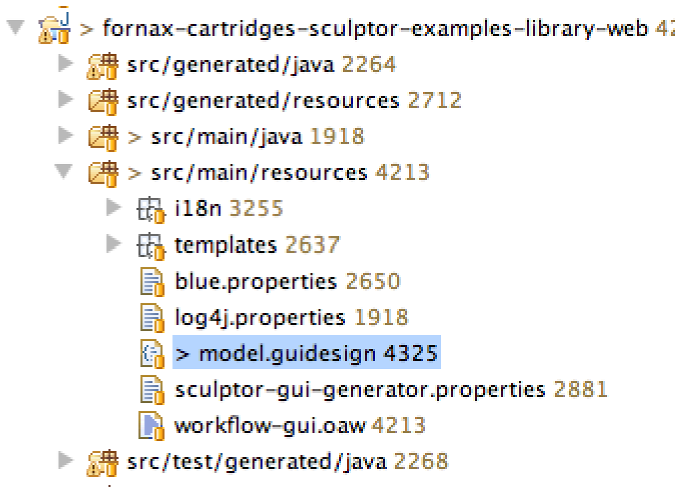
Fornax-Platform : 5.3 DSL for GRUD GUI

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

DSL for GRUD GUI

Sculptor has a Domain Specific Language (DSL) for the CRUD GUI. To use it, open your model.guidesign file.



> fornax-cartridges-sculptor-examples-library-web 4
src/generated/java 2264
src/generated/resources 2712
> src/main/java 1918
> src/main/resources 4213
i18n 3255
templates 2637
blue.properties 2650
log4j.properties 1918
> model.guidesign 4325
sculptor-gui-generator.properties 2881
workflow-gui.oaw 4213
src/test/generated/java 2268

model.guidesign is the default name of the file. You can change it to what ever you like as long as the extension is `.guidesign`. If you change it, don't forget to also change it in the pom.xml and in the workflow-gui.aow files.

When you are adding GUI DSL code, you are basing it on (imports) a business DSL. That means that if you don't add any GUI specific DSL code, the generation will be based on the business model.

In short, the GUI DSL supports:

- Specifying the name of the client application
- Specifying what attributes and references that should be shown and in what order
- Specifying what service methods to use for viewing, creating, updating, listing and deleting
- Specifying domain objects to be skipped
- Specifying tasks to be skipped

Let us use the Library application as an example and start with a minimalistic GUI DSL:

```
import 'platform:/resource/fornax-cartridges-sculptor-examples-library/src/main/resources/model.btdesign'

gui LibraryWeb for Library {

}
```

So, here you see that GUI DSL starts with an import of a business model. This is mandatory.
Next, we define a client for the business model, and here we have our first opportunity to manipulate the client code by defining the name of the client application. Here, in our case, we name it 'LibraryWeb' (since this is the Library example application and its web client).

OK, lets add some more customization. When listing persons, we want to specify what attributes should be shown and in what order:

```
import 'platform:/resource/fornax-cartridges-sculptor-examples-library/src/main/resources/model.btdesign'
```

```
gui Library for Library {
    Module for person {
        ListTask for Person {
            - name.first
            - birthDate
            - sex
            - ssn
        }
    }
}
```

First, we need to go into the right module, hence the first line.
Second, we specify what kind of task we want to customize, and in this case it is a `ListTask` and it is for the `Person` entity.
Then, we just add the attributes we want to use when listing persons. 'Simple' attributes like strings, date, integers are of course supported. But also more complex cases like enums (sex), `BasicType` and references. For basic types you can either specify it by the reference name (like ssn above), and get all the 'nested' attributes for that basic type, or, specify exactly what 'nested' attributes you are interested in (like name.first above).

Brief description of the standard CRUD User Tasks

The objective of a **CreateTask** is to create a new instance of a Domain Object and in the end store it the database. When used as a subtask it is the parent task that is responsible for storing in the database. The CreateTask UI consists of an input and a confirm screen. In the input screen you fill in a form with fields for the attributes. For refererences it is possible to select existing or create new objects. Creating a new referred object spawns another CreateTask, a subtask. When input has been entered the next step in the wizard is a confirm screen. Later, the confirm screen may be optional.

The objective of a **ListTask** is to show all or a subset of all Domain Objects and make them available for other tasks. The ListTask UI consists of a single screen showing all objects of a specific type. Later, search facility and pagination will probably be included. From the ListTask you can spawn ViewTask, DeleteTask and UpdateTask for existing objects.

UpdateTask is similar to CreateTask, but the objective is to modify an existing Domain Object.

The objective of the **ViewTask** is to present detailed information for a Domain Object. The ViewTask UI consists of a single screen, with possibility to follow references, i.e. spawn other ViewTasks as subtasks.

The objective of the **DeleteTask** is to remove a Domain Object from persistent storage. When used as subtask it is the parent task that is responsible for saving. The DeleteTask UI consists of a single screen with the purpose to confirm the delete.

Alright, lets say that when listing persons, we have a custom service method that should be used:

```
    Module for person {
        ListTask for Person {
            - name.first
            - name.last
            - birthDate
            - sex
            - ssn
            searchWith PersonService.findAllNicePersons
        }
    }
```

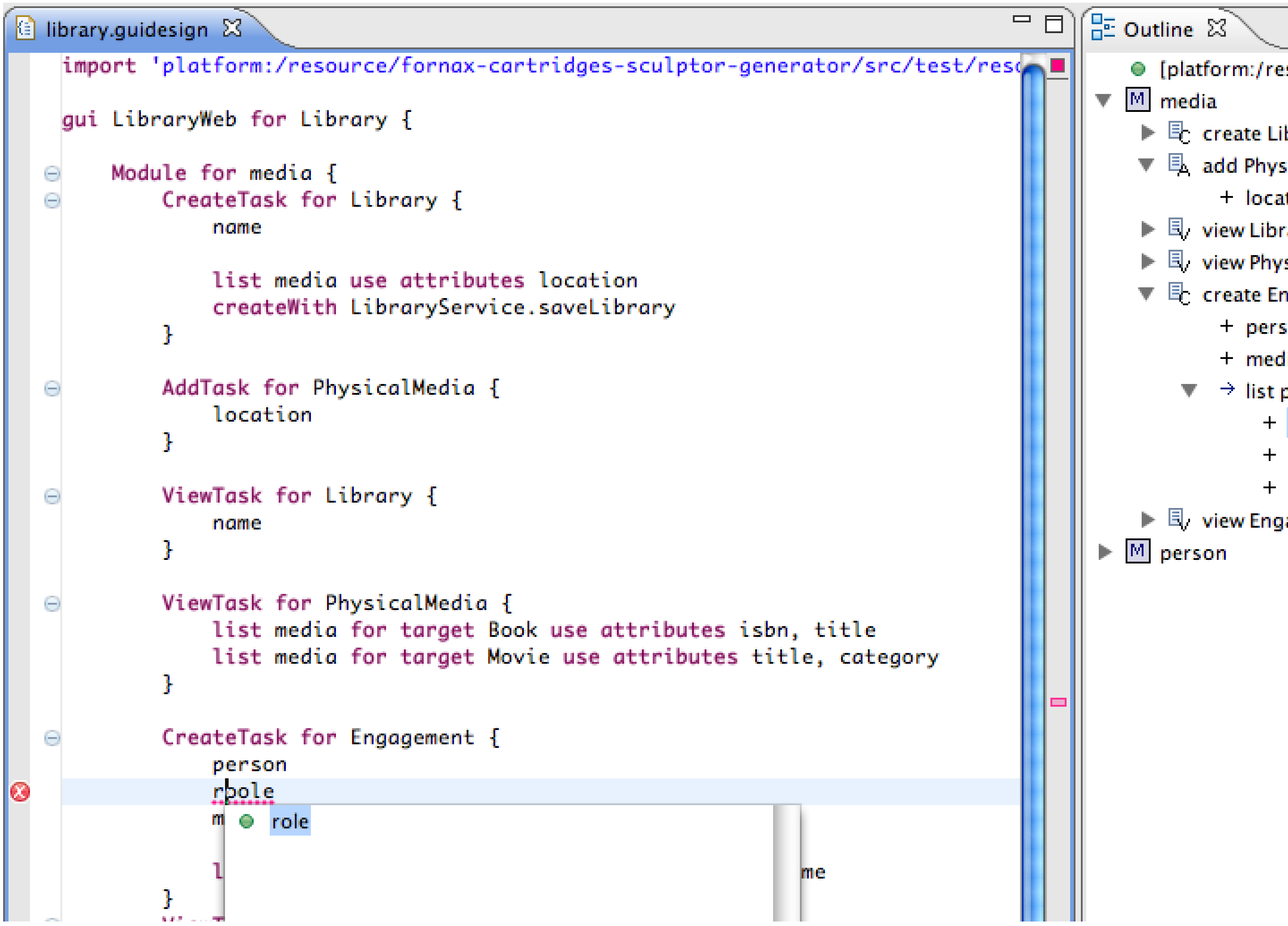
As you can see, for ListTasks the `searchWith` keyword is available, and after that you specify what service and what method to use.

When you are using the default scaffold methods you don't need to specify anything.

For other tasks it is possible to define the methods to use in a similar way as illustrated for ListTask above.

User Task	Define service operation with
CreateTask	createWith
UpdateTask	findWith, saveWith
ViewTask	findWith
DeleteTask	findWith,deleteWith
ListTask	searchWith
AddTask	findWith

The editor for the `.guidesign` files supports code completion and error highlighting, including linking to business tier model.



```
import 'platform:/resource/fornax-cartridges-sculptor-generator/src/test/reso

gui LibraryWeb for Library {

    Module for media {
        CreateTask for Library {
            name

            list media use attributes location
            createWith LibraryService.saveLibrary
        }

        AddTask for PhysicalMedia {
            location
        }

        ViewTask for Library {
            name
        }

        ViewTask for PhysicalMedia {
            list media for target Book use attributes isbn, title
            list media for target Movie use attributes title, category
        }

        CreateTask for Engagement {
            person
            rpole
            m

            l                                                        me
        }
```

So, lets move on to how to skip generation for certain parts.

Lets say that we don't want to have a library client that handles books, just movies. Easy:

```
gui Library for Library {
    Module for media {
        skip Book
    }
}
```

It's just a matter of adding a 'skip' clause for the domain object that shouldn't be part of the client.

OK, but if we just want to exclude the updating of a book, not exclude the entire domain object. Just as easy:

```
Module for media {
    skip UpdateTask for Book
}
```

The last example we will show is how references to other domain objects can be changed:

```
Module for media {
    ViewTask for Library {
        list media use attributes location, status
    }
}
```

You use the 'list' keyword, followed by the reference, followed by the 'use attributes' keywords, followed by the attributes you want to use when listing the reference.

For a little bit more complex case, when we have inheritance:

```
Module for media {
    ViewTask for PhysicalMedia {
        list media for target Book use attributes isbn, title
    }
}
```

References to skipped domain objects or tasks are naturally skipped. It is also possible to skip a reference without skipping the domain object or task. When you define any attributes or references, only those will be included, i.e. others will be skipped.

The following will include media reference (for Book and Movie), but the reference to Library will not be available, since it is not defined.

```
    UpdateTask for PhysicalMedia {
        - status
        - location
        - media
    }
```