

Projector

Event driven application framework

Daniel Florey

Projector: Event driven application framework

Daniel Florey

Copyright © 2004 Daniel Florey

Projector is an event driven workflow based web application framework.

Dedication

This book is dedicated to the Open Source community

Table of Contents

1. Introduction	4
1. Features	4
2. Justification	4
2. The big picture	7
1. Architecture	7
3. Projector basics	9
1. Processors	9
2. Configurable processors	10
3. Summary	11
4. Setting up eclipse	12
1. Install the WebDAV plug-in	12
2. Project layout	12
3. Sync the content	16
5. Applications	22
1. Assembling applications	22
2. Application description	22
3. Processor registry	23
4. Summary	24
6. Templates	25
1. Dynamic content	25
2. Optional or required content	26
3. Content type aware templating	26
4. Conditional templating	27
5. Nested conditions	28
6. Ignoring parts	29
7. Fragments	30
8. Summary	30
7. Workflows	31
1. Process definition	31
2. Fixed values	33
3. Casting values	34
4. Dynamic values	35
4.1. Loading stores values	35
4.2. Loading processor results	37
4.3. Nested dynamic values	39
4.4. Saving processor results	40
4.5. Timeout	41
4.6. Chaining processors	42
4.7. Routing	43
4.8. Putting it all together	43
4.9. Process description	46
5. Synchronous flows	46
6. Asynchronous flows	47
7. Nested flows	47
8. The scheduler	47
9. Bookmarks	47
8. Exception handling	48
9. Internationalization	49
10. Tables and Trees	50
1. Table layout	50
2. Advances tables	50
3. Table pager	50
4. Tree layout	50
5. Sitemap generation	50
11. Processing SQL	51
12. Processing XML	52
13. Form handling	53

1. Generating a standard form	53
2. Customizing a form	53
3. Designing form based workflows	53
4. Creating wizard like forms	53
14. Transactions	54
15. Extending Projector	55
1. Custom Processors	55

Chapter 1. Introduction

If you want to build complex and scalable web applications on top of a Slide or any other compliant WebDAV enabled content repository, Projector is a good choice. There are several existing frameworks for building web applications, but Projector was designed from the ground up to serve complex and highly scalable applications backed by a WebDAV server. As Projector requires support of WebDAV transactions and notifications, at least Slide and Microsoft Exchange are supported. Since the Slide 2.1 beta 1 release Projector is part of the Apache Jakarta Slide project.

1. Features

Projector offers:

- WebDAV-based content management
- Very simple templating to enforce the separation of content, layout and logic
- A workflow engine featuring synchronous, asynchronous and nested workflows
- Event triggered workflows
- Persistent workflows
- Contract based programming paradigm
- Content type aware templating
- Advanced form handling
- Prepared table and tree views
- Automatic sitemap generation
- Transaction handling
- XML processing
- Integration of relational database
- And much more...

In the following chapters we will dive into the projector framework step by step. Each section contains some examples that will finally show how to build a complete real world Projector application. This part of the book is directed to those who have some experience in developing Java based web applications.

2. Justification

Why do we need another web application framework, as there are several mature frameworks available and even for free? There have been good reasons for designing Projector beside the fact that it is designed for use in a WebDAV based environment:

- Separation of concern

If you are designing a very simple web application and you are the one and only who is working on the site, every technology might suit your needs. But if you are working on a bigger project and the layout of the web application is done by another person than the programming and the one who owns the website wants to change the content without charging the programmer or designer, you'll need a more sophisticated approach

that allows this so called separation of concerns.

The designer, who wants to change some parts of the layout, should not be able to destroy the hard and expensive work of the programmers and the editor of the website should not be able to destroy the layout that contains all the love of the websites designer. If you are familiar with JSP programming, you know that it is possible to mix the page layout and the java source code that not only renders complex parts of the page like tables and forms, but - even worse - sometimes also contains sensitive business logic. Someone not familiar with the java programming language, who wants to change the layout of the page, can by accident destroy some important parts of the application. Even if there are some new concepts, that help to avoid mixing the layout and the programming stuff, it is still possible and you will hardly find any project, where the separation of concerns is done well.

Projector tries to avoid this situation, by introducing a very simple and reduced templating mechanism that does not even offer the possibility to write some programming instructions into the template. This is for sure causing some pain, especially if you are a programmer and want to type in some very easy conditions or loops, but this is the only way to avoid the situation described before. A big advantage of the reduced templating is that anybody familiar with HTML can understand and write Projector templates himself. We have to accept that artists and programmers sometimes live in different worlds and things that are looking very easy to programmers are as hard to understand for a designer as the weird look of a webpage designed by a programmer. The Projector templating mechanism is described in detail in a later section.

- Contract based programming

What the heck is contract based programming? Well, it sounds good and in fact plays an important role in the Projector framework. It more or less means, that you describe the behavior of the methods you implement in detail and you'll earn a lot of benefits up to automatic form generation. This is a real killer feature, so the curtain for this show will not be opened now, but later.

- Event based workflow engine

Building complex web applications is a hard task. Even if the logic of the application is simple, it is sometimes difficult to do things at the right time. Projector helps you to schedule the programming logic by using a sophisticated event mechanism. If you have ever implemented a web application before, you know how to respond to user input: The servlet, JSP or whatever you've implemented will be called, after the user entered the URL of your page, submitted a form or pressed on a link or anything similar. This might be enough for simple scenarios. But what if the desired scenario is more complex?

Imagine the following: You are implementing an online store. After the user created his account, he is not buying anything for 30 days. Now you want to notify him by sending an email. If the user is not responding for another 10 days, you want to delete the account automatically. This is hard to implement, if you start from scratch, but it is very simple if you are using Projector. You only have to define the events that must occur before your code gets launched and Projector is dealing with all the details. Even if you are restarting the server, the state is still consistent.

- WebDAV-based content management

All parts of your Projector based application are stored in the underlying WebDAV repository. This is great, because every person working on the application can use his or her favorite applications to work on it: The designer can change layout templates with Dreamweaver or GoLive, the programmer can use Eclipse or any other WebDAV enabled IDE. The editor can use the office suite of his choice like MS Office or StarOffice to edit the content. Beside this wonderful experience, you gain all of the other benefits, which have been described in the previous parts of this book: Your content is versioned, can be searched and the access can be restricted and much more.

- Let processors work for you

Processors are the smallest building blocks in the Projector framework. Many prepared processors ship with the Projector framework and will help you solving common tasks in web development: You can generate tables that can be sorted and split across pages, without the need to write a single line of code. Tree view processors will help you to layout hierarchical data and generate navigation menus.

Projector offers a lot of features which let you enjoy the application development. Most web frameworks make easy life even easier. Projector was designed to let you also manage the harder parts. So, if you want to build complex web applications Projector is a good choice, even if it is brand new and not as widely spread as JSP or Struts for example.

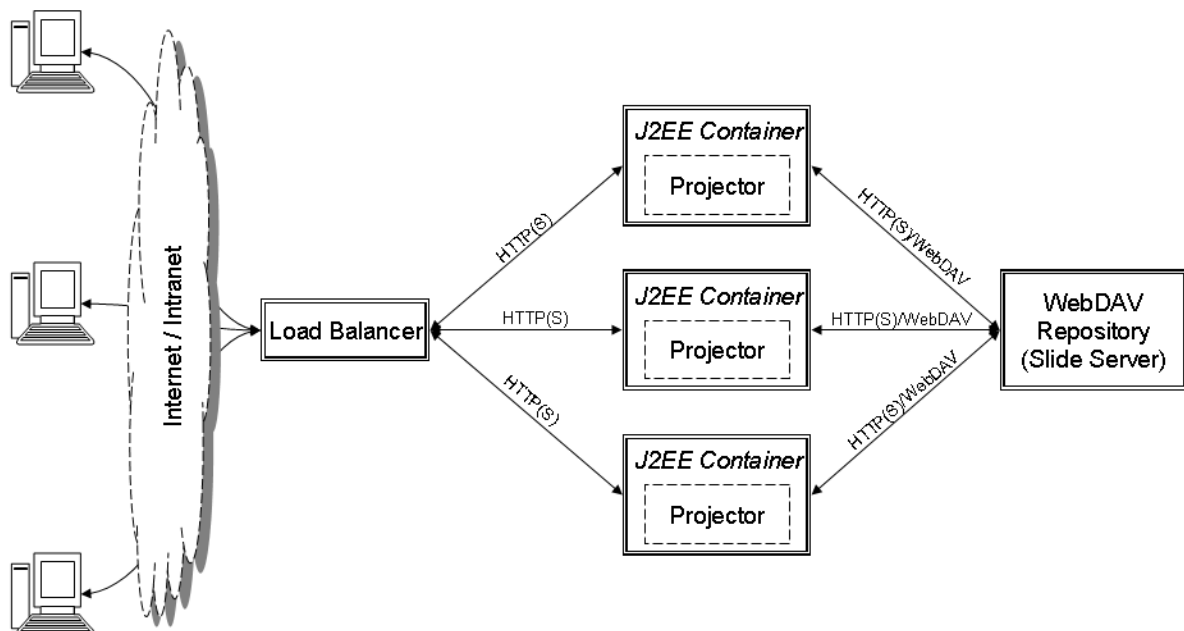
Chapter 2. The big picture

Projector is a J2EE application and as such needs a J2EE 1.3-compliant application server or at least a servlet container. If you download Projector as part of the Slide/Tomcat bundle you'll have a ready-to-run installation that will work fine for smaller projects. This installation is suited for small projects as it is sharing the same servlet container as the Slide server. If you want to serve a web application with high traffic you can install the Projector/Geronimo bundle that is preconfigured to run in a distributed environment.

1. Architecture

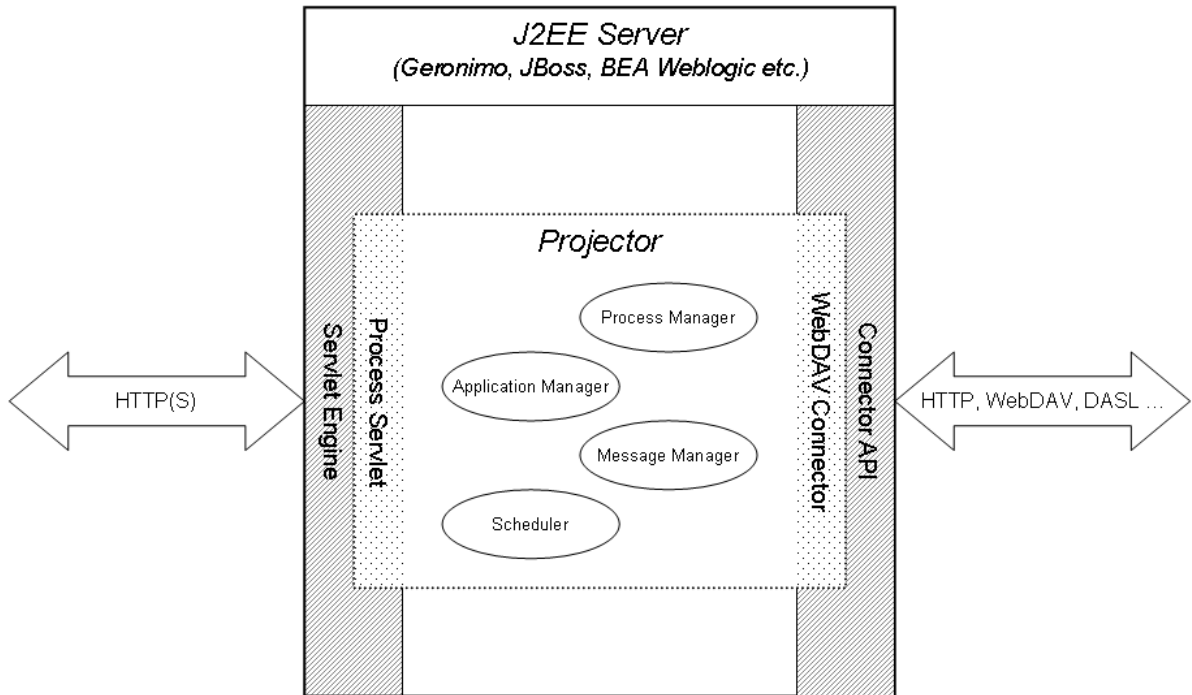
The following diagram shows a typical clustered scenario. On the left side you can see the users of a Projector based application. They connect to the application via their web browser. A load balancer is dispatching the requests to the different Projector servers to allow a highly scalable and failsafe application. The Apache web server works very well as a load balancer and can additionally do SSL encryption if needed.

Figure 2.1. Clustered Environment



The different Projector instances connect to the Slide server or and other compliant WebDAV repository via JCA. As the Slide server can also be clustered, a failsafe scenario can be achieved. The following diagram shows some more details of Projector. Projector itself is a J2EE based application that provides some services like a scheduler responsible for launching jobs when the specified conditions are met, a message manager that takes care of internationalization issues, the process manager that knows all available processors and their properties:

Figure 2.2. Projector



The main entry point for Projector is the process servlet that handles incoming HTTP-requests and invokes the corresponding processors. The result of the processor is delivered to the user as servlet response.

Chapter 3. Projector basics

Projector is very simple but yet very powerful. In this chapter you will learn the basic concepts that Projector is build upon. We try to keep this section as short as possible in order to start with practical exercises as soon as possible. It is the best way to learn a new technology by using it yourself. Don't bother if some things stay a little bit unclear as this fog will lighten later when we start with the practical parts.

1. Processors

The smallest building block in projector is called Processor. It is comparable to a method in the java world. You can do a lot inside a single method or you can do nothing. The same applies to processors in the Projector framework. Some contain a single line of code, others are fairly complex. This section will not teach you how to implement your own processor, it will just give you an impression what processors are. You will learn in a later chapter (Extending Projector) how to implement custom processors.

There are some characteristics of a processor that we need to know:

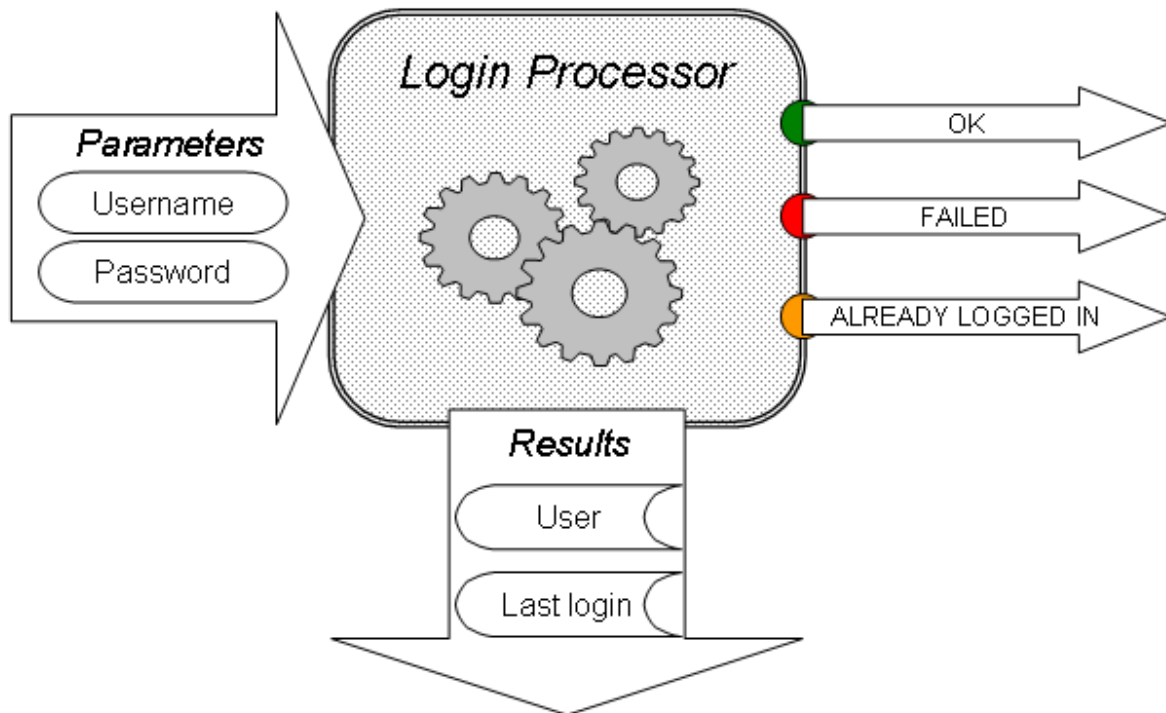
- A processor will tell you not only the types of the required parameters as we know it from Java methods but also the valid parameter ranges
- A processor can return more than one result object and can on top return a result state

The following illustration shows you a simplified login processor. This processor needs the two parameters *Username* and *Password* to perform the provided login action. It will provide two results when the action was performed successfully: The *User* object that represents the authorized user and may contain the full name and address of the user. Beside this it will return the date of the last login.

When the login action will be invoked, three states can occur:

- `OK` will be returned when the login was successful
- `FAILED` means that the user and password didn't match
- `ALREADY LOGGED IN` will be return if a user that tries to log in is already logged in

Figure 3.1. Login Processor



These result states can be used to build complex workflows by chaining different processors. In case of a login processor it might be a realistic workflow to show an error page if the login fails and to show a welcome page if the login was successful. We will see in a later chapter how to build workflows by chaining processors.

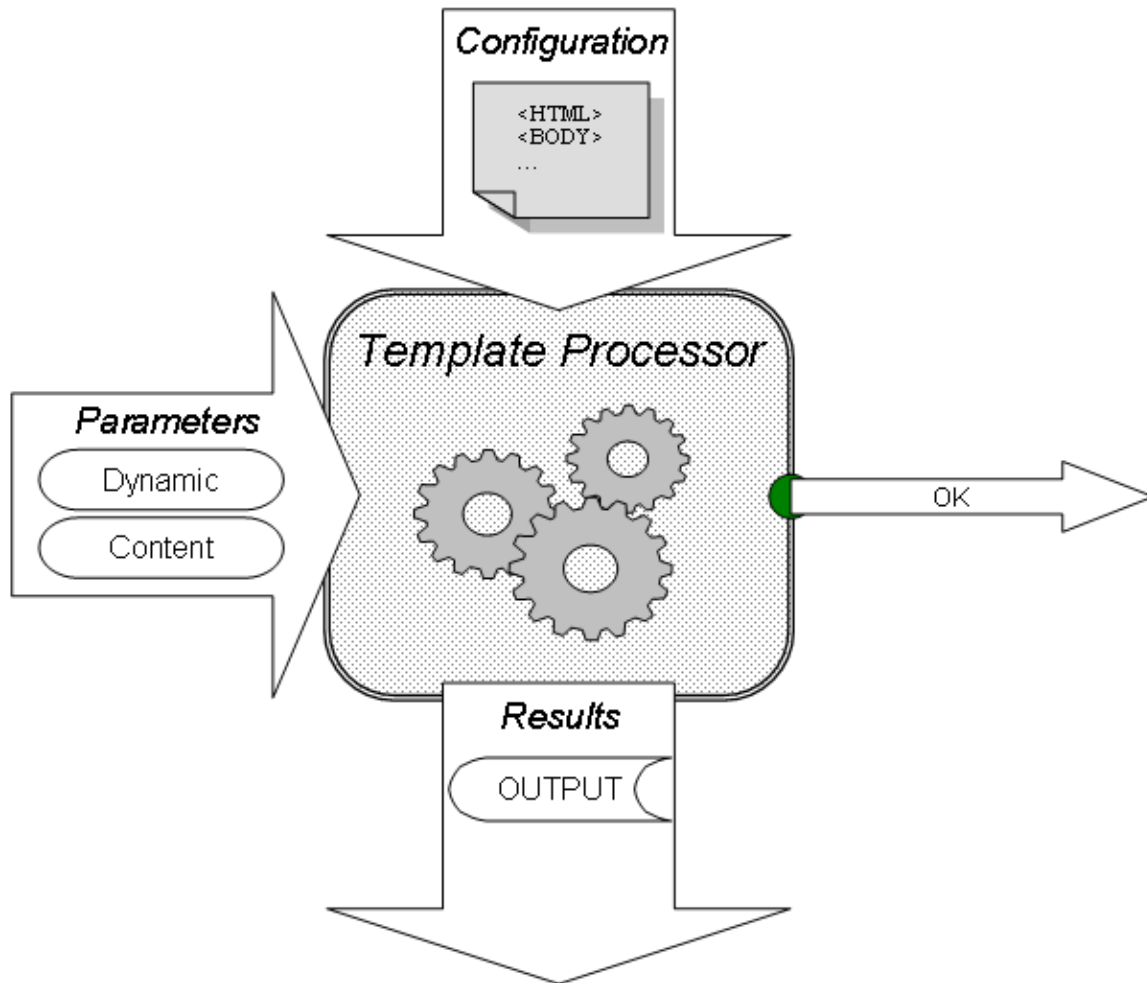
2. Configurable processors

Some processors can be configured to fit the needs of the user. In case of a login processor the configuration could contain the information that the processor needs to connect to a remote LDAP server. This information could also be passed as an input parameter, but the configuration has some significant advantages:

- Configurations are stored in the WebDAV-repository and can be edited with any client that is WebDAV aware.
- When Projector loads the processor at startup time, the configuration gets passed and can be parsed by the processor. This is a big advantage in terms of performance as it might take some time to parse the configuration and should because of that not be done each time the processor get invoked.

Let's have a look at a configurable processor that is one of the core processors of Projector: The template processor. The template processor is used to fill dynamic content into static documents like web pages, page fragments or XML documents.

Figure 3.2. Template Processor



The template processor gets configured with the template that is stored in the Slide server. When projector first starts up this template is parsed and precompiled so that the dynamic content can be filled in very fast. After parsing the template the processor knows which parameter it requires.

The only output that this processor delivers is the generated document. The only provided state is OK as the template evaluation will never fail. This is because the parameter description will assure that the processor will only be invoked if the necessary dynamic content is provided.

3. Summary

In this chapter you've learned the basic concepts of Projector.

Chapter 4. Setting up eclipse

Before we start diving into Projector, we need to set up the development environment to work on a Projector based application. This section describes the setup based on the Eclipse IDE. It is strongly recommended to use Eclipse 3.0 as the team and WebDAV support is much better and more stable than in the previous 2.x series.

This section will not cover the upcoming Projector plug-in as you will probably use another IDE and you would be lost if you only would know how to build applications using the plug-in. So even if this approach is a little bit more uncomfortable it is very useful as you'll also get a deeper understanding of the basic concepts and you'll honor the time and pain that the plug-in will save you in the future.

If you assemble a Projector application that only uses the processors that ship with Projector you do not need to setup a full java development environment. But as we later want to extend Projector by implementing some custom classes we will setup the java environment as well.

1. Install the WebDAV plug-in

First of all we have to install the FTP/WebDAV plug-in that is available via the main Eclipse download page. Download the plug-in and extract it into your Eclipse installation folder. After restarting the Eclipse IDE you are able to synchronize your workspace with any WebDAV compliant repository.

2. Project layout

Create a new Java project by using the New Project Wizard. We'll call this Project [MyProject] but this for sure is just an uninspired proposal and you can call it whatever you like. After entering the projects name click on the Finish button and let the wizard generate the basic project settings for you.

Figure 4.1. Create a new Java project...

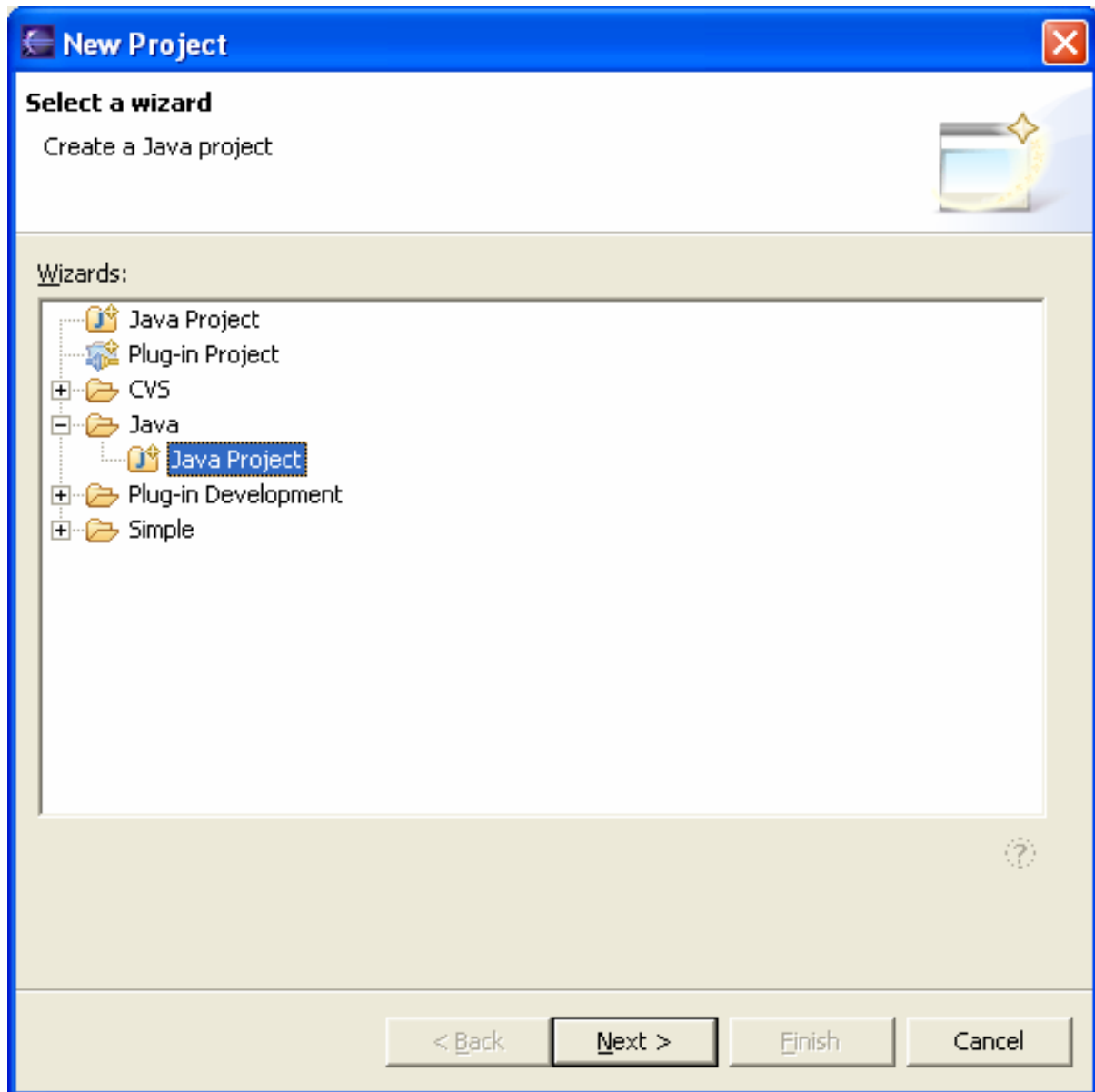
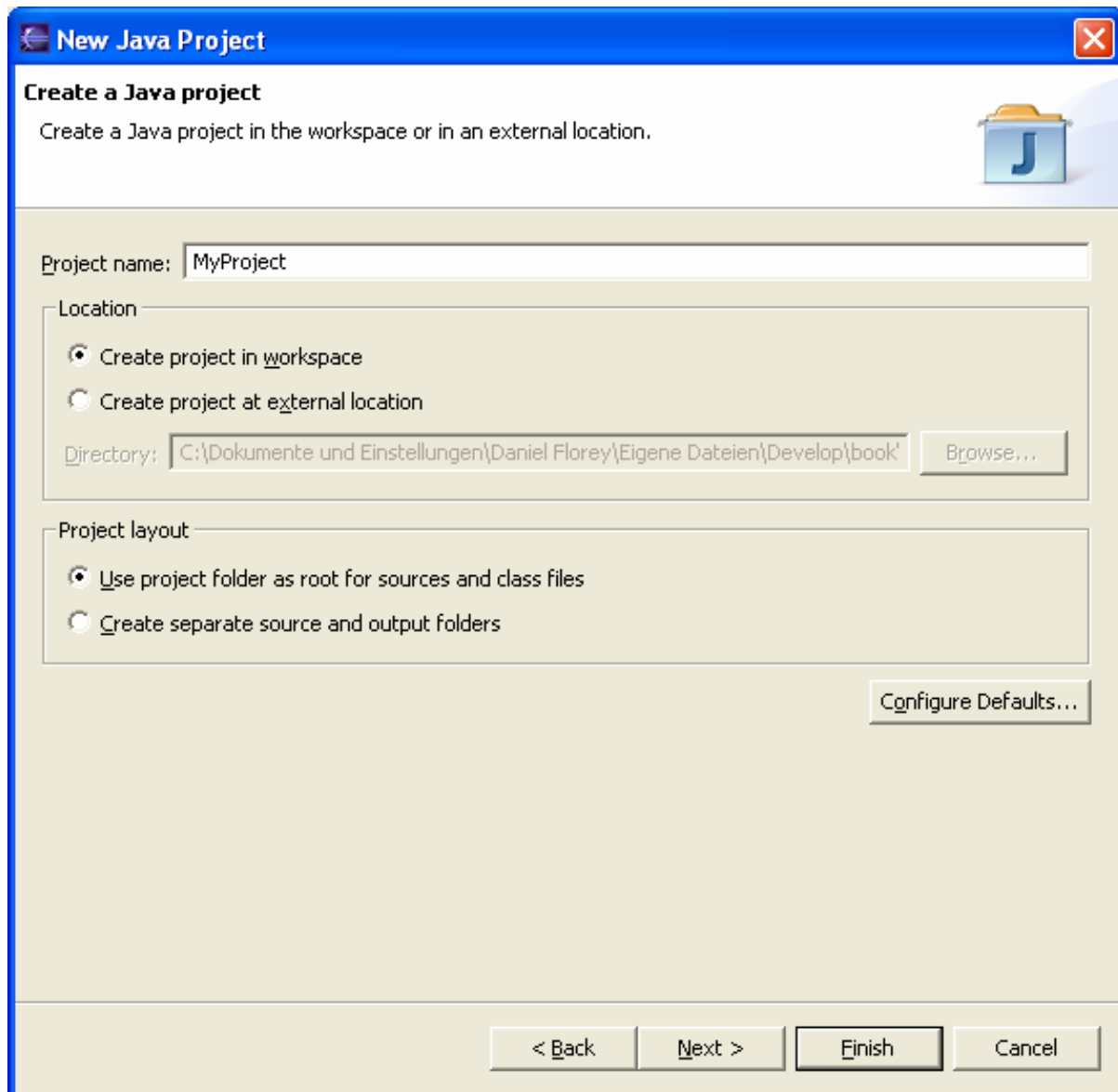


Figure 4.2. ...and call it whatever you like



Now the new project appears in your workbench and we want to create the default project layout manually. It is highly recommended to use a folder structure that is as close as possible to the proposed one, as this guarantees that every Projector user will know this structure if he is installing an application and will feel familiar.

First we need to create the source folder that will contain all the custom java classes that we develop in our project. Right-click on the project folder and select **new -> source folder**. Let's call this folder `java` as it will contain the java source code.

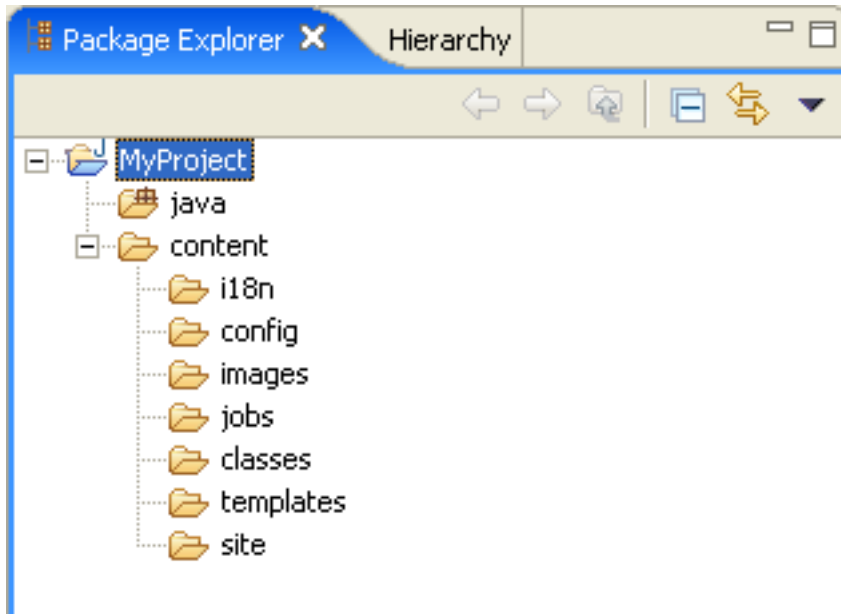
We need another folder that will contain all the project's content like templates, images and so on. Create this folder by selecting **new -> folder** and enter `content` as the folder's name.

Figure 4.3. Create folder



Now create the subfolders in the content folder that represent the default project structure. Finally your folder structure should look like this:

Figure 4.4. Create project structure



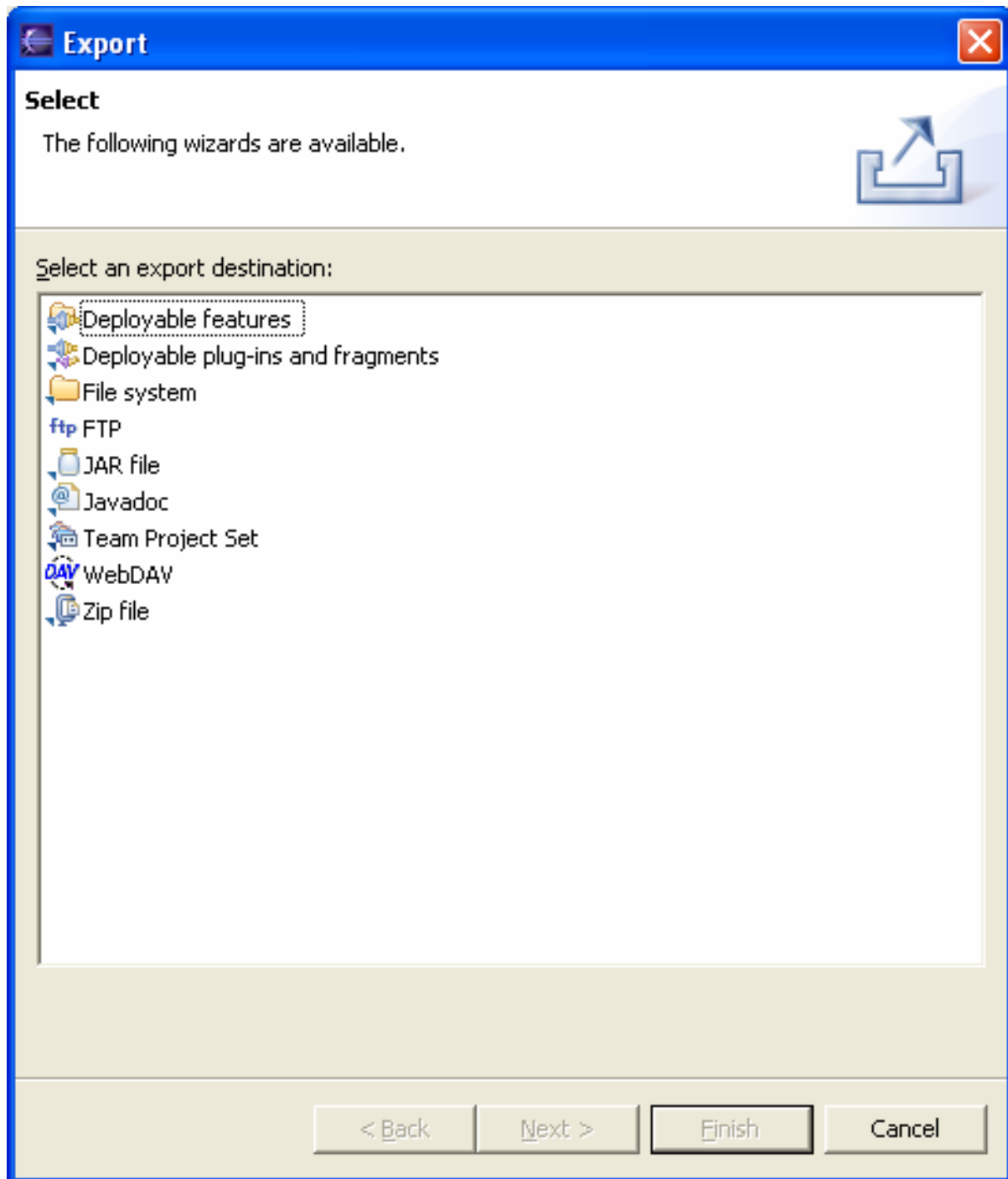
The `classes` folder will contain the compiled java classes so it is needed to set this folder as the project's default output folder. This can be done by selecting the `Java Build Path` entry in the project properties dialog and selecting this folder as the default output folder.

3. Sync the content

The content folder needs to be synchronized with the WebDAV repository as it contains the whole project. Before performing the following steps assure that your Slide server is up and running.

Right-click on this folder and select **Export**. The Export wizard appears and will show up an entry called WebDAV. If this entry is not present please check if the WebDAV plug-in has been installed successfully.

Figure 4.5. Export the content



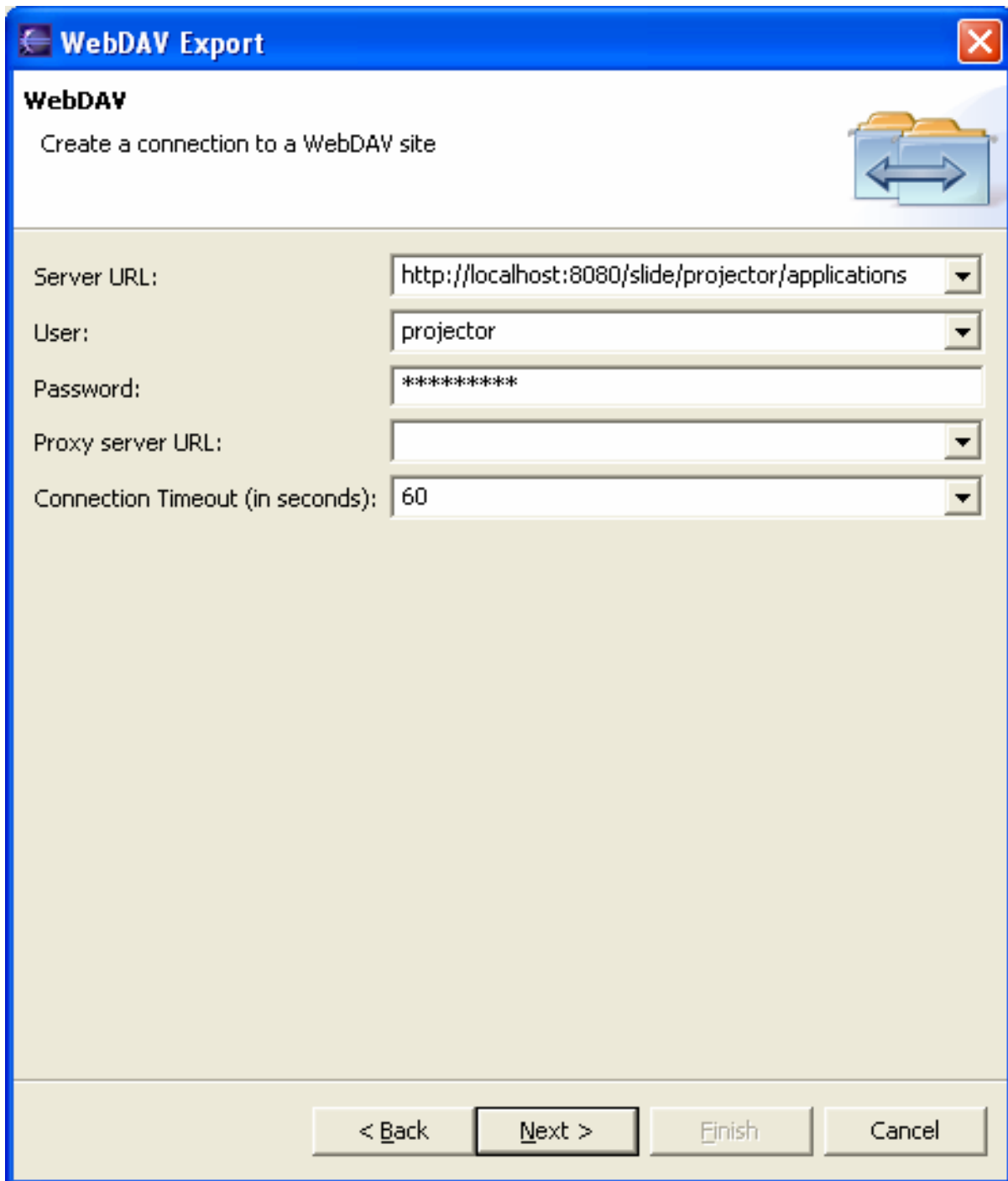
Select this entry and click the **Next**-button. Now select the content folder and proceed to the next step. Select **Create a new site** and click the **Next**-button.

Now you've to specify the location of your WebDAV-repository. When using the default Slide installation the URL to the Projector application directory will be

```
http://localhost:8080/slide/projector/applications
```

Enter this URL and specify the user and password that will be used to synchronize the content. Use the user `projector` with password `projector` if you have not changed the default Slide configuration.

Figure 4.6. Exporting content via WebDAV



After pressing the **Next**-Button, Eclipse will connect to the WebDAV-repository and will show up the desired application folder. Open it and create a new folder with the name of your application. The folder should appear and we select it in order to sync our content with the newly created remote folder. Now check the box that appears on the next screen and press the **Finish**-Button to start the synchronization.

Figure 4.7. Select the target site...

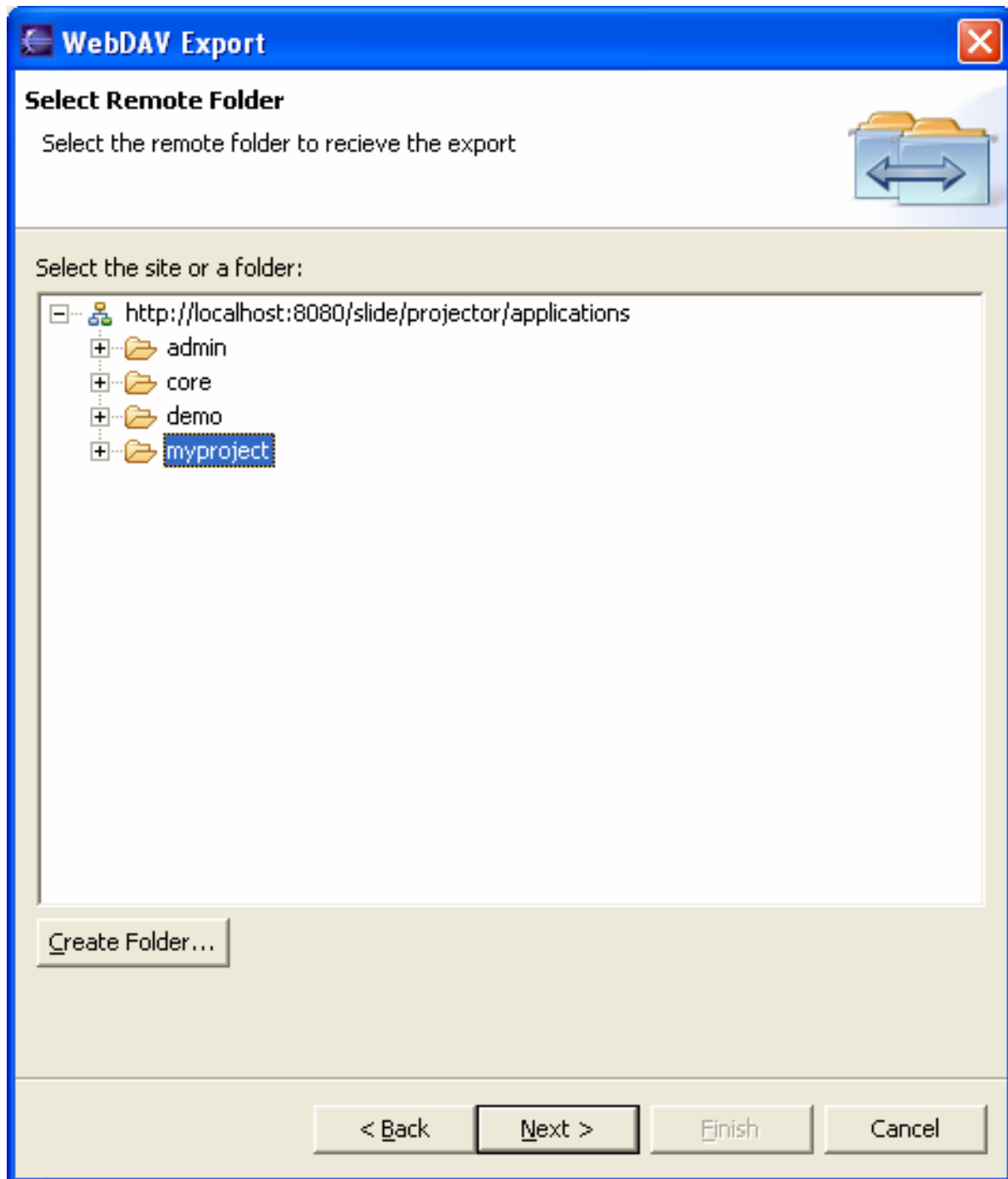
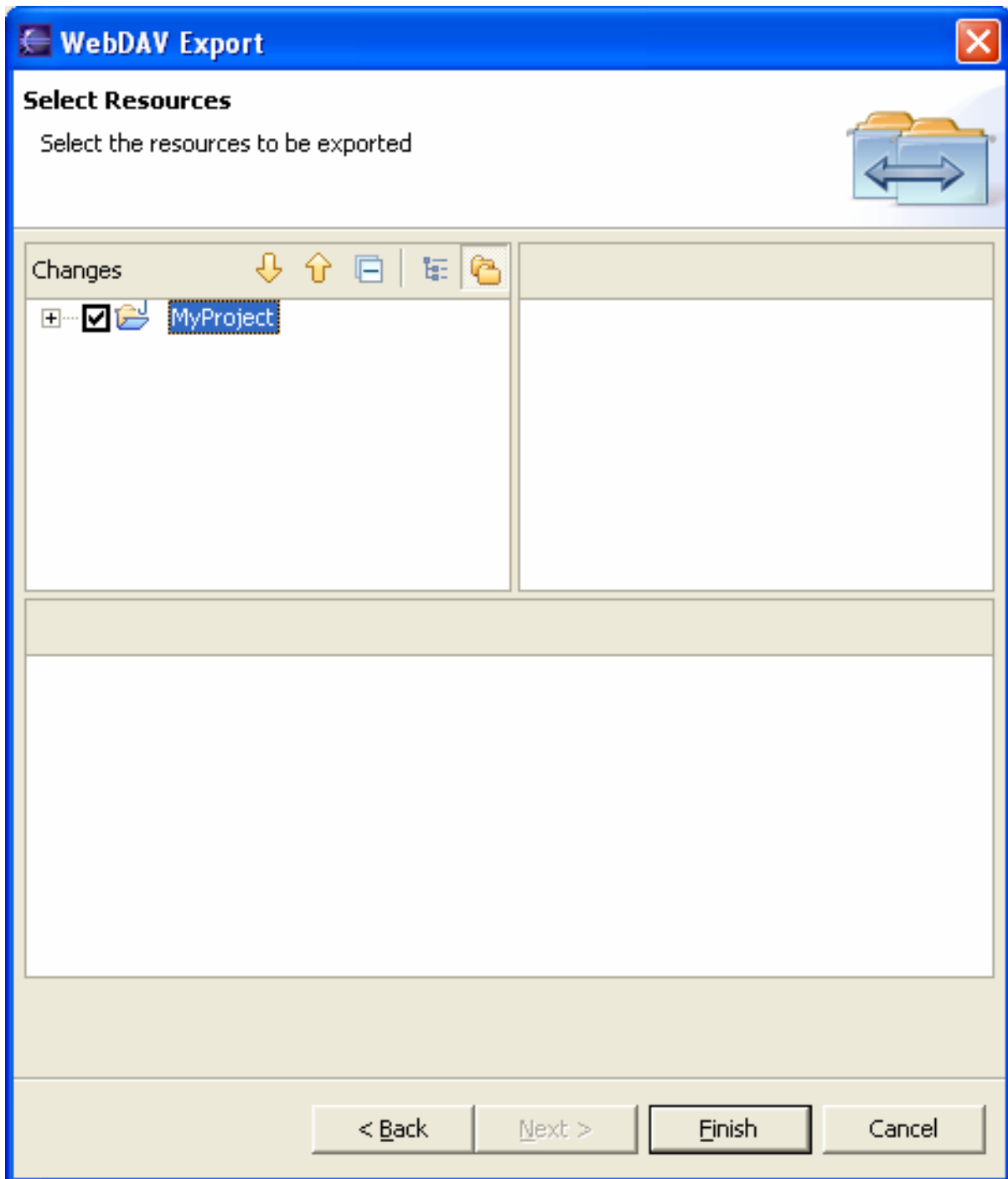
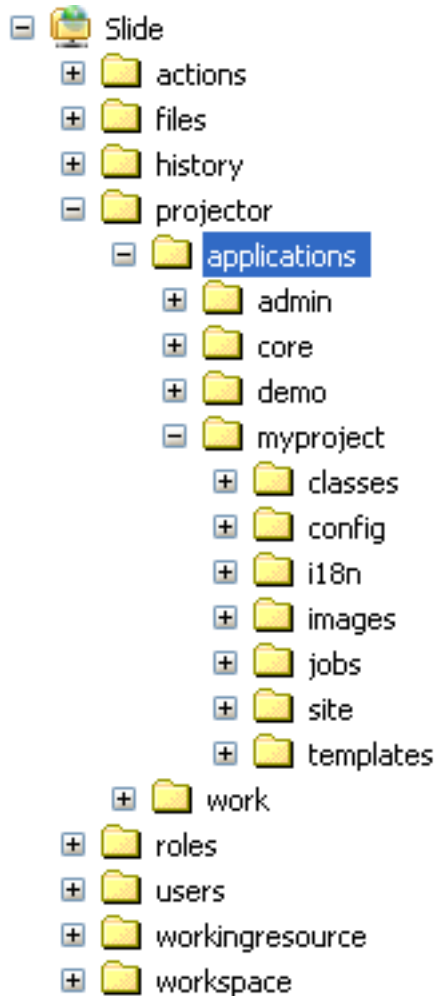


Figure 4.8. ...and the resources to be exported



After the synchronization has been completed you can check with any other tool if the content really has been created. If you are using Windows and you've mounted the Slide repository by using a Webfolder you'd see something like this:

Figure 4.9. Checking success via Webfolder



So you see that everything went fine and you are able to change the content of the application either by using Eclipse or by using any other WebDAV-enabled client. In order to keep informed about any changes that need to be synchronized you can open and configure the Eclipse Synchronize view. Click on the Synchronize-Button and select WebDAV as synchronization type.

It is a good idea to manage the whole project by using CVS as version control system. You can do this by using the team support as you know it from any Eclipse based project. So the WebDAV-synchronization is an additional option and doesn't conflict with the CVS-based team support.

That's it! Now you are ready for developing your first Projector application.

Chapter 5. Applications

As an experienced Java programmer you know that the J2EE world introduced the concept of web-applications some time ago. These applications are packed as a .war archive and can be deployed on any J2EE-compliant servlet container.

Projector applications are similar to the web-applications but there are some fundamental differences that might indicate when it's time for a Projector application and when you can stay with the good old web-applications.

First of all, web-applications normally don't get changed after they have been deployed. If you want to change some pieces of text of a web page you have to make these changes locally and package and redeploy the whole application afterwards. The content of a Projector based applications resides in the underlying WebDAV repository and as such can be changed easily on the live site with any tools that allow accessing the repository.

Second, the Projector applications can rely on each other. You can split your application into small reusable pieces. This is a big advantage if you for example want to change the layout of different applications. You can define a basic layout application that other applications depend on. Think of some small applications like a guestbook, a forum and so on that share the same layout. Changes to the application defining the layout will affect both immediately. As you can even share processors you can achieve a much higher grade of reusability.

It can be distinguished between simple applications that can be assembled by using the processors that ship with Projector and more complex applications that require the extension of the Projector framework itself by adding custom classes.

We want to start with the first as this chapter will focus on assembling and deploying Projector applications.

1. Assembling applications

We've already learned how the structure of an application looks like. Now we want to put something into our empty folders in order to develop a real world application.

First of all, we have to describe our application and tell Projector of which parts it consists. This is done by adding a description file to the `content` folder of our application. Remember that the `content` folder is the root folder of our application as we have exported this one to the WebDAV repository.

2. Application description

Right-click on the content folder and create a new file called `application.xml`.

Note

This file must not have any other name as Projector will lookup the file by this name and will fail to launch your application if you give it a different name.

You can also copy the description file from any other Projector application as a template. However this is how the description file should look like finally:

```
<application>
  <vendor>MyCompany (C) 2004</vendor>
  <author>Me, myself and I</author>
  <name>MyProject</name>
  <version>1.0</version>
  <display-name>My application</display-name>
  <description>This is my first Projector application</description>
  <dependencies>
    <requires application="core" version="1.x" /> ❶
  </dependencies>
  <content>
    <processors uri="config/processors.xml" /> ❷
  </content>
</application>
```

- ❶ The dependencies tag can contain one or more tags that indicate on which applications our application depends. If we want to use any parts as processors or templates of another application we need to declare this dependency at this point. Projector assures that these applications will be loaded and will be available before our application gets started. Optionally you can specify which version of the needed application is required.
- ❷ The content tag contains references to all files that should be loaded on startup. As we want to keep our example as simple as possible, we start by just using a single file that contains the processors used by our application. Note that the uri is relative to the root folder of our application.

Don't hesitate to replace the content of the descriptive tags with some text of your choice.

3. Processor registry

Now that we've told Projector what belongs to our application, we need to create a file called `processors.xml` that we've already referenced in our description file. This file must be created in the `config` folder as the name and location of the file must match the URI given in the application description.

You still don't know exactly what a processor is in terms of the Projector framework. This will be described in more detail later as it is part of the fundamental core of Projector. Up to now we only need to know that it is some portion of logic that is doing some fine stuff and can be invoked by mapping it to an URL. We now want to use the newly created file to define such a mapping:

```
<?xml version="1.0" encoding="UTF-8" ?>
<processors>
  <processor uri="hello"
            config-uri="templates/helloWorld.tmpl"
            class="org.apache.slide.projector.processor.TemplateRenderer" />
</processors>
```

What does this entry mean in detail? The processor with the implementing Java class `TemplateRenderer` is registered under the URI `hello`. This class is part of the Projector core, so this is the reason why our application depends on the core application. It is the base class that handles the default templating mechanism that will be described in detail in the following chapter. The `TemplateRenderer` is configurable, so we have to associate a configuration file with this processor instance. In case of the `TemplateRenderer` the configuration file means the template that will be used.

So the last step in our first example is to create a new file in the templates folder that is called `helloWorld.tmpl`. This file contains a very simple HTML-page:

```
<html>
  <body>
    <h1>Hello world!</h1>
  </body>
</html>
```

If you have a look at the Synchronize view in Eclipse you'll notice that our changes that have not been uploaded to the Slide server will show up. Upload all files by right-clicking on the root folder and selecting the **Upload** menu entry.

Now that we have made our first steps in the Projector universe we proudly want to have a look at the result. Open a web browser and enter the URL

`http://localhost:8080/projector/hello`

This URL assumes that you are running the Slide/Projector bundle on the same machine as your web browser and without changing the default configuration. If you are familiar with the J2EE world you might notice that `projector` is the name of the Projector web application itself. This can be change if you don't like to have any hints that you are using Projector in the URL of you site.

If this is the first time that you access Projector it might take some time until the result appears as Projector starts up and launches all applications that have been deployed. Finally you will see the following:

Figure 5.1. Hello world!



Congratulations! You've managed to get your first Projector application running. The result might be slightly disappointing if we take into account that the same result could have been achieved by simply dropping the HTML page into the Slide server as shown in the previous parts of this book, but – promised – soon you will see the first benefits of using Projector.

4. Summary

In this chapter you've learned how the default Projector application layout looks like and how to set up the corresponding folder structure in Eclipse. You have successfully configured the `TemplateRenderer` with a very simple template and made it accessible via an URL.

Finally you've got the obligatory "Hello world" application running.

Chapter 6. Templates

Projector offers a very simple templating mechanism. Templating is used to insert dynamic content into static web pages or any other kind of document. You may have come across different types of templating, as a java programmer you might at least know JSP. In a JSP you mix the layout of a page with programming instructions that generate the dynamic parts of the page.

This is very nice, if you have some simple web pages that need to show some data from a database or any other backend system. This is a nice thing from the programmer's point of view, but it is no good if you for example want to relaunch your website with a brand new design. The designer can by accident delete or modify some code and this can cause some real trouble.

Projector's templating is very easy and it does not allow you to write any Java source code into the template. You just can insert special tags where dynamic content should be filled in.

1. Dynamic content

We have already used a Projector template in our "Hello world" example. Now we want to improve this page by giving the user a warm welcome. We want to welcome the user with his given name so we replace "world" with a tag that indicates that this part should be filled with dynamic content, in our case with the name of the current user.

```
<html>
  <body>
    <h1>Hello <%username%!</h1>
  </body>
</html>
```

As you can see, you don't tell Projector at this stage how to retrieve the username. It is just a name of a variable that will be bound later. Why? The reusability of the template will be increased dramatically if you don't specify how to retrieve the dynamic content. If you are using the template only once this might look a bit uncomfortable and overdosed, but as your templates might be much more complex in real life, you will be happy if you can reuse them across different projects.

Thanks to this approach Projector can offer some generic templates that can be used out of the box to generate tables, forms or menus.

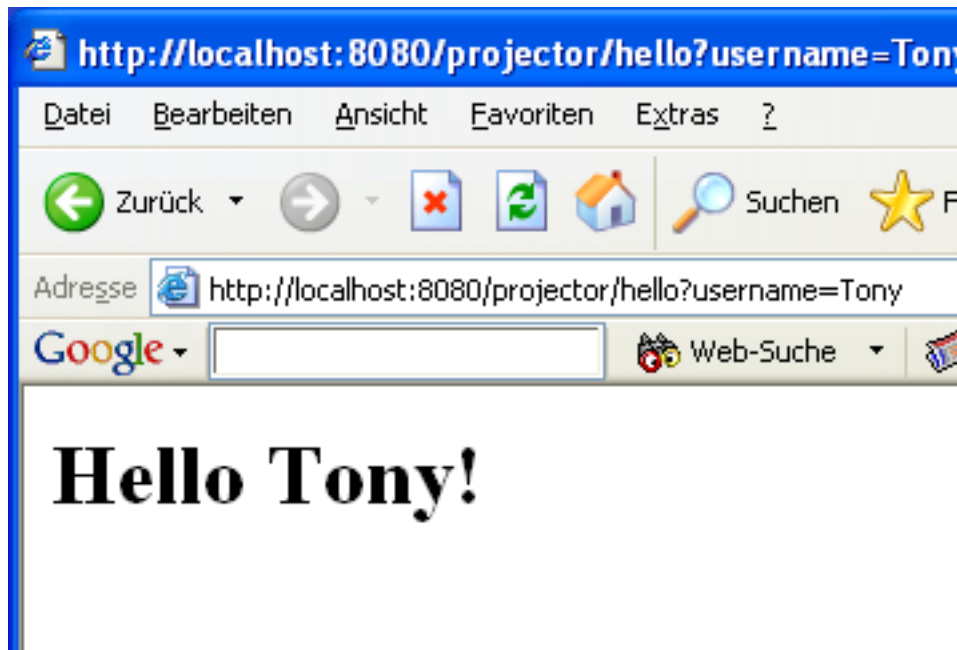
After uploading the template we try to reload the page in our web browser. But instead of seeing our lovely welcome page, Projector tells us that a variable is missing! This is very true as we just defined a variable by entering the Projector tag that defines dynamic content.

At this stage we can support our template processor with the required input by adding this information to our URL:

<http://localhost:8080/projector/hello?username=Tony>

Now that we provide the required input our page looks like this:

Figure 6.1. Dynamic content



2. Optional or required content

There are some more hints you can give Projector when designing a template. You can specify, if a variable is required or optional. We have already seen that dynamic content is required by default.

Now we want to make the username optional as we want to make our template work, even if the username is not available:

```
<html>
  <body>
    <h1>Hello <%username;optional%!</h1>
  </body>
</html>
```

Upload the file and check if this works. If you don't provide the username by using the URL of the "Hello world" example the page should still work. Of course the username is missing but this is exactly what we wanted by declaring the username optional.

3. Content type aware templating

We can restrict the allowed values that can be bound to the variable by specifying the content type. In our example we expect that the username is plain text, so we can add this to our template:

```
<html>
  <body>
    <h1>Hello <%username;optional;text/plain%></h1>
  </body>
</html>
```

If someone using our template tries to bind a number value to the username variable, Projector will tell him not to do so. You can enumerate more than one legal content type by separating them with a comma.

Note

If you want restrict the valid content types, you always have to specify, if the variable is optional or required.

As you see, it is really simple to write a Projector template if you are familiar with languages like HTML or XML. You can use this templating mechanism to generate documents of any other markup based language.

The restriction of the content type is very useful in many ways: You can show the user of the template what kind of content is expected at a special position as this is often not fully described by the variables names. You can even restrict the content type in a way that will show that for example an HTML table is expected as content of a variable. By this way it is possible for Projector to ensure that the generated pages contain valid HTML and don't get broken by inserting some malformed content.

4. Conditional templating

There is one big exception that breaks the law of logic free templates: You can skip a part of the template if a variable is not set. This one and only exception was introduced as this is needed very often and still keeps the templates readable.

Let's modify our example to get a simple use case:

```
<html>
  <body>
    <?username <h1>Hello <%username;optional;text/plain%>!</h1>?>
  </body>
</html>
```

What does this mean? If the variable following the question mark is unset, all characters up to the closing question mark tag are skipped. If the variable is set, the characters following the blank after the variable's name are printed. The variable used inside such a statement automatically gets optional.

So upload the modifications and try again with both URL we've use so far. You can see that the welcome message is skipped if the variable `username` is not set. The conditional templating was above all introduced to enable the skipping of optional attributes. This is a task needed very often as HTML is aware of many optional attributes. If you want for example let the user of your template modify the background color of the page, you can do it like this:

```
<html>
  <body <?color bgcolor="<%color%>"?>>
    <h1>Hello <%username;optional;text/plain%>!</h1>
  </body>
</html>
```

The optional attribute `bgcolor` that can be defined in the `body`-tag will not be printed if the `color` variable is not set. Upload the changes and check this behaviour by optionally adding the `color` variable to the URL:

<http://localhost:8080/projector/hello?username=Daniel&color=red>

The page will look like this:

Figure 6.2. Dynamic background color



5. Nested conditions

It is possible to use even nested conditions. This can be a very powerful feature but should be only used as a last resort as it makes the template more complicated. But to be complete we want to show this possibility in our last example:

```
<html>
  <body>
    <?username <h1 <?style style="<%style%"?>>Hello <%username%!</h1?>
  </body>
</html>
```

If you followed the previous examples carefully you can call yourself an expert of Projector templating and might be able to understand this example without explanations as it is just a combination of the previous ones: If the variable username is not set, the whole headline is skipped and the page will show up empty.

Check this by entering the URL without this parameter as you've done in the "Hello world" example. If you open the page source of the rendered page, you'll see that the body tag is empty:

```
<html>
  <body>
  </body>
</html>
```

Now add the username as parameter as we've done in the first example or this chapter. This is the page source of the generated page:

```
<html>
  <body>
    <h1 >Hello Tony!</h1>
  </body>
</html>
```

You can see that the headline is printed but the style attribute is not yet included. We can apply a style to the headline by encoding the style parameter to the URL:

http://localhost:8080/projector/hello?style=color:red&username=Tony

The page that will show up looks like this:

Figure 6.3. Dynamic style



This is the associated page source:

```
<html>
  <body>
    <h1 style="color:red;">Hello Tony!</h1>
  </body>
</html>
```

You can play around with this example and add some other style attributes to the headline.

6. Ignoring parts

Sometimes it is nice to add some text to the template that should be ignored when the template gets evaluated. In larger projects the web design is done by a different company or at least a different person than the programming. The layout is done with some HTML editor or coded by hand and the parts where dynamic content will appear, is filled with some dummy text. If you get such HTML pages and want to convert them into Projector templates you can simply skip these parts by adding tags that define the start and the end of this dummy text block.

If you get the following page and need to convert it into a Projector template...

```
<HTML>
<BODY>
<P>This is the text that will be replaced with dynamic content</P>
</BODY>
</HTML>
```

...you can simply mark the dummy text with ignore tags:

```
<HTML>
<BODY>
<!--*** Start ignore ***-->
<P>This is the text that will be replaced with dynamic content</P>
<!--*** End ignore ***-->
<%text%>
</BODY>
</HTML>
```

Even if the same visual result could be achieved by using HTML comments, this would have two disadvantages:

- The resulting page source is bloated up with all of the dummy text that should not be of interest for the users
- If the dummy block already contains some comments you'll run into trouble as it is illegal to create nested comments.

7. Fragments

If you've ever converted a given layout into a dynamic web application before, you know that you have to split up the HTML source into tiny parts that represent a single component. Think of a layout that consist of a page with two columns and some boxes or text blocks that will be reused all over the website.

The first step is to understand the HTML source code and to find the start and end of such a component. When using JSP or other technologies you normally have to split up your HTML page into a large number of tiny files that contain the reusable fragments.

When using Projector templating you can simply mark such fragments in the original page source and use them by giving them an identifying name. The following example shows a simplified HTML page and how different components can be marked as reusable fragments:

[TODO]

```
<HTML>
<BODY>
<P>This is the text that will be replaced with dynamic content</P>
</BODY>
</HTML>
```

8. Summary

In this chapter we have seen how Projector templating looks like. It is very simple and strict and enforces the separation of program logic and layout information by simply not supporting any Java constructs inside the template.

You have learned how to declare dynamic content, to make it optional or query the existence of content to skip parts of the template. We will see many more advanced examples of templating in this part of the book.

Chapter 7. Workflows

The heart of Projector is a powerful workflow engine that can solve very different tasks. Let's carefully get closer to this core technology and use our very first workflow in our example application.

A workflow is called [process] in Projector terminology as it is chaining different processors together to a single process.

1. Process definition

We want to use a process to fill the dynamic content of our template so that we don't have to encode all parameters into the calling URL.

The process definition for this tasks looks like this:

```
<?xml version="1.0" encoding="UTF-8" ?>
<process>
  <description> ❶
    <output>
      <state>ok</state> ❷
      <result name="welcomePage" ❸
        description="composedPage" ❹
        content-type="text/html" ❺
        presentable="true" /> ❻
    </output>
  </description>

  <step id="compose" processor="hello"> ❼
    <load parameter="username"> ❽
      <string>Tony</string> ❾
    </load>
    <load parameter="style"> ❿
      <string>color:red;font:italic;</string> 11
    </load>
    <save result="output" 12
      store="output" 13
      key="welcomePage" 14
      presentable="true"/> 15
    <route state="ok" return="ok" /> 16
  </step>
</process>
```

- ❶ The description element contains the contract of this process. The contract describes the required parameters, the provided results and the possible return states as described in the chapter Projector basics
- ❷ The only state that this process can return is the state with the identifier OK
- ❸ Only one result is provided. The result is accessible via its name `welcomePage`
- ❹ The description attribute of the result contains a key to the localized description. This detailed description must be added to any message file that the Message Manager is aware of
- ❺ The content type of this result is of type `text/html` as the generated page will be a valid HTML document.
- ❻ We declare this result `presentable` as it can be displayed to the user
- ❼ Our process consists of a single step with the id `compose`. When this step gets launched the processor with the given URL `hello` will be invoked. This is exactly the processor that we have registered in our previous example.
- ❽ Before the processor gets launched, values can be assigned to the required parameters. The parameter attribute identifies a parameter by its name. In this example we assign a value to the parameters `username` and `style`.
- ❾ We load the parameters with suitable string values.
- ❿ We want to save the result with the key `output`. This is the result that the template processor registered

under the URL `hello` provides. If you don't remember this, take a look back at the chapter *Projector basics*.

Note

If the processor that is associated with the step provides more than one result, you should remember that every result that is not saved will be lost.

We save the result with the key `output` to the output store of this process by using the defined key `welcomePage` that we have defined in the contract of this process. So this result will be accessible to the caller of our process.

The result that we bind to the key `welcomePage` can be displayed to the user so we define it as a presentable result.

We route the state `ok` of the template processor to the return state `ok` of our process. Return states are identified by a string. The string `ok` is the convention for indicating that everything went fine.

Create a file with this process definition in the `content/site` folder of our project and call it `welcome.xml`. In order to test our process we have to add it to the processor registry:

```
<?xml version="1.0" encoding="UTF-8" ?>
<processors>
  <processor uri="hello"
            config-uri="templates/helloWorld.tpl"
            class="org.apache.slide.projector.processor.TemplateRenderer" />
  <processor uri="welcome.html"
            config-uri="site/welcome.xml"
            class="org.apache.slide.projector.processor.process.Process" /> ❶
</processors>
```

- ❶ We add a new processor to our registry with the implementing class `Process`. We tell *Projector* that this process will be accessible via the URL `welcome.html` and the process definition can be found at the relative URL `site/welcome.xml`.

As you see in this first workflow lesson, the implementing class `Process` that deals with workflows is just another configurable processor. In opposite to the first configurable processor that we have known, the template processor, the configuration file contains no template but the process definition. We register this processor and the associated configuration file just in the same way as we have already done it using the template processor.

The best thing of it all: We can use our newly defined process as a processor in other process definitions! By this way we can easily create nested workflows. We will heavily use this great feature in our upcoming examples.

But before we continue we should test our work by uploading the changes and calling the defined process. This can be done by invoking the associated URL as we already have learned:

`http://localhost:8080/projector/welcome.html`

The result page looks well known:

Figure 7.1. The result of the process



2. Fixed values

As we have seen in our example we've loaded the processor parameters with string values using the `<string>` tag. This tag is used to create a fixed string value.

Each value type that Projector is aware of has a corresponding element to instantiate it. The following table shows the core types that are defined in the Projector core application and a simple example how to create them.

Table 7.1. Core types

Type	Example
String	<code><string>Some text</string></code>
Boolean	<code><boolean>>true</boolean></code>
Number	<code><number>4711</number></code>
Date	<code><date>537456834756</date></code>
Locale	<code><locale>en_US</locale></code>
Array	<code><array> <string>First element</string> <string>Second element</string> </array></code>
Map	<code><map> <entry key="name"> <string>Tony Tomato</string> </entry> <entry key="trusted"> <boolean>true</boolean> </entry> </map></code>

Type	Example
XML	<pre><xml> <news> <title>Slide 3.0 released</title> <author>Apache Software Foundation</author> </news> </xml></pre>
URI	<pre><uri>/projector/applications/myproject</uri></pre>
Message	<pre><message id="login successful"> <string>Tony Tomato</string> <number>2</number> </message></pre>

As you can see from this table Projector also supports complex data types like arrays, lists and maps. These complex types can be nested so that a map can contain an array or a map as a map entry. We will need this ability in some of the upcoming examples and see how it works.

3. Casting values

What will happen if we try to assign another value type to a parameter than the parameter requires? Let's try it out by loading the *username* with a date value. Modify the bold lines in the process definition and upload the file.

```
<?xml version="1.0" encoding="UTF-8" ?>
<process>
  ...

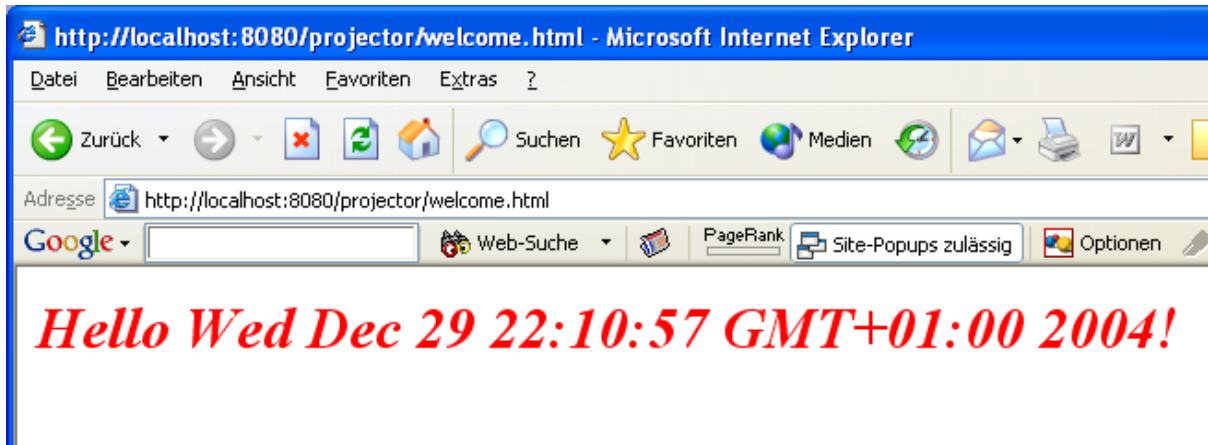
  <step id="compose" processor="hello">
    <load parameter="username">
      <date>1104354657683</date> ❶
    </load>
    <load parameter="style">
      <string>color:red;font:italic;</string>
    </load>
  </step>

  ...
</process>
```

❶ Date values are created with the desired time in milliseconds since 1970

When we reload the page we will see that no error arises as you might have expected but the date is shown in a human readable form:

Figure 7.2. Say hello to the current date



As we have not defined a required content type in the last current of our template, Projector tries to cast any given value into a printable form as the template required a printable value. Projector will always do its best to fulfill the requirements of a processor. If you for example feed a processor that requires a number value with a string value it will try to parse the string and create a matching number value. This is very nice because it will enable you to invoke many processors by encoding parameters into the calling URL as we have seen it in our very first examples.

If we would restrict variable in the template to the content type `text/plain` this would not work any more as Projector would assure that only values that represent plain text would be accepted. Modify the template so that the content type is restricted:

```
<html>
  <body>
    <?username <h1 <?style style="<%style%>"?>>Hello <%username;required;text/plain%>!</h1?>>
  </body>
</html>
```

After upload this changes and reloading the page you'll see that Projector throws a detailed exception that will tell us that it's illegal to load a date value to the parameter `username`. In summary it can be said that Projector will always try to cast the values so that they will fit the needs of a processor but it will not do something illegal but will throw an exception instead that will give you detailed information if processing was aborted.

4. Dynamic values

Processor parameters can be loaded not only with fixed values, but with dynamic values that will be evaluated at runtime. These dynamic value or [Any Values] as they are called in Projector terminology offer a rich set of possibilities how to retrieve the dynamic values. You can load values that have been stored somewhere before or you can even use these dynamic values to load a parameter with a single result entry of another processor.

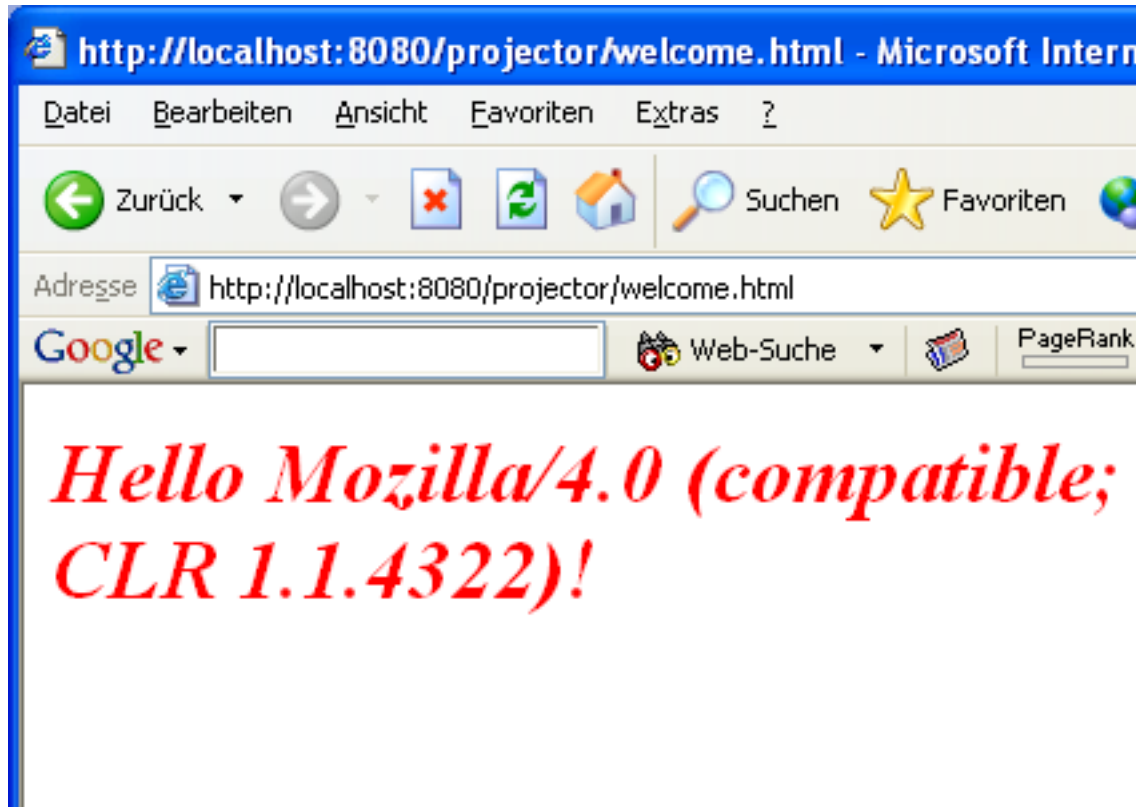
4.1. Loading stores values

Our next lesson will teach us how to load parameters with stored values. Change the highlighted lines in the process definition so that it looks like this:

```
<?xml version="1.0" encoding="UTF-8" ?>
<process>
  ...
  <step id="compose" processor="hello">
    <load parameter="username">
      <value key="user-agent" store="request-header"/>
    </load>
    <load parameter="style">
      <string>color:red;font:italic;</string>
    </load>
  </step>
  ...
</process>
```

The value tag is very important as it enables you to access different stores that are provided by Projector. One of these stores is the `request-header` store. This store can be used to access header fields of the incoming HTTP request. In our example we will load the parameter `username` with the content of the request header field `user-agent`. Upload the changes and reload the page:

Figure 7.3. Detecting the user agent



It worked! Instead of good old Tony or the date the content of the request header field `user-agent` is displayed. This might be of interest later as we could use this information to build a multi channel application that can display pages differently depending on the user-agent.

The following table will give you a brief overview of the available stores and their meaning. As we will use most stores in future examples you will get familiar with them very soon.

Table 7.2. Available stores

Store	Description	HTTP specific
request-parameter	This store contains all parameters that have been encoded in the request URL. You cannot save any data to this store as this store is a read only.	X
request-attribute	If you want to save or access values that have a scope of a single HTTP-request you can use this store. This can be useful if you call a processor out of another web technology like Struts or JSP and want to share some request related information.	X
request-header	This store can be used to read the header fields of a HTTP-request. We already used this one to retrieve the user-agent header field.	X

Store	Description	HTTP specific
Session	The session store is used to load or save values to the current session. The lifetime and clustering of sessions is done by the enclosing J2EE-container.	X
Context	The context store enables you to store information that will last for the scope of a single process invocation. It is comparable to the request-attribute store if you call a processor via HTTP but it will give the ability to call a processor without HTTP environment and will by this make it much more flexible and reusable. Use this store to share data between different processors in the same call.	
Cache	This store represents the transaction aware cache of Projector. You can share data not only between different processors in the same call but for a longer lifetime. So this is the right choice if you want to share data between different calls. This store is mostly used to cache rendered pages, page fragments or the result of time consuming search requests to speed up your application.	
Repository	This store gives you read and write access to the content that is stored in the underlying WebDAV-server. As the repository can be located on another machine and might be accessed via a secure connection as we have seen in the architectural overview, accessing this store can be slow.	
Input	Accessing the input parameters of your process is possible via the input store. You should not write any values to this store as it is available for retrieving the defined input parameters of your process.	
output	If you want to store results of your process, use the output store. This is a write only store and you should never try to read values from this store. Stored values will be available in scope of the enclosing process.	
Process	You can store data in scope of your process even if it spans multiple requests or pages in terms of HTTP.	
Persistent-process	This store is similar to the process store but will store the shared data in the underlying repository so that it will not be lost event if you start the Projector server.	
Form	The form store is used to store and retrieve data that is bound to a specific form. If you think of a web page that contains some forms, the state of each form is stored in the corresponding form store.	
Step	If you want to pass data from one processor step to the directly following step you can use the step store to do so.	

Note

All stores that are marked as HTTP specific can be used to access the well known stores available in the J2EE world. So they are a good choice if you know that you process will be invoked from a HTTP call. Projector is able to call a process in background and when doing so the HTTP environment will not be available. Each process that makes use of these HTTP specific stores can't be called in background and is limited to a HTTP based call. So try to avoid the use of these stores whenever possible.

4.2. Loading processor results

The `<value>` tag can not only be used to read from a store, but can also be used to invoke a process implicitly and retrieve a result entry of the called processor. The following example shows us how to achieve this:

...

```

<step id="compose" processor="hello">
  <load parameter="username">
    <value processor="system" result="currentDate"/>
  </load>
  <load parameter="style">
    <string>color:red;font:italic;</string>
  </load>
</step>
...

```

When the parameter `username` gets loaded, the processor registered by the URL system is called and the result with the key `currentDate` is bound to the parameter. The system processor can be used to access some information related to the system environment comparable to the `java.lang.System` class that you might know. The system processor doesn't require any input and can because of that be invoked this way.

If you want to test this example please remove the content type restriction from the template that we use as we have learned before that a date value is not accepted if the content type is restricted. If you've changed the template and have uploaded the latest changes our page should display the current date.

It is also possible to call a processor that required a single parameter as input. To show this by example we now introduce the `DateFormatter` processor. This processor can be used to format a date to the locale specific representation.

Example 7.1. Portrait of the DateFormatter

Description	This processor can be used to format a date to the locale specific representation
URL	<code>dateFormatter</code>
Class	<code>org.apache.projector.processor.text.DateFormatter</code>

Table 7.3. Parameters

Name	Description	Allowed Values	Required	Default Value
Date	The date to format	Any values of type <code>dateValue</code>	Yes	
dateFormat	Specifies the date related representation of the formatted result.	One of the following <code>stringValue</code> values: <code>none</code> , <code>short</code> , <code>medium</code> , <code>long</code> , <code>full</code>	No	<code>short</code>
timeFormat	Specifies the time related representation of the formatted result.	One of the following <code>stringValue</code> values: <code>none</code> , <code>short</code> , <code>medium</code> , <code>long</code> , <code>full</code>	No	<code>short</code>
locale	Specifies the locale	Any values of type <code>locale</code>	No	<code>default locale</code>

Table 7.4. Results

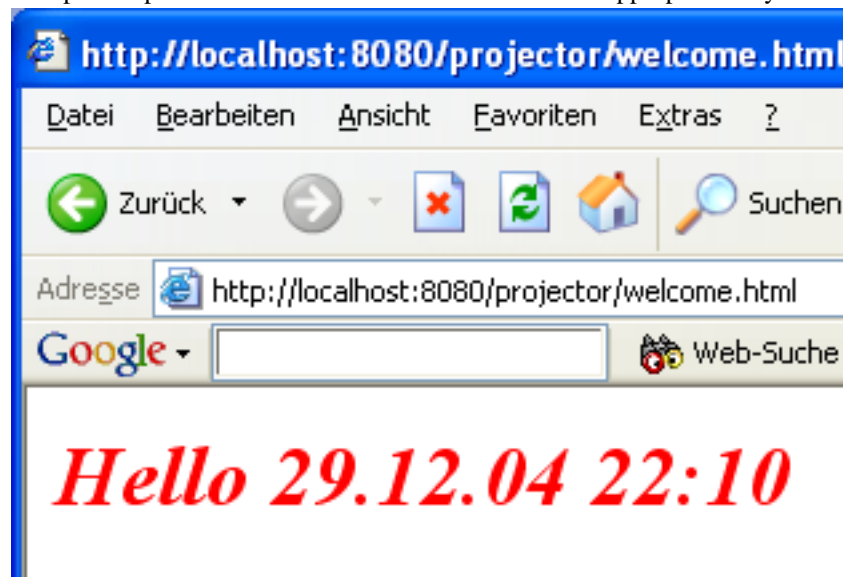
State	Name	Description	Content-Type	Presentable
OK	Output	The string representation of the formatted date	<code>text/plain</code>	<code>true</code>

The table above shows us exactly what parameters this processor accepts and what results we can expect. We can see that exactly one parameter is required, the other parameters are optional. So this processor is suited for calling it via the `<value>` element as you can call processors that require just one or zero parameters.

Change the process definition that it uses the date formatter instead of the current date delivered by the system processor:

```
<step id="compose" processor="hello">
  <load parameter="username">
    <value processor="dateFormatter" result="output">
      <date>1104354657683</date>
    </value>
  </load>
  <load parameter="style">
    <string>color:red;font:italic;</string>
  </load>
  ...
```

As you can see in this example the child element inside the `<value>` element contains the value that will be passed to the processor as input parameter. We pass the fixed date value that we already know from our previous examples. If we now take a look at the rendered page we can see that the date is formatted in the way that we could have expected: The time and date part of the format are in short form as these are the default values of the optional parameters and the date is formatted in the appropriate way for the system's default locale:



As the output of the date formatter is of content type plain/text we now can restrict the content type for our variable in our template again.

4.3. Nested dynamic values

It is even possible to nest dynamic values. In our examples this makes sense as we can combine the both processors that we are now familiar with and replace the fixed date with the current date that the system processor offers:

```
...
<step id="compose" processor="hello">
  <load parameter="username">
    <value processor="dateFormatter" result="output">
      <value processor="system" result="currentDate"/>
    </value>
  </load>
  ...
```

If we upload this and reload the page, we will see the current date in a much more beautiful representation than

we know it from our past examples!

Note

Try to avoid nesting dynamic values deeply as this makes your process definition harder to understand.

If you take a sharp look at the both usages of the value tag that we know right now, it might come into your mind to combine these. It is possible to load the result of a processor that gets feed with a stored value as input. As an example we want to use a parameter that is encoded into the URL as the date to format. This can be done either by nesting dynamic values...

```
<step id="compose" processor="hello">
  <load parameter="username">
    <value processor="dateFormatter" result="output">
      <value store="request-parameter" key="date"/>
    </value>
  </load>
...

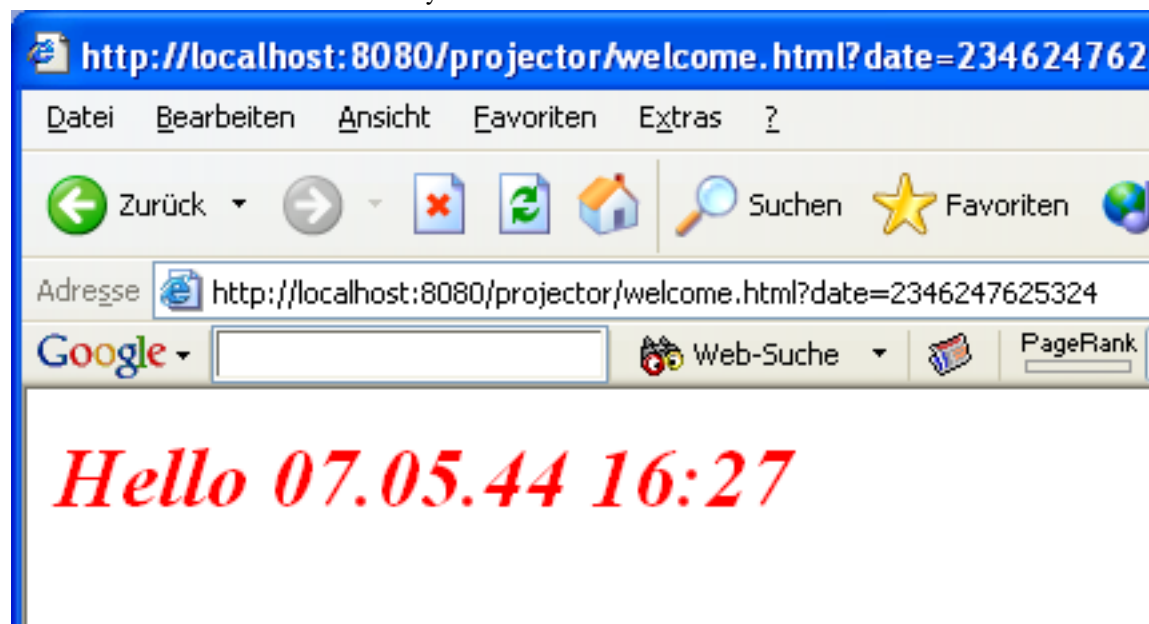
```

...or by using a more compact syntax:

```
...
<step id="compose" processor="hello">
  <load parameter="username">
    <value processor="dateFormatter" result="output"
      store="request-parameter" key="date"/>
  </value>
</load>
...

```

In both cases the result looks identically:



4.4. Saving processor results

We have learned how to load parameters with fixed and dynamic values and we know from the process definition that results can be stored to the available stores.

To complete our knowledge about storing values we should have to look again at the part of the step definition where the processor result gets saved:

...

```
<save result="output" store="output" key="welcomePage" presentable="true"/>
...
```

The result entry with the key `output` that is provided by the `TemplateRender` is stored to the output store using the key `welcomePage`. This is the key that enclosing processes can use to get the generated page in the same way as we have retrieved the formatted date from the `DateFormatter` by using the key `output`. If we would delete this line or comment it out, the outer world would not be able to retrieve the generated page. Just do it to see what happens:

```
...
<!-- Disabling the following line by using xml comment
<save result="output" store="output" key="welcomePage" presentable="true"/>
-->
...
```

If we upload this and reload the page we will see nothing. The page will be empty as Projector could not find any presentable result of the invoked processor. Instead of the rendered page we will see an entry in the Projector log file that indicates this failure:

```
> SEVERE: No presentable value found!
```

We don't bother and want to start another experiment that will show us the power and simplicity of Projector by changing the step definition in the following way:

```
...
<save result="output" store="output" key="welcomePage" presentable="true"/>
<save result="output" store="repository" key="/files/welcomePage.html" presentable="true"/>
...
```

When you reload the web page you'll see our beloved page again, as Projector is finding a presentable result. Now have a look at the WebDAV repository by opening the appropriate Webfolder or any other client of your choice. If you browse into the `files` directory you'll see that Projector has saved the generated page to this folder:



In this example we have seen that it is possible to define multiple save directives to a single processor result. It would for sure be possible to store the generated page at different locations of the repository if needed.

The ability to store a processor result to the repository is very useful as it enables you to easily create some dynamic documents that are evaluated and stored into the repository afterwards. By using this trick it is easy to create applications like weblog or guestbook entries or any other structured content that needs to be generated from user input.

4.5. Timeout

You can define a timeout for stored values. After the given timeout the stored content gets deleted or invalid. Add the following line to our example to find out how this works:

```
...
<save result="output" store="output" key="welcomePage" presentable="true"/>
<save result="output" store="repository" key="/files/welcomePage.html" presentable="true"/>
<save result="output" store="repository" key="/files/illPage.html" presentable="true" timeout="30000" />
...
```

After uploading these changes, reload the page and have a look at the repository by opening it again in the Webfolder. You will see two pages that have been stored as expected, one with the name `welcomePage.html` that we already know and a new one with the name `illPage.html`.

If you reload this Webfolder view after half a minute by either choosing this entry from the menu or pressing F5, you will see that the `illPage.html` disappeared. As we have defined a timeout of 30000 milliseconds, this is exactly what we wanted.

The timeout is very useful if you are using the cookie store that we have not used right now. If you store content in the cookie store, a cookie is automatically created on the client machine. This cookie will be deleted from the client computer when its time that we have defined using the timeout attribute is up. When using the cookie store the standard cookie technology is used to realize the timeout mechanism, otherwise Projector handles the timeout itself.

We will see the cookie store in action in later examples.

4.6. Chaining processors

Up to now our process definitions was made up of a single step. We have seen that a single step can perform fairly complex actions. We will now introduce the ability to connect multiple steps.

Let's expand the example by adding a second step:

```
<?xml version="1.0" encoding="UTF-8" ?>
<process first-step="formatDate" > ❶
  <description>
    <output>
      <state>ok</state>
      <result name="welcomePage"
        description="composedPage"
        content-type="text/html"
        presentable="true" />
    </output>
  </description>

  <step id="formatDate" processor="dateFormatter">
    <load parameter="date">
      <value processor="system" result="currentDate" />
    </load>
    <load parameter="timeFormat">
      <string>medium</string>
    </load>
    <save result="output" key="currentDate" /> ❷
      <route state="ok" step="compose" /> ❸
    </step>

    <step id="compose" processor="hello">
      <load parameter="currentDate">
        <value key="currentDate" /> ❹
      </load>
      <load parameter="style">
        <string>color:red;font:italic;</string>
      </load>
      <save result="output" store="output"
        key="welcomePage" presentable="true" />
      <route state="ok" return="ok" />
    </step>
  </process>
```

- ❶ Optional attribute to set the starting step
- ❷ If you don't specify the store, the scope of the saved value is the current running process. Saved values with no store specified can be loaded by any other step in this process definition that is called in the same request. This is the easiest way to transfer data from one step to the following steps
- ❸ The result state `ok` of the called processor is routed to the step with the id `compose`
- ❹ If you don't specify a store when loading a dynamic value, the scope is set to the current running process. This is the way to load values that have been stored by previous steps with no store specified.

We have extended our example by introducing a second step that will be performed before the step that we al-

ready know is executed. When a process gets launched, the first step that is defined in the process definition gets launched. If you want the process to start with a different step, you can explicitly specify the first step to start with. In the example you can this attribute has been used to show the syntax even if the desired first step is the first in our definition and would because of that fact have been the first one anyway.

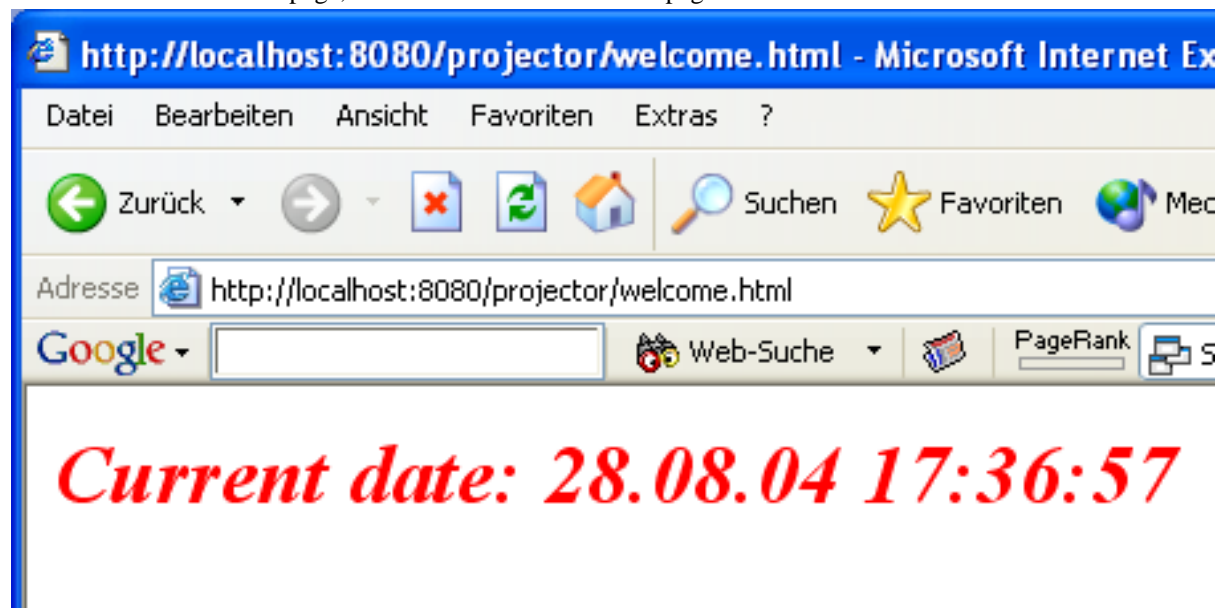
The first step is associated with the `DateFormatter` that has been introduced in the previous examples. As we are no more using this processor implicitly by assigning the result directly to a dynamic value as we did in the previous example, we can now adjust the output format that will be generated by loading all parameters that we want to use. In this example we set the time format to medium. This has the effect, that in opposite to the default short time format the seconds of the current time are also displayed.

The result of the `DateFormatter` gets save to the temporary process store that is available to pass parameters from one step to the following ones.

As we define a routing for the only result state that the `DateFormatter` provides, the process will continue with the compose step. This step is nearly identically with the previous example, but we replaced the `username` parameter with a parameter called `currentTime` as this more reflects the content of the variable. This parameter gets loaded with the value that was stored by the first step. Change our template so that it matches the newly defined parameter name:

```
<html>
  <body>
    <h1 <?style style="<%style%>"?>>Current date: <%currentDate;required;text/plain%></h1>
  </body>
</html>
```

If we now reload the web page, we will see a more attractive page than ever:



You can see that the time now includes the seconds in contrast to the previous example screenshot. It's now very easy for us to change the date or time format as we can load each parameter of the `DateFormatter`. In our previous version we could only specify the date parameter as it is only allowed to use processors with exactly one or zero required parameters to be loaded into dynamic values directly.

4.7. Routing

[TODO]

4.8. Putting it all together

Now we want to put all the things together and extend our example to show undreamed-of possibilities. We want to show the user not only the current time, but also the time when he last visited this page.

To do so we first have to extend our template so that it will show the time of the last visit when available:

```
<html>
  <body>
    <h1 <?style style="<%style%>"?>>Current date: <%currentDate;required;text/plain%></h1>
    <?lastVisit <h1 <?style style="<%style%>"?>>Last visit: <%lastVisit%></h1?>
  </body>
</html>
```

Upload and check if the page looks exactly as before. Projector should not throw any exception as we used the newly introduced variable `lastVisit` as a condition and as such it gets optional. We remember that if it is not set, the marked block gets skipped.

Now we want to fill this variable with an appropriate value. This can be achieved by using the `cookie` store. We want to save the current time to a cookie when the user calls the page. This saved time will be displayed in the subsequent page calls. As a cookie is stored on the client machine on a per user basis, this value can be displayed to each individual user.

```
<?xml version="1.0" encoding="UTF-8" ?>
<process first-step="formatDate">
  <description>
    <output>
      <state>ok</state>
      <result name="welcomePage"
        description="composedPage"
        content-type="text/html"
        presentable="true" />
    </output>
  </description>

  <step id="formatDate" processor="dateFormatter">
    <load parameter="date">
      <value processor="system" result="currentDate" />
    </load>
    <load parameter="timeFormat">
      <string>medium</string>
    </load>
    <save result="output" key="currentDate" />
    <route state="ok" step="compose" />
  </step>

  <step id="compose" processor="hello">
    <load parameter="currentDate">
      <value key="currentDate" />
    </load>
    <load parameter="lastVisit"> ❶
      <value store="cookie" key="date" />
    </load>
    <load parameter="style">
      <string>color:red;font:italic;</string>
    </load>
    <save result="output" store="output"
      key="welcomePage" presentable="true" />
    <route state="ok" step="updateCookie" /> ❷
  </step>

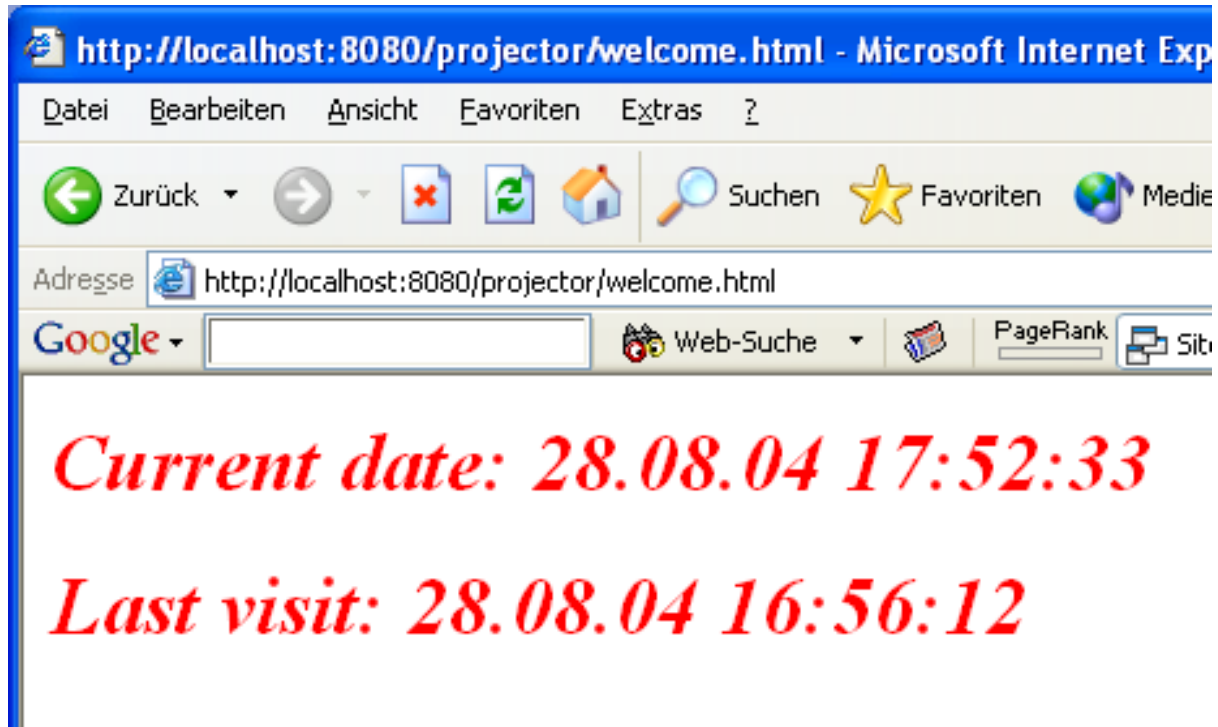
  <step id="updateCookie" processor="echo">
    <load parameter="input"><value key="currentDate" /></load>
    <save result="output" store="cookie" key="date" />
    <route state="ok" return="ok" />
  </step>
</process>
```

- ❶ The parameter `lastVisit` that has been implicitly defined by changing the template will now be loaded with the content of the cookie with the name `date`
- ❷ When the page has been composed successfully the processor will continue with the step `updateCookie`

We've added a new step to store the current date in the cookie store. This is done by using the `Echo` processor. This processor is really simple but also useful: It simply echoes the value loaded to the parameter `input` to the result with the key `output`. We can use this to store the rendered date to the cookie store using the key `date`. This key is used in the adopted `compose` step to access the previously stored value.

If you reload the page, you will see no difference at first sight as the cookie just gets stored and is not yet

present. But if you reload the page, you will see not only the current time, but also the time of your last visit of this page:



You can also store result to other stores, but with different effects as the scope of the stores differ. We can modify the template to show some other scopes:

```
<html>
<body>
  <h1 <?style style="<%style%>"?>>Current date: <%currentDate;required;text/plain%></h1>
  <?yourLastVisit <h1 <?style style="<%style%>"?>>
    Your last visit: <%yourLastVisit%></h1>
  ?>
  <?anybodysLastVisit <h1 <?style style="<%style%>"?>>
    Anybodys last visit: <%anybodysLastVisit%></h1>
  ?>
  <?lastVisitSinceRestart <h1 <?style style="<%style%>"?>>
    Last visit since restart of Projector: <%lastVisitSinceRestart%></h1>
  ?>
</body>
</html>
```

Now update the process definition that all these new parameters get loaded with the appropriate values:

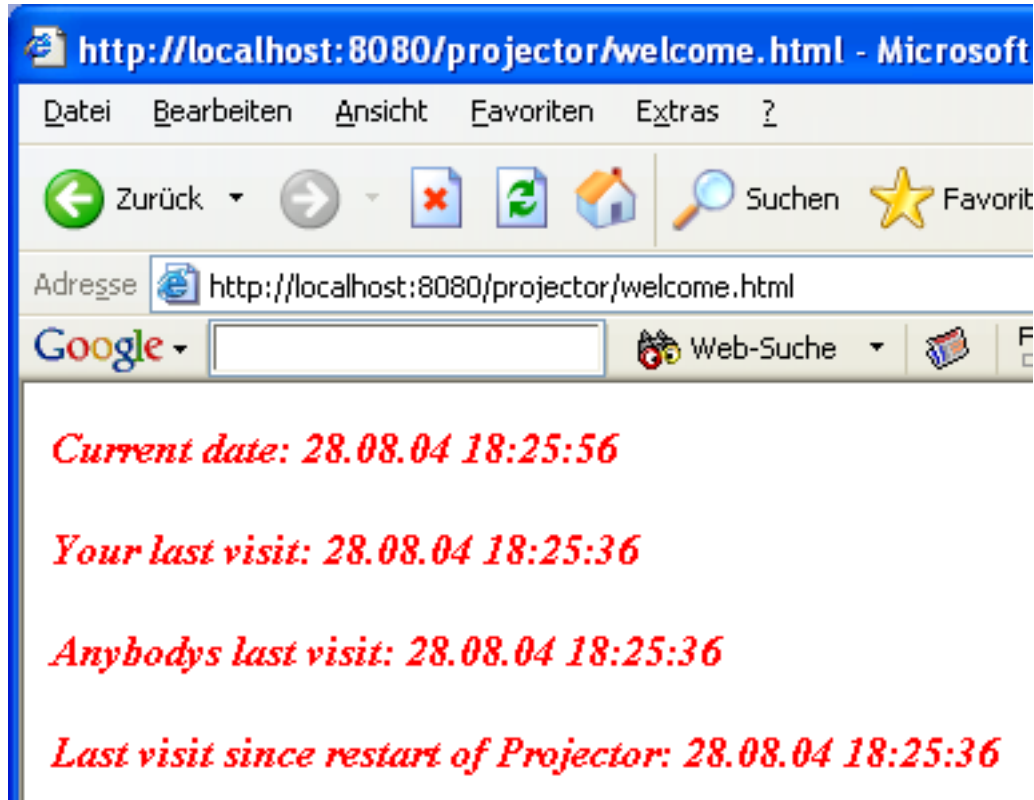
```
...
<step id="compose" processor="hello">
  <load parameter="currentDate">
    <value key="currentDate"/>
  </load>
  <load parameter="yourLastVisit">
    <value store="cookie" key="date"/>
  </load>
  <load parameter="anybodysLastVisit">
    <value store="repository" key="/files/lastVisit.txt"/>
  </load>
  <load parameter="lastVisitSinceRestart">
    <value store="cache" key="lastVisit"/>
  </load>
  <load parameter="style">
    <string>color:red;font:italic;font-size:16px;</string>
  </load>
  <save result="output" store="output"
    key="welcomePage" presentable="true"/>

  <route state="ok" step="updateCookie" />
</step>
<step id="updateCookie" processor="echo">
```

```

<load parameter="input"><value key="currentDate"/></load>
<save result="output" store="cookie" key="date"/>
<save result="output" store="repository" key="/files/lastVisit.txt"/>
<save result="output" store="cache" key="lastVisit"/>
<route state="ok" return="ok" />
</step>
...

```



4.9. Process description

We now want to have a more detailed view at the description element of our process definition that we have ignored so far.

This description contains the contract that describes what parameters our process expects, which states can occur and which results it provides. We have seen the portrait of the `DateFormatter` in the previous section and we could use the description of our process to generate a similar table.

The current version of our process defines just one return state and a single result entry that has the key `welcomePage`.

```

<description>
  <output>
    <state>ok</state> ❶
    <result name="welcomePage" description="composedPage"
            content-type="text/html" presentable="true" /> ❷
  </output>
</description>

```

- ❶ You have to define one or more result *states*. These result states can be set by routing to a return value. If you try to return a value that is not defined here, Projector will throw an exception. More result states can simply be defined by adding more state elements as child of the `output` element
- ❷ The result description provides details about each single result entry that is provided by this process

5. Synchronous flows

6. Asynchronous flows

7. Nested flows

8. The scheduler

9. Bookmarks

Chapter 8. Exception handling

Chapter 9. Internationalization

Chapter 10. Tables and Trees

1. Table layout

2. Advances tables

3. Table pager

4. Tree layout

5. Sitemap generation

Chapter 11. Processing SQL

Chapter 12. Processing XML

Chapter 13. Form handling

- 1. Generating a standard form**
- 2. Customizing a form**
- 3. Designing form based workflows**
- 4. Creating wizard like forms**

Chapter 14. Transactions

Chapter 15. Extending Projector

1. Custom Processors

You will quickly learn how to implement your own processors, but it might take some time to do it right.

You must follow the rules that apply to stateless session beans, when implementing your own processor:

The process method must be thread-safe as it gets called from many concurrent threads.

We will start with an example of a processor that can be used as a very simple calculator. Each line of code of our example is explained in detail in the following sections so that you will understand the basic concepts of Projector.