client Project

### Plugin Installation

Install the Sculptor Rich Client Feature, as described in the [Installation Guide](#), if you haven't already done so.

### Create Project

Start the Rich Client project wizard. `File > New > Project... > Sculptor > Sculptor Rich Client Project`.




Fill in the form and finish the wizard.

- The project name is the Bundle-Name of the plugin.
- The project id is the Bundle-SymbolicName of the plugin.
- Use the name of the business tier project created in previous step.

Add `<module>../helloworld.richclient</module>` in the modules section of pom.xml in the helloworld-parent.

Run the launcher `Fornax mvn-install` from helloworld-parent. This will also copy the business tier client jar file to the lib folder of the helloworld.richclient plugin.

Refresh project.

 To do a full refresh you might have to open the MANIFEST.MF and remove/add the jar file in the Runtime tab of the manifest editor.

## Generate

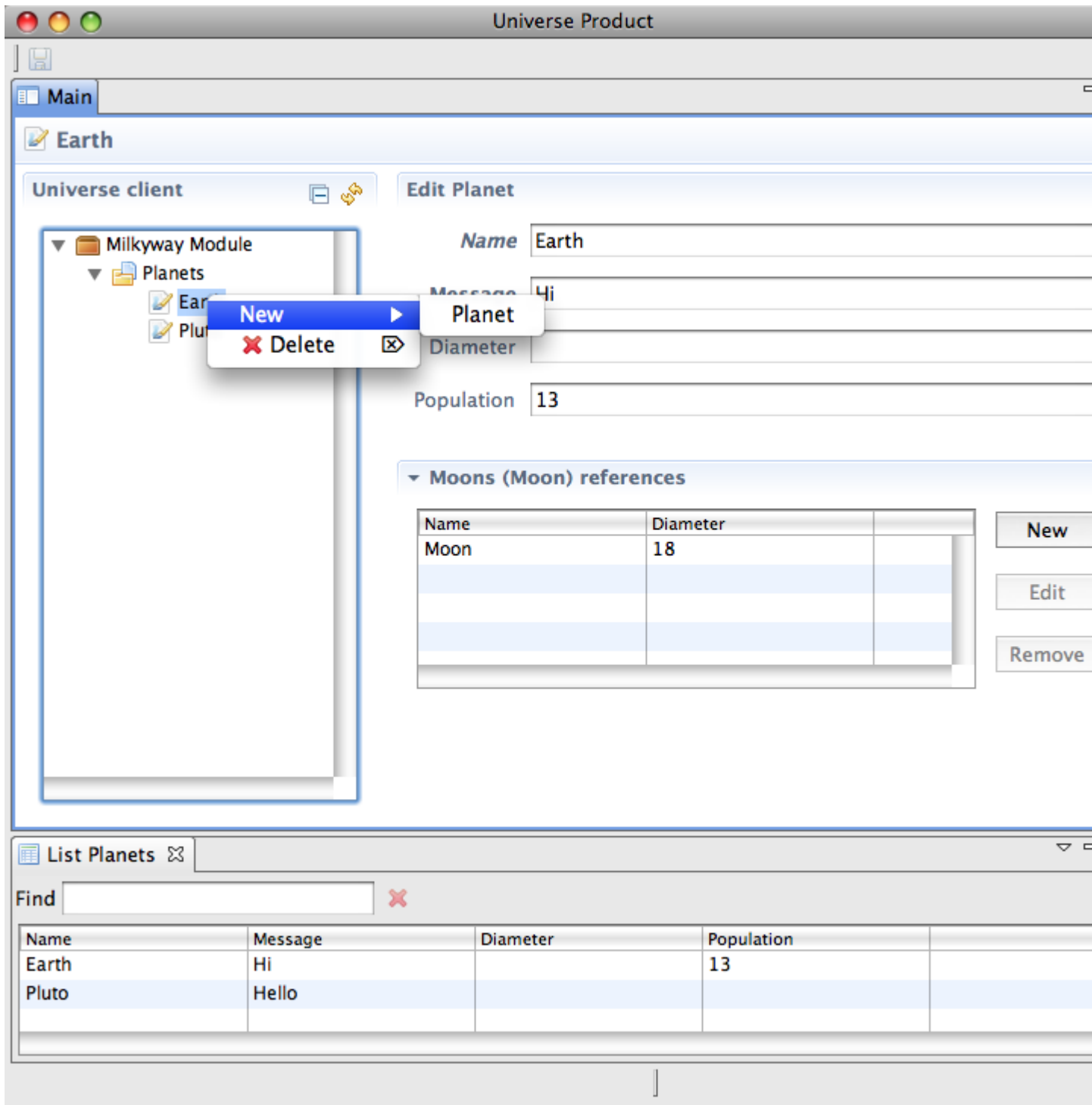
Select something in the client project and run the launcher `Fornax mvn-generate-sources`. This will generate java source and configuration files.

## Run

1. Start Jetty with `mvn jetty:run` from the helloworld-web project
2. Select `helloworld.richclient` project, right click > Run as Eclipse Application

It isn't harder 😊

Now you can try the application.



- Right click in the navigation tree to create a new planet.
- Select the planet in the navigation tree to edit the details.
- Note that when you select the `Planets` folder a new view is displayed. This is also available from the menu, `List > Planets`.

## Evolution

Let us do a redesign of the model to learn the steps of the roundtrip for doing changes.

1. Add another module in the business tier:

```

Module travel {

    ValueObject SpaceJourney {
        - @Astronaut person fetch="join"
        - @Planet fromPlanet fetch="join"
        - @Planet toPlanet fetch="join"
        Date departure
        Date arrival nullable
    }

    Repository JourneyRepository {
        findAll;
        save;
    }
}

Service SpaceJourneyService {
    findAll => @JourneyRepository.findAll;
    save => @JourneyRepository.save;
}

Entity Astronaut {
    scaffold
    String name key
}
}

```

Note that we use `scaffold` for the `Astronaut`, but define a specific `Service` and `Repository` for the `SpaceJourney`.

2. Run `Fornax mvn-generate-sources`
3. Run tests and fix failing tests
4. Run `Fornax mvn-install` from parent project
5. Restart Jetty (might not be necessary)
6. Start the client application again.

## JUnit Tests

There are a bunch of generated JUnit tests for the controllers and repositories. When you run them for the first time most of them will fail. That is because you need to create the objects to use in the tests. From the generated comments it is pretty easy to understand what you need to fill in.

```

public class RichPlanetRepositoryTest extends RichPlanetRepositoryTestBase {
    @Override
    protected RichPlanet createNewObject() {
        RichPlanet input = new RichPlanet();
        input.setName("Earth");
        input.setMessage("Hi");
        return input;
    }

    @Override
    protected RichPlanet createExistingObject() {
        Planet domainObj = new Planet("Earth") {
            {
                setId(1L);
            }
        };
    }
}

```

```

    }
};
RichPlanet input = new RichPlanet();

input.fromDomainObject(domainObj);
input.setMessage("Hello");
input.setDiameter(17);
return input;
}
}

```

```

public class PlanetDetailsControllerTest extends
    PlanetDetailsControllerTestBase {
    @Override
    protected RichPlanet createInput() {
        RichPlanet input = new RichPlanet() {};
        input.setName("Earth");
        input.setMessage("Hi");
        return input;
    }

    @Override
    protected void populateFormSuccess(
        Map<String, IObservableValue> targetObservables) {

        targetObservables.get("message").setValue("Hello");
        targetObservables.get("diameter").setValue(17);
    }

    @Override
    protected void verifyTargetToModelBinding(RichPlanet model,
        Map<String, IObservableValue> targetObservables) {
        // modify target (gui widget) and verify that model is changed correctly
        targetObservables.get("message").setValue("Hey");
        assertEquals("Expected message to change", targetObservables.get(
            "message").getValue(), model.getMessage());
    }

    @Override
    protected void verifyModelToTargetBinding(RichPlanet model,
        Map<String, IObservableValue> targetObservables) {
        // modify model and verify that target (gui widget) is changed correctly
        model.setMessage("Hey");
        assertEquals("Expected message to change", model.getMessage(),
            targetObservables.get("message").getValue());
    }
}

```

You may add own tests, which is especially useful when you implement some [custom](#) logic.

Those tests run without RCP infrastructure. They use the `HeadlessRealm`, which is part of Sculptor richclient framework. [JMock](#) is used for stubbing and verification of invocation. Sometimes it is necessary to let the UI thread execute `asyncExec` runnables. This is done by using:

```
HeadlessRealm.processDisplayEvents();
```

The controller tests run without Spring, while the repository tests run with Spring using a separate configuration: `/applicationContext-test.xml`. The services are stubbed classes keeping the objects in simple collections.

try it!

Complete the JUnit tests and look at the extended classes to get a feeling of what is tested. Also, you should take a look at the `PlanetServiceStub` and `applicationContext-test.xml`.

## Explanations

### Separated Presentation

The overall design of the generated application is inspired by MVC like design patterns. The major driver of this design is testability, i.e. possibility to test controller and model without the full user interface infrastructure. See Martin Fowler's description of:

- [SeparatedPresentation](#)
- [PresentationModel](#)

TODO: class diagram

### Model

RichObjects are the client side representation of the DomainObjects.

- property change support
- populates associations when needed
- assembly to/from DomainObjects
- RichObject instances are created with the corresponding Spring factory, normally done via the create method in the repository

All access to the server is encapsulated in client side repositories. By default, [spring\\_remoting](#) with RMI is used.

By default all services are exported as remote services, but you can use `hint="notRemote"` to skip remoting for specific services.

You can start the client with system property `org.helloworld.richclient.SpringConfig=/applicationContext-stub.xml` to use a fake stub of the services. The generated stub implementation is extremely simple, with a lot of limitations. You will probably need to implement something more intelligent.

### View

- responsible for creating widgets
- widgets are registered by name to be able bind them to the model
- only UI, no logic

### Controller

- decoupled from the view with the presentation interface
- binds widgets and model together using Eclipse `DataBindingContext`
- validation and conversion is also done using the `DataBindingContext`
- manages subtask transitions, e.g. from Planet details page it is possible to edit an associated Moon

try it!

Take a look at `RichPlanet`, `PlanetDetailsPage`, `PlanetDetailsController` and `RichPlanetRepository`. Make sure you understand the dependencies and main interaction.

## Lazy loading of associations

Loading of associations is tricky and you will probably run into a few `LazyInitializationExceptions`. The `DomainObjects` are detached from the Hibernate session when transferred to the client. Associations that are not populated can not be used in the client.

One option is to populate the needed associations before sending the object to the client. This can be done with `fetch="join"` as in the `SpaceJourney` sample above.

Another alternative is to let the client make a request to the server to populate associations when needed, i.e. lazy loading. When you use `scaffold` there will be a `populateAssociations` method that is used for this. You can define it without using `scaffold`, but it must currently be named `populateAssociations` in the service named `<domain object name>Service`. If there is such a service method there will be a corresponding method in the client side repository.

## Internationalization

All texts can be defined in resource bundles. Sculptor generates resource bundles with suggestions of English texts based on the names in the DSL model. The generated files are located in `src/generated/resources/i18n`. You can copy these to `src/main/resources/i18n` and a locale suffix when you localize to a specific language. The texts are separated in one for the common stuff (such as navigation in main view) and one file for each module:

- `messages_en.properties`
- `milkywayMessages_en.properties`
- `travel_en.properties`

To specify the language to use you start the application with `-nl` argument.

```
-nl sv
```

Some texts in `plugin.xml` are defined in `plugin.properties`. You can translate them also. Create file `plugin_en.properties`.

The `richclient` framework also has some message texts, which can be translated by defining the messages in `org.fornax.cartridges.sculptor.framework.richclient/messages_en.properties`

```
ColumnChooserDlg_add=Add
ColumnChooserDlg_availableColumns=Available Columns
ColumnChooserDlg_choseColumns=Choose columns...
ColumnChooserDlg_displayColumns=Display Columns
ColumnChooserDlg_down=Down
ColumnChooserDlg_remove=Remove
ColumnChooserDlg_up=Up
DateTimeStrategy_dateTimePattern=yyyy-MM-dd hh:mm:ss
```

Best practice is to separate the translated files in a separate plugin fragment.



## Error Handling

The error handling approach of the business tier is explained in [Advanced Tutorial](#). All interaction with the server is executed in jobs that handle exceptions (`ExceptionAwareJob`). The error code of `SystemException` and `ApplicationException` is translated to a error message using the message resource bundles. The error message of other exceptions are defined using the exception class name.

```
org_fornax_cartridges_sculptor_framework_errorhandling_SystemException=System error
org_fornax_cartridges_sculptor_framework_errorhandling_OptimisticLockingException=The information was
updated by another user. Please redo your changes.
org_helloworld_milkyway_exception_PlanetNotFoundException=Couldn't find planet
```

`ApplicationException` may include parameters, which can be used in the translated message

```
org_helloworld_milkyway_exception_TooSmallDiameterException=The diameter of a planet must be at least
{0} km.
```

Error messages are displayed with `JFace Policy.getStatusHandler().show`.

Logging is done with `JFace Policy.getLog().log`, which by default will log to the `.metadata/.log` file in the workspace.

## Multi User Features

try it!

Try this scenario:

Start two instances of the application, here called A and B.

1. A: Create a new planet, Earth
2. B: Expand the tree of Planets. Note that the new planet, Earth, is loaded when the tree is expanded first time.
3. A: Create another planet, Pluto.
4. B: Click refresh button and see that the second planet, Pluto, is also loaded.
5. A: Change population of Earth, Save.
6. B: Select Earth in the tree and see that the changed population is loaded.
7. A: Now you have selected Earth in both A and B. Change the population again, save.
8. B: Change the population, save. This will fail with error message that someone else has updated it.

## Customization

By default it is possible to run the client application without adding any hand written code. Rather soon you would like to change the default implementation and there are several mechanisms to do that.

### GUI Model

As explained in the [DSL for GRUD GUI](#) section, you can customize the client code generation by adding a gui model.

Lets say that we want to:

- give our client a specific name
- define the fields to display when listing planets

- suppress the diameter field from the edit page of a planet
- only use the name of the moons in the edit page of planet
- skip the list view for astronaut

Open the model.guidesign file and add something like:

```
import 'platform:/resource/helloworld/src/main/resources/model.btdesign'

gui TheFarFarAwayClient for Universe {
  Module for milkyway {
    ListTask for Planet {
      name
      diameter
      population
    }
    UpdateTask for Planet {
      name
      population
      moons
      list moons use attributes name
    }
  }

  Module for travel {
    ListTask for SpaceJourney {
    }
    skip ListTask for Astronaut
  }
}
```

## Not Persistent Value Objects

The structure of the persistent domain model might not always match the presentation, for example you might want a dialog for editing several domain objects in one single screen. Then you can create a non-persistent ValueObject with a corresponding Service. The application service handles the transformation between the presentation object and the persistent domain objects.

A sample is that you would like to focus on the Moons. List all moons, and for each moon display the name of the parent planet and make it possible to edit the diameter.

```
Module appservice {
  ValueObject MoonPresentation {
    not persistent
    not immutable
    String id;
    String moonName not changeable;
    Integer moonDiameter nullable;
    String planetName not changeable;
  }

  Service MoonPresentationService {
    inject @PlanetService
    @MoonPresentation findById(String id);
    @MoonPresentation save(@MoonPresentation moon);
    List<@MoonPresentation> findAll;
    delete(@MoonPresentation moon);
  }
}
```

Note that the ValueObject is marked as `not persistent`. You have to define an attribute named `id`. The type of the `id` is normally a `Long` for persistent domain objects. You can use any type, e.g. `String`. Use something that is convenient for the service to use when mapping between the persistent domain objects and this representation. In this sample I would use a `String` with concatenation of planet and moon `id`, since `Moon` is not an aggregate root and must be fetched via the `Planet`.

## Data Transfer Objects

In similar way to not persistent ValueObjects you can also use `DataTransferObject`.

```
Module presentation {
    Service PlanetDtoService {
        inject @PlanetService
        @PlanetDto save(@PlanetDto planet);
        @PlanetDto findById(Long id);
        List<@PlanetDto> findAll;
        delete(@PlanetDto planet);
    }

    DataTransferObject PlanetDto {
        Long id
        String name
        String message
        - List<MoonDto> moons
    }

    DataTransferObject MoonDto {
        Long id
        String name
        Integer diameter
    }
}
```

## Hand Written Subclasses

You can define that you need a hand written subclass (gap class) to override the default generated implementation.

In `model.guidesign` you can add `gap` to the tasks that need subclasses.

```
ListTask for Planet {
    gap
    name
    diameter
    population
}
```

This will result in several subclasses for hand written code.

Alternatively, you can specify properties in `sculptor-gui-generator.properties` if you need more detailed control of which gap classes to generate.

```
generate.gapClass.PlanetDetailsPage=true
generate.gapClass.PlanetDetailsController=true
generate.gapClass.NewPlanetController=true
generate.gapClass.RichPlanetRepositoryImpl=true
generate.gapClass.RichPlanet=true
generate.gapClass.NewPlanetWizard=true
```

```

generate.gapClass.NewPlanetWizardPage=true
generate.gapClass.ListPlanetView=true
generate.gapClass.ListPlanetTextFilter=true
generate.gapClass.PlanetServiceStub=true
generate.gapClass.UniverseRichClientPlugin=true
generate.gapClass.Application=true
generate.gapClass.ApplicationActionBarAdvisor=true
generate.gapClass.ApplicationWorkbenchAdvisor=true
generate.gapClass.ApplicationWorkbenchWindowAdvisor=true
generate.gapClass.AddPlanetDialog=true
generate.gapClass.AddPlanetPage=true
generate.gapClass.PlanetAdapter=true
generate.gapClass.MilkywayAdapterFactory=true
generate.gapClass.CommonAdapterFactory=true
generate.gapClass.DomainObjectFolderAdapter=true
generate.gapClass.ModuleFolderAdapter=true
generate.gapClass.RootNodeAdapter=true
generate.gapClass.ErrorNodeAdapter=true
generate.gapClass.MainView=true
generate.gapClass.NavigationMasterDetail=true
generate.gapClass.Perspective=true
generate.gapClass.PreferenceInitializer=true
generate.gapClass.GeneralPreferencePage=true

```

## Change layout

Let us change the layout of a simple field, the population field in `PlanetDetailsPage`. Add this in `sculptor-gui-generator.properties`:

```
generate.gapClass.PlanetDetailsPage=true
```

Remove previous `PlanetDetailsPage` and re-generate. Add this in the new `PlanetDetailsPage` gap subclass:

```

@Override
protected Text createPopulation() {
    Text result = super.createPopulation();
    GridData gd =
        new GridData(org.eclipse.swt.SWT.LEFT, org.eclipse.swt.SWT.CENTER, true, false);
    gd.widthHint = 100;
    result.setLayoutData(gd);
    return result;
}

```

## Replace widget

What if you would like to replace a widget, e.g. use a Spinner instead of a Text field.

```
generate.gapClass.PlanetDetailsPage=true
```

Override suitable method and create your own widget.

```

public class PlanetDetailsPage extends PlanetDetailsPageBase {
    private Spinner population;
}

```

```

@Override
protected Text createPopulation() {
    createPopulationSpinner();

    return null;
}

private void createPopulationSpinner() {
    Label label = getToolkit().createLabel(getPageContainer(),
        MilkywayMessages.domainObject_Planet_population,
        SWT.NONE);
    label.setForeground(getToolkit().getColors().getColor(IFormColors.TITLE));

    label.setLayoutData(new GridData(SWT.RIGHT, SWT.CENTER, false, false));

    population = new Spinner(getPageContainer(), SWT.NONE);

    GridData gd =
        new GridData(org.eclipse.swt.SWT.FILL, org.eclipse.swt.SWT.CENTER,
            true, false);
    population.setLayoutData(gd);

    getTargetObservables().put("population",
        SWTObservables.observeSelection(population));
}
}

```

When you run this you will see a binding exception. You need to replace the binding in the controller also.

```
generate.gapClass.PlanetDetailsController=true
```

Override the `bindPopulation` method.

```

@Override
protected void bindPopulation(DataBindingContext bindingContext) {
    String attributeName = "population";
    if (getTargetObservable(attributeName) == null) {
        return;
    }

    UpdateValueStrategy targetToModelUpdateStrategy =
        new UpdateValueStrategy();
    UpdateValueStrategy modelToTargetUpdateStrategy =
        new UpdateValueStrategy();

    bindingContext.bindValue(getTargetObservable(attributeName),
        getModelObservable(attributeName), targetToModelUpdateStrategy,
        modelToTargetUpdateStrategy);
}

```

## Change labels in the tree

Currently it is not possible to define in the GUI DSL what attributes to use for presentation in the navigation tree. You need a gap class to take control of them.

```
generate.gapClass.SpaceJourneyAdapter=true
```

Override `getLabel` method:

```
private SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-DD");

@Override
public String getLabel(Object object) {
    RichSpaceJourney richSpaceJourney = ((RichSpaceJourney) object);
    StringBuilder result = new StringBuilder();

    result.append(richSpaceJourney.getPerson().getName()).append(", ");
    result.append(dateFormat.format(richSpaceJourney.getDeparture()));

    return result.toString();
}
```

## Login

A default login dialog is provided, but you need to implement the actual authentication yourself. Add a gap class for the Application class.

```
generate.gapClass.Application=true
```

Override the `authenticate` method and perform the validation of the user credentials.

If you don't need the login at all, you can override the `login` method and return true.

## Preferences

A general [preference](#) page is provided and you can easily add more settings.

Add a gap class for `GeneralPreferencePage` and override `createFieldEditors` to add more fields.

## Replace Runtime Framework

It is possible to replace all classes in the Sculptor rich client runtime framework by defining the classes to use in `sculptor-gui-generator.properties`:

```
framework.richclient.controller.AbstractDetailsController=org.myown.framework.richclient.MyAbstractDetailsController
```

## Adopt Generation with Aspect-Oriented Features

It is also possible to redefine the code generation templates by using the Aspect-Oriented Programming features in oAW, as described in [Developer's Guide](#).

The rich client project, created by the wizard, is prepared for this. There is a file `src/main/resources/RichClientSpecialCases.xpt`, in which you can add your AROUND definitions.



When you edit this file you will be asked if you want to add the oAW nature to the project. Don't add the oAW nature unless you have the full Sculptor source code in the workspace. Otherwise the editor will present errors because you don't have all dependencies available.

### Skip re-generation of MANIFEST.MF

You might need to edit MANIFEST.MF manually. Add this to RichClientSpecialCases.xpt:

```
«REM»Skip re-generation of MANIFEST.MF«ENDREM»
«AROUND templates::rcp::RcpCrudGuiManifest::manifest FOR sculptorguimetamodel::GuiApplication »
«ENDAROUND»
```

### Adopt generation of plugin.xml

To add more content to plugin.xml you overwrite morePluginContent. For example, to contribute to the popup menu of the navigation tree:

```
«AROUND templates::rcp::RcpCrudGuiPlugin::morePluginContent FOR sculptorguimetamodel::GuiApplication »
<extension point="org.eclipse.ui.menus">
  <menuContribution locationURI="popup:org.helloworld.richclient.NavigationTreeMenu?after=additions">
    <command commandId="org.helloworld.common.richclient.command.doSomethingSpecial"></command>
  </menuContribution>
</extension>
<extension point="org.eclipse.ui.commands">
  <command id="org.helloworld.common.richclient.command.doSomethingSpecial"
    name="Special Action"></command>
</extension>
<extension point="org.eclipse.ui.handlers">
  <handler class="org.helloworld.common.richclient.handler.DoSomethingSpecialHandler"
    commandId="org.helloworld.common.richclient.command.doSomethingSpecial"></handler>
</extension>
«ENDAROUND»
```

You need to create the handler class also:

```
public class DoSomethingSpecialHandler extends AbstractHandler {
    public Object execute(ExecutionEvent event) throws ExecutionException {
        IWorkbenchWindow window =
            HandlerUtil.getActiveWorkbenchWindowChecked(event);
        MessageDialog.openInformation(window.getShell(), "Special", "Executed special action");
        return null;
    }
}
```

### Replace communication

[Spring remoting](#) with RMI is used by default. The type of Spring remoting can be selected with properties in sculptor-generator.properties:

```
The type of remoting can be selected with
spring.remoting.type=rmi
#spring.remoting.type=hessian
```

```
#spring.remoting.type=burlap  
#spring.remoting.type=httpInvoker
```

Hessian, Burlap and HttpInvoker requires some additional configuration in `web.xml`, as described in [Spring remoting documentation](#)

## Source

The complete source code for this tutorial is available in Subversion.  
Web Access (read only):

- <http://fisheye3.cenqua.com/browse/fornax/trunk/cartridges/sculptor/sculptor-helloworld-parent>
- <http://fisheye3.cenqua.com/browse/fornax/trunk/cartridges/sculptor/sculptor-helloworld>
- <http://fisheye3.cenqua.com/browse/fornax/trunk/cartridges/sculptor/sculptor-helloworld-web>
- <http://fisheye3.cenqua.com/browse/fornax/trunk/cartridges/sculptor/sculptor.helloworld.richclient>

Anonymous Access (read only):

- <https://fornax.svn.sourceforge.net/svnroot/fornax/trunk/cartridges/sculptor/sculptor-helloworld-parent>
- <https://fornax.svn.sourceforge.net/svnroot/fornax/trunk/cartridges/sculptor/sculptor-helloworld>
- <https://fornax.svn.sourceforge.net/svnroot/fornax/trunk/cartridges/sculptor/sculptor-helloworld-web>
- <https://fornax.svn.sourceforge.net/svnroot/fornax/trunk/cartridges/sculptor/sculptor.helloworld.richclient>