This page last changed on Jan 17, 2011 by patrik_nordwall.

Web CRUD Gui Tutorial

This tutorial describes how to generate, customize and use the CRUD gui, and it will use the Helloworld example application as base.

I.e. we start out with the Hello World model:

```
Application Universe {
    basePackage=org.helloworld

    Module milkyway {
        Service PlanetService {
            String sayHello(String planetName);
            protected findByExample => PlanetRepository.findByExample;
        }

        Entity Planet {
            String name key;
            String message;

            Repository PlanetRepository {
                findByExample;

            }
        }

    }
}
```

For guidance on environment setup, see the Archetype tutorial.   You may stop at the section named 'Run in Jetty', i.e. JBoss and MySQL are not mandatory for this tutorial.

Table of content:

- Web CRUD Gui Tutorial
    - Tutorial
        - JUnit tests
        - Redesign of model
        - GUI model
        - Not Persistent Value Objects
        - Data Transfer Objects
        - Hand Written Subclasses
        - Custom logic
            - Example 1: Custom heading
            - Example 2: Custom validation
            - Example 3: Custom logging on creation
            - Example 4: Customization of flow logic
        - Custom generation
    - Explanations
        - What's generated?
            - *Web resources*
            - Java
            - Configuration
        - Internationalization
        - Required fields
        - Error handling

        - Jpa Session
    - Resources
    -
        - Source
        - More Advanced Example
        - Links

So, before you start, set up the project structure, import to eclipse, apply an initial model, generate code, etc, all according to Archetype tutorial.

Tutorial

JUnit tests

For the business tier project see part 3 of the Hello World Tutorial.

The tests for the client are ready to run. By default there are one test that checks that there are a start state for a flow.

Redesign of model

Ok, so, according to the Hello World Tutorial, the model looks like above.

The problem with the Hello world model (see above) 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).

1. So, in step one we do a redesign of our model to look something like this:

```
Application Universe {
    basePackage=org.helloworld

    Module milkyway {
        Service PlanetService {
            String sayHello(String planetName);
            protected findByExample => PlanetRepository.findByExample;
        }
        Entity Planet {
            scaffold
            String name key;
            String message;
            Integer diameter nullable;
            Integer population nullable;
            - Set<@Moon> moons opposite planet;

            Repository PlanetRepository {
                findByExample;
                findByKeys;
                save;
            }
        }
        Entity Moon {
            not aggregateRoot // belongs to Planet Aggregate
            String name key;
            Integer diameter nullable;
            - @Planet planet opposite moons;
        }
    }
}
```

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

2. Re-generate the code in both the business tier project and the web project

✓ You can save a lot of time by just generate source instead of doing a full build, i.e. `maven install`, run the external maven task `mvn-generate-sources`

3. Start Jetty with `mvn jetty:run` from the helloworld-web project.

4. Try and run the CRUD GUI on http://localhost:8080/. Click on the 'Create planet' menu choice and create a planet, and possible, create an optional Moon in conjunction to the Planet. When the planet is created, you can browse the complete list of Planets by clicking on the 'List all planets' menu choice.

It isn't harder 🙂

GUI model

Ok, as explained in the 5.3 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
- customize how we list our planets

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
            population
            diameter
        }
    }
}
```

The first one is obvious, we gave the gui the name 'TheFarFarAwayClient'. The list customization is done by specifying what attributes and in what order. The disabling of the message attribute is simply accompliced by NOT specifying it amongst the attribute in the ListTask.

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 hand written subclass, xhtml and flow definition (gap class) to override the default generated implementation.
In `model.guidesign` you add can add `gap` to the tasks that need subclasses.

```
ListTask for Planet {
    gap
    name
```

```
            diameter
            population
        }
```

For the ListTask for Planet this will result in:

- action subclass for hand written java code (ListPlanetAction extends generated ListPlanetActionBase)
- form subclass for hand written java code (ListPlanetForm extends generated ListPlanetFormBase)
- facelet xhtml for hand written page (list.xhtml includes generated list_include.html)
- flow xml for hand written flow control (listPlanet-flow.xml with parent generated listPlanet-base.xml)

Custom logic

Ok, lets say we want to do four things:

1. We want to change the heading of the input page.
2. When a new Planet is created we want to validate that the name isn't 'Pluto'.
3. When we successfully created a planet, we want to log the id.
4. Customization of flow logic.

Example 1: Custom heading

To be able to customize the views we need gap-files, i.e. files that you can edit. According to the above description, add the 'gap' keyword to the CreateTask for Planet:

```
CreateTask for Planet {
  gap
}
```

Open the file:

src/main/webapp/WEB-INF/flows/milkyway/createPlanet/input.xhtml

and change the heading to something of your choice:

```
<h1>
   +++ <h:outputFormat value="#{msg['create.formHeader']}"> +++
      <f:param value="#{msgMedia['model.DomainObject.Planet']}" />
   </h:outputFormat>
</h1>
```

Example 2: Custom validation

In webflow there are two different ways of adding custom validation, in the model object (what we call form-object) or in a separate validator.

First, with form-object.

You need the gap-variant of the form-object, so again, make sure you have the 'gap' keyword on the CreateTask of the Planet entity.

Open the file:

src/main/java/org/fornax/cartridges/sculptor/examples/helloworld/milkyway/web/CreatePlanetForm.java

and add the method validateInput:

```
public void validateInput(ValidationContext context) {
    if (this.getName().equalsIgnoreCase("pluto")) {
        MessageContext messages = context.getMessageContext();
        messages.addMessage(new MessageBuilder().error().code("error.value.pluto").defaultText("Pluto isn't a
 planet").build());
    }
}
```

And of course, you need to add the "error.value.pluto" key and message to the resource bundles if you want the message to be localized.

You can add several validation methods. The pattern is validate"State", where "state" is the id of your view-state where you want validation to run.

Ok, how to add the same but with a standalone validator?

For this, we don't need gap-files, which in most cases is to prefer. Just add a new class:

```
@Component
public class CreatePlanetFormValidator {
    public void validateInput(CreatePlanetForm createPlanetForm, ValidationContext context) {
        if (this.getName().equalsIgnoreCase("pluto")) {
            MessageContext messages = context.getMessageContext();
            messages.addMessage(new MessageBuilder().error().code("error.value.pluto").defaultText("Pluto isn't a
 planet").build());
        }
    }
}
```

And here the pattern is that you create a validator with the name "model"Validator where model is the name of your form-bean. And as above, a method with the pattern validate"State".

For more info about webflow and validation, see here.

Example 3: Custom logging on creation

Open the file:

    src/main/java/org/fornax/cartridges/sculptor/examples/helloworld/milkyway/web/
    CreatePlanetAction.java

Override the save method to add logging:

```
@Override
public Event save(RequestContext ctx) {
    Event event = super.save(ctx);
    Planet planet =  (Planet) ctx.getFlowScope().get("Planet");
    System.out.println("Planet created with id: " + planet.getId());
    return event;
 }
```

Example 4: Customization of flow logic

For example, lets say you want to implement filter functionality for the planet list feature. Here is the steps to do that:

1) Again, you need gap-files for the list planet feature, so in your model.guidesign:

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

gui TheFarFarAwayClient for Universe {
    Module for milkyway {
        ListTask for Planet {
            gap
            name
            population
            diameter
         }
    }
}
```

2) Edit the listPlanet flow:

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
    xmlns:ns0="http://www.w3.org/2001/XMLSchema-instance"
    ns0:schemaLocation="http://www.springframework.org/schema/webflow   http://
www.springframework.org/schema/webflow/spring-webflow-2.0.xsd"
    parent="milkyway/listPlanetBase">
    <view-state id="list">
        <transition on="filterPlanets" to="listByFilter" />
    </view-state>
    <view-state id="listByFilter" model="listPlanetForm"
        view="/WEB-INF/flows/milkyway/listPlanet/list.xhtml" parent="milkyway/listPlanetBase#list">
        <on-render>
            <evaluate
                expression="listPlanetAction.findByFilter(flowRequestContext)" />
        </on-render>
    </view-state>
</flow>
```

3) Edit the ListPlanetAction, add the method:

```
public String findByFilter(RequestContext ctx) {
    getRepository().clear();
    List<Planet> allPlanets = getPlanetService().findAll(ServiceContextStore.get());
    List<Planet> filtered = new ArrayList<Planet>();
    String filter = ctx.getRequestParameters().get("planetFilter");
    for (Planet planet : allPlanets) {
        if (planet.getName().startsWith(filter)) {
           filtered.add(planet);
        }
    }
    formObject(ctx).setAllPlanets(filtered);
    return "success";
}
```

4) Add a simple form with a text field and a button to the milkyway/listPlanet/list.xhtml-file:

```
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:f="http://java.sun.com/jsf/core" xmlns:t="http://myfaces.apache.org/tomahawk"
    xmlns:h="http://java.sun.com/jsf/html" xmlns:c="http://java.sun.com/jstl/core"
    xmlns:a="ApplicationTaglib">
    <body>
        <ui:composition template="/WEB-INF/common/template.xhtml">
            <ui:define name="content">
                <h1>
                    <h:outputFormat value="#{msg['list.header']}">
                        <f:param
                            value="#{msgMilkyway['model.DomainObject.Planet.plural']}" />
                    </h:outputFormat>
                </h1>
                <h:form xmlns="http://www.w3.org/1999/xhtml"
                    xmlns:ui="http://java.sun.com/jsf/facelets" xmlns:f="http://java.sun.com/jsf/core"
                    xmlns:t="http://myfaces.apache.org/tomahawk"
                    xmlns:h="http://java.sun.com/jsf/html" xmlns:c="http://java.sun.com/jstl/core"
                    xmlns:a="ApplicationTaglib">
                <div>
                    <label for="_planetFilter">#{msgMilkyway['model.DomainObject.Planet.filter']}: </label>
                    <input type="text" value="#{requestParameters.planetFilter}" name="planetFilter"
   id="_planetFilter"/>
                    <h:commandButton value="#{msgMilkyway['model.DomainObject.Planet.filterButton']}"
   action="filterPlanets" />
                </div>
                </h:form>
                <ui:include
                    src="/WEB-INF/generated/flows/milkyway/listPlanet/list_include.html" />
            </ui:define>
        </ui:composition>
    </body>
</html>
```

That's it.

Note that the default generated table showing the list result is used by including the generated file. In case you need to modify the generated content you can either overwrite the code generation templates in WebSpecialCases.xpt or simply copy the generated file and maintain it manually

For more information of webflow and flow syntax, see here.

Custom generation

Developer's Guide describes how to customize and add your own code generation.

[TODO: add a sample]

Explanations

What's generated?

In short, a fully functional CRUD web application, based on spring webflow2, JSF and facelets.

Web resources

Flows are made up of configuration and xhtml/html files.
Normaly, a flow is put in a 'module' named after the flow, i.e:

`src/main/webapp/WEB-INF/flows/milkyway/createPlanet`

Though, in sculptor we need to split this in two:

`src/main/webapp/WEB-INF/flows/milkyway/createPlanet` `src/main/webapp/WEB-INF/generated/flows/milkyway/createPlanet`

Where the first one is generated once and then maintained manually, and the second one is always re-generated. Note that you need to add the gap keyword to `model.guidesign` to use the hand written files.

```
CreateTask for Planet {
   gap
}
```

Java

By default sculptor generates one class that holds the logic.

For each domain object there are webflow implementation classes generated:

- Action - each CRUD method gets its own action where the logic resides
- Form - each CRUD method gets its own data carrier

To make sculptor generate base class and customizable concrete (hand written) subclass, you need to add the gap keyword to the task in `model.guidesign`, as explained here here.

Configuration

As always with spring there is a lot of configuration. The positive side here is that sculptor generates it for you.

There are some application specific configuration files:

- web-inf/web.xml - the traditional servlet container configuration, here with a predefined webflow front controller
- web-inf/generated/config/faces-config.xml - JSF configuration file
- web-inf/generated/config/applicationContext.xml - the traditional spring configuration file
- web-inf/generated/config/webmvc-config.xml - the spring front controller configuration file
- web-inf/generated/config/webflow-config.xml - the webflow framework configuration file

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. You can copy these to `src/main/resources/i18n` when you localize to a specific language. Remember to suffix the name with the locale you want, i.e. messages.properties will be named messages_sv.properties for swedish custom translations. The generated files are located in `src/generated/resources/i18n`. The texts are separated in 2 files plus 1 for each module:

- defaultMessages_en.properties
- messages_en.properties
- milkywayMessages_en.properties

Date format is defined in the messages resource bundle.

```
format.YearMonthDayPattern=yyyy-MM-dd
format.DateTimePattern=yyyy-MM-dd HH:mm
```

A special feature that is useful when the domain model evolves. Sculptor will display ??? at the texts in the gui that are not defined in a resource bundle, if you add the following to `src/main/resources/sculptor-gui-generator.properties` and regenerate.

```
gui.highlightMissingMessageResources=true
```

The locale of the user is changed when the request contains a `locale` parameter. E.g. of links to switch language:

```
<a href="index.htm?locale=en"><spring:message code="language.en" text="English" /></a>
&nbsp;&nbsp;&nbsp;
<a href="index.htm?locale=sv_SE"><spring:message code="language.sv" text="Svenska" /></a>
```

Required fields

Required fields are derived from the model. For attributes the rule are that if the attribute are 'nullable', the form field isn't required. For references we have two cases, 'one' and 'many' references.

For 'one' references the same rule as for attributes apply, i.e. if the reference is 'nullable' it isn't required in the gui.

Error handling

Two types of exceptions, ApplicationException and SystemException, are used as described in Advanced Tutorial. Error messages for these two types of error situations are displayed in different ways.

ApplicationException
When an ApplicationException occurs a message is displayed in the application pages. It is displayed in the same place as validation error messages.

Let us simulate an ApplicationException. The `findById` method throws PlanetNotFoundException, which is an ApplicationException, when a requested `Planet` doesn't exist. Click on List all Planets, right click on the id link of one Planet, and select Copy Link Location. Past into the browser URL field, but modify the last id request parameter, e.g. `&id=999`. When you try to load that page an error is displayed.

The actual error message can be defined in messages resource bundle. The `errorCode`of the ApplicationException is used as key to the resource bundle. Parameters of the exception can be used as message resource parameters.

```
org.helloworld.milkyway.exception.PlanetNotFoundException=Planet with id {0} doesn't exist.
```

The error handling for ApplicationException is done by an advice that intercepts all methods in the actions. It catch exceptions and bind the error message to the form errors object.

ValidationException
ValidationExceptions throw by business tier are treated in the same way as ApplicationException, i.e. error message on the same page.

OptimisticLocking
When two users try to update the same entity simultaneously an `OptimisticLockingException` is thrown. This exception is treated in the same way as ApplicationException, i.e. error message on the same page.

SystemException
A separate error page is displayed when a `SystemException` or any other unexpected `RuntimeException`occurs. Normally it is only necessary to display a general error message, but it is possible to define individual error messages for specific exceptions. The `errorCode` in the `SystemException` or the name of the `RuntimeException` is used as key in the resource bundle.

```
org.fornax.cartridges.sculptor.framework.errorhandling.SystemException=System error ({0}), <br/>caused
 by: {1}
org.fornax.cartridges.sculptor.framework.errorhandling.DatabaseAccessException=System error. Database
 problem.
```

The fault barrier is implemented with a Spring exception resolver that directs to `error.jsp`, which resolves and displays the error message.

Jpa Session

The GRUD Gui application uses JpaFlowExecutionListener to implement persistent sessions that span over conversations. The same JPA session is used during a conversation, i.e. from a top flow and throughout its subflows, but it is disconnected in between each request. In practice this means that lazy loading of collections is possible when traversing associations.

However, all root objects are retrieved with service methods and transactional services perform all updating operations. The transaction boundary is still at the service layer, but the DomainObjects are not detached when used in the presentation tier.

Resources

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-ear

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-ear

More Advanced Example

You can try the more comprehensive Library example, which also has a generated GRUD Gui.
Checkout these four projects from Subversion.

- https://fornax.svn.sourceforge.net/svnroot/fornax/trunk/cartridges/sculptor/fornax-cartridges-sculptor-examples-library-parent
- https://fornax.svn.sourceforge.net/svnroot/fornax/trunk/cartridges/sculptor/fornax-cartridges-sculptor-examples-library
- https://fornax.svn.sourceforge.net/svnroot/fornax/trunk/cartridges/sculptor/fornax-cartridges-sculptor-examples-library-web

Links

- Spring
- Webflow