IT Mill Toolkit

Reference Manual

IT Mill Toolkit: Reference Manual

IT Mill Toolkit 5.3.0

Published: 2009-03-03

Abstract

IT Mill Toolkit enables developers to build high quality browser user interfaces with Java on server. It provides a library of ready to use high-quality user interface components and defines a clean framework for creating your own components. The focus is on ease of use, re-usability, extensibility and meeting the requirements of large enterprise applications. The toolkit has been used in production since 2001 and it is shown to be suitable for building demanding business applications.

Table of Contents

List of Figures

List of Tables

List of Examples

Preface

1. About This Book

This book is intended for software developers who use IT Mill Toolkit to develop web applications.

1.1. Prerequisites

This book assumes that you have some experience with programming in Java. If not, Java is easy to learn if you have experience with other object oriented languages, such as C++. You may have used some desktop-oriented user interface toolkits for Java, such as AWT, Swing, or SWT. Or for C++, a toolkit such as Qt. Such knowledge is useful for understanding the scope of the Toolkit, but not necessary. Regarding the web, it is good if you know the basics of HTML and CSS, so that you can develop basic presentation themes for the application. Knowledge of Google Web Toolkit (GWT), JavaScript, and AJAX is needed only if you develop special custom UI components.

1.2. Organization of This Book

The IT Mill Toolkit Reference Manual is divided into two parts: Developer's Guide and API Reference.

The Developer's guide first gives an introduction to what IT Mill Toolkit is and how you use it to develop web applications. It then proceeds to architecture, particular components and features of the Toolkit, to special topics, and finally to special design patterns for the Toolkit.

API Reference gives the full documentation for particular classes and interfaces, and their methods. The API documentation is also available in JavaDoc format, as a HTML page with frames.

1.3. Supplementary Material

Demo Application

The installation package of IT Mill Toolkit includes a demo application that you can simply run and use with a web browser. You can view the source code of the individual demo applications from the main menu. The demo application includes a feature browser, which offers demonstration of most user interface components in IT Mill Toolkit.

You can find the demo application also online at http://toolkit.itmill.com/demo/.

1.4. Online Resources

Developer's Site

The IT Mill Toolkit Developer's Site [http://dev.itmill.com/] provides various online resources, such as a development wiki, ticket (bugs and other issues) management system, source repository browsing, timeline, development milestones, and so on.

- Checkout IT Mill Toolkit source code from the Subversion repository
- Read technical articles and get more examples
- Report bugs
- Make requests for enhancements

- Follow the development of the Toolkit
- Collaborate!

The wiki provides instructions for developers, especially for those who wish to checkout and compile IT Mill Toolkit itself from the source repository. The technical wiki articles deal with integration of IT Mill Toolkit applications with various systems, such as JSP, Maven, Spring, Hibernate, and portals. The wiki also provides answers to Frequently Asked Questions.

Online Documentation

You can read this book online at http://www.itmill.com/documentation/. You can find technical articles and answers to Frequently Asked Questions also from the Developer's Site [http://dev.itmill.com/].

1.5. Support

Support Forum

An open support forum for developers is available at http://forum.itmill.com/. Please use the forum to discuss any problems you might encounter, wishes about features, and so on.

- Share your ideas and code
- Ask and you get answers
- Search answers from archived discussions

Bug Report Form

If you have found an issue with IT Mill Toolkit, demo applications or documentation, please report it to us by filing a ticket in the IT Mill Toolkit developer's site at http://dev.itmill.com/. You may want to check the existing tickets before filing a new ticket. You can make a ticket to make a request for a new feature as well, or to suggest modifications to an existing feature.

Commercial Support

IT Mill offers full commercial support and training services for the IT Mill Toolkit products. Please contact our sales at http://www.itmill.com/itmill_contact_sales.htm for details.

Part I. Developer's Guide

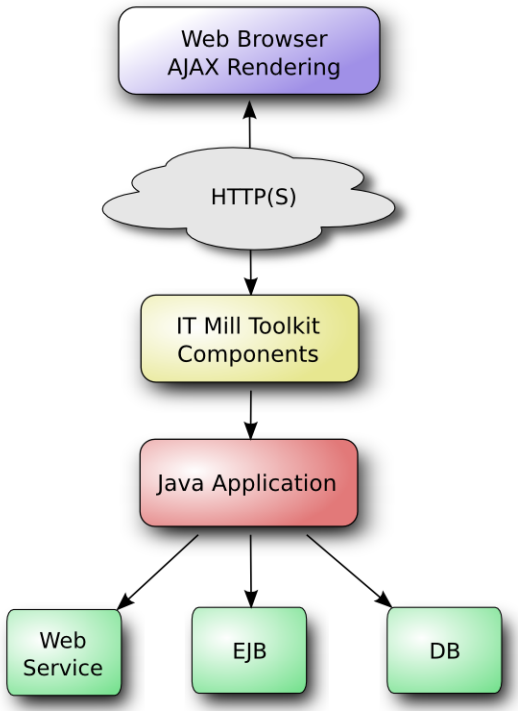
Chapter 1. Introduction

This chapter provides an introduction to software development with IT Mill Toolkit, including installation of the Toolkit, the Eclipse development environment, and any other necessary or useful utilities. We look into the design philosophy behind the Toolkit and at the changes in IT Mill Toolkit Release 5.

1.1. Overview

IT Mill Toolkit is essentially a Java library that is designed to make creation and maintenance of high quality web-based user interfaces easy. The key idea is that the Toolkit allows you to forget the web and lets you program user interfaces much like you would program any Java desktop application with conventional toolkits such as AWT, Swing, or SWT. But easier.

Figure 1.1. IT Mill Toolkit General Architecture



While traditional web programming is a fun way to spend your time learning new web technologies and debugging them, you probably want to be productive and concentrate on the application logic. The library takes care of user interface rendering in the browser and AJAX communications between the browser and the server. With Toolkit's approach, you do not need to learn and debug browser technologies, such as HTML or JavaScript.

IT Mill Toolkit makes the best use of AJAX (Asynchronous JavaScript and XML) techniques that enable the creation of web applications as responsive and interactive as desktop applications. While conventional JavaScript-enabled HTML pages can receive new content only with page updates, AJAX-enabled pages can ask the server for updated content using the asynchronous `XMLHttpRequest` JavaScript request. User interaction with UI components is communicated to the server. The IT Mill Toolkit framework interprets the events and communicates them to your application logic. Any user interface feedback is rendered in the response message to the AJAX request.

Hidden well under the hood, IT Mill Toolkit uses GWT, the Google Web Toolkit, for rendering the user interface in the browser. GWT programs are written in Java, but compiled into JavaScript, thus freeing the developer from learning JavaScript and other browser technologies. GWT is ideal for implementing advanced user interface components (or widgets in GWT terminology) and interaction logic in the browser, while IT Mill Toolkit handles the actual application logic in the server. IT Mill Toolkit is designed to be extensible, and you can indeed use any 3rd-party GWT components easily, in addition to the component repertoire offered in IT Mill Toolkit. The use of GWT also means that all the code you need to write is pure Java.

Because HTML and other browser technologies are invisible to the application logic, you can think of the web browser as only a thin client platform. A thin client displays the user interface and communicates user events to the server at a low level. The control logic of the user interface runs on a Java-based web server, together with your business logic. By contrast, a normal client-server architecture with a dedicated client application would include a lot of application specific communications between the client and the server. Essentially removing the user interface tier from the application architecture makes our approach a very effective one.

The Toolkit library defines a clear separation between user interface presentation and logic and allows you to develop them separately. Our approach to this is *themes*, which dictate the visual appearance of applications. Themes control the appearance of the user interfaces using CSS and (optional) HTML page templates. As the Toolkit provides excellent default themes, you do not usually need to make much customization, but you can if you need to. For more about themes, see Chapter 6, *Themes*.

We hope that the description above is enough about architecture for now. You can read more about it later in Chapter 2, *Architecture*. Let us next look at some of the core ideas behind IT Mill Toolkit.

1.1.1. Goals and Philosophy

Simply put, as its name implies, the Toolkit's ambition is to be the best possible tool when it comes to creating web user interfaces for business applications. It is easy to adopt, as it is designed to support both entry-level and advanced programmers, as well as usability experts and graphical designers.

When designing the Toolkit, we have followed the philosophy inscribed in the following rules.

Right tool for the right purpose

Because our goals are high, the focus must be clear. This toolkit is designed for creating web applications. It is not designed for creating websites or advertisements demos. You should use JSP/JSF and Flash for such purposes.

Simplicity and maintainability

We have chosen to emphasize robustness, simplicity, and maintainability over the possibility to "draw" user interfaces with visual design tools. For serious business applications, you have to program the user interfaces anyway and the visual design tools just get in the way.

XML is not designed for programming

The Web is inherently document centered and very much bound to the declarative presentation of user interfaces. Toolkit's framework frees the programmer from these limitations. It is far more natural to create user interfaces by programming them than by defining them in the various XML dialects.

Tools should not limit your work

There should not be any limits on what you can do with the framework: if for some reason the user interface components do not support what you need to achieve, it must be easy to add new ones to your application.

When you need to create new components, the role of the framework is critical: it makes it easy to create re-usable components that are easy to maintain.

1.1.2. Background

The library was not written overnight. After working with web user interfaces since the beginning of the Web, a group of developers got together in 2000 to form IT Mill. The team had a desire to develop a new programming paradigm that would support the creation of real user interfaces for real applications using a real programming language.

The library was originally called Millstone Library. The first version was used in a large production application that IT Mill designed and implemented for an international pharmaceutical company. IT Mill made the application already in the year 2001 and it is still in use. Since then, the company has produced dozens of large business applications with the library and it has proven its ability to solve hard problems easily.

The next generation of the library, IT Mill Toolkit Release 4, was released in 2006. It introduced an entirely new AJAX-based presentation engine. This allowed the development of AJAX applications without the need to worry about communications between the client and the server.

The latest generation, IT Mill Toolkit Release 5, takes a significant step further into AJAX. The client-side rendering of the user interface has been rewritten using GWT, the Google Web Toolkit. This allows the use of Java for developing all aspects of the framework. It also allows easy integration of existing GWT components with IT Mill Toolkit.

IT Mill Toolkit Release 5 was released under the Apache open source license and the development of the toolkit .

1.1.3. Changes in IT Mill Toolkit Release 5

Release 5 of IT Mill Toolkit introduces a number of changes in the API, the client-side customization layer, and themes. See the Release Notes in the installation package of IT Mill Toolkit for a more detailed listing of changes.

We have decided to introduce some important API improvements in Release 5. Many of the user interface components in Release 4 and before were available as styles for a basic set of components. For example, the **Select** class allowed selection of items from a list. Normally, it would show as a dropdown list, but setting `setStyle("optiongroup")` would change it to a radio button group. In Release 5, we have obsoleted the `setStyle()` method and provided distinct classes for such variations. For example, we now have **OptionGroup** that inherits the **AbstractSelect** component. In a similar fashion, the **Button** component had a *`switchMode`* attribute, set with `setSwitchMode()`, that would turn the button into a check box. Release 5 introduces a separate **CheckBox** component. The `setStyle()` method actually had a dual function, as it was also used to set the HTML element *`class`* attribute for the components to allow styling in CSS. This functionality has been changed to `addStyle()` and `removeStyle()` methods.

The **OrderedLayout** is replaced (since the first stable version 5.3.0) with specific **VerticalLayout** and **HorizontalLayout** classes.

Release 5 introduces *expansion ratio* for applicable layout components. It allows you to one or more components as expanding and set their relative expansion sizes. The components will stretch to expand the layout to maximum size inside its container. The release also introduces a number of new user interface components: **SplitPanel**, **Slider**, **Notification** to display a popup notification window, and **RichTextEditor** to allow editing formatted text.

The Client-Side Engine of IT Mill Toolkit has been entirely rewritten with Google Web Toolkit. This does not, by itself, cause any changes in the API of IT Mill Toolkit, because GWT is a browser technology that is well hidden behind the IT Mill Toolkit API. Transition from JavaScript to GWT makes the development and integration of custom components and customization of existing components much easier than before. It does, however, require reimplementation of any existing custom client-side code with GWT. See Chapter 2, *Architecture* for more information on the impact of GWT on the architecture and Chapter 8, *Developing Custom Components* for details regarding creation or integration of custom client-side components with GWT.

IT Mill Toolkit Release 5 introduces an entirely new architecture for themes. Themes control the appearance of web applications with CSS and can include images, HTML templates for custom layouts, and other related resources. The old themeing architecture in Release 4 required use of some JavaScript even in the simplest themes, and definition of a theme XML descriptor. In Release 5, you simply include the CSS file for the theme and any necessary graphics and HTML templates for custom layouts. For more details on the revised theme architecture, see Chapter 6, *Themes*. Old CSS files are not compatible with Release 5, as the HTML class style names of components have changed. As GWT implements many components with somewhat different HTML elements than what IT Mill Toolkit Release 4 used, styles may need to be updated also in that respect.

1.2. Example Application Walkthrough

Let us follow the long tradition of first saying "Hello World!" when learning a new programming environment. After that, we can go through a more detailed example that implements the model-view-controller architecture. The two examples given are really simple, but this is mostly because IT Mill Toolkit is designed to make things simple.

1.2.1. Hello World!

Example 1.1. HelloWorld.java

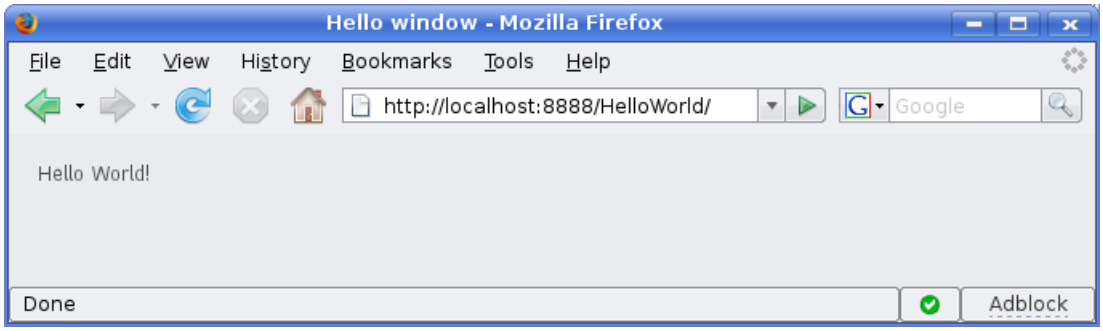
```
import com.itmill.toolkit.ui.*;

public class HelloWorld extends com.itmill.toolkit.Application {

    public void init() {
        Window main = new Window("Hello window");
        setMainWindow(main);
        main.addComponent(new Label("Hello World!"));
    }
}
```

The first thing to note is that the example application extends **com.itmill.toolkit.Application** class. The **Application** class is used as the base class for all user applications. Instances of the **Application** are essentially user sessions, and one is created for each user using the application. In the context of our HelloWorld application, it is sufficient to know that the application is started when the user first accesses it and at that time `init` method is invoked.

Initialization of the HelloWorld application first creates a new window and sets "Hello window" as its caption. The window is then set as the main window of the application. This means that when a user launches the application, the contents of the main window are shown to the user. The caption is shown as the title of the (browser) window.

A new user interface component of class `com.itmill.toolkit.ui.Label` is created. The label is set to draw the text "Hello World!". Finally, the label is added to the main window. And here we are, when the application is started, it draws the text "Hello World!" to the browser window.

The following screenshot shows what the "Hello World!" program will look like in a web browser.

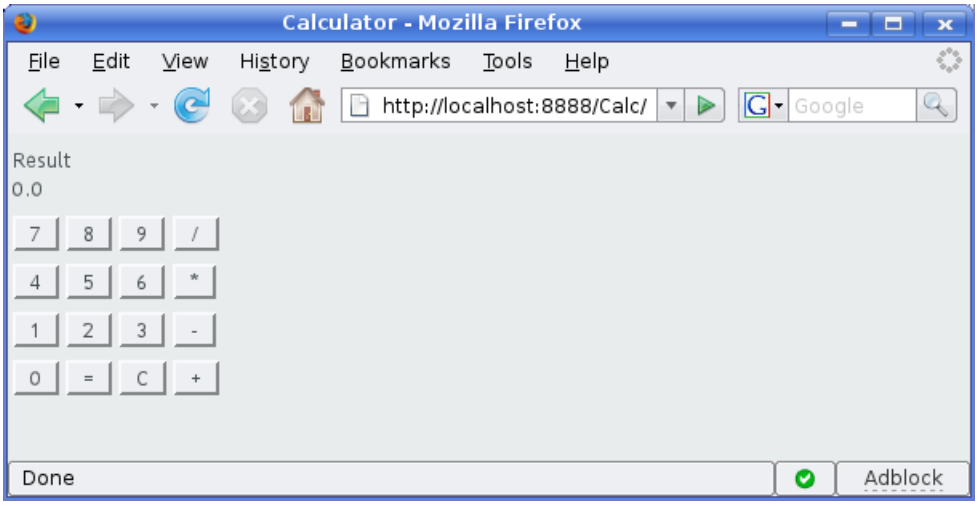


Example 1.1, “HelloWorld.java” implements our "Hello World!" program. Before going into details, we should note that this example source code is complete and does not need any additional declaratively defined template files to be run. To run the program, you can just add it to your web application explained in Section 3.7, “Application Environment”.

1.2.2. Calculator

One of the most amazing, or dare we say annoying, things about modern computers is that while they cost hundreds if not thousands of currency units, they are less useful for calculations than an abacus. Recent operating systems have taken a significant step forward by including a trivial calculator program with almost every computer. Our little version below takes some 50 lines of code.

Let us first see what it should look like.



We all have to admit it, the calculator is not very beautiful with the gray buttons and backgrounds and all. In Chapter 6, *Themes*, we will show you how to excel in beauty, but for now we keep it dull and gray.

Let us look how it was done. Example 1.2, “Calc.java” implements a simple calculator that can do add, subtract, divide and multiply operations. Of course, this is not too useful to do with a server-based application, but it demonstrates event management, layout management, and a few other features of IT Mill Toolkit nicely.

Example 1.2. Calc.java

```
import com.itmill.toolkit.ui.*;

public class Calc extends com.itmill.toolkit.Application implements
  Button.ClickListener {

 private Label display = null;
 private double stored = 0.0;
 private double current = 0.0;
 private String operation = "C";
 private static String[] captions = // Captions for the buttons
 { "7", "8", "9", "/", "4", "5", "6", "*", "1", "2", "3", "-", "0", "=",
   "C", "+" };

 public void init() {

  // Create a new layout for the components used by the calculator
  GridLayout layout = new GridLayout(4, 5);

  // Create a new label component for displaying the result
  display = new Label(Double.toString(current));
  display.setCaption("Result");

  // Place the label to the top of the previously created grid.
  layout.addComponent(display, 0, 0, 3, 0);

  // Create the buttons and place them in the grid
  for (int i = 0; i < captions.length; i++) {
   Button button = new Button(captions[i], this);
   layout.addComponent(button);
  }

  // Create the main window with a caption and add it to the application.
  addWindow(new Window("Calculator", layout));

 }

 public void buttonClick(Button.ClickEvent event) {

  try {
   // Number button pressed
   current = current * 10
     + Double.parseDouble(event.getButton().getCaption());
   display.setValue(Double.toString(current));
  } catch (java.lang.NumberFormatException e) {

   // Operation button pressed
   if (operation.equals("+"))
    stored += current;
   if (operation.equals("-"))
    stored -= current;
   if (operation.equals("*"))
    stored *= current;
   if (operation.equals("/"))
    stored /= current;
   if (operation.equals("C"))
    stored = current;
   if (event.getButton().getCaption().equals("C"))
    stored = 0.0;
   operation = event.getButton().getCaption();
   current = 0.0;
   display.setValue(Double.toString(stored));
  }
 }
}
```

Let us next look at the architecture of this marvellous piece of modern technology by considering it from the perspective of the model-view-controller (MVC) design pattern. MVC is central to any decent user interface design, so we will revisit it frequently later in this guide. Our first example is actually not a very good example of MVC, but that is mainly to keep the size of the example below 50 lines. The main benefit is learning to think through the design pattern.

Model

The application has an internal state that is simply stored in the member variables of the the `Calculator`:

- `stored` is the last value entered to the calculator before the value that is now displayed.
- `current` is the value that is currently shown on the calculator's display.
- `operation` is the previously selected operation that will be done between `stored` and `current` when entering the current value is finished.

When thinking in the terms of the Model-View-Controller paradigm, the above variables form the model part together with the calculation logic. In most applications, the model should be well separated from the view and controller. IT Mill Toolkit provides good support for completely isolating the model from user interface logic. In this example, further separation would only make the application more complex.

View

As with `HelloWorld`, the application extends **com.itmill.toolkit.Application** and initializes itself in the `init()` method. In this method, you need to initialize the user interface of the application and anything else that needs to be initialized when the application starts.

The user interface is laid out with the **GridLayout** layout management object. It provides an easy way to position components in a tabular layout; you just specify that the layout has four columns and add the components to the layout. In the Calculator example, we first create and add the display component of the calculator as a **Label** and then all the **Button** components in a `for` loop.

Construction of user interface component trees can be done freely in any order the programmer wants to. In the HelloWorld example, we added the **Label** component to the **Window**. To provide an example, we set the layout in the constructor of the **Window** object. This is in essence almost the same: if the layout is not specified for the **Window**, it uses **VerticalLayout** by default and adding components to a **Window** adds them to its layout. Here we replace the default **VerticalLayout** and use **GridLayout** instead. Then we just add the **Window** to the **Application**. As this is the first **Window** in the application, it is automatically recognized as the main window.

Notice that programming the user interface is done on logical level -- nothing is said about colors, fonts, how buttons behave, and such. This is all specified in the *theme* of the application. This separation of appearance is important as it frees the application programmer from the compexity of details related to look and feel. It also makes it possible to maintain the looks as separate themes -- possibly left for the visual designers.

Controller

Programming a "controller" in IT Mill Toolkit is not very different from Swing or any other event-based user interface framework. You just attach event handlers to components and program the application logic in those handlers. The toolkit manages all the complexity of creating event handlers in the browser, their memory management issues, network transport of actions with AJAX, security, etc. Keeping all these "out of the way", the programmer can focus on the application itself.

In the Calculator example, all buttons are assigned to send their events to the application by saying in the **Button** constructor that the **Button** should send the event to `this` and implementing **Button.Clicklistener** interface. All the events in this example are sent to the same listener method, `buttonClick()`. To keep the example simple, the listener decides what to do according to the caption of the button from the received event.

The logic in the listener is fairly simple - if the caption is a number, it is added as the next digit to the `current` variable and the value of `current` is copied to the `display` component. Notice that IT Mill Toolkit handles all sending of UI changes to web browser automatically -- it decides what parts of the screen need to be repainted and updates only them. For other buttons, the application state is updated according to the button pressed and the display component is also updated.

1.3. What's Inside the IT Mill Toolkit Package?

This section gives an overview of the IT Mill Toolkit package and its installation.

1.3.1. Installing

Installing IT Mill Toolkit is very straight-forward:

1. Download the newest IT Mill Toolkit from the download page at http://www.itmill.com/itmill_toolkit_download.htm. Select the proper download package for your operating system: Windows, Linux, or Mac OS X.

2. Uncompress the installation package to a directory using an uncompressor program appropriate for the package type (see below) and your operating system.

   - In Windows, use ZIP uncompressor to install the package to your chosen directory.

     Warning

     At least with Windows XP default unzipper or when using WinRAR to uncompress the installation package, uncompression can result in an error such as "*The system cannot find the file specified.*" This is because the uncompressor is unable to handle long file paths where the total length exceeds 256 characters. This occurs, for example, if you try to uncompress the package under Desktop. You should uncompress the package directly under `C:\` or some other short path.

   - In Linux, use GNU `tar` and BZIP2 uncompression with **tar jxf itmill-toolkit-linux-5.x.x.tar.bz2** command.

   - In Mac OS X, use `tar` and Gzip uncompression with **tar zxf itmill-toolkit-mac-5.x.x.tar.gz** command.

   The files will be, by default, uncompressed under a directory with the name `itmill-toolkit-<operatingsystem>-5.x.x`.

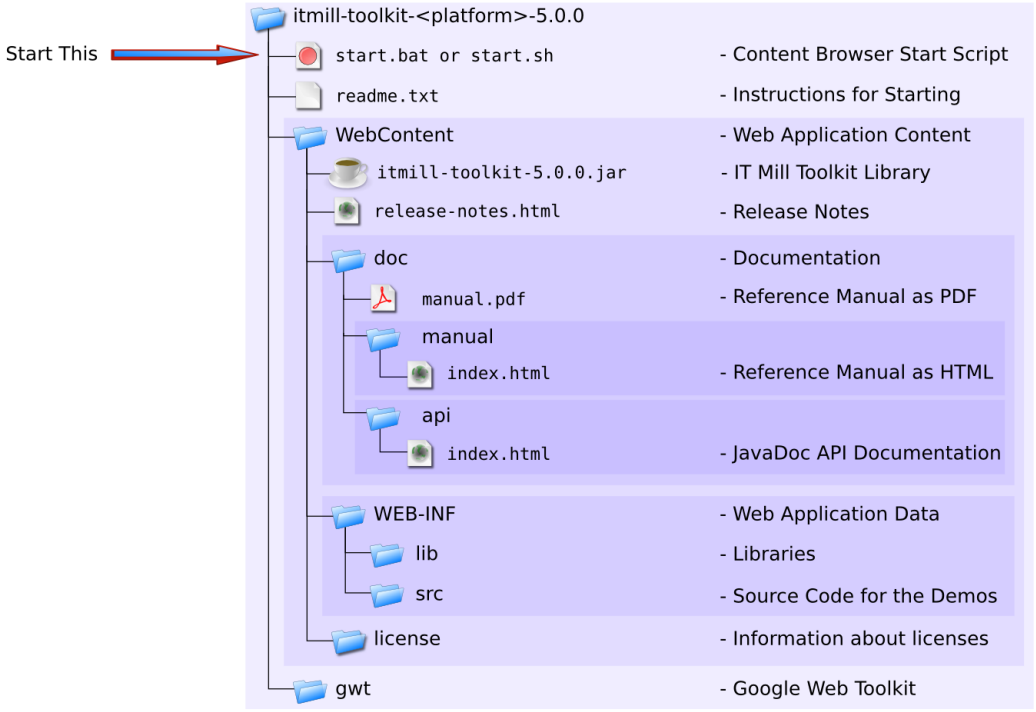
You can rename the installation directory as you wish, especially if you wish to use the installation as a skeleton for your own project, as described in Section 1.5.4, “Using QuickStart as a Project Skeleton” below.

When using IT Mill Toolkit in a project, you will need to copy or import the JAR packages of the library to the web application, and possibly also to your project directory during development, depending on your development environment. This is the case at least with Eclipse. See Section 1.4, “Getting the Development Environment Up and Running” below for details.

1.3.2. Package Contents

At the top level of the installation directory, you can find the `start.bat` (Windows) or `start.sh` (Linux and Mac) script. Execute it from a file manager or command prompt, as instructed in Section 1.3.4, “Running the Demo Applications”. The script launches the IT Mill Toolkit Content Browser web application and a web browser to view its start page.

Figure 1.2. IT Mill Toolkit Package Contents



The `WebContent` is a web application directory that contains the material available from the Content Browser. If you do not wish to or can not run the Content Browser, you can open the `index.html` with a web browser to view the installation package contents and documentation, although the demos will not be usable. The `release-notes.html` contains information about changes in the latest release and the release history. The `license` subdirectory contains copying information (`COPYING`) and licensing guidelines (`licensing-guidelines.html`).

The IT Mill Toolkit Library itself is located at `WebContent/itmill-toolkit-5.x.x.jar`. The JAR package contains, in addition to the compiled files, full source code of the libraries.

The `WebContent/doc` directory contains full documentation for IT Mill Toolkit, including JavaDoc API Reference Documentation and this manual in both HTML and printable PDF format.

The `WebContent/WEB-INF` directory contains source code for the demo applications in the `src` subdirectory and required libraries in the `lib` subdirectory.

The `gwt` folder contains the full Google Web Toolkit installation package, including runtime libraries for the selected operating system platform, full documentation, and examples. You will need GWT if you intend to compile custom client-side widgets for IT Mill Toolkit. The root directory contains also `build-widgetsets.xml`, which is an Ant file for compiling GWT widget sets as described in Section 8.7.4, “Compiling GWT Widget Sets”.

In addition, the installation directory contains project files to allow importing the directory as a project in the Eclipse IDE. See Section 1.5, “QuickStart with Eclipse” for details on how to import the installation directory as a QuickStart project in Eclipse.

1.3.3. Starting the Content Browser

The Content Browser is your best friend when using IT Mill Toolkit. It allows you to browse documentation and example source code and run the demo applications. The demo applications demonstrate most of the core features of IT Mill Toolkit. You can find the demo application also from the IT Mill website, at http://toolkit.itmill.com/demo/.

To start the Content Browser, run the start script in IT Mill Toolkit installation directory as instructed below for your specific platform. It launches a stand-alone web server running on the local host at port 8888, and a web browser at address http://localhost:8888/.

The Content Browser will open the default web browser configured in your system. Please make sure that the browser is compatible with IT Mill Toolkit or otherwise the demo applications may not work properly.

If the Content Browser fails to start, make sure that no other service is using port 8888.

Windows

Run the `start.bat` batch file by double-clicking on the icon.

JRE must be installed

You must have Java Runtime Environment (JRE) installed or the batch file will fail and close immediately.

Starting the web server and the web browser can take a while.

Notice that executing the Content Browser locally may cause a security warning from your firewall software. This is due to the started web service. You have to ignore warnings or temporarily accept connections to port 8888 on your firewall software.

Linux / UNIX

Open a shell window, change to the IT Mill Toolkit installation directory, and run the `start.sh` shell script. You have to run it with the following command:

```
$ sh start.sh
------------------------------------------------
Starting IT Mill Toolkit in Desktop Mode.
Running in http://localhost:8888
------------------------------------------------

2007-12-04 12:44:55.657::INFO:  Logging to STDERR via org.mortbay.log.StdErrLog
2007-12-04 12:44:55.745::INFO:  jetty-6.1.5
2007-12-04 12:45:03.642::INFO:  NO JSP Support for , did not find
org.apache.jasper.servlet.JspServlet
2007-12-04 12:45:03.821::INFO:  Started SelectChannelConnector@0.0.0.0:8888
```

Starting the web server and the web browser can take a while.

Some web browsers for Linux, such as Konqueror, are not well supported, so you may have problems in running the demo applications. Please use Mozilla Firefox or some other compatible browser.

Mac OS X

Double-click on the **Start IT Mill Toolkit** icon.

Starting the web server and the web browser can take a while.

If the start icon fails in your environment for some reason, you can start the Content Browser by following the instructions for Linux/UNIX above: open a shell window, change to the installation directory, and execute **sh start.sh**.

1.3.4. Running the Demo Applications

The Content Browser allows you to run demo applications. The start page features four demos:

Feature Browser	The Feature Browser allows you to view a demonstration of the standard components available in IT Mill Toolkit. Select the example from the tree on the left. The top-right panel will display a list of the examples in the category and shows which examples you have already viewed. The bottom-right panel will display the selected example. You can click on the **Open in sub-window** or **Open in native window** to open the example in a child window or native window, respectively.
Notification	The Notification demo demonstrates the four types of notification boxes: humanized, warning, error, and tray notifications. Select the notification type, enter the caption and message in the text fields, and click **Show notification**.
Reservation Application	The Reservation Application demonstrates use of various components in a semi-real application connected to a local database. Most importantly, it shows how to use a Google Maps view inside an application. *Notice: starting the demo can take several seconds.*
Windowed Demos	Windowed Demos are small examples that run inside child windows, which you can open from the list on left.

Clicking on the **Additional demos** opens a list of other small examples, which you can view. You can click on the **sources** to view the source code of each demo application.

1.4. Getting the Development Environment Up and Running

This section gives a step-by-step guide for setting up a development environment. IT Mill Toolkit supports a wide variety of tools, so you can use any IDE for writing the code, most web browsers for viewing the results, any operating system or processor supported by the Java 1.5 platform, and almost any Java server for deploying the results.

In this example, we use the following toolchain:

- Windows XP [http://www.microsoft.com/windowsxp/]
- Sun Java 2 Standard Edition 6.0 Update 1 [http://java.sun.com/javase/downloads/index_jdk5.jsp]
- Eclipse IDE for Java EE Developers (Europa version) [http://www.eclipse.org/downloads/]

- Apache Tomcat 6.0 (Core) [http://tomcat.apache.org/]
- Firefox 2.0.0.1 [http://www.mozilla.com/]
- Firebug 1.01 [http://www.getfirebug.com/]
- IT Mill Toolkit 5.x.x [http://www.itmill.com/]

The above is a good choice of tools, but you can use almost any tools you are comfortable with.

Figure 1.3. Development Toolchain and Process

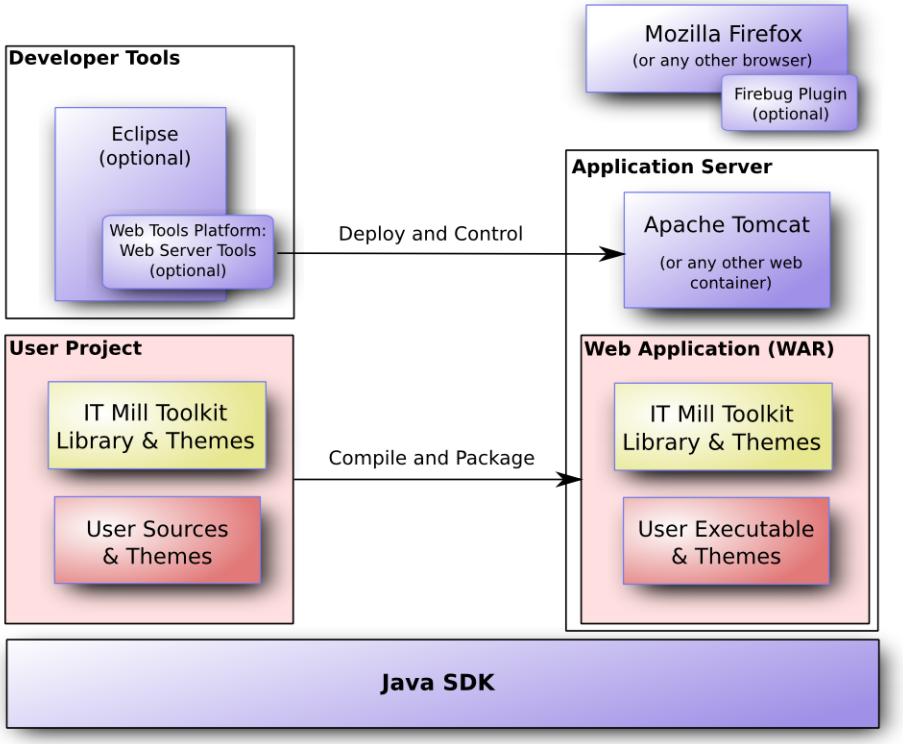


Figure 1.3, “Development Toolchain and Process” above illustrates the development environment and process. You develop your application as an Eclipse project. The project must include, in addition to your source code, the IT Mill Toolkit Library package, and the theme package. It can, optionally, include your project-specific themes. When the project is compiled and packaged as a web application (WAR), the IT Mill components are copied to the package. Web Tools Platform for Eclipse allows easy deployment of web applications and debugging them under Tomcat.

1.4.1. Installing Java SDK

Java SDK is required by the Eclipse IDE during development. You may also need it for some other tasks. IT Mill Toolkit is compatible with Java 1.5 and later editions. (Java 1.4 is no longer supported since version 5.3.0 because of the requirements of GWT 1.5.)

Windows

Setting up the Java in Windows XP is really straightforward.

1. Download the installation packages from:

   - Sun Java 2 Standard Edition 6.0 Update 1 from http://java.sun.com/javase/downloads/index.jsp [http://java.sun.com/javase/downloads/index.jsp]

2. Install the Java SDK by running the installer. The default options are fine.

Linux / UNIX

Download the following package:

- Sun Java 2 Standard Edition 6.0 Update 1 from http://java.sun.com/javase/downloads/index.jsp [http://java.sun.com/javase/downloads/index.jsp]

Decompress Java SDK under a suitable base directory, such as `/opt`. For example, for Java SDK, enter (either as root or with **sudo** in Linux):

```
# cd /opt
# sh (path-to-installation-package)/jdk-6u1-linux-i586.bin
```

and follow the instructions in the installer.

1.4.2. Installing Eclipse IDE

Windows

Setting up the Eclipse IDE in Windows XP is really straightforward.

1. Download the installation package from:

   - Eclipse IDE for Java EE Developers (Europa version) from http://www.eclipse.org/downloads/ [http://www.eclipse.org/downloads/]

2. Decompress the Eclipse IDE package to a suitable directory. You are free to select any directory and to use any ZIP decompressor, but in this example we decompress the ZIP file by just double-clicking it and selecting "Extract all files" task from Windows compressed folder task. In our installation example, we use `C:\dev` as the target directory.

3. Eclipse is now installed in `C:\dev\eclipse` and can be started from there (by clicking eclipse.exe).

Linux / UNIX

You have two basic options for installing Eclipse in Linux and UNIX: you can either install it using the package manager of your operating system or by downloading and installing the packages manually. The *manual installation method is recommended*, because the latest versions of the packages available in a Linux package repository may be incompatible with Eclipse plugins that are not installed using the package management. The versions mentioned above have been tested to work.

Download the following package:

- Eclipse IDE for Java EE Developers from http://www.eclipse.org/downloads/ [http://www.eclipse.org/downloads/]

Decompress the Eclipse package under a suitable base directory. It is important to make sure that there is no old installation in a directory with the same name as installing a new version on top of an old one would probably make Eclipse unusable.

Eclipse should normally be installed as a regular user, as this makes installation of plugins easier. Eclipse also stores some user settings in the installation directory. To install the package, enter:

```
$ tar zxf (path-to-installation-package)/eclipse-jee-europa-fall-linux-gtk.tar.gz
```

This will extract the package to a subdirectory with the name `eclipse`.

You may wish to add the Eclipse installation directory and the `bin` subdirectory in the installation directory of Java SDK to your system or user PATH.

Alternatively, the package management system of your operating system may provide the packages. For example, in Ubuntu Linux, which includes Sun Java SDK and Eclipse in its APT repository, you can install the programs from a package manager GUI or from command-line with a command such as:

```
$ sudo apt-get install sun-java6-jdk eclipse
```

This is not, however, recommended, because the Eclipse package may not include all the necessary Java EE tools, most importantly the Web Standard Tools, and it may cause incompatibilities with some components that are not installed with the package management system of your operating system.

1.4.3. Installing Apache Tomcat

Apache Tomcat is a lightweight Java web server suitable for both development and production. There are many ways to install it, but here we simply decompress the installation package.

Apache Tomcat should be installed with user permissions. During development, you will be running the Eclipse or some other IDE with user permissions, but Eclipse can not deploy web applications to Tomcat that is installed system-wide with administrator or root permissions.

1. Download the installation package:

   Apache Tomcat 6.0 (Core Binary Distribution) from `http://tomcat.apache.org/`

2. Decompress Apache Tomcat package to a suitable target directory, such as `C:\dev` in Windows or `/opt` in Linux or Mac OS X. The Apache Tomcat home directory will be `C:\dev\apache-tomcat-6.0.x` or `/opt/apache-tomcat-6.0.x`, respectively.

3. We are now ready to start and configure Eclipse. Start it by running `C:\dev\eclipse\eclipse.exe` (Windows) or `/opt/eclipse/eclipse` (Linux or OS X).

4. When starting Eclipse for the first time, it asks where to save the workspace. You can select any directory, but here we select `C:\dev\workspace` (Windows) or `/home/<user>/workspace` (Linux or OS X). We suggest that you also set this as the default.

5. You can see some Eclipse tutorials on the "Welcome" -screen or go to workbench to continue.

6. Configure the Tomcat Server by selecting **Window** → **Preferences**. Select **Web Services** → **Server and Runtime** from the tree on the left. Set Tomcat version to 6.x. Other defaults are fine.

1.4.4. Firefox and Firebug

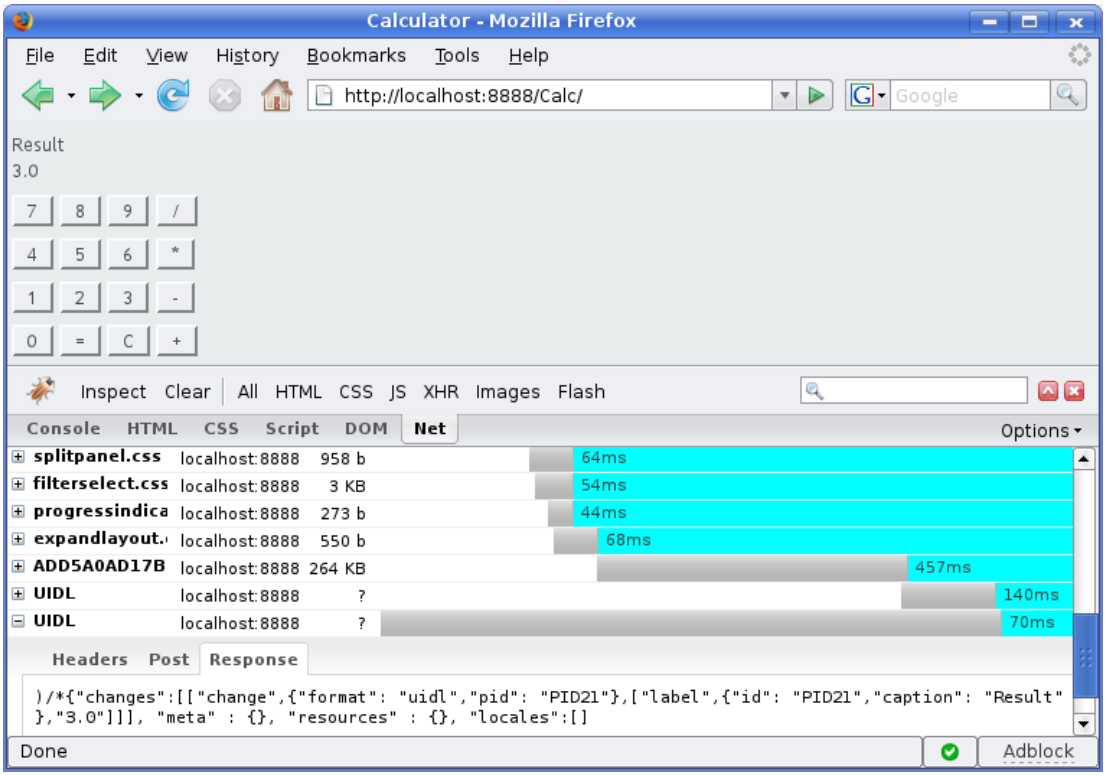
Because IT Mill Toolkit supports many web browsers, you can use any of them for development. If you also plan to build any theme parts, like CSS files, customized layouts, or even completely new user interface components, we recommend that you use Firefox for debugging. The toolkit specially supports Firebug debugger and shows special support information there.

To install Firefox, just go to www.mozilla.com [http://www.mozilla.com/] and download and run the installer.

After installing Firefox, use it to open http://www.getfirebug.com/ [http://www.getfirebug.com/] to install latest stable version of Firebug available for the browser. If clicking the **Install Firebug 1.0** -button does not open the install window, allow installs from the domain by clicking the yellow warning bar at the top of the browser-window.

When Firebug is installed, it can be enabled at any time from the bottom right corner of the Firefox window. See the example on debugging in Figure 1.4, “Firebug Debugger for Firefox” below.

Figure 1.4. Firebug Debugger for Firefox



Now that you have installed the development environment, you can proceed to making your first application.

1.5. QuickStart with Eclipse

Just want to have a quick try with IT Mill Toolkit? This section presents a QuickStart into running and debugging IT Mill Toolkit demos under Eclipse. The QuickStart includes a web server, so you do not need to install a full-weight web container such as Apache Tomcat.

1.5.1. Importing IT Mill Toolkit as a Project

The installation directory of IT Mill Toolkit contains all the necessary files to allow importing it as a ready-to-run Eclipse project. Do the following steps.

1. Start Eclipse. If necessary, switch to Java Perspective from menu **Window** → **Open Perspective** → **Java**.

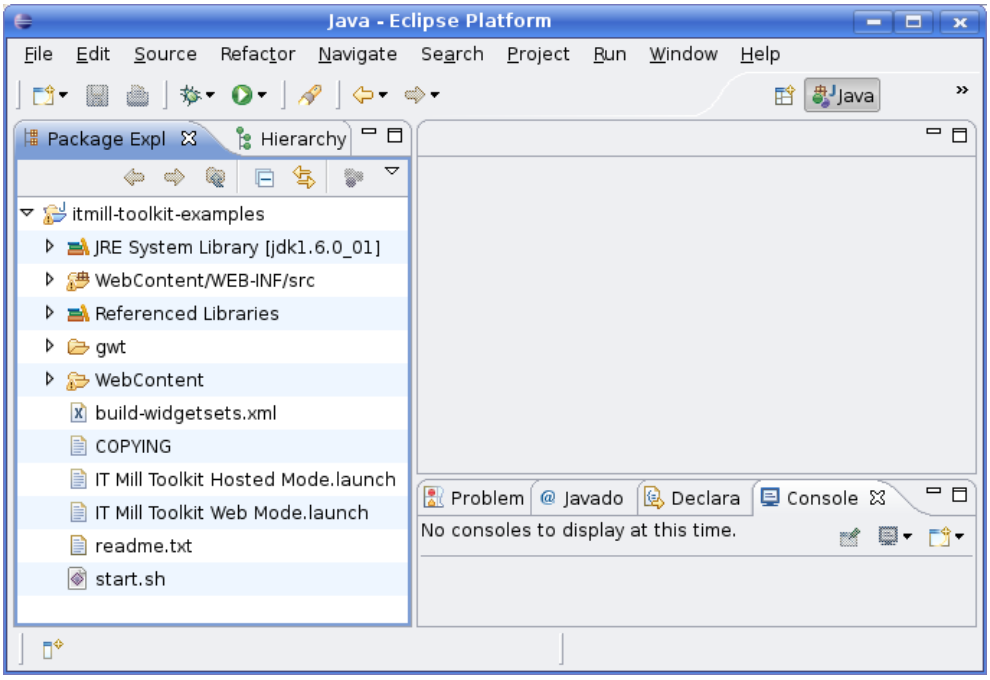
2. Select **File** → **Import...** and the dialog for importing opens.

3. In the **Import** dialog, select **General** → **Existing Project Into Workspace** and click **Next**.

4. In the **Select root directory** option, click on the **Browse** button, and select the folder where you unpacked IT Mill Toolkit, such as `itmill-toolkit-5.x.x`. Click **OK** in the selection window. Click **Finish** in the **Import** window to finish importing the project.

   The newly imported project will look as follows.

Figure 1.5. IT Mill Toolkit Imported as a Project in Eclipse



You can browse the source code of the demo application, and run the demo in a web browser by following the instructions given in the next section.

1.5.2. How to Run the Demo Applications in Eclipse?

Once the project is imported, as described above, you can run the Content Browser, including the demo applications, as follows. Either:

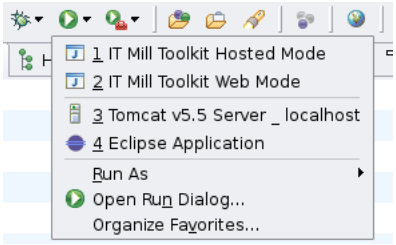
1. From the main menu, select **Run** → **Open Run Dialog...**.

2. From the list on the left, select **Java Application** → **IT Mill Toolkit Web Mode**.

3. Click **Run**.

Notice that after the application is launched once, it appears on the Favourites list. You can then either:

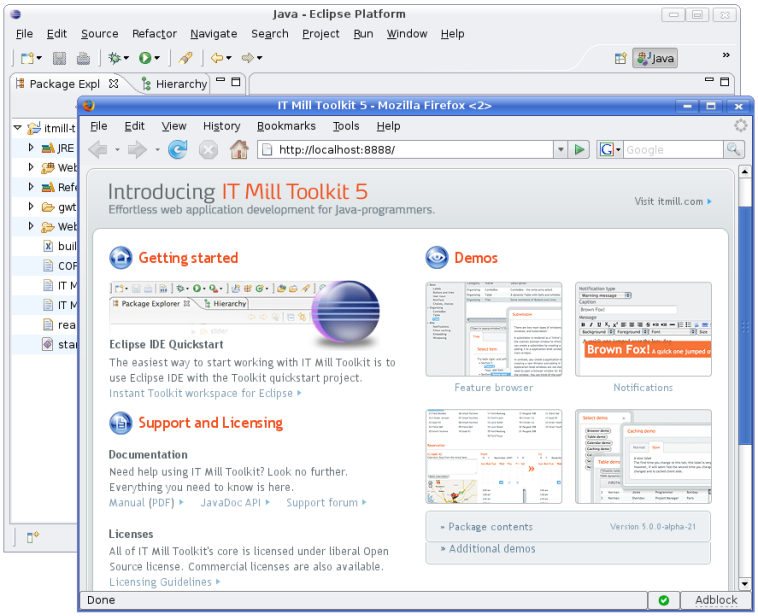
- Click on the small dropdown arrow on the right side of the **Run** button on Eclipse toolbar and select **IT Mill Toolkit Web Mode**.



- or... Select **Run** → **Run History** → **IT Mill Toolkit Web Mode**.

Running the application in Web Mode will open a browser window with the Content Browser. The default system web browser is opened; make sure that the browser is compatible with IT Mill Toolkit. The **Console** view in the lower pane of Eclipse will display text printed to standard output by the application. Clicking on the red **Terminate** button will stop the server.

Figure 1.6. IT Mill Toolkit Content Browser Started Under Eclipse



Notice that executing the web application locally may provide a security warning from your firewall software. This is caused by the Web Service which is started to run the Content Browser. You have to ignore the warnings or temporarily accept connections to port 8888 on your firewall software. Also, if the web service fails to start, make sure that no other service is using port 8888.

Launching the Hosted Mode Browser

The Hosted Mode Browser of Google Web Toolkit is a special web browser that runs the client-side GWT Java code as Java runtime instead of JavaScript, thereby allowing you to debug the client-side components in an IDE such as Eclipse.

Note

Hosted Mode Browser of Google Web Toolkit 1.4.62 does not work with Linux/Mozilla (Issue #1636 in IT Mill Toolkit version 5.2.0). As a workaround, you have to use a hand-made loader page as explained in `http://dev.itmill.com/ticket/1636`.

To run the demo applications in the Hosted Mode Browser of Google Web Toolkit, follow the following steps:

1. If not already started, start the demo application in Web Mode as described above. This launches the web server, which is used also when using the hosted mode.
2. From the main menu, select **Run** → **Open Debug Dialog...**.
3. From the list select **Java Application** → **IT Mill Toolkit Hosted Mode**.
4. Click **Debug**.

Starting demo applications under the Hosted Mode Browser can take considerable time! This is especially true for the Reservation and Color Picker applications, which require compilation of custom widget sets. During this time, the Hosted Mode Browser is unresponsive and does not update its window. Compiling widgets can take 5-30 seconds, depending on the hardware.

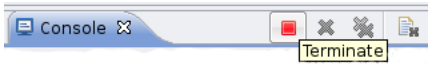
As with the Web Mode launcher, after you have run the launch once, you can select **Run** → **Debug History** → **IT Mill Toolkit Hosted Mode**, or click the dropdown marker on right of the **Debug** button in the toolbar and select **IT Mill Toolkit Hosted Mode**.

To use the Hosted Mode Browser in your own projects, you need to create a launch configuration in Eclipse. See Section 8.7.6, “Hosted Mode Browser” for more detailed information about the Hosted Mode Browser and how to create the launch configuration.

How to Stop the Run

To stop the launched Jetty web container that serves the Content Browser web application, select the **Console** tab and click on the **Terminate** button.

Figure 1.7. Terminating a Launch



To clean up all terminated launches from the **Console** window, click on the **Remove All Terminated Launches** button.

Figure 1.8. Removing Terminated Launches

1.5.3. How to Debug the Demo Applications in Eclipse?

You can inspect and experiment with the imported project as you like. When you develop an application, you may want to debug it. Running a web application in debug mode is easy in Eclipse. Next, we will show you how to debug the demo applications by inserting a breakpoint in the Calc example.

1. Make sure to stop any previous **Run** command as instructed above at the end of Section 1.3.4, “Running the Demo Applications”.
2. Select from menu **Run** → **Debug...** and the Debug configuration window will open.

3. Select **Java Application → IT Mill Toolkit Web Mode** and click **Debug**. The server will start and the web browser will open.

4. Open the Calc application by selecting on the start page **Additional demos → Calculator**.

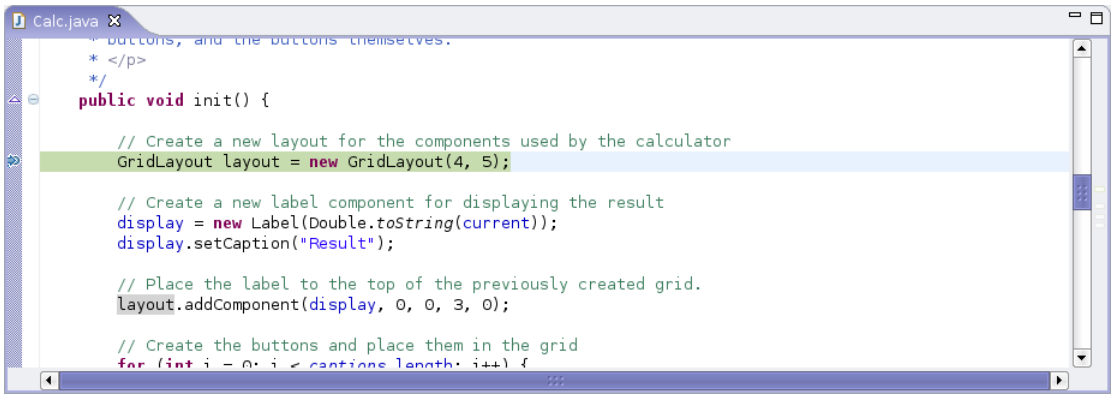
5. Open the source code for the Calc program. It is located in `WebContent/WEB-INF/src/com.itmill.toolkit.demo.Calc`. Doubleclick the class to open the source code in the editor.

6. Insert a breakpoint in the `init()` (line 57) by clicking on the gray bar on the left of the editor window to open the context menu, and select **Toggle Breakpoint**.

7. Switch to the browser window and click on the **Calc** link to open it.

8. Eclipse encouters the breakpoint and asks to switch to the Debug perspective. Click **Yes**. The debug window will show the current line where the execution stopped as follows:

Figure 1.9. Execution Stopped at Breakpoint in Debug Perspective in Eclipse



1.5.4. Using QuickStart as a Project Skeleton

If you like, you can also use the imported Toolkit as a skeleton for your project. Just remove any unnecessary files or files related to the demo applications from the project. You may also want to rename the IT Mill Toolkit installation directory with a name more proper for your project.

If you want to go the long way, which is probably preferred for a real project, especially a large one, you should follow the instructions in Section 1.6, “Your First Project with IT Mill Toolkit”.

1.6. Your First Project with IT Mill Toolkit

This section gives detailed instructions into creating a new project that uses IT Mill Toolkit. The task will include the following steps:

1. Create a new project in the Eclipse IDE.

2. Import IT Mill Toolkit library JAR into the project.

3. Write the source code.

4. Write the `web.xml` Deployment Descriptor for the web application.

5. Configure and start Tomcat (or some other web server).

6. Open a web browser to use the web application.

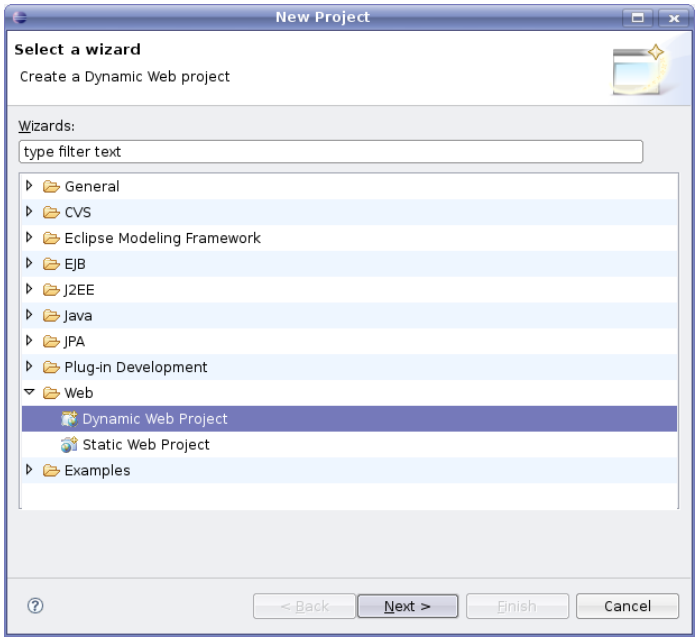
We also show you how to debug the application in the debug mode in Eclipse.

1.6.1. Creating the Project

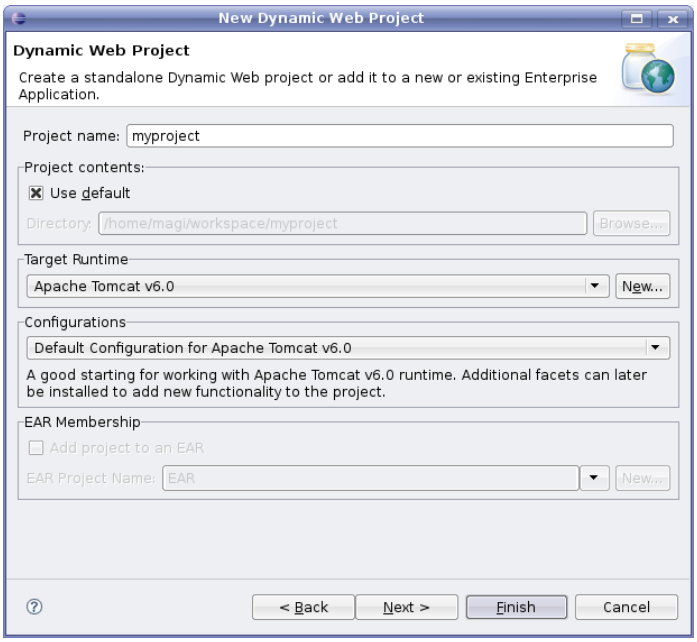
Let us create the first application project with the tools installed in the previous section. First launch Eclipse and follow the following steps:

1. Start creating a new project by selecting from the menu **File** → **New** → **Project...**.

2. From the New Project window that opens, select **Web** → **Dynamic Web Project** and click **Next**.



3. Enter the **Project name**, such as "`myproject`", and leave **Use default location** selected to create the new project under the default workspace location. Check that the **Target runtime**, that is the web container, is correct. For example, if you installed Apache Tomcat, check that it reads here. Click **Next** twice.

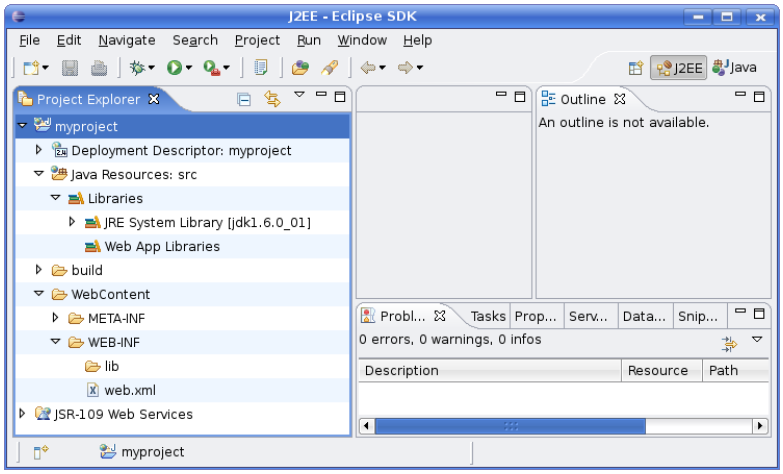


4. The wizard will suggest `myproject` for a context name. This will be the subpath in the URL, for example `http://localhost:8080/myproject`. The default for the application root will be `/` (root).

   You can just accept the defaults and click **Finish**. The wizard closes and creates the project.

5. Eclipse asks to switch to J2EE perspective. A Dynamic Web Project uses an external web server and the J2EE perspective provides tools to control the server and manage application deployment. Click **Yes**.

Figure 1.10. A New Dynamic Web Service Project



Feel free to explore the contents of the newborn project. Your source code will usually go under the `src` folder. IT Mill Toolkit libraries and any resource files will be placed under the `WebContent` folder, which contains all material that is to be published to the web server.

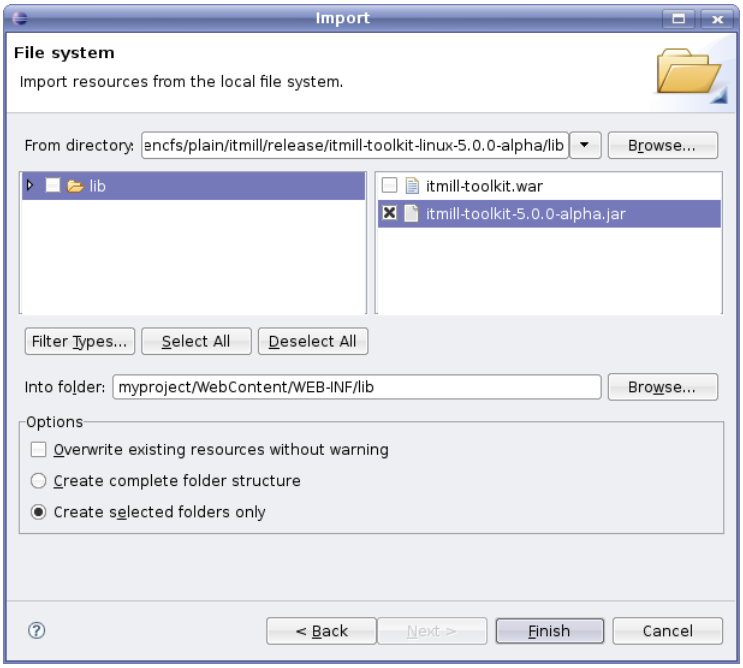
1.6.2. Including IT Mill Toolkit Libraries

You need to include the IT Mill Toolkit library package in the project. Copy the following JAR package from the directory where you unpacked IT Mill Toolkit distribution to `WebContent/WEB-INF/lib` folder:

- `WebContent/itmill-toolkit-5.x.x.jar`

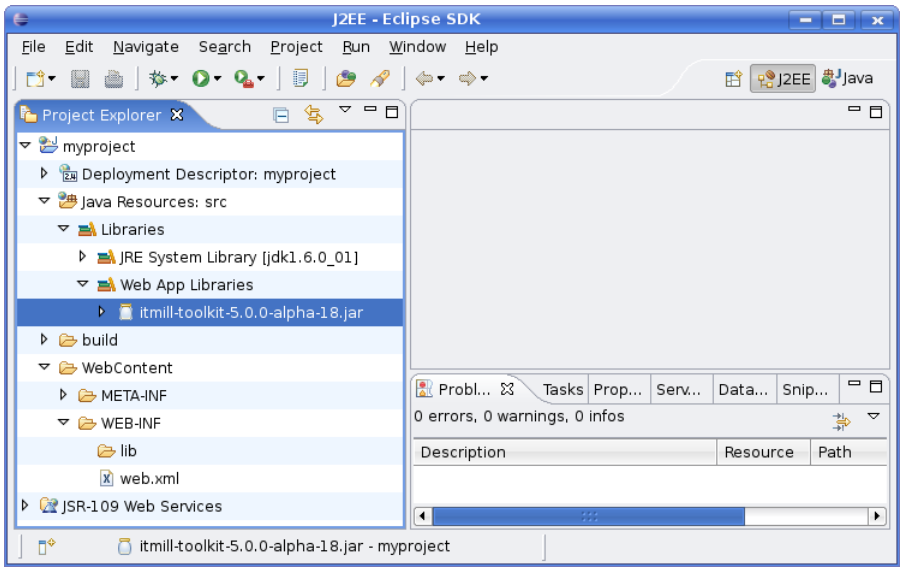
Perhaps the easiest way to include the library is to import it.

1. Select the `WebContent/WEB-INF/lib` folder in the **Project Explorer**, right-click on the folder and select **Import...**.
2. Select **General** → **File System** and click **Next**.
3. Click **Browse** to select the `WebContent` directory under the IT Mill Toolkit installation directory and click **Ok**.
4. The **Import** window will show the libraries contained in the directory.

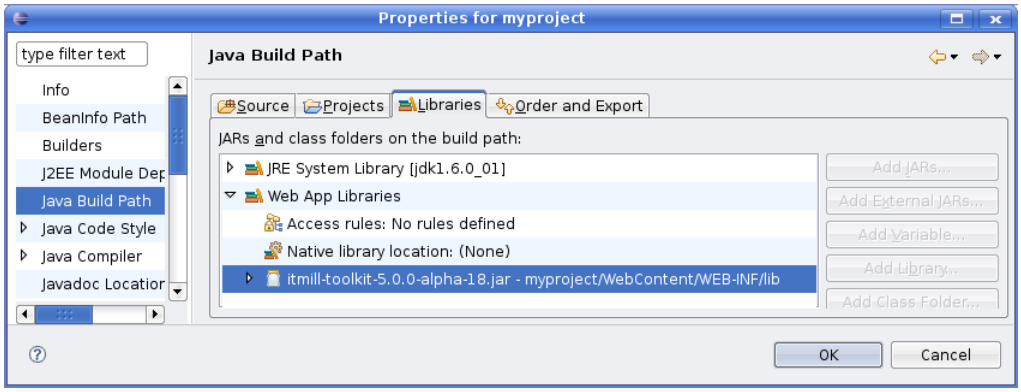


   Check the `itmill-toolkit-5.x.x.jar` item as shown above. Click **Finish** to import the selected library.

Notice that Eclipse does not show the imported library under `WebContent/WEB-INF` folder where you imported it, but under **Java Resources** → **Libraries** → **Web App Libraries**.



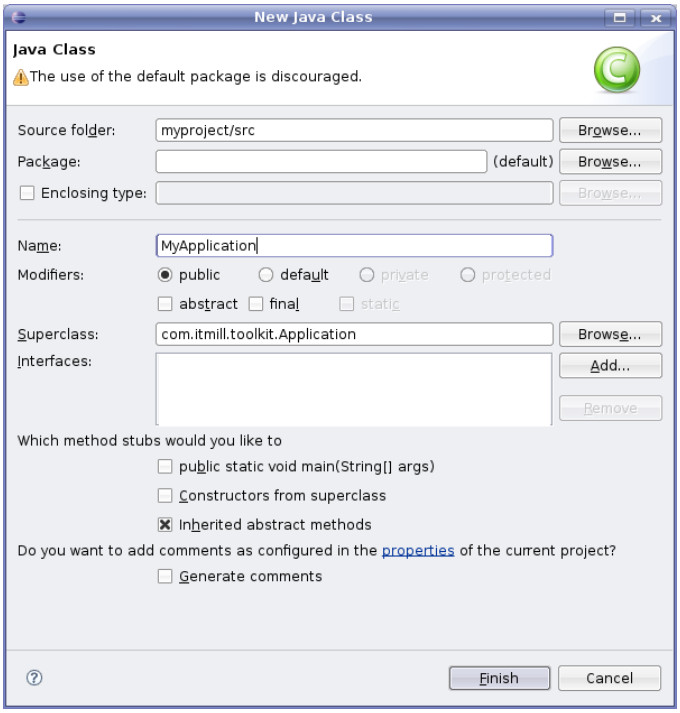
You can observe that the library has appeared in the project classpath by selecting **Project** → **Properties** and in the **Properties** window selecting **Java Build Path** → **Libraries** → **Web App Libraries**.



1.6.3. Writing the Code

Next, we will look into how to create the application class.

1. Right-click on the **Java Resources: src** folder and select **New** → **Class**.



2. Enter the class name in the **Name** field, for example **MyApplication**.

3. Enter a package name in the **Package** field or leave it empty to create the class in the default package.

4. You can enter the superclass **com.itmill.toolkit.Application** to create stubs for inherited abstract methods automatically, or leave it empty to define the inheritance manually in editor.

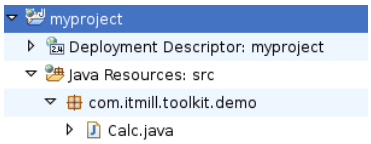
5. Click **Finish** to create the class and its source file.

The skeleton of the file will be opened in the editor and will look as follows.

```
import com.itmill.toolkit.Application;

public class MyApplication extends Application {

 @Override
 public void init() {
  // TODO Auto-generated method stub
 }
}
```

You can now write the source code. The Hello World application above provides a simple example for creating a minimal application.

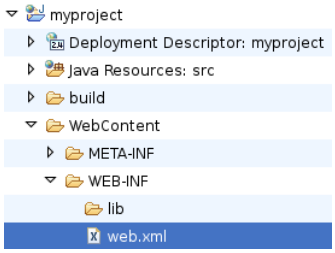
We will use the Calculator demo application in the rest of this section as an example. You can import the source file into the project by right-clicking the project folder and selecting **Import** → **Import...**. From the **Import** dialog, select **General** → **File System**, click **Next**. Click **Browse** to select the demo directory from the installation package, browse to `WebContent/src/itmill/toolkit/demo`, and click **Ok**. Check `Calc.java` in the list on the right. In the **Into folder** field, enter `myproject/src/com/itmill/toolkit/demo` to import the source file under the com.itmill.toolkit.demo package. Finally, click **Finish**.



1.6.4. Defining Deployment Descriptor

You need to set up the application environment as described in Section 3.7, “Application Environment” and provide a deployment descriptor `WebContent/WEB-INF/web.xml` for the application.

The new web project in Eclipse contains a template for the deployment descriptor. By default, Eclipse opens the file with XML editor. To use text editor, right-click on the `web.xml` file and select **Open With → Text Editor**. The template contains a `<welcome-file-list>` block, which you can remove if you like.



The contents of the descriptor for the Calc application are given in the example below.

Example 1.3. Web.xml Deployment Descriptor for a Project

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_ID" version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                        http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

    <servlet>
        <servlet-name>myservlet</servlet-name>
        <servlet-class>com.itmill.toolkit.terminal.gwt.server.ApplicationServlet
        </servlet-class>
        <init-param>
            <param-name>application</param-name>
            <param-value>com.itmill.toolkit.demo.Calc</param-value>
        </init-param>
    </servlet>

    <servlet-mapping>
        <servlet-name>myservlet</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

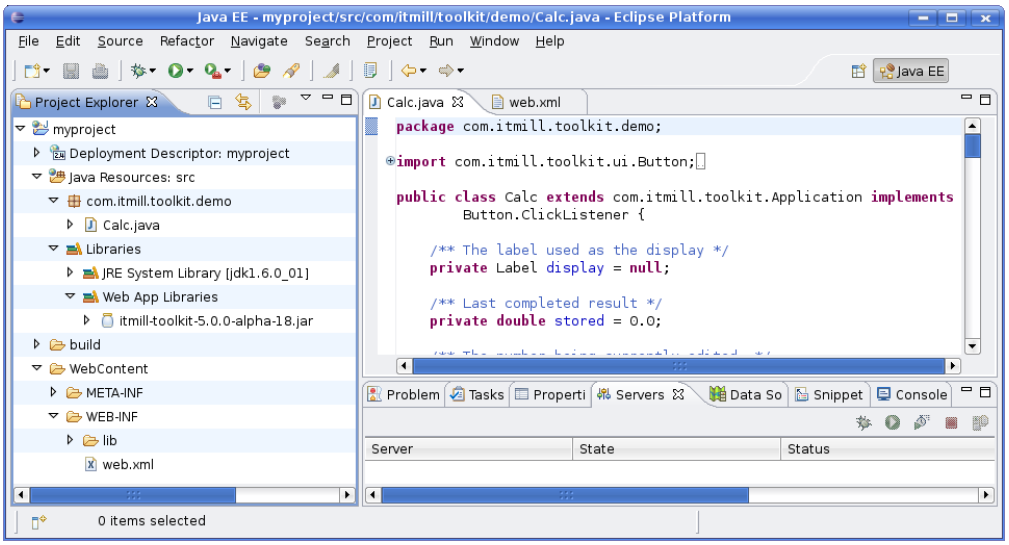
The descriptor defines a servlet with the name `myservlet`. The servlet class, **com.itmill.toolkit.terminal.gwt.server.ApplicationServlet**, is provided by IT Mill Toolkit framework and it should be the same for all IT Mill Toolkit projects. The servlet takes the class name **Calc** of the user application class as a parameter, including the full package path to the class. If the class is in the default package the package path is obviously not used.

For a more detailed treatment of the `web.xml` file, see Section 3.7.3, “Deployment Descriptor `web.xml`”.

1.6.5. Ready to Go!

Now everything should be in place and your Eclipse should look like this:

Figure 1.11. A Ready Project



In a production-ready project, you should also have widget sets and themes in the `WebContent/ITMILL` directory. The default widget sets and themes are included in the JAR library, but accessing them from a JAR is inefficient. We recommend installing the `ITMILL` directory so that it can be accessed directly from the web server. You can copy the directory from under the IT Mill Toolkit installation directory to the `WebContent` directory in your project, or extract it from the JAR package.

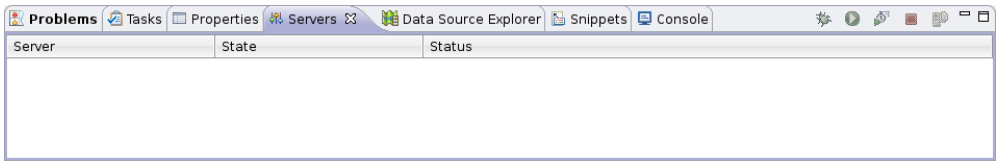
1.6.6. Starting the Web Server

Eclipse IDE for Java EE Developers has the Web Standard Tools package installed, which supports control of various web servers and automatic deployment of web content to the server when changes are made to a project.

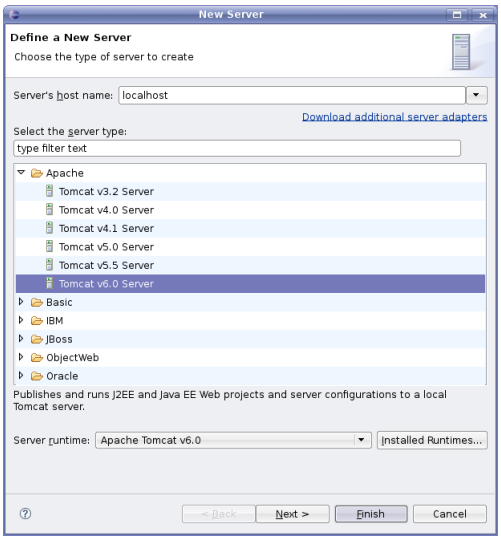
Make sure that Tomcat was installed with user permissions. Configuration of the web server in Eclipse will fail if the user does not have write permissions to the configuration and deployment directories under the Tomcat installation directory.

Follow the following steps.

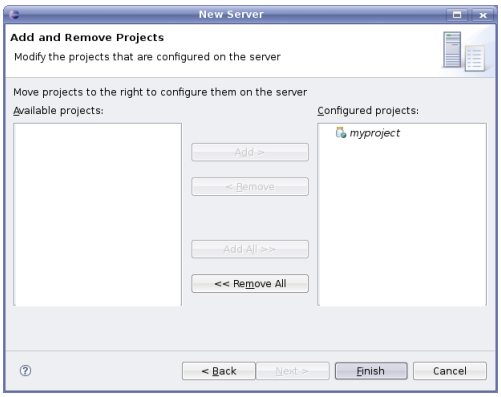
1. Switch to the **Servers** tab in the lower panel in Eclipse. List of servers should be empty after Eclipse is installed. Right-click on the empty area in the panel and select **New** → **Server**.



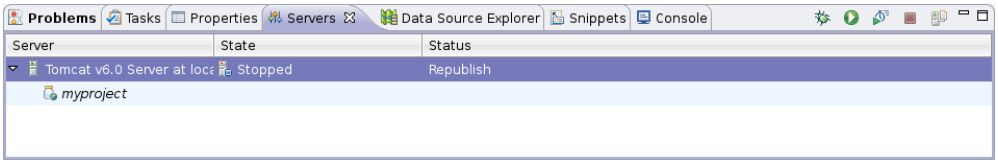
2. Select **Apache** → **Tomcat v6.0 Server** and set **Server's host name** as `localhost`, which should be the default. If you have only one Tomcat installed, **Server runtime** has only one choise. Click **Next**.



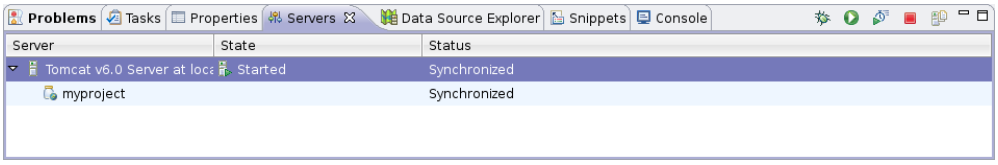
3. Add your project to the server by selecting it on the left and clicking **Add** to add it to the configured projects on the right. Click **Finish**.



4. The server and the project are now installed in Eclipse and are shown in the **Servers** tab. To start the server, right-click on the server and select **Debug**. To start the server in non-debug mode, select **Start**.



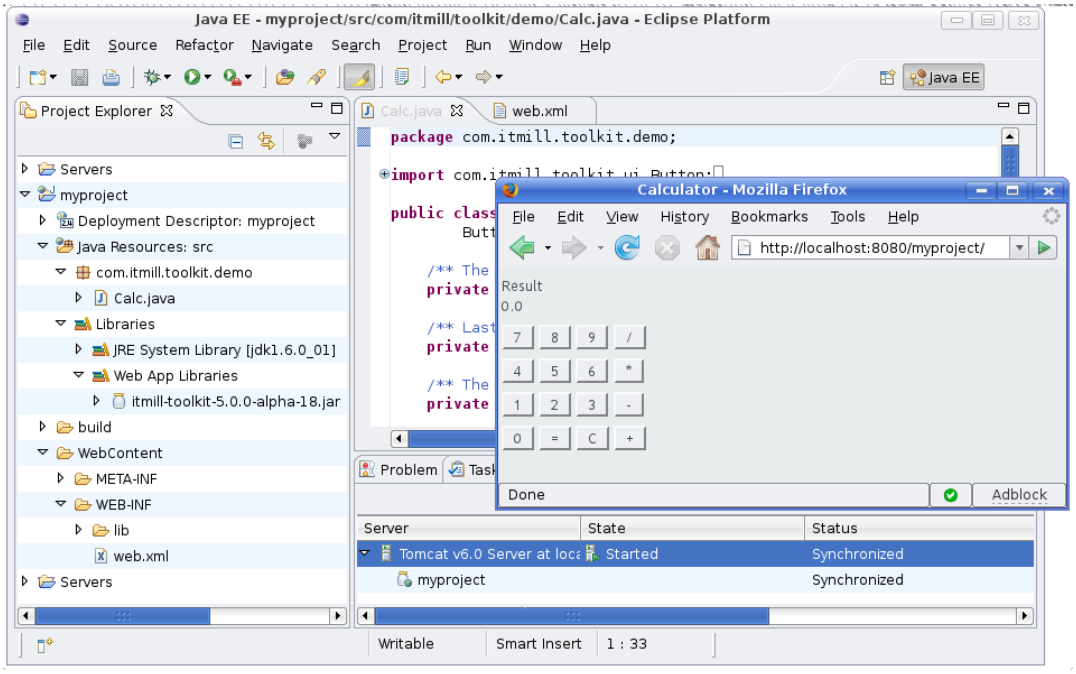
5. The server starts and the WebContent directory of the project is published to the server.



1.6.7. Running and Debugging

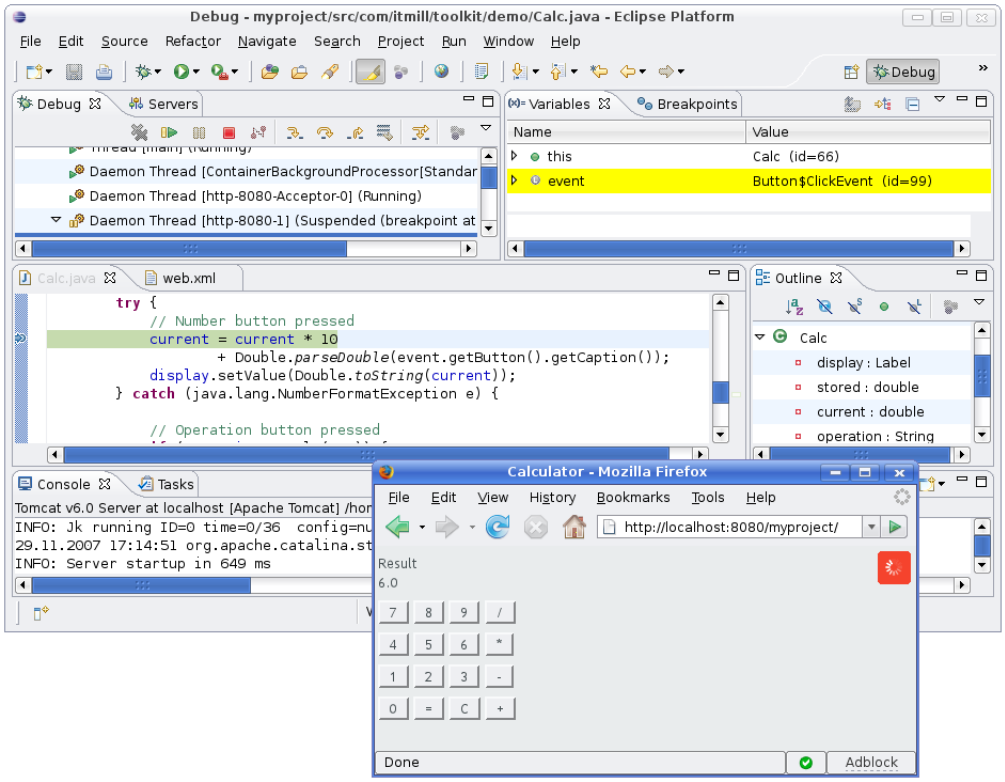
If you have everything set up as described above, all the rest is easy. Just head your web browser to `http://localhost:8080/myproject/`.

Figure 1.12. Running an IT Mill Toolkit Application



To examine how the application works, you can insert break points in the Java code by double-clicking on the left margin bar of the source code window. A small magnifying glass will indicate the breakpoint. If you insert a breakpoint in the `buttonClick()` event handling method and click any button in the calculator, eclipse will ask to switch to the Debug perspective. Answer **Yes** and the Debug perspective will open where the execution stopped at the breakpoint. You can examine the state of the application and even make some changes and then select **Resume** from **Run** menu to continue the execution.

Figure 1.13. Debugging an IT Mill Toolkit Application



The procedure described above allows debugging the server-side application. If you develop client-side widgets with Google Web Toolkit (GWT), the GWT Hosted Mode Browser allows you to debug the widgets. For more information on debugging client-side widgets, see Section 8.7.6, “Hosted Mode Browser”.

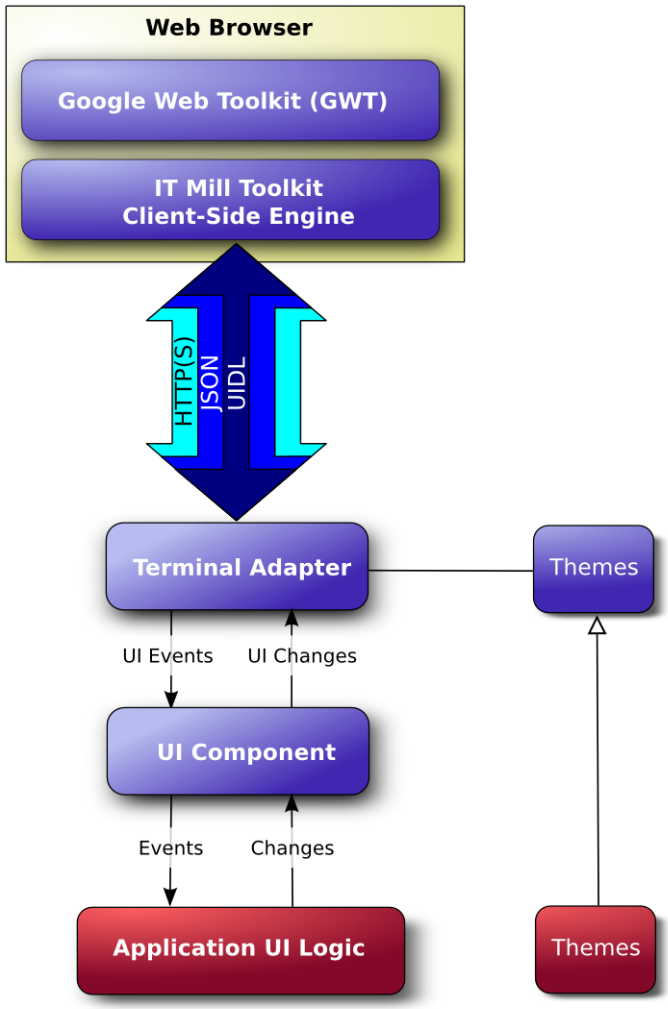
Chapter 2. Architecture

This chapter provides a description of the architecture of IT Mill Toolkit.

2.1. Overview

In Chapter 1, *Introduction*, we gave a short introduction to the general architecture of IT Mill Toolkit. Let us now look deeper into it. Figure 2.1, “IT Mill Toolkit Architecture” below illustrates the architecture.

Figure 2.1. IT Mill Toolkit Architecture



IT Mill Toolkit consists of a *web application API*, a horde of *user interface components*, *themes* for controlling the appearance, and a *data model* that allows binding the user interface components directly to data. Behind the curtains it also employs a *terminal adapter* to receive requests from web browsers and make responses by rendering the pages.

An application using IT Mill Toolkit runs as a servlet in a Java web server, serving HTTP requests. The terminal adapter receives client requests through the web server's Java Servlet API, and inteprets them to

user events for a particular session. An event is associated with a UI component and delivered to the application. As the application logic makes changes to the UI components, the terminal adapter renders them in the web browser by generating a response. In AJAX rendering mode, a client-side JavaScript component receives the responses and uses them to make any necessary changes to the page in the browser.

The top level of a user application consists of an application class that inherits **com.itmill.toolkit.Application**. It creates the UI components (see below) it needs, receives events regarding them, and makes necessary changes to the components. For detailed information about inheriting the **Application**, see Chapter 3, *Writing a Web Application*.

The major parts of the architecture and their function are as follows:

User Interface Components	The user interface consists of UI components that are created and laid out by the application. The components render themselves using a terminal adapter, which in return creates user events (see below) for the components. The components relay these events to the application logic. Most components are bound to some data using the Data Model (see below). For a complete description of UI component architecture, see Chapter 4, *User Interface Components*.
Client-Side Engine	The Client-Side Engine of IT Mill Toolkit manages the rendering in the web browser using Google Web Toolkit (GWT). It communicates user interaction and UI changes with the server-side Terminal Adapter using the User Interface Definition Language (UIDL), a JSON-based language. The communications are done using asynchronous HTTP or HTTPS requests. See Section 2.3, “Client-Side Engine”.
Terminal Adapter	The UI components do not render themselves directly as a web page, but use a *Terminal Adapter*. This abstraction layer allows users to use IT Mill Toolkit applications with practically any web browser. Releases 3 and 4 of IT Mill Toolkit supported HTML and simple AJAX based rendering, while Release 5 supports advanced AJAX-based rendering using Google Web Toolkit (GWT). You could imagine some other browser technology, not even based on HTML, and you - or we for that matter - could make it work just by writing a new adapter. Your application would still just see the Toolkit API. To allow for this sort of abstraction, UI components communicate their changes to the Terminal Adapter, which renders them for the user's browser. When the user does something in the web page, the events are communicated to the terminal adapter (through the web server) as asynchronous AJAX requests. The terminal adapter delivers the user events to the UI components, which deliver them to the application's UI logic.
Themes	The user interface separates between presentation and logic. While the UI logic is handled as Java code, the presentation is defined in *themes* as CSS. IT Mill Toolkit provides a default themes. User themes can, in addition to style sheets, include HTML templates that define custom layouts and other resources, such as images. Themes are discussed in detail in Chapter 6, *Themes*.

UIDL	The Terminal Adapter draws the user interface to the web page and any changes to it using a special User Interface Definition Language (UIDL). The UIDL communications are done using JSON (JavaScript Object Notation), which is a lightweight data interchange format that is especially efficient for interfacing with JavaScript-based AJAX code in the browser. See Section 2.2.3, "JSON" and Chapter 10, *User Interface Definition Language (UIDL)* for details.
Events	User interaction with UI components creates events, which are first processed on the client side with JavaScript and then passed all the way through the HTTP server, terminal adapter, and user component layers to the application. See Section 2.4, "Events and Listeners".
Data Model	In addition to the user interface model, IT Mill Toolkit provides a *data model* for interfacing data presented in UI components. Using the data model, the user interface components can update the application data directly, without the need for any control code. All the UI components use this data model internally, but they can be bound to a separate data source as well. For example, you can bind a table component to an SQL query response. For a complete overview of the IT Mill Toolkit Data Model, please refer to Chapter 7, *Data Model*.

2.2. Technological Background

This section provides an introduction to the various technologies and designs on which IT Mill Toolkit is based: AJAX-based web applications in general, Google Web Toolkit, and JSON data interchange format. This knowledge is not necessary for using IT Mill Toolkit, but provides some background if you need to make low-level extensions to IT Mill Toolkit.

2.2.1. AJAX

AJAX (Asynchronous JavaScript and XML) is a technique for developing web applications with responsive user interaction, similar to traditional desktop applications. Communications between the browser and the server can be done without needing to reload the page, but only small parts of the data can be loaded, as necessary. This goal is archieved by the use of a certain set of technologies: XHTML, CSS, DOM, JavaScript, XMLHttpRequest, and XML.

AJAX requests can be made by using the `XMLHttpRequest` API in JavaScript. The API is available in all major browsers and, as of 2006, the API is under way to become a W3C standard.

Communications between the browser and the server usually require some sort of *serialization* (or *marshalling*) of data objects. AJAX suggests the use of XML for data representation in communications between the browser and the server. While IT Mill Toolkit Release 4 used XML for data interchange, Release 5 uses the more efficient JSON. For more important about JSON and its use in IT Mill Toolkit, see Section 2.2.3, "JSON" below.

Section 9.2, "Special Characteristics of AJAX Applications" discusses the history and motivations for AJAX-based web applications, as well as some special characteristics that differ from both traditional web applications and desktop applications.

2.2.2. Google Web Toolkit

Google Web Toolkit is a software development kit for developing client-side web applications easily, without having to use JavaScript or other browser technologies directly. Applications using GWT are developed with Java and compiled into JavaScript with the GWT Compiler.

GWT is essentially a client-side technology, normally used to develop user interface logic in the web browser. GWT applications still need to communicate with a server using RPC calls and by serializing any data. IT Mill Toolkit effectively hides all client-server communications, allows handling user interaction logic in a server application, and allows software development in a single server-side application. This makes the architecture of an AJAX-based web application much simpler.

IT Mill Toolkit uses GWT to render user interfaces in the web browser and handle the low-level tasks of user interaction in the browser. Use of GWT is largely invisible in IT Mill Toolkit for applications that do not need any custom GWT components.

See Section 2.3, “Client-Side Engine” for a description of how GWT is used in the Client-Side Engine of IT Mill Toolkit. Chapter 8, *Developing Custom Components* provides information about the integration of GWT-based user interface components with IT Mill Toolkit.

2.2.3. JSON

JSON is a lightweight data-interchange format that is easy and fast to generate and parse. JSON messages are said to be possibly a hundred times faster to parse than XML with current browser technology. The format is a subset of the JavaScript language, which makes it possible to evaluate JSON messages directly as JavaScript expressions. This makes JSON very easy to use in JavaScript applications and therefore also for AJAX applications.

The Client-Side Engine of IT Mill Toolkit uses JSON through Google Web Toolkit, which supports JSON communications in the **com.google.gwt.json.client** package. Together with advanced update optimization and caching, IT Mill Toolkit is able to update changes in the user interface to the browser in an extremely efficient way.

The use of JSON is completely invisible to a developer using IT Mill Toolkit. Implementation of client-server serialization in custom widgets uses abstract interfaces that may be implemented as any low-level interchange format, such as XML or JSON. Details on JSON communications are given in Section 10.2, “JSON Rendering”.

2.3. Client-Side Engine

This section gives an overview of the client-side architecture of IT Mill Toolkit. Knowledge of the client-side technologies is generally not needed unless you develop or use custom GWT components. The client-side engine is based on Google Web Toolkit (GWT), which allows the development of the engine and client-side components solely with Java.

Chapter 8, *Developing Custom Components* provides information about the integration of GWT-based user interface components with IT Mill Toolkit.

Figure 2.2. Architecture of IT Mill Toolkit Client-Side Engine

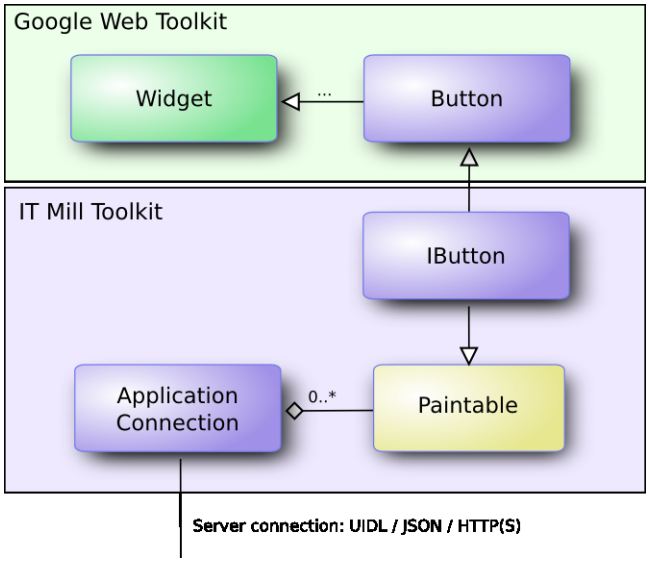


Figure 2.2, “Architecture of IT Mill Toolkit Client-Side Engine” illustrates the architecture of the client-side engine using a button component as an example. The user interface is managed by the **ApplicationConnection** class, which handles AJAX requests to the server and renders the user interface according to responses. Communications are done over HTTP(S) using the JSON data interchange format and the User Interface Definition Language (UIDL). In the server-side application, the button is used with the **Button** class of IT Mill Toolkit. On the client-side, the user interface consists of various GWT components that inherit **Widget** class. In the figure above, the GWT class **Button** is used to render the button in the browser (the inheritance of **Button** is simplified in the figure). IT Mill Toolkit provides an **IButton** class, which implements the **Paintable** interface needed for rendering the component with GWT.

The actual initial web page that is loaded in the browser is an empty page that loads the JavaScript code of the IT Mill Toolkit Client-Side Engine. After it is loaded and started, it handles the AJAX requests to the server. All server communications are done through the **ApplicationConnection** class.

The communication with the server is done as UIDL (User Interface Definition Language) messages using the JSON message interchange format over a HTTP(S) connection. UIDL is described in Chapter 10, *User Interface Definition Language (UIDL)* and JSON in Section 2.2.3, “JSON” and Section 10.2, “JSON Rendering”.

2.4. Events and Listeners

When a user does something, such as clicks a button or selects an item, the application needs to know about it. Many Java-based user interface frameworks follow the *Observer* design pattern to communicate user input to the application logic. So does IT Mill Toolkit. The design pattern involves two kinds of elements: an object and a number of observers that listen for events regarding the object. When an event related to the object occurs, the observers receive a notification regarding the event. In most cases there is only one observer, defined in the application logic, but the pattern allows for multiple observers. As in the event-listener framework of Java SE, we call the observing objects *listeners*.

In the ancient times of C programming, *callback functions* filled largely the same need as listeners do now. In object-oriented languages, we have only classes and methods, not functions, so the application has to give a class interface instead of a callback function pointer to the framework. However, IT Mill Toolkit supports defining a method as a listener as well.

Events can serve many kinds of purposes. In IT Mill Toolkit, the usual purpose of events is handling user interaction in a user interface. Session management can require special events, such as time-out, in which case the event is actually the lack of user interaction. Time-out is a special case of timed or scheduled events, where an event occurs at a specific date and time or when a set time has passed. Database and other asynchronous communications can cause events too.

To receive events of a particular type, an application must include a class that implements the corresponding listener interface. In small applications, the application class itself could implement the needed listener interfaces. Listeners are managed by the **AbstractComponent** class, the base class of all user interface components. This means that events regarding any component can listened to. The listeners are registered in the components with `addListener()` method.

Most components that have related events define their own event class and corresponding listener classes. For example, the **Button** has **Button.ClickEvent** events, which can be listened to through the **Button.ClickListener** interface. This allows an application to listen to many different kinds of events and to distinguish between them at class level. This is usually not enough, as applications usually have many components of the same class and need to distinguish between the particular components too. We will look into that more closely below. The purpose of this sort of class level separation is to avoid having to make type conversions in the handlers.

Notice that many listener interfaces inherit the **java.util.EventListener** superinterface, but it is not generally necessary to inherit it.

Figure 2.3. Class Diagram of a Button Click Listener

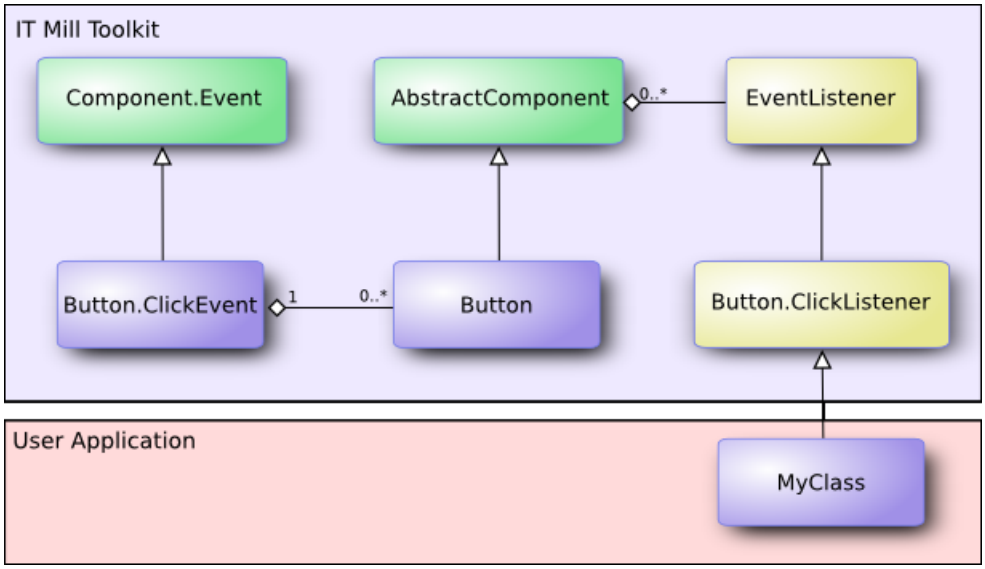


Figure 2.3, “Class Diagram of a Button Click Listener” illustrates an example where an application-specific class inherits the **Button.ClickListener** interface to be able to listen for button click events. The application must instantiate the listener class and register it with `addListener()`. When an event occurs, an event object is instantiated, in this case a **ClickEvent**. The event object knows the related UI component, in this case the **Button**.

The following example follows a typical pattern where you have a **Button** component and a listener that handles user interaction (clicks) communicated to the application as events. Here we define a class that listens click events.

```
public class TheButton implements Button.ClickListener {
    Button thebutton;

    /** Creates button into given container. */
    public TheButton(AbstractComponentContainer container) {
        thebutton = new Button ("Do not push this button");
        thebutton.addListener(this);
        container.addComponent(thebutton);
    }

    /** Handle button click events from the button. */
    public void buttonClick (Button.ClickEvent event) {
        thebutton.setCaption ("Do not push this button again");
    }
}
```

As an application often receives events for several components of the same class, such as multiple buttons, it has to be able to distinguish between the individual components. There are several techniques to do this, but probably the easiest is to use the property of the received event, which is set to the object sending the event. This requires keeping at hand a reference to every object that emits events.

```
public class TheButtons implements Button.ClickListener {
    Button thebutton;
    Button secondbutton;

    /** Creates two buttons in given container. */
    public TheButtons(AbstractComponentContainer container) {
        thebutton = new Button ("Do not push this button");
        thebutton.addListener(this);
        container.addComponent(thebutton);

        secondbutton = new Button ("I am a button too");
        secondbutton.addListener(this);
        container.addComponent (secondbutton);
    }

    /** Handle button click events from the two buttons. */
    public void buttonClick (Button.ClickEvent event) {
        if (event.getButton() == thebutton)
            thebutton.setCaption ("Do not push this button again");
        else if (event.getButton() == secondbutton)
            secondbutton.setCaption ("I am not a number");
    }
}
```

Another solution to handling multiple events of the same class involves attaching an event source to a listener method instead of the class. An event can be attached to a method using another version of the `addListener()` method, which takes the event handler method as a parameter either as a name of the method name as a string or as a **Method** object. In the example below, we use the name of the method as a string.

```
public class TheButtons2 {
    Button thebutton;
    Button secondbutton;

    /** Creates two buttons in given container. */
    public TheButtons2(AbstractComponentContainer container) {
        thebutton = new Button ("Do not push this button");
        thebutton.addListener(Button.ClickEvent.class, this, "theButtonClick");
        container.addComponent(thebutton);

        secondbutton = new Button ("I am a button too");
        secondbutton.addListener(Button.ClickEvent.class, this, "secondButtonClick");
        container.addComponent (secondbutton);
    }
```

```
    public void theButtonClick (Button.ClickEvent event) {
        thebutton.setCaption ("Do not push this button again");
    }

    public void secondButtonClick (Button.ClickEvent event) {
        secondbutton.setCaption ("I am not a number!");
    }
}
```

Adding a listener method with `addListener()` is really just a wrapper that creates a **com.itmill.toolkit.event.ListenerMethod** listener object, which is an adapter from a listener class to a method. It implements the **java.util.EventListener** interface and can therefore work for any event source using the interface. Notice that not all listener classes necessarily inherit the **EventListener** interface.

The third way, which uses anonymous local class definitions, is often the easiest as it does not require cumbering the managing class with new interfaces or methods. The following example defines an anonymous class that inherits the **Button.ClickListener** interface and implements the `buttonClick()` method.

```
public class TheButtons3 {
    Button thebutton;
    Button secondbutton;

    /** Creates two buttons in given container. */
    public TheButtons3(AbstractComponentContainer container) {
        thebutton = new Button ("Do not push this button");
        thebutton.addListener(new Button.ClickListener() {
            /* Define the method in the anonymous class to handle the click. */
            public void buttonClick(ClickEvent event) {
                thebutton.setCaption ("Do not push this button again");
            }
        });
        container.addComponent(thebutton);

        secondbutton = new Button ("I am a button too");
        secondbutton.addListener(new Button.ClickListener() {
            /* Define the method in the anonymous class to handle the click. */
            public void buttonClick(ClickEvent event) {
                secondbutton.setCaption ("I am not a number!");
            }
        });
        container.addComponent (secondbutton);
    }
}
```

Other techniques for separating between different sources also exist. They include using object properties, names, or captions to separate between them. Using captions or any other visible text is generally discouraged, as it may create problems for internationalization. Using other symbolic strings can also be dangerous, because the syntax of such strings is checked only runtime.

Events are usually emitted by the framework, but applications may need to emit them too in some situations, such as when updating some part of the UI is required. Events can be emitted using the `fireEvent(Component.Event)` method of **AbstractComponent**. The event is then relayed to all the listeners of the particular event class for the object. Some components have a default event type, for example, a **Button** has a nested **Button.ClickEvent** class and a corresponding **Button.ClickListener** interface. These events can be triggered with `fireComponentEvent()`.

Chapter 3. Writing a Web Application

This chapter gives the fundamentals of web application development with IT Mill Toolkit. The overview gives an introduction to the **Application** class that every user application must inherit. Every application has a main window. The main window and other windows are also managed by the application object, so we give a detailed description of the various window classes, together with some common design patterns. Also various resources, such as images and downloadable documents, are managed by the **Application**, so we look into the basic resource interfaces and classes. Finally, we look into the deployment of applications as Java Servlets in a web container.

Related topics in other chapters include the use of events and listeners, the basis of all user interaction in applications, which are are detailed in Section 2.4, “Events and Listeners”.

To gain more insight about application design with IT Mill Toolkit, you may want to read Chapter 9, *Advanced Web Application Topics*. For a newcomer into AJAX development, it explains the role of pages in AJAX web applications, and provides some basic design patterns for applications.

3.1. Overview

An application that uses IT Mill Toolkit must define an application class that inherits the abstract **com.itmill.toolkit.Application** class. The application class must implement the `init()` method.

```
public class MyApplication extends com.itmill.toolkit.Application {

    public void init() {
        ... initialization code goes here ...
    }
}
```

The web application API may seem similar to Java Servlet API, but that is only superficial. IT Mill Toolkit framework associates requests with sessions so that an application class instance is really a session object. Because of this, you can develop web applications much like you would develop desktop applications.

How does it work? IT Mill Toolkit framework does basically everything it does on top of the Java Servlet API, which lies hidden deep under the hood, with the terminal adapter being the lowest level layer for handling requests from the web container. When the web container gets the first request for a URL registered for an application, it creates an instance of the **ApplicationServlet** class in IT Mill Toolkit framework that inherits the **HttpApplet** class defined in Java Servlet API. It follows sessions by using **HttpSession** interface and associates an **Application** instance with each session. During the lifetime of a session, the framework relays user actions to the proper application instance, and further to a user interface component.

Application class also provides facilities for window access, execution control, and theme selection.

As the application instance is really a session, it can end. If the user quits the application through the user interface, an event handler should call the `close()` method in **Application**. However, as the user interface runs under a web browser, a user can simply close the browser window and the application has no way of knowing it happened. When the user starts the browser again and opens the application URL, the application window will be rendered where the user left off. This can be desired behaviour in many cases, but often it is not and it creates security problems. A common solution is to use a timeout to terminate a session automatically.

The user application class needs to be registered in the web application. This is done in the web application package, see Section 3.7, “Application Environment”.

3.2. Managing the Main Window

As explained in Section 9.2, “Special Characteristics of AJAX Applications”, an AJAX web application usually runs in a single "web page" in a browser window. The page is generally not reloaded after it is opened initially, but it communicates user interaction with the server through AJAX communications. A window in an AJAX application is therefore more like a window in a desktop application and less like a page.

A **Window** is the top-level container of a user interface displayed in a browser window. As an AJAX application typically runs on a single "page" (URL), there is usually just one window -- the main window. The main window can be accessed using the URL of the application. You set the main window with the `setMainWindow()` method of the **Application** class.

```
import com.itmill.toolkit.ui.*;

public class HelloWorld extends com.itmill.toolkit.Application {
    public void init() {
        Window main = new Window("The Main Window");
        setMainWindow(main);

        ... fill the main window with components ...
    }
}
```

IT Mill Toolkit has two basic kinds of windows: application-level windows such as the main window and child windows inside the application-level windows. These two types of windows are covered in the subsequent sections.

3.3. Child Windows

An application-level window can have a number of floating child windows. They are managed by the client-side JavaScript runtime of IT Mill Toolkit using HTML features. IT Mill Toolkit allows opening and closing child windows, refreshing one window from another, resizing windows, and scrolling the window content. Child windows are typically used for *Dialog Windows* and *Multiple Document Interface* applications. Child windows are by default not modal; you can set them modal as described in Section 3.3.3, “Modal Windows”.

As with all user interface components, the appearance of a window and its contents is defined with themes.

User control of a child window is limited to moving, resizing, and closing the window. Maximizing or minimizing are not yet supported.

3.3.1. Opening and Closing a Child Window

You can open a new window by creating a new **Window** object and adding it to the main window with `addWindow()` method of the **Application** class.

```
mywindow = new Window("My Window");
mainwindow.addWindow(mywindow);
```

You close the window in a similar fashion, by calling the `removeWindow()` of the **Application** class:

```
myapplication.removeWindow (mywindow);
```

The user can, by default, close a child window by clicking the close button in the upper-right corner of the window. You can disable the button by setting the window as *read-only* with `setReadOnly(true)`. Notice that you could disable the button also by making it invisible in CSS with a "`display: none`"

formatting. The problem with such a cosmetic disabling is that a malicious user might re-enable the button and close the window, which might cause problems and possibly be a security hole. Setting the window as read-only not only disables the close button on the client side, but also prevents processing the close event on the server side.

The following example demonstrates the use of a child window in an application. The example manages the window using a custom component that contains a button for opening and closing the window.

```
/** Component contains a button that allows opening a window. */
public class WindowOpener extends CustomComponent
                          implements Window.CloseListener {
    Window mainwindow;  // Reference to main window
    Window mywindow;    // The window to be opened
    Button openbutton;  // Button for opening the window
    Button closebutton; // A button in the window
    Label  explanation; // A descriptive text

    public WindowOpener(String label, Window main) {
        mainwindow = main;

        /* The component consists of a button that opens the window. */
        final VerticalLayout layout = new VerticalLayout();

        openbutton = new Button("Open Window", this, "openButtonClick");
        explanation = new Label("Explanation");
        layout.addComponent(openbutton);
        layout.addComponent(explanation);

        setCompositionRoot(layout);
    }

    /** Handle the clicks for the two buttons. */
    public void openButtonClick(Button.ClickEvent event) {
        /* Create a new window. */
        mywindow = new Window("My Dialog");
        mywindow.setPositionX(200);
        mywindow.setPositionY(100);

        /* Add the window inside the main window. */
        mainwindow.addWindow(mywindow);

        /* Listen for close events for the window. */
        mywindow.addListener(this);

        /* Add components in the window. */
        mywindow.addComponent(new Label("A text label in the window."));
        closebutton = new Button("Close", this, "closeButtonClick");
        mywindow.addComponent(closebutton);

        /* Allow opening only one window at a time. */
        openbutton.setEnabled(false);

        explanation.setValue("Window opened");
    }

    /** Handle Close button click and close the window. */
    public void closeButtonClick(Button.ClickEvent event) {
        /* Windows are managed by the application object. */
        mainwindow.removeWindow(mywindow);

        /* Return to initial state. */
        openbutton.setEnabled(true);

        explanation.setValue("Closed with button");
    }
```

```
    /** In case the window is closed otherwise. */
    public void windowClose(CloseEvent e) {
        /* Return to initial state. */
        openbutton.setEnabled(true);

        explanation.setValue("Closed with window controls");
    }
}
```

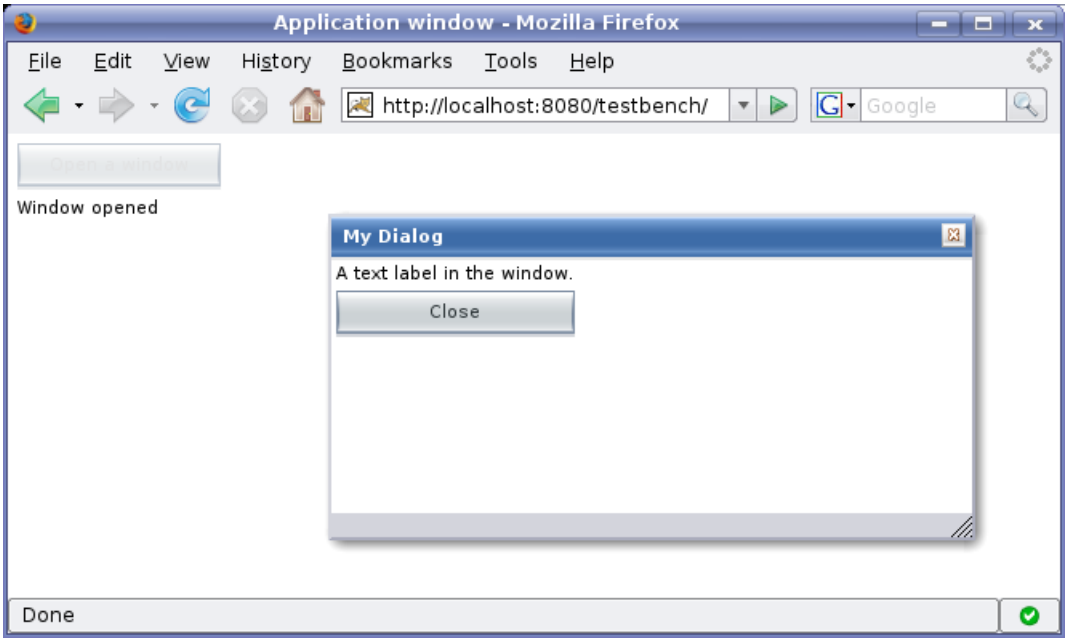
You can use the above custom component in the application class with:

```
    public void init() {
        Window main = new Window("The Main Window");
        setMainWindow(main);

        addComponent(new WindowOpener("Window Opener", main));
    }
```

The example implements a custom component that inherits the **CustomComponent** class. It consists of a **Button** that it uses to open a window and a **Label** to describe the state of the window. When the window is open, the button is disabled. When the window is closed, the button is enabled again.

When added to an application, the screen will look as illustrated in the following screenshot:

Figure 3.1. Opening a Child Window



3.3.2. Window Positioning

When created, a window will have a default size and position. You can specify the size of a window with `setHeight()` and `setWidth()` methods. You can set the position of the window with `setPositionX()` and `setPositionY()` methods.

```
/* Create a new window. */
mywindow = new Window("My Dialog");

/* Set window size. */
mywindow.setHeight("200px");
mywindow.setWidth("400px");
```

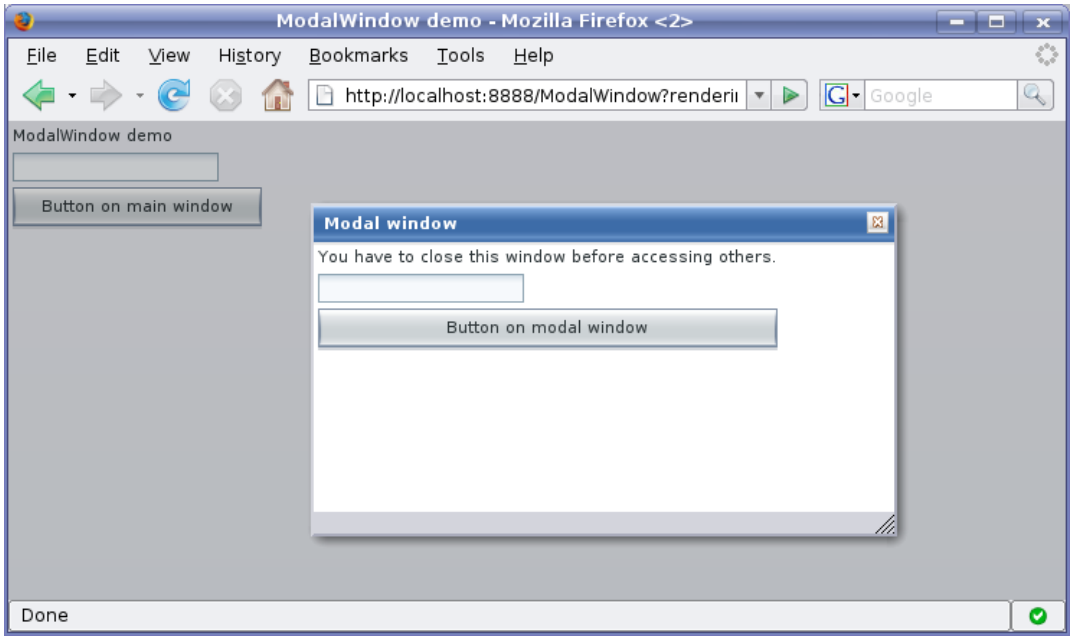
```
/* Set window position. */
mywindow.setPositionX(200);
mywindow.setPositionY(50);
```

Notice that the size of the main window is unknown and the `getHeight` and `getWidth` methods will return -1.

3.3.3. Modal Windows

A modal window is a child window that has to be closed by the user before the use of the parent window can continue. Dialog windows are typically modal. The advantage of modal windows is the simplification of user interaction, which may contribute to the clarity of the user interface. Modal windows are also easy to use from a development perspective, because as user interaction is isolated to them, changes in application state are more limited while the modal window is open. The disadvantage of modal windows is that they can restrict workflow too much.

Figure 3.2. Screenshot of the Modal Window Demo Application



Depending on theme settings, the parent window may be grayed while the modal window is open.

The demo application of IT Mill Toolkit includes an example of using modal windows. Figure 3.2, “Screenshot of the Modal Window Demo Application” above is from the demo application. The example includes the source code.

Security Warning

Modality of child windows is purely a client-side feature and can be circumvented with client-side attack code. You should not trust in the modality of child windows in security-critical situations such as login windows.

3.4. Application-Level Windows

IT Mill Toolkit Release 5 introduces support for multiple application-level windows that can be used just like the main window. All such windows use the same application session. Each window is identified with a URL that is used to access it. This makes it possible to bookmark application-level windows. Such windows can even be created dynamically based on URLs.

Application-level windows allow several uses important for the usability of browser-based applications.

- *Native child windows*. An application can open child windows that are not floating windows inside a parent window.
- *Page-based browsing*. The application can allow the user to open certain content to different windows. For example, in a messaging application, it can be useful to open different messages to different windows so that the user can browse through them while writing a new message.
- *Bookmarking*. Bookmarks in the web browser can provide an entry-point to some content provided by an application.
- *Embedding windows*. Windows can be embedded in web pages, thus making it possible to provide different views to an application from different pages or even from the same page, while keeping the same session. See Section 3.8, “Embedding Applications in Web Pages”.

Because of the special nature of AJAX applications, these uses require some caveats. We will go through them later in Section 3.4.4, “Caveats in Using Multiple Windows”.

3.4.1. Creating New Application-Level Windows

Creating a new application-level window is much like creating a child window, except that the window is added with `addWindow()` to the application object instead of the main window.

```
public class WindowTestApplication extends Application {
    public void init() {
        final Window main = new Window ("Window Test Application");
        setMainWindow(main);

        /* Create a new window. */
        final Window mywindow = new Window("Second Window");

        /* Manually set the name of the window. */
        mywindow.setName("mywindow");

        /* Add some content to the window. */
        mywindow.addComponent(new Label("This is a second window."));

        /* Add the window to the application. */
        addWindow(mywindow);
    }
}
```

This creates the window object that a user can view by opening the URL in a browser. Creating an application-level window object does not open a new browser window automatically to view the object, but if you wish to open one, you have to do it explicitly as shown below. An application-level window has a unique URL, which is based on the application URL and the name of the window given with the `setName()` method. For example, if the application URL is `http://localhost:8080/myapp/` and the window name is `mywindow`, the URL for the window will be `http://localhost:8080/myapp/mywindow/`. If the name of a window is not explicitly set with

`setName()`, an automatically generated name will be used. The name can be retrieved with the `getName()` method and the entire URL with `getURL()`.

There are three typical ways to open a new window: using the `open()` method of **Window** class, a **Link**, or referencing it from HTML or JavaScript code written inside a **Label** component.

The **Window** `open()` method takes as parameters a resource to open and the target name. You can use **ExternalResource** to open a specific URL, which you get from the window to be opened with the `getURL()` method.

```
/* Create a button to open a new window. */
main.addComponent(new Button("Click to open new window",
                  new Button.ClickListener() {
    public void buttonClick(ClickEvent event) {
        // Open the window.
        main.open(new ExternalResource(mywindow.getURL()), "_new");
    }
}));
```

The target name is one of the default HTML target names (*`_new`*, *`_blank`*, *`_top`*, etc.) or a custom target name. How the window is exactly opened depends on the browser. Browsers that support tabbed browsing can open the window in another tab, depending on the browser settings.

Another typical way to open windows is to use a **Link** component with the window URL as an **ExternalResource**.

```
/* Add a link to the second window. */
Link link = new Link("Click to open second window",
                     new ExternalResource(mywindow.getURL()));
link.setTargetName("second");
link.setTargetHeight(300);
link.setTargetWidth(300);
link.setTargetBorder(Link.TARGET_BORDER_DEFAULT);
main.addComponent(link);
```

Using a **Link** allows you to specify parameters for the window that opens by clicking on the link. Above, we set the dimensions of the window and specify what window controls the window should contain. The *`Link.TARGET_BORDER_DEFAULT`* specifies to use the default, which includes most of the usual window controls, such as the menu, the toolbar, and the status bar.

Another way to allow the user to open a window is to insert the URL in HTML code inside a **Label**. This allows even more flexibility in specifying how the window should be opened.

```
/* Add the link manually inside a Label. */
main.addComponent(new Label("Second window: <a href='"
                            +mywindow.getURL()+"' target='second'>click to open</a>",
                  Label.CONTENT_XHTML));
main.addComponent(new Label("The second window can be accessed through URL: "
                            +mywindow.getURL()));
```

When an application-level window is closed in the browser the `close()` method is called just like for a child window and the **Window** object is purged from the application.

3.4.2. Creating Windows Dynamically

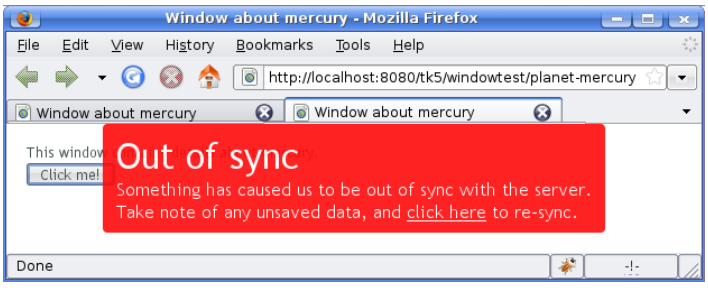
You can create a window object dynamically by its URL path by overriding the `getWindow()` method of the **Application** class. The method gets a window name as its parameter and must return the corresponding **Window** object. The window name is determined from the first URL path element after the application URL (window name may not contain slashes). See the notes below for setting the actual name of the dynamically created windows below.

The following example allows opening windows with a window name that begins with "`planet-`" prefix. Since the method is called for *every* browser request for the application, we filter only the requests where a window with the given name does not yet exist.

```
public class WindowTestApplication extends Application {
    ...

    @Override
    public Window getWindow(String name) {
        // If a dynamically created window is requested, but it does
        // not exist yet, create it.
        if (name.startsWith("planet-") &&
              super.getWindow(name) == null) {
            String planetName = name.substring("planet-".length());

            // Create the window object.
            Window newWindow = new Window("Window about " + planetName);

            // DANGEROUS: Set the name explicitly. Otherwise, an
            // automatically generated name is used, which is usually safer.
            newWindow.setName(name);

            // Put some content in it.
            newWindow.addComponent(new Label("This window contains details about " +
                                             planetName + "."));

            // Add it to the application as a regular application-level window.
            addWindow(newWindow);

            return newWindow;
        }

        // Otherwise the Application object manages existing windows by their name.
        return super.getWindow(name);
    }
```

The window name is and must be a unique indentifier for each **Window** object instance. If you use `setName()` to set the window name explicitly, as we did above, any browser window that has the same URL (within the same browser) would open the *same* window object. This is dangerous and *generally not recommended*, because the browser windows would share the same window object. Opening two windows with the same static name would immediately lead to a synchronization error, as is shown in Figure 3.3, “Synchronization Error Between Windows with the Same Name” below. (While also the window captions are same, they are irrelevant for this problem.)

Figure 3.3. Synchronization Error Between Windows with the Same Name



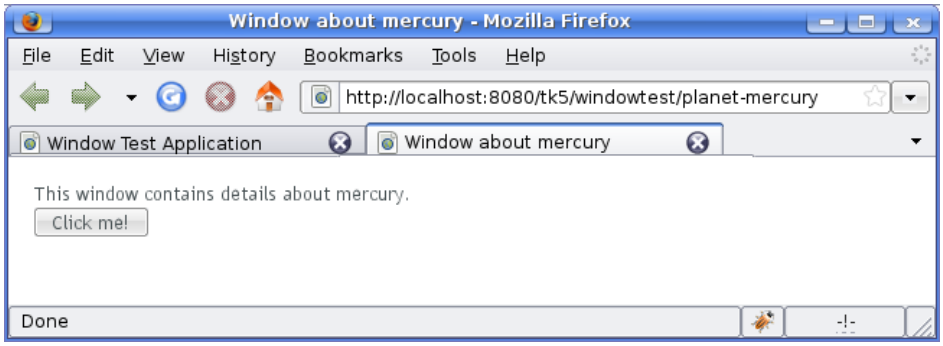
There are some cases where setting the name explicitly is useful. The launch application below is one example, as it always opens the other windows in a window target that is specific to the window name, thereby never creating two windows with the same URL. Similarly, if you had embedded the application in a browser frame and the link would open the window in a frame, you would not have problems. Having a

single window instance for a URL is also useful if the browser crashes and the user opens the window again, as it will have kept its previous (server-side) state.

Leaving the window name to be automatically generated allows opening multiple windows with the same URL, while each of the windows will have a separate state. The URL in the location bar stays unchanged and the generated window name is used only for the Ajax communications to identify the window object. A generated name is a string representation of a unique random number, such as "`1928676448`". You should be aware of the generated window names when overriding the `getWindow()` method (and not unintentionally create a new window instance dynamically for each such request). The condition in the above example would also filter out the requests for an already existing window with a generated name.

Figure 3.4, “A Dynamically Created Window” shows a dynamically created application-level window with the URL shown in the address bar. The URL for the application is here `http://localhost:8080/tk5/windowexample/`, including the application context, and the dynamically created window's name is `planet-mars`.

Figure 3.4. A Dynamically Created Window

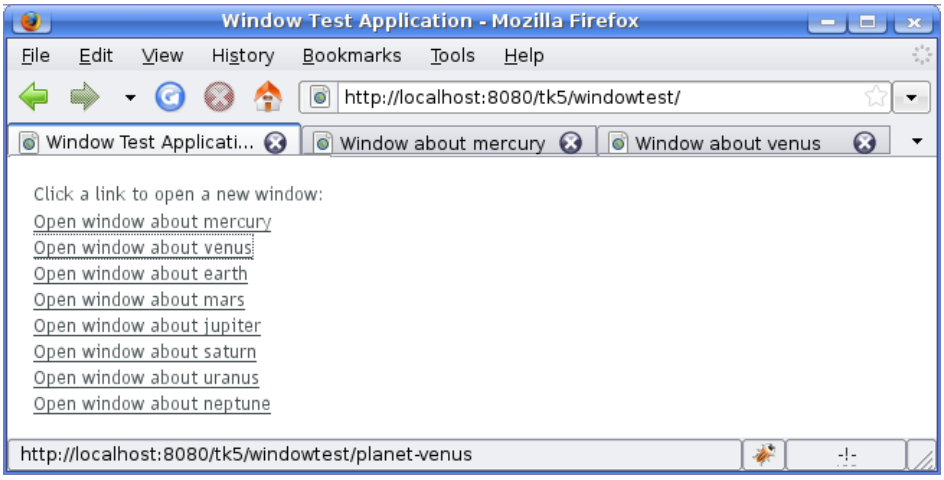


The application knows the windows it already has and can return them after the creation. The application also handles closing and destruction of application-level window objects, as discussed in Section 3.4.3, “Closing Windows”.

Such dynamic windows could be opened as in the following example:

```
public void init() {
    final Window main = new Window("Window Test Application");
    setMainWindow(main);

    // Have some IDs for the dynamically creatable windows.
    final String[] items = new String[] { "mercury", "venus", "earth",
            "mars", "jupiter", "saturn", "uranus", "neptune" };

    // Create a list of links to each of the available window.
    for (int i = 0; i < items.length; i++) {
        // Create a URL for the window.
        String windowUrl = getURL() + "planet-" + items[i];

        // Create a link to the window URL.
        // Using the item ID for the target also opens it in a new
        // browser window (or tab) unique to the window name.
        main.addComponent(new Link("Open window about " + items[i],
                                   new ExternalResource(windowUrl),
                                   items[i], -1, -1, Window.BORDER_DEFAULT));
    }
}
```

Figure 3.5. Opening Windows



3.4.3. Closing Windows

When the user closes an application-level window, the Client-Side Engine running in the browser will report the event to the server before the page is actually removed. You can catch the event with a **Window.CloseListener**, as is done in the example below.

```
newWindow.addListener(new Window.CloseListener() {
    @Override
    public void windowClose(CloseEvent e) {
        // Do something.
        System.out.println(e.getWindow().getName() + " was closed");

        // Add a text to the main window about closing. (This does
        // not update the main window.)
        getMainWindow().addComponent(
            new Label("Window '" + e.getWindow().getName() +
                      "' was closed."));
    }
});
```

Notice that the change to the server-side state of the main window (or another application-level window) does not refresh the window in the browser, so the change will be unseen until user interaction or polling refreshes the window. This problem and its dangers are discussed in Section 3.4.4, “Caveats in Using Multiple Windows” below.

The close event does not occur if the browser crashes or the connection is otherwise severed violently. In such a situation, the window object will be left hanging, which could become a resource problem if you allow the users to open many such application-level windows. The positive side is that the user can reconnect to the window using the window URL.

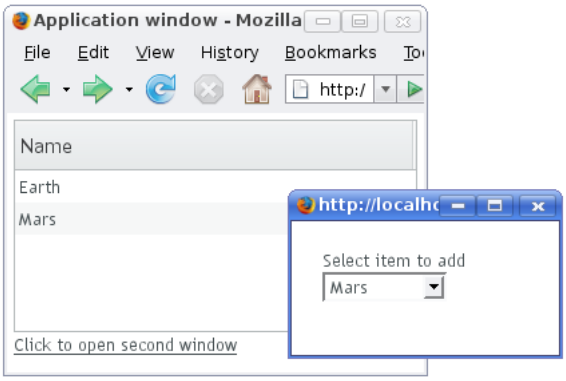
3.4.4. Caveats in Using Multiple Windows

Communication Between Windows

For cases where you need communication between windows, we recommend using floating child windows. In IT Mill Toolkit Release 5, an application window can not update the data in other windows. The contents of a window can only be updated when the particular window makes a request to the server. The request can be caused by user input or through polling.

Changing the server-side state of a window while processing a user event from another window can potentially cause serious problems. Changing the client-side state of a window does not always immediately communicate the changes to the server. The server-side state can therefore be out of sync with the client-side state.

Figure 3.6. Communication Between Two Application-Level Windows



The following example creates a second window that changes the contents of the main window, as illustrated in the figure above. In this simple case, changing the main window contents is safe.

```
// Create a table in the main window to hold items added in the second window
final Table table = new Table();
table.setPageLength(5);
table.getSize().setWidth(100, Size.UNITS_PERCENTAGE);
table.addContainerProperty("Name", String.class, "");
main.addComponent(table);

// Create the second window
final Window adderWindow = new Window("Add Items");
adderWindow.setName("win-adder");
main.getApplication().addWindow(adderWindow);

// Create selection component to add items to the table
final NativeSelect select = new NativeSelect("Select item to add");
select.setImmediate(true);
adderWindow.addComponent(select);

// Add some items to the selection
String items[] = new String[]{"-- Select --", "Mercury", "Venus",
        "Earth", "Mars", "Jupiter", "Saturn", "Uranus", "Neptune"};
for (int i=0; i<items.length; i++)
    select.addItem(items[i]);
select.setNullSelectionItemId(items[0]);

// When an item is selected in the second window, add
// table in the main window
select.addListener(new ValueChangeListener() {
    public void valueChange(ValueChangeEvent event) {
        // If the selected value is something else but null selection item.
        if (select.getValue() != null) {
            // Add the selected item to the table in the main window
            table.addItem(new Object[]{select.getValue()}, new Integer(table.size()));
        }
    }
});

// Link to open the selection window
Link link = new Link("Click to open second window",
                     new ExternalResource(adderWindow.getURL()),
```

```
                    "_new", 50, 200, Link.TARGET_BORDER_DEFAULT);
main.addComponent(link);

// Enable polling to update the main window
ProgressIndicator poller = new ProgressIndicator();
poller.addStyleName("invisible");
main.addComponent(poller);
```

The example uses an invisible **ProgressIndicator** to implement polling. This is sort of a trick and a more proper API for polling is under design. Making the progress indicator invisible requires the following CSS style definition:

```
.i-progressindicator-invisible {
 display: none;
}
```

3.5. Referencing Resources

Web applications work over the web and have various resources, such as images or downloadable files, that the web browser has to get from the server. These resources are typically used in **Embedded** (images) or **Link** (downloadable files) user interface components. Various components, such as **TabSheet**, can also include icons, which are also handled as resources.

A web server can handle many of such requests for static resources without having to ask them from the application, or the **Application** object can provide them. For dynamic resources, the user application must be able to create them dynamically. IT Mill Toolkit provides resource request interfaces for applications so that they can return various kinds of resources, such as files or dynamically created resources. These include the **StreamResource** class and URI and parameter handlers described in Section 9.3.1, “URI Handlers” and Section 9.3.2, “Parameter Handlers”, respectively.

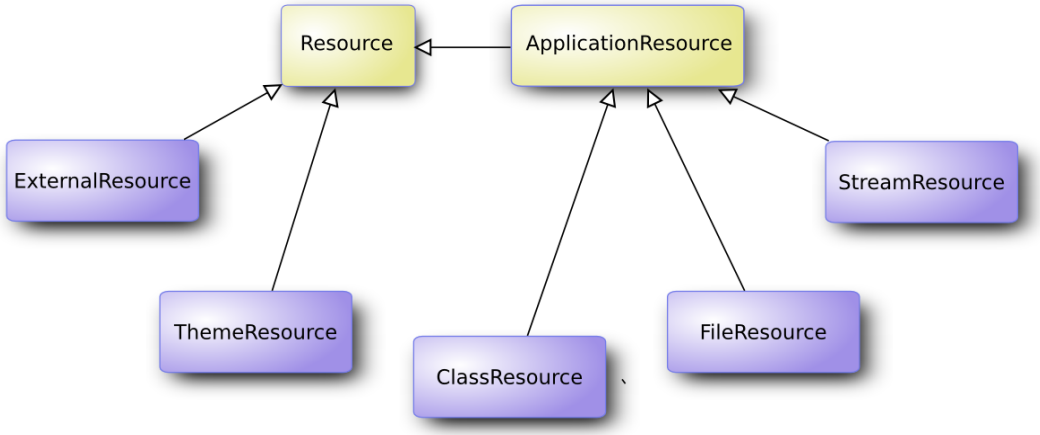
IT Mill Toolkit provides also low-level facilities for retrieving the URI and other parameters of a HTTP request. We will first look into how applications can provide various kinds of resources and then look into low-level interfaces for handling URIs and parameters to provide resources and functionalities.

Notice that using URI or parameter handlers to create "pages" is not meaningful in IT Mill Toolkit or in AJAX applications generally. Please see Section 9.2, “Special Characteristics of AJAX Applications” for a detailed explanation.

3.5.1. Resource Interfaces and Classes

IT Mill Toolkit has two interfaces for resources: a generic **Resource** interface and a more specific **ApplicationResource** interface for resources provided by the application.

Figure 3.7. Resource Interface and Class Diagram



ApplicationResource resources are managed by the **Application** class. When you create such a resource, you give the application object to the constructor. The constructor registers the resource in the application using the **addResource** method.

Application manages requests for the resources and allows accessing resources using a URI. The URI consists of the base name of the application and a relative name of the resource. The relative name is `"APP/"+resourceid+"/"+filename`, for example `"APP/1/myimage.png"`. The `resourceid` is a generated numeric identifier to make resources unique, and `filename` is the file name of the resource given in the constructor of its class. However, the application using a resource does not usually need to consider its URI. It only needs to give the resource to an appropriate **Embedded** or **Link** or some other user interface component, which manages the rendering of the URI.

3.5.2. File Resources

File resources are files stored anywhere in the file system. The use of file resources falls into two main categories: downloadable files and embedded images.

The file that can be retrieved using a file resource is defined with standard **java.io.File** class. You can create the file either with an absolute or relative path, but the base path of the relative path depends on the installation of the web server. For example, in Tomcat, the default current directory is the installation path of Tomcat.

3.5.3. Class Loader Resources

The **ClassResource** allows resources to be loaded from the deployed package of the application using Java Class Loader. The one-line example below loads an image resource from the application package and displays it in an **Embedded** component.

```
mainwindow.addComponent(new Embedded ("", new ClassResource("smiley.jpg",
mainwindow.getApplication())));
```

3.5.4. Theme Resources

Theme resources are files included in a theme, typically images. See Chapter 6, *Themes* for more information on themes.

3.5.5. Stream Resources

Stream resources are application resources that allow creating dynamic resource content. Charts are typical examples of dynamic images. To define a stream resource, you need to implement the **StreamResource.StreamSource** interface and its `getStream` method. The method needs to return an **InputStream** from which the stream can be read.

The following example demonstrates the creation of a simple image in PNG image format.

```
import java.awt.image.*;

public class MyImageSource implements StreamResource.StreamSource {
    ByteArrayOutputStream imagebuffer = null;
    int reloads = 0;

    /* Must implement this method that returns the resource as a stream.*/
    public InputStream getStream () {
        /* Create an image and draw something on it. */
        BufferedImage image = new BufferedImage (200, 200, BufferedImage.TYPE_INT_RGB);
        Graphics drawable = image.getGraphics();
        drawable.setColor(Color.lightGray);
        drawable.fillRect(0,0,200,200);
        drawable.setColor(Color.yellow);
        drawable.fillOval(25,25,150,150);
        drawable.setColor(Color.blue);
        drawable.drawRect(0,0,199,199);
        drawable.setColor(Color.black);
        drawable.drawString("Reloads="+reloads, 75, 100);
        reloads++;

        try {
            /* Write the image to a buffer. */
            imagebuffer = new ByteArrayOutputStream();
            ImageIO.write(image, "png", imagebuffer);

            /* Return a stream from the buffer. */
            return new ByteArrayInputStream(imagebuffer.toByteArray());
        } catch (IOException e) {
            return null;
        }
    }
}
```

The content of the generated image is dynamic, as it updates the reloads counter with every call. The **ImageIO**.`write()` method writes the image to an output stream, while we had to return an input stream, so we stored the image contents to a temporary buffer.

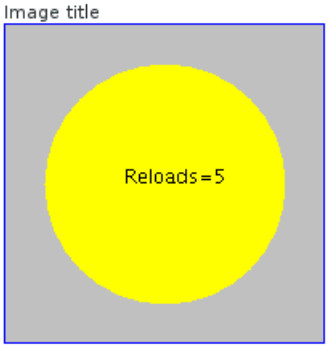
You can use resources in various ways. Some user interface components, such as **Link** and **Embedded**, take their parameters as a resource.

Below we display the image with the **Embedded** component. The **StreamResource** constructor gets a reference to the application and registers itself in the application's resources. Assume that `main` is a reference to the main window and `this` is the application object.

```
/* Create an instance of our stream source. */
StreamResource.StreamSource imagesource = new MyImageSource ();

/* Create a resource that uses the stream source and give it a name. The
 * constructor will automatically register the resource in the application. */
StreamResource imageresource = new StreamResource(imagesource, "myimage.png", this);

/* Create an embedded component that gets its contents from the resource. */
main.addComponent(new Embedded("Image title", imageresource));
```

The image will look as follows:

Figure 3.8. Screenshot of the stream resource example with an embedded image



We named the resource as `myimage.png`. The application adds a resource key to the file name of the resource to make it unique. The full URI will be like `http://localhost:8080/testbench/APP/1/myimage.png`. The end `APP/1/myimage.png` is the *relative* part of the URI. You can get the relative part of a resource's URI from the application with `Application.getRelativeLocation()`.

Another solution for creating dynamic content is an URI handler, possibly together with a parameter handler. See Section 9.3.1, “URI Handlers” and Section 9.3.2, “Parameter Handlers”.

3.6. Error Handling

3.6.1. Error Indicator and message

All components have a built-in error indicator that can be set explicitly with `setComponentError()` or can be turned on implicitly if validating the component fails. As with component caption, the placement of the indicator is managed by the layout in which the component is contained. Usually, the error indicator is placed right of the caption text. Hovering the mouse pointer over the field displays the error message.

The following example shows how you can set the component error explicitly. The example essentially validates field value without using an actual validator.

```
// Create a field.
final TextField textfield = new TextField("Enter code");
main.addComponent(textfield);

// Let the component error be initially clear. (It actually is by default.)
textfield.setComponentError(null);

// Have a button right of the field (and align it properly).
final Button button = new Button("Ok!");
main.addComponent(button);
((VerticalLayout)main.getLayout()).setComponentAlignment(button, Alignment.BOTTOM_LEFT);

// Handle button clicks
button.addListener(new Button.ClickListener() {
    public void buttonClick(ClickEvent event) {
        // If the field value is bad, set its error.
        // (Here the content must be only alphanumeric characters.)
        if (! ((String) textfield.getValue()).matches("^\\w*$")) {
            // Put the component in error state and set the error message.
            textfield.setComponentError(new UserError("Must be letters and numbers"));
        } else {
```

```
            // Otherwise clear it.
            textfield.setComponentError(null);
        }
    }
});
```

Figure 3.9. Error indicator active

The **Form** component handles and displays also the errors of its contained fields so that it displays both the error indicator and the message in a special error indicator area. See Section 4.15, "**Form**" and Section 4.15.3, "Validating Form Input" for details on the **Form** component and validation of form input.

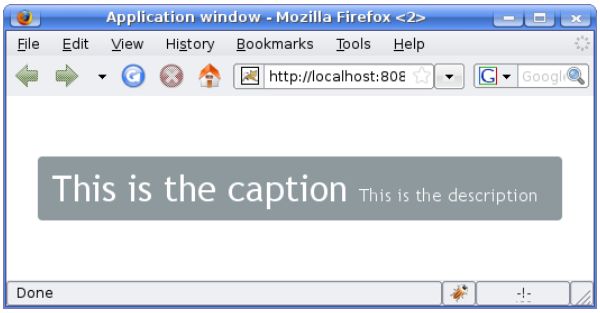
3.6.2. Notifications

Notifications are error or information boxes that appear typically at the center of the screen. A notification box has a caption and optional description and icon. The box stays on the screen either for a defined time or until the user clicks it. The notification type defines the default appearance and behaviour of a notification.

Notifications are always associated with a window object, which can be a child window (the positioning is always relative to the entire browser view). The **Window** class provides a `showNotification()` method for displaying notifications. The method takes the caption and an optional description and notification type as parameters. The method also accepts a notification object of type **Window.Notification**, as described further below.

```
mainwindow.showNotification("This is the caption",
                            "This is the description");
```

Figure 3.10. Notification



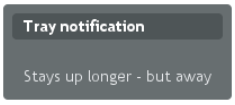
The caption and description are, by default, written on the same line. If you want to have a line break between them, use the XHTML line break markup "`<br/>`". You can use any XHTML markup in the caption and description of a notification.

```
main.showNotification("This is a warning",
                            "<br/>This is the <i>last</i> warning",
                            Window.Notification.TYPE_WARNING_MESSAGE);
```

Figure 3.11. Notification with Formatting

The notification type defines the overall default style and behaviour of a notification. If no notification type is given, the "humanized" type is used as the default. The notification types, listed below, are defined in the **Window.Notification** class.

Table 3.1. Types of Notifications

	TYPE_HUMANIZED_MESSAGE	A user-friendly message that does not annoy too much: it does not require confirmation by clicking and disappears very quickly. It is centered and has a neutral gray color.
	TYPE_WARNING_MESSAGE	Warnings are messages of medium importance. They are displayed with colors that are neither neutral nor too distractive. A warning is displayed for 1.5 seconds, but the user can click the message box to dismiss it. The user can continue to interact with the application while the warning is displayed.
	TYPE_ERROR_MESSAGE	Error messages are notifications that require the highest user attention, with alert colors and by requiring the user to click the message to dismiss it. The error message box does not itself include an instruction to click the message, although the close box in the upper right corner indicates it visually. Unlike with other notifications, the user can not interact with the application while the error message is displayed.
	TYPE_TRAY_NOTIFICATION	Tray notifications are displayed in the "system tray" area, that is, in the lower-right corner of the browser view. As they do not usually obsure any user interface, they are displayed longer than humanized or warning messages, 3 seconds by default. The user can continue to interact with the application normally while the tray notification is displayed.

All of the features of specific notification types can be controlled with the attributes of **Window.Notification**. You can pass an explicitly created notification object to the `showNotification()` method.

```
// Create a notification with the default settings for a warning.
Window.Notification notif = new Window.Notification(
        "Be warned!", "This message lurks in the top-left corner!",
        Window.Notification.TYPE_WARNING_MESSAGE);

// Set the position.
notif.setPosition(Window.Notification.POSITION_TOP_LEFT);

// Let it stay there until the user clicks it
notif.setDelayMsec(-1);
```

```
// Show it in the main window.
main.showNotification(notif);
```

The `setPosition()` method allows setting the positioning of the notification. The method takes as its parameter any of the constants:

`Window.Notification.POSITION_CENTERED`
`Window.Notification.POSITION_CENTERED_TOP`
`Window.Notification.POSITION_CENTERED_BOTTOM`
`Window.Notification.POSITION_TOP_LEFT`
`Window.Notification.POSITION_TOP_RIGHT`
`Window.Notification.POSITION_BOTTOM_LEFT`
`Window.Notification.POSITION_BOTTOM_RIGHT`

The `setDelayMSec()` allows you to set the time in milliseconds for how long the notification is displayed. Parameter value `-1` means that the message is displayed until the user clicks the message box. It also prevents interaction with other parts of the application window, as is default behaviour for error messages. It does not, however, add a close box that the error notification has.

3.7. Application Environment

While more and more server based frameworks, libraries, standards, and architectures for Java are invented to make the programmer's life easier, software deployment seems to get harder and harder. For example, Java Enterprise Beans tried to make the creation of persistent and networked objects easy and somewhat automatic, but the number of deployment descriptions got enormous. As IT Mill Toolkit lives in a Java Servlet container, it must follow the rules, but it tries to avoid adding extra complexity.

All IT Mill Toolkit applications are deployed as Java web applications, which can be packaged as WAR files. For a detailed tutorial on how web applications are packaged, please refer to any Java book that discusses Servlets. Sun has an excellent reference online on http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/WCC3.html [http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/WCC3.html] .

3.7.1. Creating Deployable WAR in Eclipse

To deploy the created application to a web server, you need to create a WAR package. Here we give the instructions for Eclipse.

Open project properties and first set the name and destination of the WAR file in Tomcat **Export to WAR settings** tab. Exporting to WAR is done by selecting **Export to WAR** from **Tomcat Project** in project context menu (just click calc with the right mouse button on **Package contents tree**).

3.7.2. Web Application Contents

The following files are required in a web application in order to run it.

Web application organization

`WEB-INF/web.xml` — This is the standard web application descriptor that defines how the application is organized. You can refer to any Java book about the contents of this file. Also see an example in Example 3.1, “web.xml”.

WEB-INF/lib/itmill-toolkit-5.0.0.jar	This is the IT Mill Toolkit library. It is included in the product package in lib directory.
Your application classes	You must include your application classes either in a JAR file in WEB-INF/lib or as classes in WEB-INF/classes
Your own theme files (OPTIONAL)	If your application uses a special theme (look and feel), you must include it in WEB-INF/lib/themes/themename directory.

3.7.3. Deployment Descriptor web.xml

The deployment descriptor is an XML file with the name web.xml in the WEB-INF directory of a web application. It is a standard component in Java EE describing how a web application should be deployed. The structure of the deployment descriptor is illustrated by the following example. You simply deploy applications as servlets implemented by the special com.itmill.toolkit.terminal.gwt.server.ApplicationServlet wrapper class. The class of the actual application is specified by giving the *application* parameter with the name of the specific application class to the servlet. The servlet is then connected to a URL in a standard way for Java Servlets.

Example 3.1. web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_ID" version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                        http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

    <servlet>
        <servlet-name>myservlet</servlet-name>
        <servlet-class>com.itmill.toolkit.terminal.gwt.server.ApplicationServlet
        </servlet-class>
        <init-param>
            <param-name>application</param-name>
            <param-value>MyApplicationClass</param-value>
        </init-param>
    </servlet>
    <servlet-mapping>
        <servlet-name>myservlet</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

The descriptor defines a servlet with name myservlet. The servlet class, **com.itmill.toolkit.terminal.gwt.server.ApplicationServlet**, is provided by IT Mill Toolkit framework and it should be the same for all IT Mill Toolkit projects. The servlet takes the class name **Calc** of the user application class as a parameter, including the full package path to the class. If the class is in the default package the package path is obviously not used.

The url-pattern is defined above as /*. This matches to any URL under the project context. We defined above the project context as myproject so the application URL will be http://localhost:8080/myproject/. If the project were to have multiple applications or servlets, they would have to be given different names to distinguish them. For example, url-pattern /myapp/* would match a URL such as http://localhost:8080/myproject/myapp/. Notice that the slash and the asterisk *must* be included at the end of the pattern.

Notice also that if the URL pattern is other than root `/*` (such as `/myapp/*`), you will also need to make a servlet mapping to `/ITMILL/*` (unless you are serving it statically as noted below). For example:

```
...
<servlet-mapping>
    <servlet-name>myservlet</servlet-name>
    <url-pattern>/myurl/*</url-pattern>
</servlet-mapping>

<servlet-mapping>
    <servlet-name>myservlet</servlet-name>
    <url-pattern>/ITMILL/*</url-pattern>
</servlet-mapping>
```

You do not have to provide the above `/ITMILL/*` mapping if you serve both the widget sets and (custom and default) themes statically in `WebContent/ITMILL/` directory. The mapping simply allows serving them dynamically from the IT Mill Toolkit JAR. Serving them statically is recommended for production environments as it is much faster.

For a complete example on how to deploy applications, see the demos included in the IT Mill Toolkit installation package, especially the `WebContent/WEB-INF` directory.

Deployment Descriptor Parameters

Deployment descriptor can have many parameters and options that control the execution of a servlet. You can find a complete documentation of the deployment descriptor in Java Servlet Specification at `http://java.sun.com/products/servlet/`.

By default, IT Mill Toolkit applications run in *debug mode*, which should be used during development. This enables various debugging features. For production use, you should have put in your `web.xml` the following parameter:

```
<context-param>
  <param-name>productionMode</param-name>
  <param-value>true</param-value>
  <description>IT Mill Toolkit production mode</description>
</context-param>
```

The parameter and the debug and production modes are described in detail in Section 9.1, “Debug and Production Mode”.

One often needed option is the session timeout. Different servlet containers use varying defaults for timeouts, such as 30 minutes for Apache Tomcat. You can set the timeout with:

```
<session-config>
    <session-timeout>30</session-timeout>
</session-config>
```

After the timeout expires, the `close()` method of the **Application** class will be called. You should implement it if you wish to handle timeout situations.

3.8. Embedding Applications in Web Pages

Many web applications and especially web sites are not all AJAX, but AJAX is used only for specific functionalities. In practice, many web applications are a mixture of dynamic web pages and AJAX applications embedded to such pages.

Embedding IT Mill Toolkit applications is easy. There are two basic ways to embed them. One is to have a `<div>` placeholder for the web application and load the IT Mill Toolkit Client-Side Engine with a simple JavaScript code. The second method is even easier, which is to simply use the `<iframe>` element. Both of these methods have advantages and disadvantages. The `<div>` method can only embed one application in a page, while the `<iframe>` method can embed as many as needed. One disadvantage of the `<iframe>` method is that the size of the `<iframe>` element is not flexible according to the content while the `<div>` method allows such flexibility. The following sections look closer into these two embedding methods.

3.8.1. Embedding Inside a Element

The loading code for the Client-Side Engine has changed in IT Mill Toolkit version 5.1.2 and the explanation below is no longer compatible with 5.1.2 and later versions. Please view the source code of the initial page of your application in your browser or see the `WebContent/multiapp.html` for an example.

You can embed an IT Mill Toolkit application inside a web page with a method that is equivalent to loading the initial page content from the application servlet in a non-embedded application. Normally, the **ApplicationServlet** servlet generates an initial page that contains the correct parameters for the specific application. You can easily configure it to load multiple IT Mill Toolkit applications on the same page, assuming that they use the same widget set.

You can view the initial page for your application easily simply by opening the application in a web browser and viewing the HTML source code. You could just copy and paste the embedding code from the default initial page. It has, however, some extra functionality that is not normally needed: it generates some of the script content with `document.write()` calls, which is useful only when you are running the application as a portlet in a portal. The method outlined below is much simpler.

The `WebContent/multiapp.html` file included in the IT Mill Toolkit installation package provides an example of embedding (multiple) IT Mill Toolkit applications in a page. After launching the demo application, you can view the example at URL `http://localhost:8888/multiapp.html`. Notice that the example assumes the use of root context for the applications (`/`).

Embedding requires four elements inside the HTML document:

1. In the `<head>` element, you need to define the application URI and parameters and load the IT Mill Toolkit Client-Side Engine. The `itmill` variable is an associative map that can contain various runtime data used by the Client-Side Engine of IT Mill Toolkit. The `toolkitConfigurations` item is itself an associate map that contains parameters for each of the applications embedded in the page. The map must contain the following items:

Table 3.2. toolkitConfigurations parameters

appUri	The application URI consists of the context and the application path. If the context is `/mycontext` and the application path is `myapp`, the *`appUri`* would be `/mycontext/myapp`. The `multiapp.html` example assumes the use of root context, which is used in the demo application.
pathInfo	The *`PATHINFO`* parameter for the Servlet.
themeUri	URI of the application theme. The URI must include application context and the path to the theme directory. Themes are, by default, stored under the `/ITMILL/themes/` path.
versionInfo	This item is itself an associative map that contains two parameters: *`toolkitVersion`* contains the version number of the IT Mill Toolkit version used by the application. The *`applicationVersion`* parameter contains the version of the particular application.

The following example defines two applications to run in the same window: the Calculator and Hello World examples. In the example, the application context is `/tk5`.

```
<script type="text/javascript">
    var itmill = {
        toolkitConfigurations: {
            'calc': {
                appUri:'/tk5/Calc',
                pathInfo: '/',
                themeUri: '/tk5/ITMILL/themes/example',
                versionInfo : {
                    toolkitVersion:"5.9.9-INTERNAL-NONVERSIONED-DEBUG-BUILD",
                    applicationVersion:"NONVERSIONED"
                }
            },
            'hello': {
                appUri:'/tk5/HelloWorld',
                pathInfo: '/',
                themeUri: '/tk5/ITMILL/themes/example',
                versionInfo : {
                    toolkitVersion:"5.9.9-INTERNAL-NONVERSIONED-DEBUG-BUILD",
                    applicationVersion:"NONVERSIONED"
                }
            }
        }};
</script>
```

2. Loading the IT Mill Toolkit Client-Side Engine is done with the following kind of line in the `<head>` element:

   ```
   <script language='javascript'
       src='/itmill-toolkit-examples/ITMILL/widgetsets/com.itmill.toolk
   it.terminal.gwt.DefaultWidgetSet/com.itmill.toolkit.terminal.gwt.DefaultWi
   dgetSet.nocache.js'></script>
   ```

   The engine URI consists of the context of the web application, `itmill-toolkit-examples` above, followed by the path to the JavaScript (`.js`) file of the widget set, relative to the `WebContent` directory. The file contains the Client-Side Engine compiled for the particular widget set. The line above assumes the use of the default widget set of IT Mill Toolkit. If you have made custom widgets that are defined in a custom widget set, you need to use the path to the compiled widget set file. Widget sets must be compiled under the `WebContent/ITMILL/widgetsets` directory.

3. In the `<html>` element, you need to do a routine inclusion of GWT history `iframe` element as follows:

   ```
   <iframe id="__gwt_historyFrame" style="width:0;height:0;border:0"></iframe>
   ```

4. The location of the IT Mill Toolkit application is defined with a `div` placeholder element having `id="itmill-ajax-window"` as follows:

   ```
   <div id="itmill-ajax-window"/>
   ```

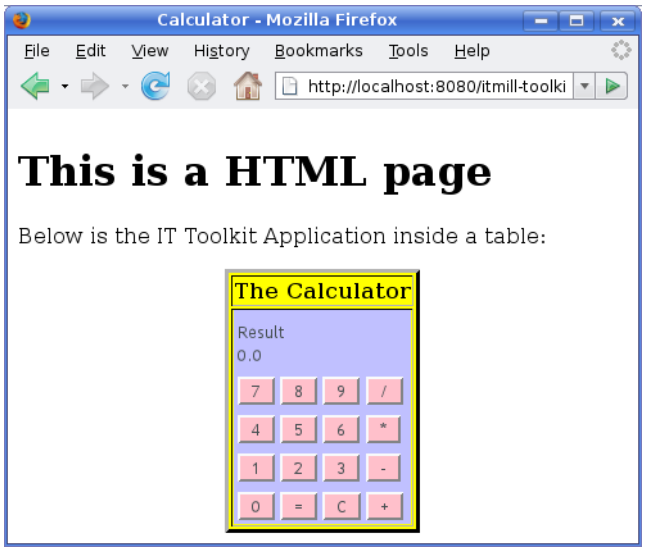
Below is a complete example of embedding an application. It works out-of-the-box with the Calculator demo application.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
  <head>
    <title>Embedding Example</title>
```

```
    <!-- Set parameters for the IT Mill Toolkit Client-Side Engine. -->
    <script type="text/javascript">
     var itmill = {appUri:'Calc', pathInfo: '/'};
    </script>

    <!-- Load the IT Mill Toolkit Client-Side Engine. -->
    <script language='javascript' src='/itmill-toolkit-examples/ITMILL/widgetse
ts/com.itmill.toolkit.terminal.gwt.DefaultWidgetSet/com.itmill.toolkit.terminal
.gwt.DefaultWidgetSet.nocache.js'></script>

    <!-- We can stylize the web application. -->
    <style>
        #itmill-ajax-window {background: #c0c0ff;}
        .i-button {background: pink;}
    </style>
  </head>

  <body>
    <!-- This <iframe> element is required by GWT. -->
    <iframe id="__gwt_historyFrame" style="width:0;height:0;border:0"></iframe>

    <h1>This is a HTML page</h1>
 <p>Below is the IT Toolkit Application inside a table:</p>
 <table align="center" border="3" style="background: yellow;">
   <tr><th>The Calculator</th></tr>
   <tr>
     <td>
          <!-- Placeholder <div> for the IT Mill Toolkit application -->
          <div id="itmill-ajax-window"/>
        </td>
      </tr>
    </table>
  </body>
</html>
```

The page will look as follows:

Figure 3.12. Embedded Application



You can style the web application with themes as described in Chapter 6, *Themes*. The Client-Side Engine loads the style sheets required by the application. In addition, you can do styling in the embedding page, as was done in the example above.

The Reservation Demo and Windowed Demos provide similar examples of embedding an application in a web page. The embedding web pages are `WebContent/reservr.html` and `WebContent/windoweddemos.html`, respectively.

The disadvantage of this embedding method is that there can only be one web application embedded in a page. One is usually enough, but if it is not, you need to use the `<iframe>` method below.

3.8.2. Embedding Inside an <iframe> Element

Embedding an IT Mill Toolkit application inside an `<iframe>` element is even easier than the method described above, as it does not require definition of any IT Mill Toolkit specific definitions. The use of `<iframe>` makes it possible to embed multiple web applications or two different views to the same application on the same page.

You can embed an application with an element such as the following:

```
<iframe src="/itmill-toolkit-examples/Calc"></iframe>
```

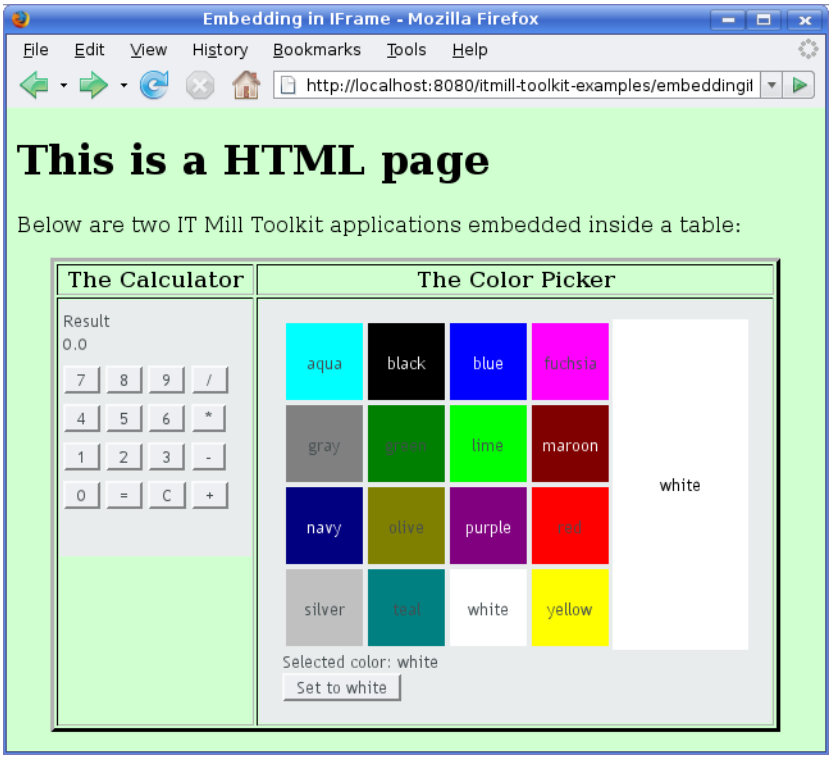
The problem with `<iframe>` elements is that their size of is not flexible depending on the content of the frame, but the content must be flexible to accommodate in the frame. You can set the size of an `<iframe>` element with `height` and `width` attributes.

Below is a complete example of using the `<iframe>` to embed two applications in a web page.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
  <head>
    <title>Embedding in IFrame</title>
  </head>

  <body style="background: #d0ffd0;">
    <h1>This is a HTML page</h1>
    <p>Below are two IT Mill Toolkit applications embedded inside a table:</p>

    <table align="center" border="3">
      <tr>
        <th>The Calculator</th>
        <th>The Color Picker</th>
      </tr>
      <tr valign="top">
        <td>
          <iframe src="/itmill-toolkit-examples/Calc" height="200"
                  width="150" frameborder="0"></iframe>
        </td>
        <td>
          <iframe src="/itmill-toolkit-examples/colorpicker" height="330" width="400"
                  frameborder="0"></iframe>
        </td>
      </tr>
    </table>
  </body>
</html>
```

The page will look as shown in Figure 3.13, “IT Mill Toolkit Applications Embedded Inside IFrames” below.

Figure 3.13. IT Mill Toolkit Applications Embedded Inside IFrames



Chapter 4. User Interface Components

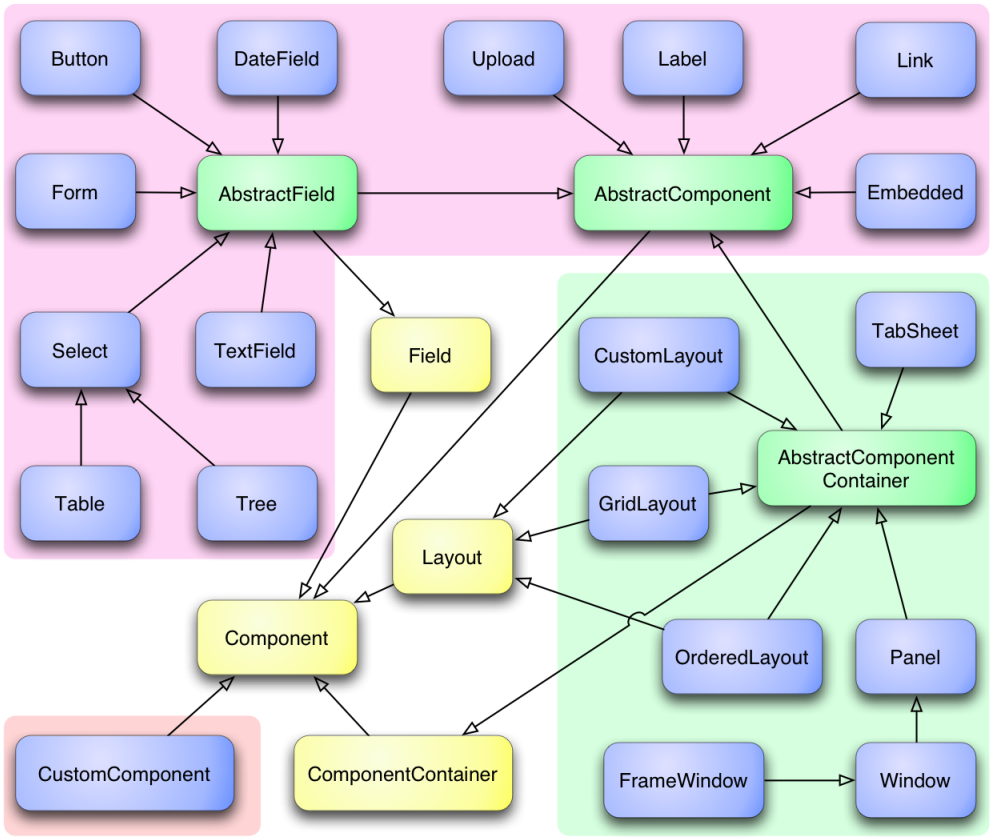
This chapter provides an overview and a detailed description of all non-layout components in IT Mill Toolkit. Many features of the components are designed for various use cases and design patterns, which are mentioned below.

As most components are used inside a layout, you should familiarize yourself with the layout components in Chapter 5, *Managing Layout*.

4.1. Overview

IT Mill Toolkit provides a comprehensive set of user interface components and allows you to define custom components. Figure 4.1, “UI Component Inheritance Diagram” illustrates the inheritance hierarchy of the UI component classes and interfaces.

Figure 4.1. UI Component Inheritance Diagram



Interfaces are displayed in yellow, abstract classes in red, and regular classes in blue. At the bottom of the interface hierarchy, we have the **Component** interface. At the bottom of the class hierarchy, we have the **AbstractComponent** class. It is inherited by two other abstract classes: **AbstractField**, inherited further by field components and **AbstractComponentContainer**, inherited by various container components. Some miscellaneous components, such as labels and links, inherit this base class directly.

The layout of the various components in a window is controlled, logically, by layout components, just like in conventional Java UI toolkits for desktop applications. In addition, with the **CustomLayout** component, you can write a custom layout as an XHTML template that includes the locations of any contained components. Looking at the inheritance diagram, we can see that layout components inherit the **AbstractComponentContainer** and the **Layout** interface. Layout components are described in detail in Chapter 5, *Managing Layout*.

Looking at it from the perspective of an object hierarchy, we would have a **Window** object, which contains a hierachy of layout components, which again contain other layout components, field components, and other visible components.

You can browse the available UI components in the Feature Browser of the IT Mill Toolkit Demo Application. The Feature Browser shows a description, a list of properties, JavaDoc documentation, and a code sample for each of the components. On the right side of the screen, you can find the Properties panel, which you can use to edit the properties of the displayed component.

4.2. Label

Label is a text component that you can use to display non-editable text. The text will wrap around if the width of the containing component limits the length of the lines, except for the preformatted text.

```
/* Some container for the Label. */
Panel panel = new Panel("Panel Containing a Label");
main.addComponent(panel);

panel.addComponent(new Label("This is a Label inside a Panel. There is enough " +
                             "text in the label to make the text wrap if it " +
                             "exceeds the width of the panel."));
```

Figure 4.2. The Label Component

The contents of a label are formatted depending on the content mode. By default, the text is assumed to be plain text and any contained XML-specific characters will be quoted appropriately to allow rendering the contents of a label in XHTML in a web browser. The content mode can be set in the constructor or with `setContentMode()`, and can have the following values:

Table 4.1. Content Modes for Label

CONTENT_DEFAULT	The default content mode is CONTENT_TEXT (see below).
CONTENT_PREFORMATTED	Content mode, where the label contains preformatted text. It will be, by default, rendered with a fixed-width typewriter font. Preformatted text can contain line breaks, written in Java with the \n escape sequence for a newline character (ASCII 0x0a), or tabulator characters written with \t (ASCII 0x08).
CONTENT_RAW	Content mode, where the label contains raw text. Output is not required to be valid XML. It can be, for example, HTML, which can be unbalanced or otherwise invalid XML. The example below uses the tag in HTML. While XHTML should be preferred in most cases, this can be useful for some specific purposes where you may need to display loosely formatted HTML content. The raw mode also preserves character entities, some of which might otherwise be interpreted incorrectly.
CONTENT_TEXT	Content mode, where the label contains only plain text. All characters are allowed, including the special <, >, and & characters in XML or HTML, which are quoted properly in XHTML while rendering the component. This is the default mode.
CONTENT_UIDL	Formatted content mode, where the contents are XML that is restricted to UIDL 1.0, the internal language of IT Mill Toolkit for AJAX communications between the server and the browser. Obsolete since IT Mill Toolkit 5.0.
CONTENT_XHTML	Content mode, where the label contains XHTML. The content will be enclosed in a DIV element having the namespace "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd".
CONTENT_XML	Content mode, where the label contains well-formed and well-balanced XML. Each of the root elements must have their default namespace specified.

Warning

Notice that the validity of XML or XHTML in a Label is not checked in the server during rendering of the component and any errors can result in an error in the browser! You should validate the content before displaying it in the component, especially if it comes from an uncertain source.

The following example demonstrates the use of **Label** in different modes.

```
GridLayout labelgrid = new GridLayout (2,1);
labelgrid.addComponent (new Label ("CONTENT_DEFAULT"));
labelgrid.addComponent (new Label ("This is a label in default mode: <plain text>",
                                   Label.CONTENT_DEFAULT));
labelgrid.addComponent (new Label ("CONTENT_PREFORMATTED"));
labelgrid.addComponent (new Label ("This is a preformatted label.\n"+
                                   "The newline character \\n breaks the line.",
                                   Label.CONTENT_PREFORMATTED));
labelgrid.addComponent (new Label ("CONTENT_RAW"));
labelgrid.addComponent (new Label ("This is a label in raw mode.<br>It can contain, "+
                                   "for example, unbalanced markup.",
                                   Label.CONTENT_RAW));
labelgrid.addComponent (new Label ("CONTENT_TEXT"));
labelgrid.addComponent (new Label ("This is a label in (plain) text mode",
                                   Label.CONTENT_TEXT));
labelgrid.addComponent (new Label ("CONTENT_XHTML"));
labelgrid.addComponent (new Label ("<i>This</i> is an <b>XHTML</b> formatted label",
                                   Label.CONTENT_XHTML));
labelgrid.addComponent (new Label ("CONTENT_XML"));
labelgrid.addComponent (new Label ("This is an <myelement>XML</myelement> formatted "+
                                   "label",
```

```
                                        Label.CONTENT_XML));
main.addComponent(labelgrid);
```

The rendering will look as follows:

Figure 4.3. Label Modes Rendered on Screen

CONTENT_DEFAULT	This is a label in default mode: <plain text>
CONTENT_PREFORMATTED	This is a preformatted label. The newline character \n breaks the line.
CONTENT_RAW	This is a label in raw mode. It can contain, for example, unbalanced markup.
CONTENT_TEXT	This is a label in (plain) text mode
CONTENT_XHTML	*This* is an **XHTML** formatted label
CONTENT_XML	This is an XML formatted label

Using the XHTML, XML, or raw modes allow inclusion of, for example, images within the text flow, which is not possible with any regular layout components. The following example includes an image within the text flow, with the image coming from a class loader resource.

```
ClassResource labelimage = new ClassResource ("labelimage.jpg", this);
main.addComponent(new Label("Here we have an image <img src=\"" +
                            this.getRelativeLocation(labelimage) + "\"/> within text.",
                            Label.CONTENT_XHTML));
```

When you use a class loader resource, the image has to be included in the JAR of the web application. In this case, the `labelimage.jpg` needs to be in the default package. When rendered in a web browser, the output will look as follows:

Figure 4.4. Referencing An Image Resource in Label

Another solution would be to use the **CustomLayout** component, where you can write the component content as an XHTML fragment in a theme, but such a solution may be too heavy for most cases, and not flexible enough if the content needs to be dynamically generated.

Notice that the rendering of XHTML depends on the assumption that the client software and the terminal adapter are XHTML based. It is possible to write a terminal adapter for a custom thin client application, which may not be able to render XHTML at all. There are also differences between web browsers in their support of XHTML.

4.3. Link

The **Link** component allows making references to resources that are either external or provided by the web server or by the application itself. While a **Link** appears like a hyperlink, it is not handled in the web browser. When a user clicks a link, the server receives an event and typically opens the referenced resource in the target window of the link. Resources are explained in Section 3.5, “Referencing Resources”.

Links to external resources can be made by using a URI as follows:

```
main.addComponent(new Link ("link to a resource",
                            new ExternalResource("http://www.itmill.com/")));
```

With the simple contructor used in the above example, the link is opened in the current window. Using the constructor that takes the target window as a parameter, or by setting the window with `setWindow`, you can open the resource in another window, such as a native popup window or a **FrameWindow**. As the target window can be defined as a target string managed by the browser, the target can be any window, including windows not managed by the application itself.

When the user clicks the link, the application will receive an event regarding the click and handle it to provide the resource. The link is therefore not an `<a href>` element in HTML and it does not have an URI. This has some additional consequences, such as that a link can not be marked as "visited" by the browser, unlike normal hyperlinks. If you wish to have an actual HTML anchor element, you need to customize the rendering of the component or use a **Label** with XHTML content mode and write the anchor element by yourself.

CSS Style Rules

The **Link** component has *`i-link`* style by default.

```
.i-link { }
```

When the mouse pointer hovers over the link, it will also have the `over` style.

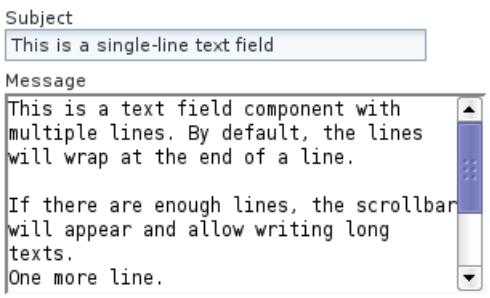
4.4. TextField

TextField is one of the most common user interface components and is highly versatile. It supports both single- and multi-line editing, password input, and buffering.

The following example creates two simple text fields: a single-line and a multi-line **TextField**.

```
/* Add a single-line text field. */
TextField subject = new TextField("Subject");
subject.setColumns(40);
main.addComponent(subject);

/* Add a multi-line text field. */
TextField message = new TextField("Message");
message.setRows(7);
message.setColumns(40);
main.addComponent(message);
```

Figure 4.5. Single- and Multi-Line Text Field Example



Notice how font size affects the width of the text fields even though the width was set with the same number of columns. This is a feature of HTML.

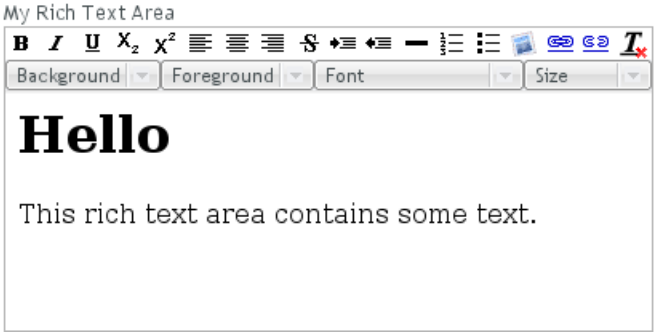
4.5. Rich Text Area

The **RichTextArea** field allows entering or editing formatted text. The toolbar provides all basic editing functionalities. The text content of **RichTextArea** is represented in HTML format. **RichTextArea** inherits **TextField** and does not add any API functionality over it. You can add new functionality by extending the client-side components **IRichTextArea** and **IRichTextToolbar**.

As with **TextField**, the textual content of the rich text area is the **Property** of the field and can be set with `setValue()` and read with `getValue()`.

```
// Create a rich text area
final RichTextArea rtarea = new RichTextArea();
rtarea.setCaption("My Rich Text Area");

// Set initial content as HTML
rtarea.setValue("<h1>Hello</h1>\n<p>This rich text area contains some text.</p>");
```

Figure 4.6. Rich Text Area Component



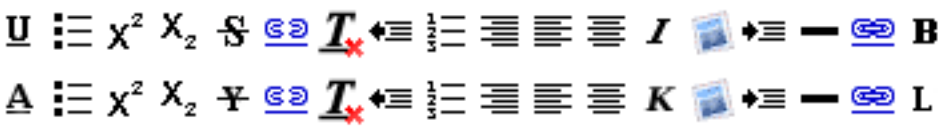
Above, we used context-specific tags such as `<h1>` in the initial HTML content. The rich text area component does not allow creating such tags, only formatting tags, but it does preserve them unless the user edits them away. Any non-visible whitespace such as the new line character (`\n`) are removed from the content. For example, the value set above will be as follows when read from the field with `getValue()`:

```
<h1>Hello</h1> <p>This rich text area contains some text.</p>
```

The rich text area is one of the few components in IT Mill Toolkit that contain textual labels. The selection boxes in the toolbar are in English, and not be localized currently otherwise but by inheriting or reimplementing the client-side **IRichTextToolbar** widget. The buttons can be localized simply with CSS by downloading a copy of the toolbar background image, editing it, and replacing the default toolbar. The toolbar is a single image file from which the individual button icons are picked, so the order of the icons is different from the rendered. The image file depends on the client-side implementation of the toolbar.

```
.i-richtextarea-richtextexample .gwt-ToggleButton .gwt-Image {
  background-image: url(img/richtextarea-toolbar-fi.png) !important;
}
```

Figure 4.7. Regular English and a Localized Rich Text Area Toolbar



CSS Style Rules

```
.i-richtextarea { }
.i-richtextarea .gwt-RichTextToolbar { }
.i-richtextarea .gwt-RichTextArea { }
```

The rich text area consists of two main parts: the toolbar with overall style `.gwt-RichTextToolbar` and the editor area with style `.gwt-RichTextArea`. The editor area obviously contains all the elements and their styles that the HTML content contains. The toolbar contains buttons and drop-down list boxes with the following respective style names:

```
.gwt-ToggleButton { }
.gwt-ListBox { }
```

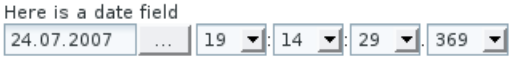
4.6. Date and Time Input

The **DateField** component provides the means to display andinput date and time. The field comes in two variations: **PopupDateField** with numeric input fields and a popup calendar view and **InlineDateField** with the calendar view always visible and the numeric input fields only for time. The **DateField** base class defaults to the popup variation.

The example below illustrates the use of the **DateField** with the default style. We set the time of the **DateField** to current time with the default constructor of the **java.util.Date** class.

```
/* Create a DateField with the default style. */
DateField date = new DateField();

/* Set the date and time to present. */
date.setValue(new java.util.Date());
```

Figure 4.8. Example of the Date Field with Default Style



The default style provides date input using a text box for the date and combo boxes for the time, down to milliseconds. Pressing the "..." button right of the date opens a month view for selecting the date.

You probably will not need milliseconds in most applications, and might not even need the time, but just the date. The visibility of the input components is controlled by *resolution* of the field which can be set with `setResolution()` method. The method takes as its parameters the lowest visible component, typically *`RESOLUTION_DAY`* for just dates and *`RESOLUTION_MIN`* for dates with time in hours and minutes. Please see the API Reference for a complete list of resolution parameters.

4.6.1. Calendar

The *`calendar`* style of the **DateField** provides a date picker component with a month view, just like the one in the default style that opens by clicking the "..." button. The user can navigate months and years by clicking the appropriate arrows.

```
/* Create a DateField with the calendar style. */
DateField date = new DateField("Here is a calendar field");
date.setStyle("calendar");

/* Set the date and time to present. */
date.setValue(new java.util.Date());

main.addComponent(date);
```

Figure 4.9. Example of the Date Field with Calendar Style

4.6.2. DateField Locale

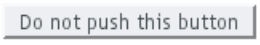
The date fields use the locale set for the component, which defaults to the system locale. You can set a custom locale with the `setLocale()` method of **AbstractComponent**.

4.7. Button

The **Button** is the primary user interface component that is normally used for finalizing input and initiating some action. When the user clicks a button, a **Button.ClickEvent** is emitted. A listener that inherits the **Button.ClickListener** interface can handle clicks with the `buttonClick()` method.

```
public class TheButton extends CustomComponent implements Button.ClickListener {
    Button thebutton;

    public TheButton() {
        /* Create a Button with the given caption. */
        thebutton = new Button ("Do not push this button");

        /* Listen for ClickEvents. */
        thebutton.addListener(this);

        setCompositionRoot(thebutton);
    }

    /** Handle button click events from the button. */
    public void buttonClick (Button.ClickEvent event) {
        thebutton.setCaption ("Do not push this button again");
    }
}
```

Figure 4.10. An Example of a Button



As a user interface often has several buttons, you can differentiate between them either by comparing the **Button** object reference returned by the `getButton()` method of **Button.ClickEvent** to a kept reference or by using a separate listener method for each button. The listening object and method can be given to the constructor. For a detailed description of these patterns together with some examples, please see Section 2.4, “Events and Listeners”.

CSS Style Rules

```
.i-button { }
```

The exact CSS style name can be different if a **Button** has the *`switchMode`* attribute enabled. See the alternative CSS styles below.

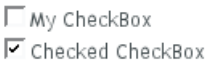
4.8. Check Box

Check box is a two-state selection component that can be either checked or unchecked. The caption of the check box will be placed right of the actual check box. IT Mill Toolkit provides two ways to create check boxes: individual check boxes with the **CheckBox** component described in this section and check box groups with the **OptionGroup** component in multiple selection mode, as described in Section 4.9.3, "Radio Button and Check Box Groups with **OptionGroup**".

Clicking on a check box will change its state. The state is the **Boolean** property of the **Button**, and can be set with `setValue()` and obtained with `getValue()` method of the **Property** interface. Changing the value of a check box will cause a **ValueChangeEvent**, which can be handled by a **ValueChangeListener**.

```
/* A check box with default state (not checked, i.e., false). */
final CheckBox checkbox1 = new CheckBox("My CheckBox");
main.addComponent(checkbox1);

/* Another check box with explicitly set checked state. */
final CheckBox checkbox2 = new CheckBox("Checked CheckBox");
checkbox2.setValue(true);
main.addComponent(checkbox2);

/* Make some application logic. We use anynymous listener classes here.
 * The above references were defined as "final" to allow accessing them
 * from inside anonymous classes. */
checkbox1.addListener(new ValueChangeListener() {
    public void valueChange(ValueChangeEvent event) {
        /* Copy the value to the other checkbox. */
        checkbox2.setValue(checkbox1.getValue());
    }
});
checkbox2.addListener(new ValueChangeListener() {
    public void valueChange(ValueChangeEvent event) {
        /* Copy the value to the other checkbox. */
        checkbox1.setValue(checkbox2.getValue());
    }
});
```

Figure 4.11. An Example of a Check Box



For an example on the use of check boxes in a table, see Section 4.10, "**Table**".

CSS Style Rules

```
.i-checkbox { }
```

4.9. Selecting Items

IT Mill Toolkit provides several alternative choices for selecting one or more items from a list. The selection components allow selecting one or more items from a list of items. The items are **Item** objects contained in a **Container**. The choices are based on the **AbstractSelect** base class.

The following selection classes are available:

Table 4.2. Selection Components

Select	Provides a drop-down list for single selection and a multi-line list in multiselect mode.
NativeSelect	Provides selection using the native selection component in the browser, typically a drop-down list for single selection and a multi-line list in multiselect mode. This uses the `<select>` element in HTML.
OptionGroup	Shows the items as a vertically arranged group of radio buttons in the single selection mode and of check boxes in multiple selection mode.
TwinColSelect	Shows two list boxes side by side where the user can select items from a list of available items and move them to a list of selected items using control buttons.

In addition, the **Tree** and **Table** components allow special forms of selection. They also inherit the **AbstractSelect**.

The selection components provide the current selection as an item identifier from the **Property** interface of the component, that is, as the value of the component. You can get the value, which is an item identifier object, with `getValue()` of the **Property** interface. In multiselect mode, the property will be an unmodifiable set of item identifiers. If no item is selected, the property will be *`null`* in single selection mode or an empty collection in multiselect mode.

New items are added with the `addItem()` method, implemented for the **Container** interface. The method takes the *item identifier* (IID) object as a parameter, and by default uses the identifier also as the caption of the item. The identifier is typically a **String**. The `addItem()` method also creates an empty **Item**, which itself has little relevance in the **Select** component, as the properties of an item are not used in any way by the component.

```
/* Create a Select component and add it to a layout. */
Select select = new Select ("Select something here");
main.addComponent(select);

/* Fill the component with some items. */
final String[] planets = new String[] {"Mercury", "Venus", "Earth", "Mars",
                                       "Jupiter", "Saturn", "Uranus", "Neptune"};
for (int i=0; i<planets.length; i++)
    select.addItem(planets[i]);
```

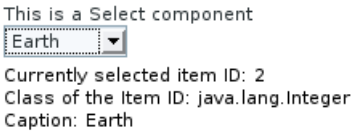
We could as well have added the item identifiers as integers, for example, and set the captions explicitly.

The **Select** and **NativeSelect** components will show "-" selection when no actual item is selected. This is the *null selection item identifier*. You can set an alternative ID with `setNullSelectionItemId()`. Setting the alternative null ID is merely a visual text; the `getValue()` will still return *`null`* value if no item is selected, or an empty set in multiselect mode.

The item identifier of the currently selected item will be set as the property of the **Select** object. You can access it with the `getValue` method of the **Property** interface of the component. Also, when handling changes in a **Select** component with the **Property.ValueChangeListener** interface, the **Prop-**

erty.ValueChangeEvent will have the selected item as the property of the event, accessible with the `getProperty` method.

Figure 4.12. Retrieval of the Currently Selected Item



The item and its identifier can be of any object type. The caption of the items can be retrieved from various sources, as defined with the caption mode of the component, which you can set with the `setItemCaptionMode()` method. The default mode is *`ITEM_CAPTION_MODE_EXPLICIT_DEFAULTS_ID`*. In addition to a caption, an item can have an icon. The icon of an item is set with `setItemIcon()`.

Table 4.3. Caption Modes for Selection Components

`ITEM_CAPTION_MODE_EXPLICIT_DEFAULTS_ID`	This is the default caption mode and its flexibility allows using it in most cases. By default, the item identifier will be used as the caption. The caption is retrieved with `toString()` method of the item identifier object. If the caption is specified explicitly with `setItemCaption()`, it overrides the item identifier.
`ITEM_CAPTION_MODE_EXPLICIT`	Captions must be explicitly specified with `setItemCaption()`. If they are not, the caption will be empty. Such items with empty captions will nevertheless be displayed in the **Select** component as empty rows. If they have an icon, they will be visible.
`ITEM_CAPTION_MODE_ICON_ONLY`	Only icons are shown, captions are hidden. Notice that icons are not supported in the themes in IT Mill Toolkit version 4 (see below).

`ITEM_CAPTION_MODE_ID`	String representation of the item identifier object is used as caption. This is useful when the identifier is actually an application specific object. For example:
	`class Planet extends Object {`
	`String planetName;`
	`Planet (String name) {`
	`planetName = name;`
	`}`
	`public String toString () {`
	`return "The Planet " + planetName;`
	`}`
	`}`
	`...`
	`SelectExample (Application application) {`
	`...`
	`for (int i=0; i<planets.length; i++)`
	`select.addItem(new Planet(planets[i]));`
	`...`
	`}`

`ITEM_CAPTION_MODE_INDEX`	Index number of item is used as caption. This caption mode is applicable only to data sources that implement the **Container.Indexed** interface. If the interface is not available, the component will throw a **ClassCastException**. The **Select** component itself does not implement this interface, so the mode is not usable without a separate data source. An **IndexedContainer**, for example, would work.

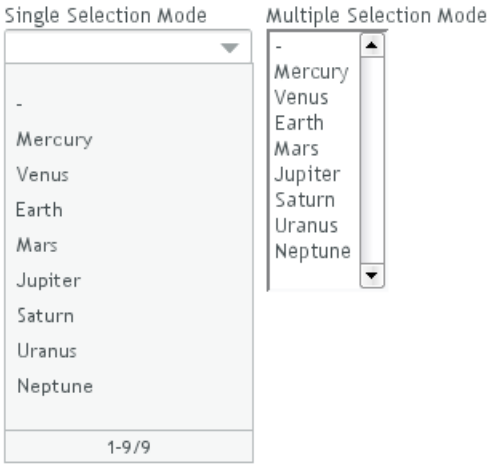
`ITEM_CAPTION_MODE_ITEM`	**String** representation of item, acquired with `toString()`, is used as the caption. This is applicable mainly when using a custom **Item** class, which also requires using a custom **Container** that is used as a data source for the **Select** component.
`ITEM_CAPTION_MODE_PROPERTY`	Item captions are read from the **String** representation of the property with the identifier specified with `setItemCaptionPropertyId()`. This is useful, for example, when you have a **Table** component that you use as the data source for the **Select**, and you want to use a specific table column for captions.

Notice that while the **Select** component allows associating an icon with each item with `setItemIcon()`, the icons are not supported in the themes in IT Mill Toolkit version 4. This is because HTML does not support images inside `select` elements. Icons are also not really visually applicable for *`optiongroup`* and *`twincol`* styles.

4.9.1. Basic Select Component

The **Select** component allows, in single selection mode, selecting an item from a drop-down list, or in multiple selection mode, from a list box that shows multiple items.

Figure 4.13. The Select Component



Combo Box Behaviour

The **Select** component will act as a combo box in single selection mode, allowing either to choose the value from the drop-down list or to write the value in the text field part of the component.

Filtered Selection

The **Select** component allows filtering the items available for selection. The component shows as an input box for entering text. The text entered in the input box is used for filtering the available items shown in a

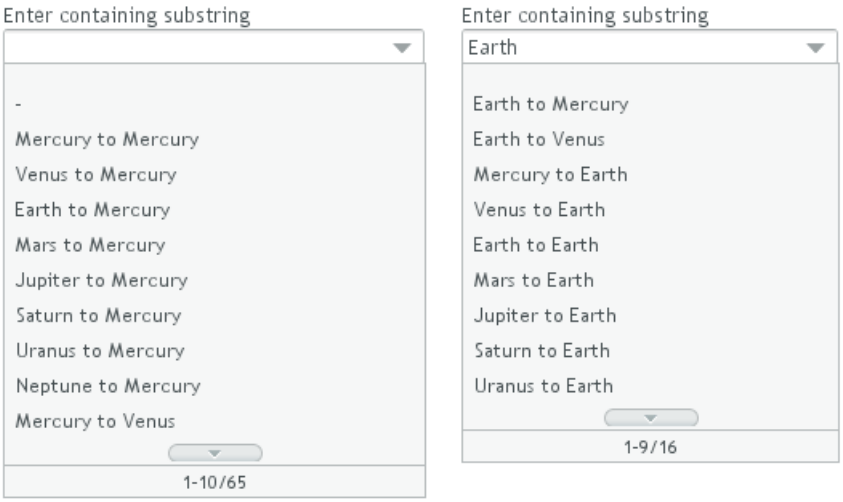
drop-down list. Pressing **Enter** will complete the item in the input box. Pressing **Up**- and **Down**-arrows can be used for selecting an item from the drop-down list. The drop-down list is paged and clicking on the scroll buttons will change to the next or previous page. The list selection can also be done with the arrow keys on the keyboard. The shown items are loaded from the server as needed, so the number of items held in the component can be quite large.

IT Mill Toolkit provides two filtering modes: *`FILTERINGMODE_CONTAINS`* matches any item that contains the string given in the text field part of the component and *`FILTERINGMODE_STARTSWITH`* matches only items that begin with the given string. The filtering mode is set with `setFilteringMode()`. Setting the filtering mode to the default value *`FILTERINGMODE_OFF`* disables filtering.

```
Select select = new Select("Enter containing substring");

select.setFilteringMode(AbstractSelect.Filtering.FILTERINGMODE_CONTAINS);

/* Fill the component with some items. */
final String[] planets = new String[] {"Mercury", "Venus", "Earth",
        "Mars", "Jupiter", "Saturn", "Uranus", "Neptune" };

for (int i = 0; i < planets.length; i++)
    for (int j = 0; j < planets.length; j++) {
        select.addItem(planets[j] + " to " + planets[i]);
```

The above example uses the containment filter that matches to all items containing the input string. As shown in Figure 4.14, “Filtered Selection” below, when we type some text in the input area, the drop-down list will show all the matching items.

Figure 4.14. Filtered Selection



The FilterSelect demo in the IT Mill Toolkit Demo Application provides an example of filtering items in a **Select** component.

CSS Style Rules

```
.i-filterselect { }
.i-filterselect-input { }
.i-filterselect-button { }
.i-filterselect-suggestpopup { }
.i-filterselect-prefpage-off { }
.i-filterselect-suggestmenu { }
.i-filterselect-status { }
```

In its default state, only the input field of the **Select** component is visible. The entire component is enclosed in `i-filterselect` style, the input field has `i-filterselect-input` style and the button in the right end that opens and closes the drop-down result list has `i-filterselect-button` style.

The drop-down result list has an overall `i-filterselect-suggestpopup` style. It contains the list of suggestions with `i-filterselect-suggestmenu` style and a status bar in the bottom with `i-filterselect-status` style. The list of suggestions is padded with an area with `i-filterselect-prefpage-off` style above and below the list.

4.9.2. Native Selection Component NativeSelect

NativeSelect offers the native selection component in web browsers, using an HTML `<select>` element. In single selection mode, the component is shown as a drop-down list, and in multiple selection mode as a list box.

CSS Style Rules

```
.i-select-optiongroup {}
.i-checkbox, .i-select-option {}
.i-radiobutton, .i-select-option {}
```

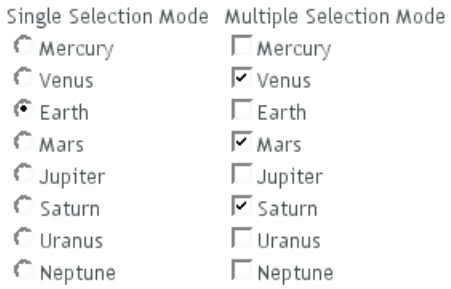
The `i-select-optiongroup` is the overall style for the component. Each check box will have the `i-checkbox` style and each radio button the `i-radiobutton` style. Both the radio buttons and check boxes will also have the `i-select-option` style that allows styling regardless of the option type.

4.9.3. Radio Button and Check Box Groups with OptionGroup

The **OptionGroup** class provides selection from alternatives using a group of radio buttons in single selection mode. In multiple selection mode, the items show up as check boxes.

```
OptionGroup optiongroup = new OptionGroup("My Option Group");

/* Use multiple selection mode. */
myselect.setMultiSelect(true);
```

Figure 4.15. Radio Button Group



It is also possible to create the check boxes individually using the **CheckBox** class, as described in Section 4.8, “Check Box”. The advantages of the **OptionGroup** component are that as it maintains the individual check box objects, you can get the array of all currently selected items easily, and that you can easily change the appearance of the component to another style.

CSS Style Rules

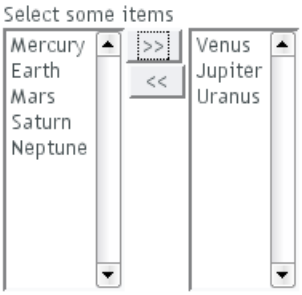
```
.i-select-optiongroup {}
.i-checkbox, .i-select-option {}
.i-radiobutton, .i-select-option {}
```

The `i-select-optiongroup` is the overall style for the component. Each check box will have the `i-checkbox` style and each radio button the `i-radiobutton` style. Both the radio buttons and check boxes will also have the `i-select-option` style that allows styling regardless of the option type.

4.9.4. Twin Column Selection with TwinColSelect

The **TwinColSelect** class provides a multiple selection component that shows two lists side by side. The user can select items from the list on the left and click on the ">>" button to move them to the list on the right. Items can be moved back by selecting them and clicking on the "<<" button.

Figure 4.16. Twin Column Selection



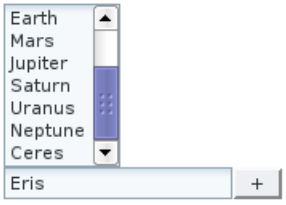
CSS Style Rules

```
.i-select-twincol {}
.i-select-twincol-options {}
.i-select-twincol-selections {}
.i-select-twincol-buttons {}
.i-select-twincol-deco {}
```

4.9.5. Allowing Adding New Items

The selection components allow the user to add new items, with a user interface similar to combo boxes in desktop user interfaces. If the *`newItemsAllowed`* mode is enabled with the `setNewItemsAllowed()` method, a text box for entering new items will be displayed beside the selection component. Clicking on the "+" button adds the item to the component.

Figure 4.17. Select Component with Adding New Items Allowed



The identifier of an item added by the user will be a **String** object identical to the caption of the item. You should take this into account if the item identifier of automatically filled items is some other type or otherwise not identical to the caption.

Adding new items is possible in both single and multiple selection modes and in all styles. Adding new items may not be possible if the **Select** is bound to an external **Container** that does not allow adding new items.

4.9.6. Multiple Selection Mode

Setting the **Select**, **NativeSelect**, or **OptionGroup** components to multiple selection mode with the `setMultiSelect()` method changes their appearance to allow selecting multiple items. By holding the **Ctrl** or **Shift** key pressed, the user can select multiple items.

See also the behaviour of the component in *`multiSelect`* mode with other styles below. With the *`twincol`* style, the selection is done by moving items from a list to a list of selected items. With the *`optiongroup`* style, the items are displayed as check boxes.

```
myselect.setMultiSelect(true);
```

In multiple selection mode, the property of a **Select** object will be an array of currently selected items.

```
/* Let us add an implementation of the ValueChangeListener interface. */
public class SelectExample extends CustomComponent implements Property.ValueChangeListener
 {
    /* Create a Select object with a caption. */
    Select select = new Select("This is a Select component");

    VerticalLayout layout = new VerticalLayout();
    Label status = new Label("-");

    SelectExample () {
        setCompositionRoot (layout);
        layout.addComponent(select);

        /* Fill the component with some items. */
        final String[] planets = new String[] {"Mercury", "Venus", "Earth", "Mars",
                                               "Jupiter", "Saturn", "Uranus", "Neptune"};

        for (int i=0; i<planets.length; i++)
            select.addItem(planets[i]);

        /* By default, the change event is not triggered immediately
         * when the selection changes. This enables it. */
        select.setImmediate(true);

        /* Listen for changes in the selection. */
        select.addListener(this);

        layout.addComponent(status);
    }

    /* Respond to change in the selection. */
    public void valueChange(Property.ValueChangeEvent event) {
        /* The event.getProperty() returns the Item ID (IID) of the
         * currently selected item in the component. */
        status.setValue("Currently selected item ID: " + event.getProperty());
    }
}
```

4.10. Table

The **Table** component is intended for presenting tabular data organized in rows and columns. The **Table** is one of the most versatile components in IT Mill Toolkit. Table cells can include text or arbitrary UI components. You can easily implement editing of the table data, for example clicking on a cell could change it to a text field for editing.

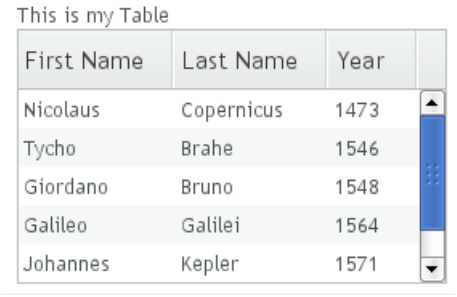
The data contained in a **Table** is managed using the Data Model of IT Mill Toolkit (see Chapter 7, *Data Model*), through the **Container** interface of the **Table**. This makes it possible to bind a table directly to a data souce such as a database query. Only the visible part of the table is loaded into the browser and moving the visible window with the scrollbar loads content from the server. While the data is being loaded, a tooltip will be displayed that shows the current range and total number of items in the table. The rows of the table are *items* in the container and the columns are *properties*. Each table row (item) is identified with an *item identifier* (IID), and each column (property) with a *property identifier* (PID).

When creating a table, you first need to define columns with `addContainerProperty()`. This method comes in two flavours. The simpler one takes the property ID of the column and uses it also as the caption of the column. The more complex one allows differing PID and header for the column. This may make, for example, internationalization of table headers easier, because if a PID is internationalized, the internationalization has to be used everywhere where the PID is used. The complex form of the method also allows defining an icon for the column from a resource. The "default value" parameter is used when new properties (columns) are added to the table, to fill in the missing values. (This default has no meaning in the usual case, such as below, where we add items after defining the properties.)

```
/* Create the table with a caption. */
Table table = new Table("This is my Table");

/* Define the names and data types of columns.
 * The "default value" parameter is meaningless here. */
table.addContainerProperty("First Name", String.class,  null);
table.addContainerProperty("Last Name",  String.class,  null);
table.addContainerProperty("Year",       Integer.class, null);

/* Add a few items in the table. */
table.addItem(new Object[] {"Nicolaus","Copernicus",new Integer(1473)}, new Integer(1));
table.addItem(new Object[] {"Tycho",   "Brahe",     new Integer(1546)}, new Integer(2));
table.addItem(new Object[] {"Giordano","Bruno",     new Integer(1548)}, new Integer(3));
table.addItem(new Object[] {"Galileo", "Galilei",   new Integer(1564)}, new Integer(4));
table.addItem(new Object[] {"Johannes","Kepler",    new Integer(1571)}, new Integer(5));
table.addItem(new Object[] {"Isaac",   "Newton",    new Integer(1643)}, new Integer(6));
```

In this example, we used an increasing **Integer** object as the Item Identifier, given as the second parameter to `addItem()`. The actual rows are given simply as object arrays, in the same order in which the properties were added. The objects must be of the correct class, as defined in the `addContainerProperty()` calls.

Figure 4.18. Basic Table Example



Scalability of the **Table** is largely dictated by the container. The default **IndexedContainer** is relatively heavy and can cause scalability problems, for example, when updating the values. Use of an optimized application-specific container is recommended. Table does not have a limit for the number of items and is just as fast with hundreds of thousands of items as with just a few. With the current implementation of scrolling, there is a limit around 500 000 rows, depending on the browser and the pixel height of rows.

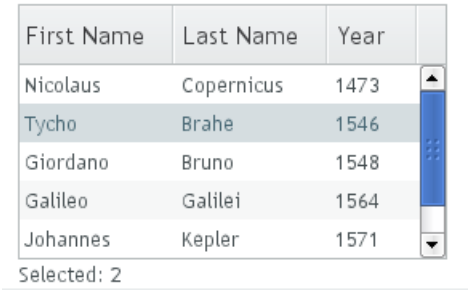
4.10.1. Selecting Items in a Table

The **Table** allows selecting one or more items by clicking them with the mouse. When the user selects an item, the IID of the item will be set as the property of the table and a **ValueChangeEvent** is triggered. To enable selection, you need to set the table *selectable*. You will also need to set it as *immediate* in most cases, as we do below, because without it, the change in the property will not be communicated immediately to the server.

The following example shows how to enable the selection of items in a **Table** and how to handle **ValueChangeEvent** events that are caused by changes in selection. You need to handle the event with the `valueChange()` method of the **Property.ValueChangeListener** interface.

```
/* Allow selecting items from the table. */
table.setSelectable(true);

/* When an item is selected, the selection is sent immediately to server. */
table.setImmediate(true);

/* Feedback from selection. */
final Label current = new Label("Selected: -");

/* Handle selection change. */
table.addListener(new Property.ValueChangeListener() {
    public void valueChange(ValueChangeEvent event) {
        current.setValue("Selected: " + table.getValue());
    }
});
```

Figure 4.19. Table Selection Example



If the user clicks on an already selected item, the selection will deselected and the table property will have *`null`* value. You can disable this behaviour by setting `setNullSelectionAllowed(false)` for the table.

A table can also be in *multiselect* mode, where a user can select and unselect any item by clicking on it. The mode is enabled with the `setMultiSelect()` method of the **Select** interface of **Table**. Selecting an item triggers a **ValueChangeEvent**, which will have as its parameter an array of item identifiers.

4.10.2. CSS Style Rules

```
.i-table {}
  .i-table-header-wrap {}
    .i-table-header {}
      .i-table-header-cell {}
        .i-table-resizer {}             /* Column resizer handle. */
        .i-table-caption-container {}
  .i-table-body {}
    .i-table-row-spacer {}
    .i-table-table {}
      .i-table-row {}
        .i-table-cell-content {}
```

Notice that some of the widths and heights in a table are calculated dynamically and can not be set in CSS.

Setting Individual Cell Styles

The **Table.CellStyleGenerator** interface allows you to set the CSS style for each individual cell in a table. You need to implement the `getStyle()`, which gets the row (item) and column (property) identifiers as parameters and can return a style name for the cell. The returned style name will be concatenated to prefix "`i-table-cell-content-`".

Alternatively, you can use a **Table.ColumnGenerator** (see Section 4.10.4, “Generated Table Columns”) to generate the actual UI components of the cells and add style names to them. A cell style generator is not used for the cells in generated columns.

```
Table table = new Table("Table with Cell Styles");
table.addStyleName("checkerboard");

// Add some columns in the table. In this example, the property IDs
// of the container are integers so we can determine the column number
// easily.
table.addContainerProperty("0", String.class, null, "", null, null); // Row header
for (int i=0; i<8; i++)
    table.addContainerProperty(""+(i+1), String.class, null,
                               String.valueOf((char) (65+i)), null, null);

// Add some items in the table.
```

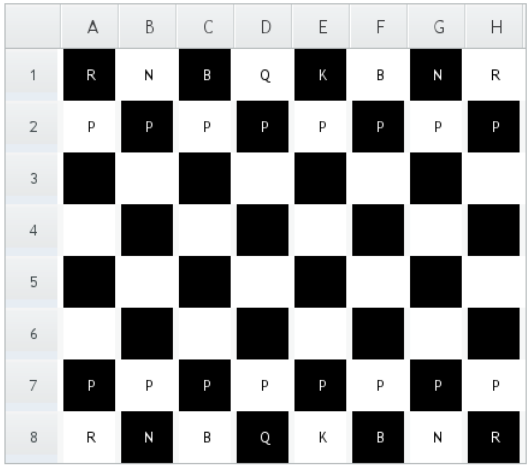
```
table.addItem(new Object[]{"1", "X", "X", "X", "X", "X", "X", "X", "X"}, new Integer(0));
table.addItem(new Object[]{"2", "P", "P", "P", "P", "P", "P", "P", "P"}, new Integer(1));
for (int i=2; i<6; i++)
    table.addItem(new Object[]{String.valueOf(i+1), "", "", "", "", "", "", "", ""}, new
 Integer(i));
table.addItem(new Object[]{"7", "P", "P", "P", "P", "P", "P", "P", "P"}, new Integer(6));
table.addItem(new Object[]{"8", "X", "X", "X", "X", "X", "X", "X", "X"}, new Integer(7));
table.setPageLength(8);

// Set cell style generator
table.setCellStyleGenerator(new Table.CellStyleGenerator() {
    public String getStyle(Object itemId, Object propertyId) {
        int row = ((Integer)itemId).intValue();
        int col = Integer.parseInt((String)propertyId);

        // The first column.
        if (col == 0)
            return "rowheader";

        // Other cells.
        if ((row+col)%2 == 0)
            return "black";
        else
            return "white";
    }
});
```

You can then style the cells, for example, as follows:

```
/* Center the text in header. */
.i-table-header-cell {
    text-align: center;
}

/* Basic style for all cells. */
.i-table-checkerboard .i-table-cell-content {
    text-align: center;
    vertical-align: middle;
    padding-top: 12px;
    width: 20px;
    height: 28px;
}

/* Style specifically for the row header cells. */
.i-table-cell-content-rowheader {
 background: #E7EDF3 url(../default/table/img/header-bg.png) repeat-x scroll 0 0;
}

/* Style specifically for the "white" cells. */
.i-table-cell-content-white {
    background: white;
    color: black;
}

/* Style specifically for the "black" cells. */
.i-table-cell-content-black {
    background: black;
    color: white;
}
```

The table will look as shown in the figure below.

Figure 4.20. Cell Style Generator for a Table



4.10.3. Table Features

Page Length and Scrollbar

The default style for **Table** provides a table with a scrollbar. The scrollbar is located at the right side of the table and becomes visible when the number of items in the table exceeds the page length, that is, the number of visible items. You can set the page length with `setPageLength()`.

Setting the page length to zero makes all the rows in a table visible, no matter how many rows there are. Notice that this also effectively disables buffering, as all the entire table is loaded to the browser at once. Using such tables to generate reports does not scale up very well, as there is some inevitable overhead in rendering a table with Ajax. For very large reports, generating HTML directly is a more scalable solution.

Organizing Columns

The default scrollable style supports most of the table features. User can resize the columns by dragging their borders, change the sorting by clicking on the column headers, collapse the columns if *`columnCollapsingAllowed`* is *`true`*, and reorder them if *`columnReorderingAllowed`* is *`true`*. You can set the column width of individual columns with `setColumnWidth()`.

Components Inside a Table

The cells of a **Table** can contain any user interface components, not just strings. If the rows are higher than the row height defined in the default theme, you have to define the proper row height in a custom theme.

When handling events for components inside a **Table**, such as for the **Button** in the example below, you usually need to know the item the component belongs to. Components do not themselves know about the table or the specific item in which a component is contained. Therefore, the handling method must use some other means for finding out the Item ID of the item. There are a few possibilities. Usually the easiest way is to use the `setData()` method to attach an arbitrary object to a component. You can subclass the component and include the identity information there. You can also simply search the entire table for the item with the component, although that solution may not be so scalable.

The example below includes table rows with a **Label** in XHTML formatting mode, a multiline **TextField**, a **CheckBox**, and a **Button** that shows as a link.

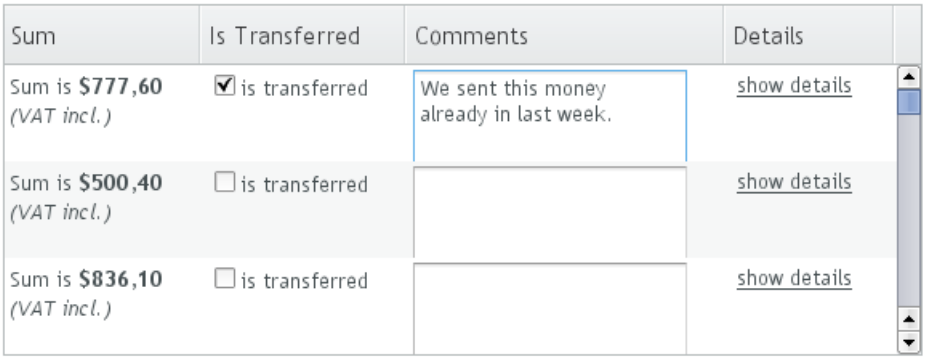
```
// Create a table and add a style to allow setting the row height in theme.
final Table table = new Table();
table.addStyleName("components-inside");

/* Define the names and data types of columns.
 * The "default value" parameter is meaningless here. */
table.addContainerProperty("Sum",            Label.class,     null);
table.addContainerProperty("Is Transferred", CheckBox.class,  null);
table.addContainerProperty("Comments",       TextField.class, null);
table.addContainerProperty("Details",        Button.class,    null);

/* Add a few items in the table. */
for (int i=0; i<100; i++) {
    // Create the fields for the current table row
    Label sumField = new Label(String.format("Sum is <b>$%04.2f</b><br/><i>(VAT
incl.)</i>",
                           new Object[] {new Double(Math.random()*1000)}),
                           Label.CONTENT_XHTML);
    CheckBox transferredField = new CheckBox("is transferred");

    // Multiline text field. This required modifying the height of the
    // table row.
    TextField commentsField = new TextField();
    commentsField.setRows(3);

    // The Table item identifier for the row.
    Integer itemId = new Integer(i);

    // Create a button and handle its click. A Button does not know
    // the item it is contained in, so we have to store the item
    // ID as user-defined data.
    Button detailsField = new Button("show details");
    detailsField.setData(itemId);
    detailsField.addListener(new Button.ClickListener() {
        public void buttonClick(ClickEvent event) {
            // Get the item identifier from the user-defined data.
            Integer itemId = (Integer)event.getButton().getData();
            getWindow().showNotification("Link "+itemId.intValue()+" clicked.");
        }
    });
    detailsField.addStyleName("link");

    // Create the table row.
    table.addItem(new Object[] {sumField, transferredField,
                                commentsField, detailsField},
                  itemId);
}

/* Show just three rows because they are so high. */
table.setPageLength(3);
```

The row height has to be set higher than the default with a style rule such as the following:

```
/* Table rows contain three-row TextField components. */
.i-table-components-inside .i-table-cell-content {
 height: 54px;
}
```

The table will look as shown in the figure below.

Figure 4.21. Components in a Table



Editing the Values of a Table

Normally, a **Table** simply displays the items and their fields as text. If you want to allow the user to edit the values, you can either put them inside components as we did above, or you can simply call `setEditable(true)` and the cells are automatically turned into editable fields.

Let us begin with a regular table with a some columns with usual Java types, namely a **Date**, **Boolean**, and a **String**.

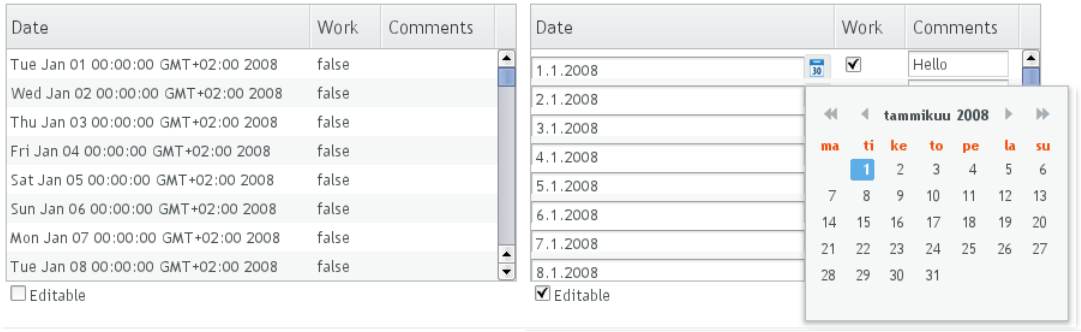
```
// Create a table. It is by default not editable.
final Table table = new Table();

// Define the names and data types of columns.
table.addContainerProperty("Date",     Date.class,  null);
table.addContainerProperty("Work",     Boolean.class, null);
table.addContainerProperty("Comments", String.class,  null);

// Add a few items in the table.
for (int i=0; i<100; i++) {
    Calendar calendar = new GregorianCalendar(2008,0,1);
    calendar.add(Calendar.DAY_OF_YEAR, i);

    // Create the table row.
    table.addItem(new Object[] {calendar.getTime(),
                                new Boolean(false),
                                ""},
                  new Integer(i)); // Item identifier
}

table.setPageLength(8);
layout.addComponent(table);
```

You could put the table in editable mode right away if you need to. We'll continue the example by adding a mechanism to switch the **Table** from and to the editable mode.

```
final CheckBox switchEditable = new CheckBox("Editable");
switchEditable.addListener(new Property.ValueChangeListener() {
    public void valueChange(ValueChangeEvent event) {
        table.setEditable(((Boolean)event.getProperty().getValue()).booleanValue());
    }
});
switchEditable.setImmediate(true);
layout.addComponent(switchEditable);
```

Now, when you check to checkbox, the components in the table turn into editable fields, as shown in Figure 4.22, “A Table in Normal and Editable Mode” below.

Figure 4.22. A Table in Normal and Editable Mode



The field components that allow editing the values of particular types are defined in a field factory that implements the **FieldFactory** interface. The default implementation is **BaseFieldFactory**, which offers the following crude mappings:

Table 4.4. Type to Field Mappings in BaseFieldFactory

Property Type	Mapped to Field Class
Date	A **DateField**.
Boolean	A **CheckBox**.
Item	A **Form**. The fields of the form are automatically created from the item's properties using the default field factory, that is, **BaseFieldFactory**. The normal use for this property type is inside a **Form** and is less useful inside a **Table**.
others	A **TextField**. The text field manages conversions from the basic types, if possible.

Field factories are covered with more detail in Section 4.15.2, “Binding Form to Data”. In the default **BaseFieldFactory** (you might want to look the source code), the mappings are defined in `createField(Class type, Component uiContext)` method, but you can implement any other of the abstract **FieldFactory** methods, depending on your needs. You could just implement the **FieldFactory** interface, but We recommend that you extend the **BaseFieldFactory** according to your needs.

Iterating Over a Table

As the items in a **Table** are not indexed, iterating over the items has to be done using an iterator. The `getItemIds()` method of the **Container** interface of **Table** returns a **Collection** of item identifiers over which you can iterate using an **Iterator**. For an example about iterating over a **Table**, please see Section 7.4, “Collecting items in Containers”. Notice that you may not modify the **Table** during iteration, that is, add or remove items. Changing the data is allowed.

4.10.4. Generated Table Columns

You might want to have a column that has values calculated from other columns. Or you might want to format table columns in some way, for example if you have columns that display currencies. The **ColumnGenerator** interface allows defining custom generators for such columns.

You add new generated columns to a **Table** with `addGeneratedColumn()`. It takes the column identifier as its parameters. Usually you want to have a more user-friendly and possibly internationalized column header. You can set the header and a possible icon by calling `addContainerProperty()` *before* adding the generated column.

```
// Define table columns.
table.addContainerProperty("date",        Date.class,   null, "Date",
null, null);
table.addContainerProperty("quantity",    Double.class, null, "Quantity (l)",
null, null);
table.addContainerProperty("price",       Double.class, null, "Price (e/l)",
null, null);
table.addContainerProperty("total",       Double.class, null, "Total (e)",
null, null);

// Define the generated columns and their generators.
table.addGeneratedColumn("date",        new DateColumnGenerator());
table.addGeneratedColumn("quantity",    new ValueColumnGenerator("%.2f l"));
table.addGeneratedColumn("price",       new PriceColumnGenerator());
table.addGeneratedColumn("total",       new ValueColumnGenerator("%.2f e"));
```

Notice that the `addGeneratedColumn()` always places the generated columns as the last column, even if you defined some other order previously. You will have to set the proper order with `setVisibleColumns()`.

```
table.setVisibleColumns(new Object[] { "date", "quantity", "price", "total"});
```

The generators are objects that implement the **Table.ColumnGenerator** interface and its `generateCell()` method. The method gets the identity of the item and column as its parameters, in addition to the table object. It has to return a component object.

The following example defines a generator for formatting **Double** valued fields according to a format string (as in **java.util.Formatter**).

```
/** Formats the value in a column containing Double objects. */
class ValueColumnGenerator implements Table.ColumnGenerator {
    String format; /* Format string for the Double values. */

    /** Creates double value column formatter with the given format string. */
    public ValueColumnGenerator(String format) {
        this.format = format;
    }

    /**
     * Generates the cell containing the Double value. The column is
     * irrelevant in this use case.
     */
    public Component generateCell(Table source, Object itemId, Object columnId) {
        Property prop = source.getItem(itemId).getItemProperty(columnId);
        if (prop.getType().equals(Double.class)) {
            Label label = new Label(String.format(format,
                    new Object[] { (Double) prop.getValue() }));

            // Set styles for the column: one indicating that it's a value and a more
            // specific one with the column name in it. This assumes that the column
            // name is proper for CSS.
            label.addStyleName("column-type-value");
            label.addStyleName("column-" + (String) columnId);
            return label;
        }
        return null;
    }
}
```

If you wish to have a custom style for the cells, you have to set it in the generator. A **CellStyleGenerator** defined for a table will not be called for the cells of generated columns.

The generator is called for all the visible (or more accurately cached) items in a table. If the user scrolls the table to another position in the table, the columns of the new visible rows are generated dynamically.

The columns in the visible (cached) rows are also generated always when an item has a value change. It is therefore usually safe to calculate the value of generated cells from the values of different rows (items).

When you set a table as *`editable`*, regular fields will change to editing fields. When the user changes the values in the fields, the generated columns will be updated automatically. Putting a table with generated columns in editable mode has a few quirks. The editable mode of **Table** does not affect generated columns. You have two alternatives: either you generate the editing fields in the generator or, in case of formatter generators, remove the generator in the editable mode. The example below uses the latter approach.

```
// Have a check box that allows the user to make the quantity
// and total columns editable.
final CheckBox editable = new CheckBox("Edit the input values - calculated columns are
regenerated");
editable.setImmediate(true);
editable.addListener(new ClickListener() {
    public void buttonClick(ClickEvent event) {
        table.setEditable(editable.booleanValue());

        // The columns may not be generated when we want to have them
        // editable.
        if (editable.booleanValue()) {
            table.removeGeneratedColumn("quantity");
            table.removeGeneratedColumn("total");
        } else {
            // In non-editable mode we want to show the formatted values.
            table.addGeneratedColumn("quantity", new ValueColumnGenerator("%.2f l"));
            table.addGeneratedColumn("total",    new ValueColumnGenerator("%.2f e"));
        }
        // The visible columns are affected by removal and addition of
        // generated columns so we have to redefine them.
        table.setVisibleColumns(
              new Object[] { "date","quantity","price","total","consumption", "dailycost"
 });
    }
});
```

You will also have to set the editing fields in *`immediate`* mode to have the update occur immediately when an edit field loses the focus. You can set the fields in *`immediate`* mode with the a custom **FieldFactory**, such as the one given below:

```
public class ImmediateFieldFactory extends BaseFieldFactory {
    public Field createField(Class type, Component uiContext) {
        // Let the BaseFieldFactory create the fields
        Field field = super.createField(type, uiContext);

        // ...and just set them as immediate
        ((AbstractField)field).setImmediate(true);

        return field;
    }
}
...
table.setFieldFactory(new ImmediateFieldFactory());
```

If you generate the editing fields with the column generator, you avoid having to use such a field factory, but of course have to generate the fields for both normal and editable modes.

Figure 4.23, “Table with Generated Columns in Normal and Editable Mode” below shows a table with columns calculated (blue) and simply formatted (black) with column generators.

Figure 4.23. Table with Generated Columns in Normal and Editable Mode

Date	Quantity (l)	Price (€/l)	Total (€)	Consumption (l/day)	Daily Cost (€/day)
2005-02-19	44,96 l	1,14 €	51,21 €	N/A	N/A
2005-03-30	44,91 l	1,20 €	53,67 €	1,15 l	1,38 €
2005-04-20	42,96 l	1,14 €	49,06 €	2,05 l	2,34 €
2005-05-23	47,37 l	1,17 €	55,28 €	1,44 l	1,68 €
2005-06-06	35,34 l	1,17 €	41,52 €	2,52 l	2,97 €
2005-06-30	16,07 l	1,24 €	20,00 €	0,67 l	0,83 €
2005-07-02	36,40 l	0,99 €	36,19 €	18,20 l	18,10 €

Date	Quantity (l)	Price (€/l)	Total (€)	Consumption (l/day)	Daily Cost (€/day)
2005-02-19	44.96	1,14 €	51.21	N/A	N/A
2005-03-30	44.91	1,20 €	53.67	1,15 l	1,38 €
2005-04-20	42.96	1,14 €	49.06	2,05 l	2,34 €
2005-05-23	47.37	1,17 €	55.28	1,44 l	1,68 €
2005-06-06	35.34	1,17 €	41.52	2,52 l	2,97 €
2005-06-30	16.07	1,24 €	20.0	0,67 l	0,83 €
2005-07-02	36.4	0,99 €	36.19	18,20 l	18,10 €

You can find the complete generated columns example in the Feature Browser demo application in the installation package, in **com.itmill.toolkit.demo.featurebrowser.GeneratedColumnExample.java**.

4.11. Tree

The **Tree** component allows a natural way to represent data that has hierarchical relationships, such as filesystems or message threads. The **Tree** component in IT Mill Toolkit works much like the tree components of most modern desktop user interface toolkits, for example in directory browsing.

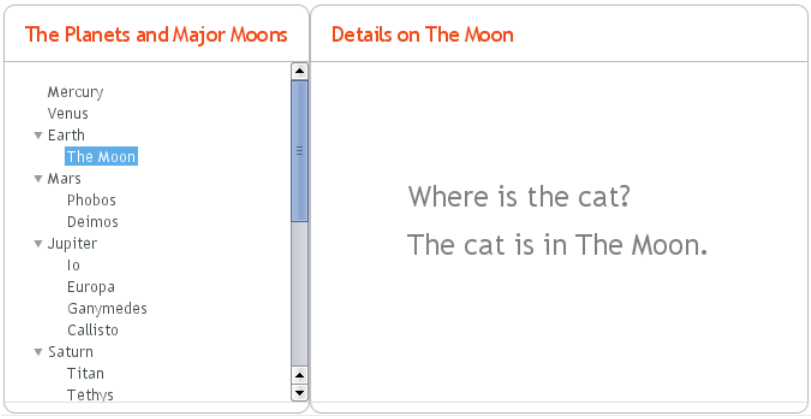
The typical use of the **Tree** component is for displaying a hierachical menu, like a menu on the left side of the screen, as in Figure 4.24, “A **Tree** Component as a Menu” below, or for displaying filesystems or other hierarchical datasets. The *menu* style makes the appearance of the tree more suitable for this purpose.

```
final Object[][] planets = new Object[][]{
        new Object[]{"Mercury"},
        new Object[]{"Venus"},
        new Object[]{"Earth", "The Moon"},
        new Object[]{"Mars", "Phobos", "Deimos"},
        new Object[]{"Jupiter", "Io", "Europa", "Ganymedes", "Callisto"},
        new Object[]{"Saturn", "Titan", "Tethys", "Dione", "Rhea", "Iapetus"},
        new Object[]{"Uranus", "Miranda", "Ariel", "Umbriel", "Titania", "Oberon"},
        new Object[]{"Neptune", "Triton", "Proteus", "Nereid", "Larissa"}};

Tree tree = new Tree("The Planets and Major Moons");

/* Add planets as root items in the tree. */
for (int i=0; i<planets.length; i++) {
    String planet = (String) (planets[i][0]);
    tree.addItem(planet);

    if (planets[i].length == 1) {
        /* The planet has no moons so make it a leaf. */
```

```
            tree.setChildrenAllowed(planet, false);
        } else {
            /* Add children (moons) under the planets. */
            for (int j=1; j<planets[i].length; j++) {
                String moon = (String) planets[i][j];

                /* Add the item as a regular item. */
                tree.addItem(moon);

                /* Set it to be a child. */
                tree.setParent(moon, planet);

                /* Make the moons look like leaves. */
                tree.setChildrenAllowed(moon, false);
            }

            /* Expand the subtree. */
            tree.expandItemsRecursively(planet);
        }
    }

    main.addComponent(tree);
```

Figure 4.24, “A **Tree** Component as a Menu” below shows the tree from the code example in a practical situation.

Figure 4.24. A Tree Component as a Menu



You can read or set the currently selected item by the value property of the **Tree** component, that is, with `getValue()` and `setValue()`. When the user clicks an item on a tree, the tree will receive an **ValueChangeEvent**, which you can catch with a **ValueChangeListener**. To receive the event immediately after the click, you need to set the tree as **setImmediate(true)**.

The **Tree** component uses **Container** data sources much like the **Table** component, with the addition that it also utilizes hierarchy information maintained by a **HierarchicalContainer**. The contained items can be of any item type supported by the container. The default container and its `addItem()` assume that the items are strings and the string value is used as the item ID.

4.12. MenuBar

The **MenuBar** component allows creating horizontal dropdown menus, much like the main menu in desktop applications.

```
// Create a menu bar
final MenuBar menubar = new MenuBar();
main.addComponent(menubar);
```

You insert the top-level menu items to a **MenuBar** object with the `addItem()` method. It takes a string label, an icon resource, and a command as its parameters. The icon and command are not required and can be *`null`*.

```
MenuBar.MenuItem beverages = menubar.addItem("Beverages", null, null);
```

The command is called when the user clicks the item. A menu command is a class that implements the **MenuBar.Command** interface.

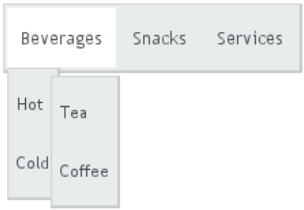
```
// A feedback component
final Label selection = new Label("-");
main.addComponent(selection);

// Define a common menu command for all the menu items.
MenuBar.Command mycommand = new MenuBar.Command() {
    public void menuSelected(MenuItem selectedItem) {
        selection.setValue("Ordered a " + selectedItem.getText() + " from menu.");
    }
};
```

The `addItem()` method returns a **MenuBar.MenuItem** object, which you can use to add sub-menu items. The **MenuItem** has an identical `addItem()` method.

```
// Put some items in the menu hierarchically
MenuBar.MenuItem beverages = menubar.addItem("Beverages", null, null);
MenuBar.MenuItem hot_beverages = beverages.addItem("Hot", null, null);
hot_beverages.addItem("Tea", null, mycommand);
hot_beverages.addItem("Coffee", null, mycommand);
MenuBar.MenuItem cold_beverages = beverages.addItem("Cold", null, null);
cold_beverages.addItem("Milk", null, mycommand);

// Another top-level item
MenuBar.MenuItem snacks = menubar.addItem("Snacks", null, null);
snacks.addItem("Weisswurst", null, mycommand);
snacks.addItem("Salami", null, mycommand);

// Yet another top-level item
MenuBar.MenuItem services = menubar.addItem("Services", null, null);
services.addItem("Car Service", null, mycommand);
```

The menu will look as follows:

Figure 4.25. Menu Bar



CSS Style Rules

```
.i-menubar { }
.gwt-MenuItem {}
.gwt-MenuItem-selected {}
```

The menu bar has the overall style name `.i-menubar`. Each menu item has `.gwt-MenuItem` style normally and `.gwt-MenuItem-selected` when the item is selected.

4.13. Embedded

The **Embedded** component allows displaying embedded media objects, such as images, animations, or any embeddable media type supported by the browser. The contents of an **Embedded** component are managed as *resources*. For documentation on resources, see Section 3.5, “Referencing Resources”.

The following example displays an image from the same Java package as the class itself using the class loader.

```
Embedded image = new Embedded("Yes, logo:", new ClassResource("toolkit-logo.png", this));
main.addComponent(image);
```

Figure 4.26. Embedded Image

The **Embedded** component supports several different content types, which are rendered differently in HTML. You can set the content type with `setType()`, although for images, as in the above example, the type is determined automatically.

Table 4.5. Embedded Object Types

`Embedded.TYPE_OBJECT`	The default embedded type, allows embedding certain file types inside HTML `<object>` and `<embed>` elements.
`Embedded.TYPE_IMAGE`	Embeds an image inside a HTML `<img>` element.
`Embedded.TYPE_BROWSER`	Embeds a browser frame inside a HTML `<iframe>` element.

4.13.1. Embedded Objects

The *`Embedded.TYPE_OBJECT`* is the default and most generic embedded type, which allows embedding media objects inside HTML `<object>` and `<embed>` elements. You need define the MIME type for the object type.

Currently, only Shockwave Flash animations are supported (MIME type `application/x-shockwave-flash`).

```
final ClassResource flashResource = new ClassResource("itmill_spin.swf", getApplication());
final Embedded embedded = new Embedded("Embedded Caption", flashResource);
embedded.setType(Embedded.TYPE_OBJECT);
embedded.setMimeType("application/x-shockwave-flash");
```

You can set object parameters with `setParameter()`, which takes a parameter's name and value as strings. The object parameters are included in the HTML as `<param>` elements.

4.13.2. Embedded Images

Images are embedded with the type *`Embedded.TYPE_IMAGE`*, although you do not normally need to set the type explicitly, as it is recognized automatically from the MIME type of the resource, as in the example above.

You can find another example of displaying an image from **FileResource** in Section 4.14, "**Upload**". Another example, in Section 3.5.5, "Stream Resources", shows how you can generate the content of an **Embedded** component dynamically using a **StreamResource**.

If you have a dynamically generated image, for example with a **StreamResource**, and the data changes, you need to reload the image in the browser. Because of how caching is handled in some browsers, you are best off by renaming the filename of the resource with a unique name, such as one including a timestamp. You should set cache time to zero with `setCacheTime()` for the resource object when you create it.

```
// Create the stream resource with some initial filename.
StreamResource imageResource = new StreamResource(imageSource, "initial-filename.png",
                                                  getApplication());

imageResource.setCacheTime(0);

Embedded embedded = new Embedded("", imageResource);
```

When refreshing, you also need to call `requestRepaint()` for the **Embedded** object.

```
// This needs to be done, but is not sufficient.
embedded.requestRepaint();

// Generate a filename with a timestamp.
SimpleDateFormat df = new SimpleDateFormat("yyyyMMddHHmmssSSS");
String filename = "myfilename-" + df.format(new Date()) + ".png";

// Replace the filename in the resource.
imageResource.setFilename(makeImageFilename());
```

You can find more detailed information about the **StreamResource** in Section 3.5.5, "Stream Resources".

4.13.3. Browser Frames

The browser frame type allows you to embed external content inside an HTML `<iframe>` element. You can refer to a URL with an **ExternalResource** object. URLs are given with the standard Java **URL** class.

```
URL url = new URL("http://dev.itmill.com/");
Embedded browser = new Embedded("", new ExternalResource(url));
browser.setType(Embedded.TYPE_BROWSER);
main.addComponent(browser);
```

4.14. Upload

The **Upload** component allows a user to upload files to the server. It displays a file name entry box, a file selection button, and an upload submit button. The user can either write the filename in the text area or click the **Browse** button to select a file. After the file is selected, the user sends the file by pressing the upload submit button.

```
// Create the Upload component.
final Upload upload = new Upload("Upload the file here", this);
```

Figure 4.27. Upload Component

You can set the text of the upload button with `setButtonCaption()`, as in the example above, but it is difficult to change the look of the **Browse** button. This is a security feature of web browsers. The language of the **Browse** button is determined by the browser, so if you wish to have the language of the **Upload** component consistent, you will have to use the same language in your application.

```
upload.setButtonCaption("Upload Now");
```

The uploaded files are typically stored as files in a file system, in a database, or as temporary objects in memory. The upload component writes the received data to an **java.io.OutputStream** so you have plenty of freedom in how you can process the upload content.

To use the **Upload** component, you need to define a class that implements the **Upload.Receiver** interface. The `receiveUpload()` method is called when the user clicks the submit button. The method must return an **OutputStream**. To do this, it typically creates a **File** or a memory buffer where the stream is written. The method gets the file name and MIME type of the file, as reported by the browser.

When an upload is finished, successfully or unsuccessfully, the **Upload** component will emit the **Upload.FinishedEvent** event. To receive it, you need to implement the **Upload.FinishedListener** interface, and register the listening object in the **Upload** component. The event object will also include the file name, MIME type, and length of the file. Notice that the more specific **Upload.FailedEvent** and **Upload.SucceededEvent** events will be called in the cases where the upload failed or succeeded, respectively.

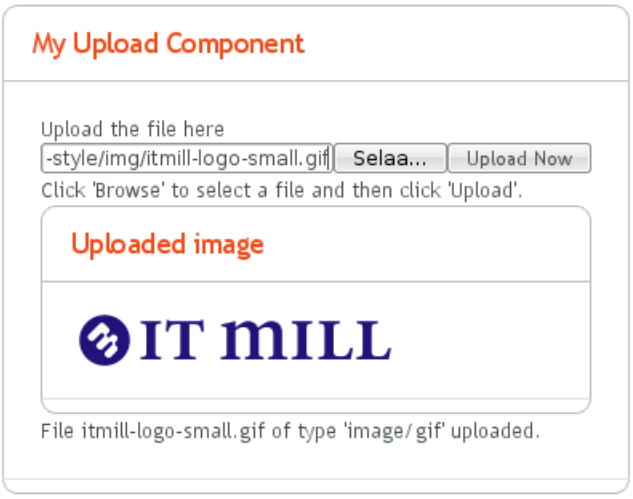
The following example allows uploading images to `/tmp/uploads` directory in (UNIX) filesystem (the directory must exist or the upload fails). The component displays the last uploaded image in an **Embedded** component.

```
import java.io.File;
import java.io.FileOutputStream;
import java.io.OutputStream;
import com.itmill.toolkit.terminal.FileResource;
import com.itmill.toolkit.ui.*;

public class MyUploader extends CustomComponent
implements Upload.SucceededListener, Upload.FailedListener, Upload.Receiver {
    Panel root;         // Root element for contained components.
    Panel imagePanel;   // Panel that contains the uploaded image.
    File  file;         // File to write to.

    MyUploader() {
        root = new Panel("My Upload Component");
        setCompositionRoot(root);

        // Create the Upload component.
        final Upload upload = new Upload("Upload the file here", this);
        upload.setButtonCaption("Upload Now");

        // Listen for Upload.SucceededEvent and FailedEvent events.
        upload.addListener((Upload.SucceededListener) this);
        upload.addListener((Upload.FailedListener) this);

        root.addComponent(upload);
        root.addComponent(new Label("Click 'Browse' to select a file and then click
'Upload'."));

        // Create a panel for displaying the uploaded file (image).
        imagePanel = new Panel("Uploaded image");
        imagePanel.addComponent(new Label("No image uploaded yet"));
```

```
        root.addComponent(imagePanel);
    }

    // Callback method to begin receiving the upload.
    public OutputStream receiveUpload(String filename, String MIMEType) {
        FileOutputStream fos = null; // Output stream to write to.
        file = new File("/tmp/uploads/" + filename);
        try {
            // Open the file for writing.
            fos = new FileOutputStream(file);
        } catch (final java.io.FileNotFoundException e) {
            // Error while opening the file. Not reported here.
            e.printStackTrace();
            return null;
        }

        return fos; // Return the output stream to write to
    }

    // This is called if the upload is finished.
    public void uploadSucceeded(Upload.SucceededEvent event) {
        // Log the upload on screen.
        root.addComponent(new Label("File " + event.getFilename()
                + " of type '" + event.getMIMEType() + "' uploaded."));

        // Display the uploaded file in the image panel.
        final FileResource imageResource = new FileResource(file, getApplication());
        imagePanel.removeAllComponents();
        imagePanel.addComponent(new Embedded("", imageResource));
    }

    // This is called if the upload fails.
    public void uploadFailed(Upload.FailedEvent event) {
        // Log the failure on screen.
        root.addComponent(new Label("Uploading " + event.getFilename()
                + " of type '" + event.getMIMEType() + "' failed."));
    }
}
```

The example does not check the type of the uploaded files in any way, which will cause an error if the content is anything else but an image. The program also assumes that the MIME type of the file is resolved correctly based on the file name extension. After uploading an image, the component will look as show in Figure 4.28, “Image Upload Example” below. The browser shows the **Browse** button localized.

Figure 4.28. Image Upload Example



4.15. Form

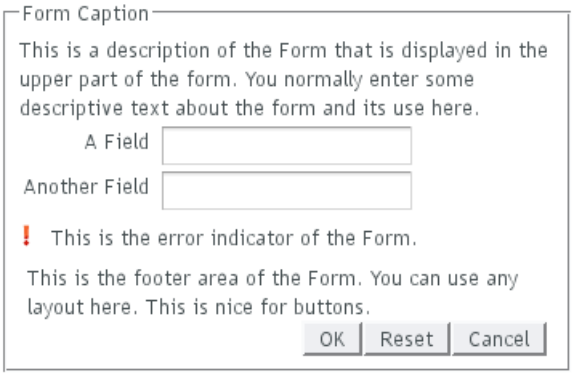
Most web applications need forms. The **Form** component in IT Mill Toolkit offers an easy way to create forms where the fields can be automatically generated from a data source that is bound to the form. The **BeanItem** adapter allows the data sources to be just JavaBeans or Plain Old Java Objects (POJOs) with just the setter and getter methods. **From** manages buffering so that the form contents can be committed to the data source only when filling the form is complete, and before that, the user can discard any changes.

The **Form** component is also a layout, with a bounding box, a caption, a description field, and a special error indicator. As such, it can also be used within logical forms to group input fields.

4.15.1. Form as a User Interface Component

To begin with the **Form**, it is a UI component with a layout suitable for its purpose. A **Form** has a caption, a description, a layout that contains the fields, an error indicator, and a footer, as illustrated in Figure 4.29, “Layout of the Form Component” below. Unlike with other components, the caption is shown within the border. (See the details below on how to enable the border with CSS, as it may not be enabled in the default style.)

Figure 4.29. Layout of the Form Component



Unlike most components, **Form** does not accept the caption in the constructor, as forms are often captionless, but you can give the caption with the `setCaption()`. While the description text, which you can set with **setDescription()**, is shown as a tooltip in most other components, a **Form** displays it in top of the form box as shown in the figure above.

```
Form form = new Form();
form.setCaption("Form Caption");
form.setDescription("This is a description of the Form that is " +
        "displayed in the upper part of the form. You normally enter some " +
        "descriptive text about the form and its use here.");
```

Form has **FormLayout** as its default layout, but you can set any other layout with `setLayout()`.

The **Form** is most of all a container for fields so it offers many kinds of automation for creating and managing fields. You can, of course, create fields directly in the layout, but it is usually more desirable to bind the fields to the connected data source.

```
// Add a field directly to the layout. This field will not be bound to
// the data source Item of the form.
form.getLayout().addComponent(new TextField("A Field"));

// Add a field and bind it to an named item property.
form.addField("another", new TextField("Another Field"));
```

Binding forms and their fields to data objects is described further in Section 4.15.2, “Binding Form to Data” below.

The **Form** has a special error indicator inside the form. The indicator can show the following types of error messages:

- Errors set with the `setComponentError()` method of the form. For example:

  ```
  form.setComponentError(new UserError("This is the error indicator of the Form."));
  ```

- Errors caused by a validator attached to the **Form** with `addValidator()`.
- Errors caused by validators attached to the fields inside forms, if `setValidationVisible(true)` is set for the form. This type of validation is explained futher in Section 4.15.3, “Validating Form Input” below.
- Errors from automatic validation of fields set as *required* with `setRequired(true)` if an error message has also been set with `setRequiredError()`.

Only a single error is displayed in the error indicator at a time.

Finally, **Form** has a footer area. The footer is a **HorizontalLayout** by default, but you can change it with `setFooter()`.

```
// Set the footer layout and add some text.
form.setFooter(new VerticalLayout());
form.getFooter().addComponent(new Label("This is the footer area of the Form. "+
                                        "You can use any layout here. This is nice for
buttons."));

// Add an Ok (commit), Reset (discard), and Cancel buttons for the form.
HorizontalLayout okbar = new HorizontalLayout();
okbar.setHeight("25px");
Button okbutton = new Button("OK", form, "commit");
okbar.addComponent(okbutton);
okbar.setComponentAlignment(okbutton, Alignment.TOP_RIGHT);
okbar.addComponent(new Button("Reset", form, "discard"));
```

```
okbar.addComponent(new Button("Cancel"));
form.getFooter().addComponent(okbar);
```

CSS Style Rules

```
.i-form {}
.i-form fieldset {}
```

The top-level style name of a **Form** component is `i-form`. It is important to notice that the form is implemented as a HTML `<fieldset>`, which allows placing the caption (or "legend") inside the border. It would not be so meaningful to set a border for the top-level form element. The following example sets a border around the form, as is done in Figure 4.29, “Layout of the Form Component” above.

```
.i-form fieldset {
    border: thin solid;
}
```

4.15.2. Binding Form to Data

The main purpose of the **Form** component is that you can bind it to a data source and let the **Form** generate and manage fields automatically. The data source can be any class that implements the **Item** interface, which is part of the IT Mill Toolkit Data Model, as described in Chapter 7, *Data Model*. You can either implement the **Item** interface yourself, which can be overly complicated, or use the ready **BeanItem** adapter to bind the form to any JavaBean object. You can also use **PropertysetItem** to bind the form to an ad hoc set of **Property** objects.

Let us consider the following simple JavaBean with proper setter and getter methods for the member variables.

```
/** A simple JavaBean. */
public class PersonBean {
    String name;
    String city;

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setCity(String city) {
        this.city = city;
    }

    public String getCity() {
        return city;
    }
}
```

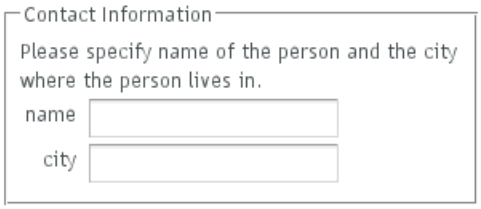
We can now bind this bean to a **Form** using the **BeanItem** adapter as follows.

```
// Create a form and use FormLayout as its layout.
final Form form = new Form();

// Set form caption and description texts
form.setCaption("Contact Information");
form.setDescription("Please specify name of the person and the city where the person
lives in.");

// Create the custom bean.
PersonBean bean = new PersonBean();
```

```
// Create a bean item that is bound to the bean.
BeanItem item = new BeanItem(bean);

// Bind the bean item as the data source for the form.
form.setItemDataSource(item);
```

The **Form** uses **FormLayout** layout by default and automatically generates the fields for each of the bean properties, as shown in Figure 4.30, “Form Automatically Generated from a Bean” below.

Figure 4.30. Form Automatically Generated from a Bean



The automatically determined order of the fields can be undesirable. To set the order properly, you can use the `setVisibleItemProperties()` method of the **Form**, which takes an ordered collection as its parameter. Fields that are not listed in the collection are not included in the form.

```
// Set the order of the items in the form.
Vector order = new Vector();
order.add("city");
order.add("name");
form.setVisibleItemProperties(order);
```

The form uses the property identifiers as the captions of the fields by default. If you want to have more proper captions for the fields, which is often the case, you need to use a **FieldFactory** to create the fields, as is shown in the section below.

Generating Proper Fields with a FieldFactory

The form generates the fields automatically using very coarse logic. A **String**, **int**, or **double** will result in a **TextField** alike, regardless of the meaning of the field. You might want to have a city name to be input with a combo box, for example. You can create such custom fields by implementing the proper methods in the **FieldFactory** interface.

The **FieldFactory** interface has four different **createField()** methods for creating the fields, each with a slightly different set of parameters. Each of the methods is used in different situations; please use debugger to find which one is used in your case.

The *`type`* is the class of the item property: **String** for both of the bean properties in our example. The *`uiContext`* is reference to UI component that will contain the fields, in this case the **Form** component. The *`propertyId`* is the identifier of the property, usually a **String**. In our example, it can be either the "`name`" or "`city`" property of the bean. The *`item`* is a reference to the **Item** implementation instance, which is in the above example a **BeanItem** bound to a bean object. The *`property`* parameter is a plain property (object-type pair). You can use these parameters in the logic for creating the proper field object.

The easiest and safest way to make a custom field factory is to extend the default **BaseFieldFactory** implementation, as we don in the example below:

```
class MyFieldFactory extends BaseFieldFactory {
    @Override
    public Field createField(Item item, Object propertyId,
```

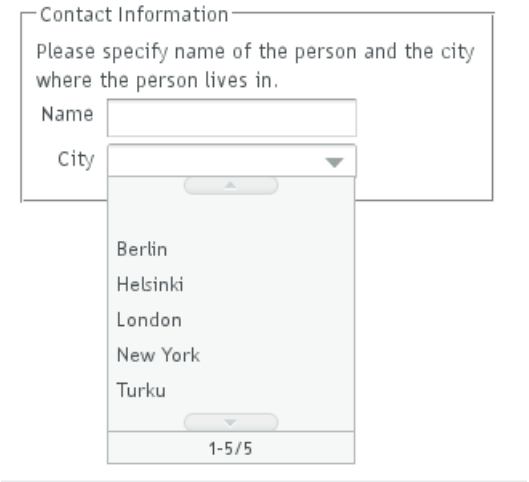
```
                                Component uiContext) {

        // Identify the fields by their Property ID.
        String pid = (String) propertyId;
        if (pid.equals("name")) {
            return new TextField("Name");
        } else if (pid.equals("city")) {
            Select select = new Select("City");
            select.addItem("Berlin");
            select.addItem("Helsinki");
            select.addItem("London");
            select.addItem("New York");
            select.addItem("Turku");
            select.setNewItemsAllowed(true);
            return select;
        }

        // Let BaseFieldFactory create other possible fields.
        return super.createField(item, propertyId, uiContext);
    }
}
```

You set the custom field factory as the field factory of the **Form** with `setFieldFactory()`:

```
form.setFieldFactory(new MyFieldFactory());
```

Our example will now look as shown below:

Figure 4.31. Form Fields Generated with a FieldFactory



4.15.3. Validating Form Input

Validation of the form input is one of the most important tasks in handling forms. The fields in IT Mill Toolkit can be bound to validators. The validation provides feedback about bad input and the forms can also manage validation results and accept the input only if all validations are successful. Fields can also be set as *required*, which is a special built-in validator. The validators work on the server-side.

Using Validators in Forms

Validators check the validity of input and, if the input is invalid, can provide an error message through an exception. Validators are classes that implement the **Validator** interface. The interface has two methods that you must implement: `isValid()` that returns the success or failure as a truth value, and

`validate()`, which reports a failure with an exception. The exception can be associated with an error message describing the details of the error.

```
// Postal code that must be 5 digits (10000-99999).
TextField field = new TextField("Postal Code");
field.setColumns(5);

// Create the validator
Validator postalCodeValidator = new Validator() {

    // The isValid() method returns simply a boolean value, so
    // it can not return an error message.
    public boolean isValid(Object value) {
        if (value == null || !(value instanceof String)) {
            return false;
        }

        return ((String) value).matches("[0-9]{5}");
    }

    // Upon failure, the validate() method throws an exception with an error message.
    public void validate(Object value) throws InvalidValueException {
        if (!isValid(value)) {
           throw new InvalidValueException("Postal code must be a number 10000-99999.");

        }
    }
};
field.addValidator(postalCodeValidator);
```

If you are using a custom **FieldFactory** to generate the fields, you may want to set the validators for fields there. It is useful to have the form in *immediate* mode:

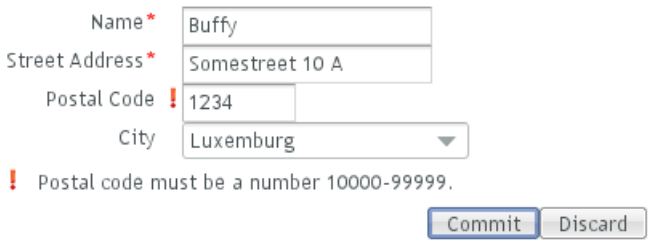
```
// Set the form to act immediately on user input. This is
// necessary for the validation of the fields to occur immediately when
// the input focus changes and not just on commit.
form.setImmediate(true);
```

Validation is done always when you call the `commit()` method of the **Form**.

```
// The Commit button calls form.commit().
Button commit = new Button("Commit", form, "commit");
```

If any of the validators in the form fail, the commit will fail and a validation exception message is displayed in the error indicator of the form. If the commit is successful, the input data is written to the data source. Notice that `commit()` also implicitly sets `setValidationVisible(true)` (if `setValidationVisibleOnCommit()` is *`true`*, as is the default). This makes the error indicators visible even if they were previously not visible.

Figure 4.32. Form Validation in Action



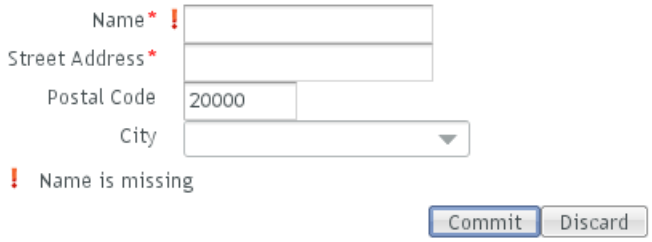
Required Fields in Forms

Setting a field as *required* outside a form is usually just a visual clue to the user. Leaving a required field empty does not display any error indicator in the empty field as a failed validation does. However, if you set a form field as required with `setRequired(true)` and give an error message with `setRequiredError()` and the user leaves the required field empty, the form will display the error message in its error indicator.

```
form.getField("name").setRequired(true);
form.getField("name").setRequiredError("Name is missing");
form.getField("address").setRequired(true); // No error message
```

To have the validation done immediately when the fields lose focus, you should set the form as *immediate*, as was done in the section above.

Figure 4.33. Empty Required Field After Clicking Commit



It is important that you provide the user with feedback from failed validation of required fields either by setting an error message or by providing the feedback by other means. Otherwise, when a user clicks the **Ok** button (commits the form), the button does not appear to work and the form does not indicate any reason. As an alternative to setting the error message, you can handle the validation error and provide the feedback about the problem with a different mechanism.

4.15.4. Buffering Form Data

Buffering means keeping the edited data in a buffer and writing it to the data source only when the `commit()` method is called for the component. If the user has made changes to a buffer, calling `discard()` restores the buffer from the data source. Buffering is actually a feature of all **Field** components and **Form** is a **Field**. **Form** manages the buffering of its contained fields so that if `commit()` or `discard()` is called for the **Form**, it calls the respective method for all of its managed fields.

```
final Form form = new Form();
...add components...

// Enable buffering.
form.setWriteThrough(false);

// The Ok button calls form.commit().
Button commit = new Button("Ok", form, "commit");

// The Restore button calls form.discard().
Button restore = new Button("Restore", form, "discard");
```

The Form example in the Feature Browser of IT Mill Toolkit demonstrates buffering in forms. The *Widget caching demo* in Additional demos demonstrates buffering in other **Field** components, its source code is available in `BufferedComponents.java`.

4.16. ProgressIndicator

The **ProgressIndicator** component allows displaying the progress of a task graphically. The progress is given as a floating-point value between 0.0 and 1.0.

Figure 4.34. The Progress Indicator Component

The progress indicator polls the server for updates for its value. If the value has changed, the progress is updated. Notice that the user application does not have to handle any polling event, but updating the component is done automatically.

Creating a progress indicator is just like with any other component. You can give the initial progress value as a parameter for the constructor. The default polling frequency is 1000 milliseconds (one second), but you can set some other interval with the `setPollingInterval()` method.

```
// Create the indicator
final ProgressIndicator indicator = new ProgressIndicator(new Float(0.0));
main.addComponent(indicator);

// Set polling frequency to 0.5 seconds.
indicator.setPollingInterval(500);
```

CSS Style Rules

```
.i-progressindicator {} /* Base element */
.i-progressindicator div {} /* Progress indication element */
```

The default style for the progress indicator uses an animated GIF image (`img/base.gif`) as the base background for the component. The progress is a `<div>` element inside the base. When the progress element grows, it covers more and more of the base background. By default, the graphic of the progress element is defined in `img/progress.png` under the default style directory. See `com.itmill.toolkit.terminal.gwt/public/default/progressindicator/progressindicator.css`.

4.16.1. Doing Heavy Computation

The progress indicator is often used to display the progress of a heavy server-side computation task. In the following example, we create a thread in the server to do some "heavy work". All the thread needs to do is to set the value of the progress indicator with `setValue()` and the current progress is displayed automatically when the browser polls the server.

```
// Create an indicator that makes you look busy
final ProgressIndicator indicator = new ProgressIndicator(new Float(0.0));
main.addComponent(indicator);

// Set polling frequency to 0.5 seconds.
indicator.setPollingInterval(500);

// Add a button to start working
final Button button = new Button("Click to start");
main.addComponent(button);

// Another thread to do some work
class WorkThread extends Thread {
    public void run () {
        double current = 0.0;
        while (true) {
```

```
            // Do some "heavy work"
            try {
                sleep(50); // Sleep for 50 milliseconds
            } catch (InterruptedException) {}

            // Show that you have made some progress:
            // grow the progress value until it reaches 1.0.
            current += 0.01;
            if (current>1.0)
                indicator.setValue(new Float(1.0));
            else
                indicator.setValue(new Float(current));

            // After all the "work" has been done for a while, take a break.
            if (current > 1.2) {
                // Restore the state to initial.
                indicator.setValue(new Float(0.0));
                button.setVisible(true);
                break;
            }
        }
    }
}

// Clicking the button creates and runs a work thread
button.addListener(new Button.ClickListener() {
    public void buttonClick(ClickEvent event) {
        final WorkThread thread = new WorkThread();
        thread.start();

        // The button hides until the work is done.
        button.setVisible(false);
    }
});
```

Figure 4.35. Starting Heavy Work

4.17. Custom Composite Components

The ease of making custom user interface components is one of the core features of IT Mill Toolkit. Custom components can be created at several levels. Typically, you simply combine existing built-in components to produce composite components. In many applications, such composite components make up the majority of the user interface. You can also create your own low-level components, for example existing GWT components. It is also possible to extend the functionality of existing components.

The architecture of user interface components is described in Chapter 4, *User Interface Components*. For more information about the overall architecture of IT Mill Toolkit, please see Chapter 2, *Architecture*. Use of custom GWT components is covered in Chapter 8, *Developing Custom Components*.

The easiest way of creating new components is combining existing components. This can be done in two basic ways: inheritance and management. With inheritance, you inherit some containing class, typically **CustomComponent** or some abstract class such as **AbstractComponent**, **AbstractField**, or **AbstractComponentContainer**. With management, you create a class that creates the needed components under some layout and handles their events. Both of these patterns are used extensively in the examples in Chapter 4, *User Interface Components* and elsewhere.

4.17.1. CustomComponent

The **CustomComponent** class is a simple implementation of the **Component** interface that provides a simple way for creating new user interface components by the composition of existing components.

Composition is done by inheriting the **CustomComponent** class and setting the *composite root* inside the component with `setCompositionRoot()`. The composite root is typically a layout component that contains multiple components.

4.18. Common Component Features

4.18.1. Sizing Components through Sizeable interface

IT Mill Toolkit components are sizeable; not in the sense that they were fairly large or that the number of the components and their features are sizeable, but in the sense that you can make them fairly large on the screen if you like. Or small, or whatever size.

The **Sizeable** interface, shared by all components, provides a number of manipulation methods and constants for setting the height and width of a component in absolute or relative units, or for leaving the size undefined.

The size of a component can be set with `setWidth()` and `setHeight()` methods. The methods take the size as a floating-point value. You need to give the unit of the measure as the second parameter for the above methods. The available units are listed in Table 4.6, “Size Units” below.

```
mycomponent.setWidth(100, Sizeable.UNITS_PERCENTAGE);
mycomponent.setWidth(400, Sizeable.UNITS_PIXELS);
```

Alternatively, you can speficy the size as a string. The format of such a string must follow the HTML/CSS standards for specifying measures.

```
mycomponent.setWidth("100%");
mycomponent.setHeight("400px");
```

The "`100%`" percentage value makes the component take all available size in the particular direction (see the description of *`Sizeable.UNITS_PERCENTAGE`* in the table below). You can also use the shorthand method `setSizeFull()` to set the size to 100% in both directions.

The size can be *undefined* in either or both dimensions, which means that the component will take the minimum necessary space. Most components have undefined size by default, but some layouts have full size in horizontal direction. You can set the height or width as undefined with *`Sizeable.SIZE_UNDEFINED`* parameter for `setWidth()` and `setHeight()`.

You always need to keep in mind that *a layout with undefined size may not contain components with defined relative size*, such as "full size". See Section 5.3.1, “Layout Size” for details.

The following table lists the available units and their codes.

Table 4.6. Size Units

Sizeable.UNITS_PIXELS	px	The *pixel* is the basic hardware-specific measure of one physical display pixel.
Sizeable.UNITS_POINTS	pt	The *point* is a typographical unit, which is usually defined as 1/72 inches or about 0.35 mm. However, on displays the size can vary significantly depending on display metrics.
Sizeable.UNITS_PICAS	pc	The *pica* is a typographical unit, defined as 12 points, or 1/7 inches or about 4.233 mm. On displays, the size can vary depending on display metrics.
Sizeable.UNITS_EM	em	A unit relative to the used font, the width of the upper-case "M" letter.
Sizeable.UNITS_EX	ex	A unit relative to the used font, the height of the lower-case "x" letter.
Sizeable.UNITS_MM	mm	A physical length unit, millimeters on the surface of a display device. However, the actual size depends on the display, its metrics in the operating system, and the browser.
Sizeable.UNITS_CM	cm	A physical length unit, *centimeters* on the surface of a display device. However, the actual size depends on the display, its metrics in the operating system, and the browser.
Sizeable.UNITS_INCH	in	A physical length unit, *inches* on the surface of a display device. However, the actual size depends on the display, its metrics in the operating system, and the browser.
Sizeable.UNITS_PERCENTAGE	%	A relative percentage of the available size. For example, for the top-level layout *100%* would be the full width or height of the browser window. The percentage value must be between 0 and 100.

If a component inside **HorizontalLayout** or **VerticalLayout** has full size in the namesake direction of the layout, the component will expand to take all available space not needed by the other components. See Section 5.3.1, “Layout Size” for details.

Chapter 5. Managing Layout

This chapter gives an overview of layout components, starting with their history, and then gives more specific description of the components together with some examples.

Layouts are required to place components to specific places in the user interface. You can use plain Java to accomplish sophisticated component layouting. Another option is to use **CustomLayout** class and let the web page designers take responsibility of component layouting using their own set of tools.

Layouts are often strongly coupled with themes that specify various layout attributes such as backgrounds, borders, alignment, and so on. Themes are detailed in Chapter 6, *Themes*.

5.1. Background for Layout

Ever since the ancient xeroxians invented graphical user interfaces, programmers have wanted to make GUI programming ever easier for themselves. Solutions started simple. When GUIs appeared on PC desktops, practically all screens were of the VGA type and fixed into 640x480 size. Mac or X Window System on UNIX were not much different. Everyone was so happy with such awesome graphics resolutions that they never thought that an application would have to work on a radically different screen size. At worst, screens could only grow, they thought, giving more space for more windows. In the 80s, the idea of having a computer screen in your pocket was simply not realistic. Hence, the GUI APIs allowed placing UI components using screen coordinates. Visual Basic and some other systems provided an easy way for the designer to drag and drop components on a fixed-sized window. One would have thought that at least translators would have complained about the awkwardness of such a solution, but apparently they were not, as non-engineers, heard or at least cared about. At best, engineers could throw at them a resource editor that would allow them to resize the UI components by hand. Such was the spirit back then.

After the web was born, layout design was doomed to change for ever. At first, layout didn't matter much, as everyone was happy with plain headings, paragraphs, and a few hyperlinks here and there. Designers of HTML wanted the pages to run on any screen size. The screen size was actually not pixels but rows and columns of characters, as the baby web was really just hyper*text*, not graphics. That was soon to be changed. The first GUI-based browser, NCSA Mosaic, launched a revolution that culminated in Netscape Navigator. Suddenly, people who had previously been doing advertisement brochures started writing HTML. This meant that layout design had to be easy not just for programmers, but also allow the graphics designer to do his or her job without having to know a thing about programming. The W3C committee designing web standards came up with the CSS (Cascading Style Sheet) specification, which allowed trivial separation of appearance from content. Later versions of HTML followed, XHTML appeared, as did countless other standards.

Page description and markup languages are a wonderful solution for static presentations, such as books and most web pages. Real applications, however, need to have more control. They need to be able to change the state of user interface components and even their layout on the run. This creates a need to separate the presentation from content on exactly the right level.

Thanks to the attack of graphics designers, desktop applications were, when it comes to appearance, far behind web design. Sun Microsystems had come in 1995 with a new programming language, Java, for writing cross-platform desktop applications. Java's original graphical user interface toolkit, AWT (Abstract Windowing Toolkit), was designed to work on multiple operating systems as well as embedded in web browsers. One of the special aspects of AWT was the layout manager, which allowed user interface components to be flexible, growing and shrinking as needed. This made it possible for the user to resize the windows of an application flexibly and also served the needs of localization, as text strings were not limited to some fixed size in pixels. It became even possible to resize the pixel size of fonts, and the rest of the layout adapted to the new size.

Layout management of IT Mill Toolkit is a direct successor of the web-based concept for separation of content and appearance and of the Java AWT solution for binding the layout and user interface components into objects in programs. IT Mill Toolkit layout components allow you to position your UI components on the screen in a hierarchical fashion, much like in conventional Java UI toolkits such as AWT, Swing, or SWT. In addition, you can approach the layout from the direction of the web with the **CustomLayout** component, which you can use to write your layout as a template in XHTML that provides locations of any contained components.

The moral of the story is that, because IT Mill Toolkit is intended for web applications, appearance is of high importance. The solutions have to be the best of both worlds and satisfy artists of both kind: code and graphics. On the API side, the layout is controlled by UI components, particularly the layout components. On the visual side, it is controlled by themes. Themes can contain any HTML, CSS, and JavaScript that you or your web artists create to make people feel good about your software.

5.2. Layout Components

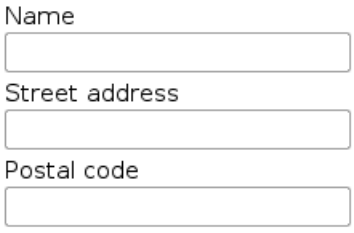
5.2.1. VerticalLayout and HorizontalLayout

VerticalLayout and **HorizontalLayout** components are containers for laying out components either vertically or horizontally, respectively. Some components, such as **Window**, have a **VerticalLayout** as the root layout, which you can set with `setLayout()`.

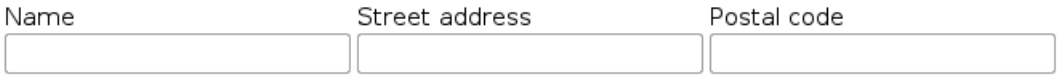
Typical use of the layouts goes as follows:

```
VerticalLayout vertical = new VerticalLayout ();
vertical.addComponent(new TextField("Name"));
vertical.addComponent(new TextField("Street address"));
vertical.addComponent(new TextField("Postal code"));
main.addComponent(vertical);
```

The text fields have a label attached, which will by default be placed above the field. The layout will look on screen as follows:



Using **HorizontalLayout** gives the following layout:



The layouts can have spacing between the horizontal or vertical cells, defined with `setSpacing()`, as described in Section 5.3.3, “Layout Cell Spacing”. The contained components can be aligned within their cells with `setComponentAlignment()`, as described in Section 5.3.2, “Layout Cell Alignment”.

You can use `setWidth()` and `setHeight()` to specify width and height of a component in either fixed units or relatively with a percentage.

Sizing Contained Components

The components contained within an ordered layout can be laid out in a number of different ways depending on how you specify their height or width in the primary direction of the layout component.

Figure 5.1. Component Widths in HorizontalLayout

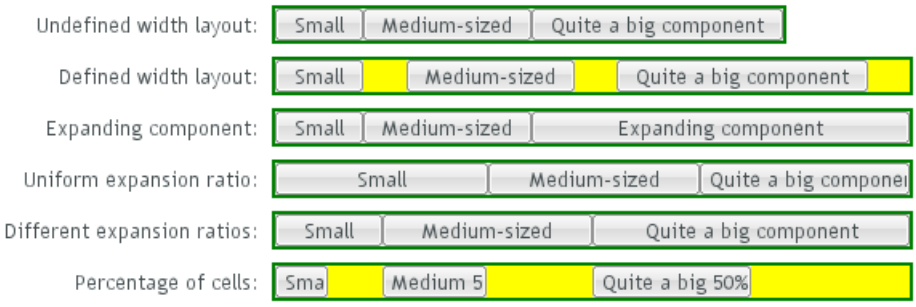


Figure 5.1, “Component Widths in **HorizontalLayout**” above gives a summary of the sizing options for a **HorizontalLayout**. Let us break down the figure as follows.

If a **VerticalLayout** has undefined height or **HorizontalLayout** undefined width, the layout will shrink to fit the contained components so that there is no extra space between them.

```
HorizontalLayout fittingLayout = new HorizontalLayout();
fittingLayout.setWidth(Sizeable.SIZE_UNDEFINED, 0);
fittingLayout.addComponent(new Button("Small"));
fittingLayout.addComponent(new Button("Medium-sized"));
fittingLayout.addComponent(new Button("Quite a big component"));
parentLayout.addComponent(fittingLayout);
```

If such a vertical layout continues below the bottom of a window (a **Window** object), the window will pop up a vertical scroll bar on the right side of the window area. This way, you get a "web page".

If you set a **HorizontalLayout** to a defined size horizontally or a **VerticalLayout** vertically, and there is space left over from the contained components, the extra space is distributed equally between the component cells. The components are aligned within these cells according to their alignment setting, top left by default, as in the example below.

```
fixedLayout.setWidth("400px");
```

Using percentual sizes for contained components requires answering the question, "Percentage of what?" There is no sensible default answer for this question in the current implementation of the layouts, so in practice, you may not define "100%" size alone.

Often, you want to have one component that takes all the available space left over from other components. You need to set its size as 100% and set it as *expanding* with `setExpandRatio()`. The second parameter for the method is an expansion ratio, which is relevant if there are more than one expanding component, but its value is irrelevant for a single expanding component.

```
HorizontalLayout layout = new HorizontalLayout();
layout.setWidth("400px");
```

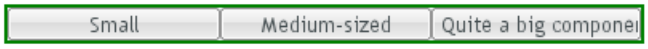
```
// These buttons take the minimum size.
layout.addComponent(new Button("Component"));
layout.addComponent(new Button("Component"));

// This button will expand.
Button expandButton = new Button("Component");

// Use 100% of the expansion cell's width.
expandButton.setWidth("100%");

// The component must be added to layout before setting the ratio.
layout.addComponent(expandButton);

// Set the component's cell to expand.
layout.setExpandRatio(expandButton, 1.0f);

parentLayout.addComponent(layout);
```

Notice that you must call `setExpandRatio()` *after* `addComponent()`, because the layout can not operate on an component that it doesn't (yet) include.

Warning: *A layout that contains components with percentual size must have a defined size*! If a layout has undefined size and component has, say, 100% size, the component would fill the space given by the layout, while the layout would shrink to fit the space taken by the component, which is a paradox. This requirement holds for height and width separately. The debug mode allows detecting such invalid cases; see Section 9.1.1, “Debug Mode”.

If you specify an expand ratio for multiple components, they will all try to use the available space according to the ratio.

```
HorizontalLayout layout = new HorizontalLayout();
layout.setWidth("400px");

// Create three equally expanding components.
String[] captions = { "Small", "Medium-sized", "Quite a big component" };
for (int i = 1; i <= 3; i++) {
    Button button = new Button(captions[i-1]);
    button.setWidth("100%");
    layout.addComponent(button);

    // Have uniform 1:1:1 expand ratio.
    layout.setExpandRatio(button, 1.0f);
}
```



As we used the same ratio for each components, the ones with more content may be have the content cut. Below, we use differing ratios:

```
// Expand ratios for the components are 1:2:3.
layout.setExpandRatio(button, i * 1.0f);
```

If the size of the expanding components is defined as a percentage (typically "100%"), the ratio is calculated from the *overall* space available for the relatively sized components. For example, if you have a 100 pixels wide layout with two cells with 1.0 and 4.0 respective expansion ratios, and both the components in the

layout are set as `setWidth("100%")`, the cells will have respective widths of 20 and 80 pixels, regardless of the minimum size of the components.

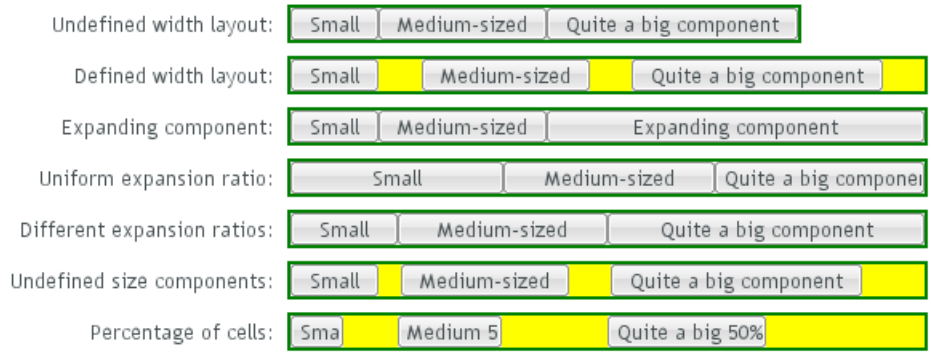
However, if the size of the contained components is undefined or fixed, the expansion ratio is of the *excess* available space. In this case, it is the excess space that expands, not the components.

```
        for (int i = 1; i <= 3; i++) {
    // Button with undefined size.
    Button button = new Button(captions[i - 1]);

    layout4.addComponent(button);

    // Expand ratios are 1:2:3.
    layout4.setExpandRatio(button, i * 1.0f);
}
```

It is not meaningful to combine expanding components with percentually defined size and components with fixed or undefined size. Such combination can lead to a very expected size for the percentually sized components.

A percentual size of a component defines the size of the component *within it's cell*. Usually, you use "100%", but a smaller percentage or a fixed size (smaller than the cell size) will leave an empty space in the cell and align the component within the cell according to its alignment setting, top left by default.

```
HorizontalLayout layout50 = new HorizontalLayout();
layout50.setWidth("400px");

String[] captions1 = { "Small 50%", "Medium 50%", "Quite a big 50%" };
for (int i = 1; i <= 3; i++) {
    Button button = new Button(captions1[i-1]);
    button.setWidth("50%");
    layout50.addComponent(button);

    // Expand ratios for the components are 1:2:3.
    layout50.setExpandRatio(button, i * 1.0f);
}
parentLayout.addComponent(layout50);
```



5.2.2. GridLayout

GridLayout container lays components out on a grid of defined width and height. The columns and rows of the grid serve as coordinates that are used for laying out components on the grid. Each component can use multiple cells from the grid, defined as an area (x1,y1,x2,y2), although they typically take up only a single grid cell.

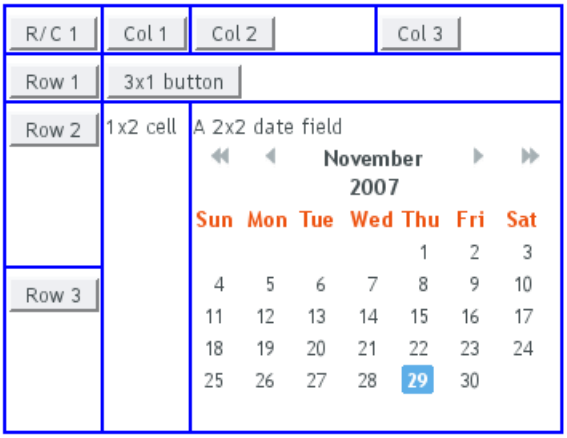
The grid layout maintains a cursor for adding components in left-to-right, top-to-bottom order. If the cursor goes past the bottom-right corner, it will automatically extend the grid downwards.

The following example demonstrates the use of **GridLayout**. The `addComponent` takes a component and optional coordinates. The coordinates can be given for a single cell or for an area in x,y (column,row) order. The coordinate values have a base value of 0. If coordinates are not given, the cursor will be used.

```
/* Create a 4 by 4 grid layout. */
GridLayout grid = new GridLayout(4, 4);
grid.addStyleName("example-gridlayout");

/* Fill out the first row using the cursor. */
grid.addComponent(new Button("R/C 1"));
for (int i = 0; i < 3; i++) {
    grid.addComponent(new Button("Col " + (grid.getCursorX() + 1)));
}

/* Fill out the first column using coordinates. */
for (int i = 1; i < 4; i++) {
    grid.addComponent(new Button("Row " + i), 0, i);
}

/* Add some components of various shapes. */
grid.addComponent(new Button("3x1 button"), 1, 1, 3, 1);
grid.addComponent(new Label("1x2 cell"), 1, 2, 1, 3);
InlineDateField date = new InlineDateField("A 2x2 date field");
date.setResolution(DateField.RESOLUTION_DAY);
grid.addComponent(date, 2, 2, 3, 3);
```

The resulting layout will look as follows. The borders have been made visible to illustrate the layout cells.

Figure 5.2. The Grid Layout Component



For a more complete example of grid layout, please see Section 1.2.2, “Calculator”.

A component to be placed on the grid must not overlap with existing components. A conflict causes throwing a **GridLayout.OverlapsException**.

Sizing Grid Cells

You can define the size of both a grid layout and its components in either fixed or percentual units, or leave the size undefined altogether, as described in Section 4.18.1, “Sizing Components through **Sizeable** interface”. Section 5.3.1, “Layout Size” gives an introduction to sizing of layouts.

The size of the **GridLayout** component is undefined by default, so it will shrink to fit the size of the components placed inside it. In most cases, especially if you set a defined size for the layout but do not set the contained components to full size, there will be some unused space. The position of the non-full components within the grid cells will be determined by their *alignment*. See Section 5.3.2, "Layout Cell Alignment" for details on how to align the components inside the cells.

The components contained within a **GridLayout** layout can be laid out in a number of different ways depending on how you specify their height or width. The layout options are similar to **HorizontalLayout** and **VerticalLayout**, as described in Section 5.2.1, "**VerticalLayout** and **HorizontalLayout**".

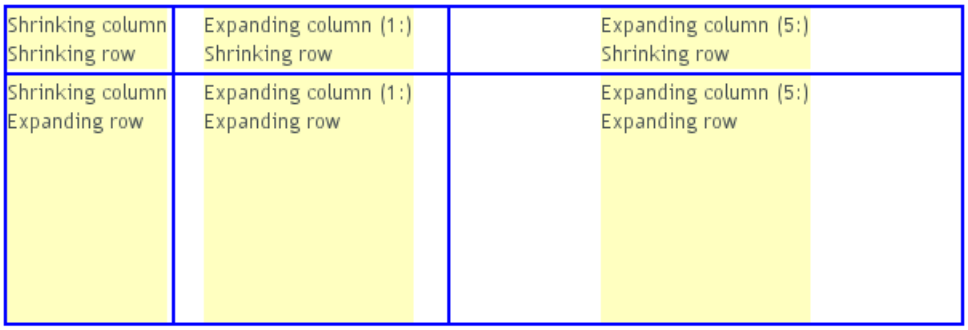
Warning: *A layout that contains components with percentual size must have a defined size*! If a layout has undefined size and component has, say, 100% size, the component would fill the space given by the layout, while the layout would shrink to fit the space taken by the component, which is a paradox. This requirement holds for height and width separately. The debug mode allows detecting such invalid cases; see Section 9.1.1, "Debug Mode".

Often, you want to have one or more rows or columns that take all the available space left over from non-expanding rows or columns. You need to set the rows or columns as *expanding* with `setRowExpandRatio()` and `setColumnExpandRatio()`. The first parameter for these methods is the index of the row or column to set as expanding. The second parameter for the methods is an expansion ratio, which is relevant if there are more than one expanding row or column, but its value is irrelevant if there is only one. With multiple expanding rows or columns, the ratio parameter sets the relative portion how much a specific row/column will take in relation with the other expanding rows/columns.

```
GridLayout grid = new GridLayout(3,2);
grid.addStyleName("gridexpandratio");

// Layout containing relatively sized components must have a defined size.
grid.setWidth("600px");
grid.setHeight("200px");

// Add content
grid.addComponent(new Label("Shrinking column<br/>Shrinking row", Label.CONTENT_XHTML));
grid.addComponent(new Label("Expanding column (1:)<br/>Shrinking row",
Label.CONTENT_XHTML));
grid.addComponent(new Label("Expanding column (5:)<br/>Shrinking row",
Label.CONTENT_XHTML));

grid.addComponent(new Label("Shrinking column<br/>Expanding row", Label.CONTENT_XHTML));
grid.addComponent(new Label("Expanding column (1:)<br/>Expanding row",
Label.CONTENT_XHTML));
grid.addComponent(new Label("Expanding column (5:)<br/>Expanding row",
Label.CONTENT_XHTML));

// Set different expansion ratios for the two columns
grid.setColumnExpandRatio(1, 1);
grid.setColumnExpandRatio(2, 5);

// Set the bottom row to expand
grid.setRowExpandRatio(1, 1);

// Align and size the labels.
for (int col=0; col<grid.getColumns(); col++) {
    for (int row=0; row<grid.getRows(); row++) {
        Component c = grid.getComponent(col, row);
        grid.setComponentAlignment(c, Alignment.TOP_CENTER);

        // Make the labels high to illustrate the empty horizontal space.
        if (col != 0 || row != 0) {
            c.setHeight("100%");
        }
```

```
    }
}
```

Figure 5.3. Expanding Rows and Columns in GridLayout



If the size of the contained components is undefined or fixed, the expansion ratio is of the *excess* space, as in Figure 5.3, “Expanding Rows and Columns in **GridLayout**” (excess horizontal space shown in white). However, if the size of the all the contained components in the expanding rows or columns is defined as a percentage, the ratio is calculated from the *overall* space available for the percentually sized components. For example, if we had a 100 pixels wide grid layout with two columns with 1.0 and 4.0 respective expansion ratios, and all the components in the grid were set as `setWidth("100%")`, the columns would have respective widths of 20 and 80 pixels, regardless of the minimum size of their contained components.

CSS Style Rules

```
.i-gridlayout {}
.i-gridlayout-margin {}
```

The i-gridlayout is the root element of the **GridLayout** component. The i-gridlayout-margin is a simple element inside it that allows setting a padding between the outer element and the cells.

For styling the individual grid cells, you should style the components inserted in the cells. The implementation structure of the grid can change, so depending on it, as is done in the example below, is not generally recommended. Normally, if you want to have, for example, a different color for a certain cell, just make set the component inside it `setSizeFull()`, and add a style name for it. Sometimes you may need to use a layout component between a cell and its actual component just for styling.

The following example shows how to make the grid borders visible, as in Figure 5.3, “Expanding Rows and Columns in **GridLayout**”.

```
.i-gridlayout-gridexpandratio {
    background: blue; /* Creates a "border" around the layout grid. */
    margin:     10px; /* Empty space around the layout. */
}

/* Add padding through which the background color of the grid shows. */
.i-gridlayout-gridexpandratio .i-gridlayout-margin {
    padding: 2px;
}

/* Add cell borders and make the cell backgrounds white.
 * Warning: This depends heavily on the HTML structure. */
.i-gridlayout-gridexpandratio > div > div > div {
    padding:    2px;   /* Layout's background will show through. */
    background: white; /* The cells will be colored white. */
}

/* Components inside the layout. This is a safe way to style the cells. */
```

```
.i-gridlayout-gridexpandratio .i-label {
    text-align: left;
    background: #ffffc0; /* Pale yellow */
}
```

You should beware of `margin`, `padding`, and `border` settings in CSS as they can mess up the layout. The dimensions of layouts are calculated in the Client-Side Engine of IT Mill Toolkit and some settings can interfere with these calculations.

5.2.3. Panel

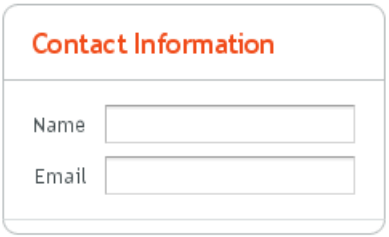
Panel is a simple container with a frame and an optional caption. The content are has an inner layout component for laying out the contained components.

The caption can have an icon in addition to the text.

```
// Create a panel with a caption.
final Panel panel = new Panel("Contact Information");
panel.addStyleName("panelexample");

// The width of a Panel is 100% by default, make it
// shrink to fit the contents.
panel.setWidth(Sizeable.SIZE_UNDEFINED, 0);

// Create a layout inside the panel, and have some margin around it.
final FormLayout form = new FormLayout();
form.setMargin(true);

// Add some components
form.addComponent(new TextField("Name"));
form.addComponent(new TextField("Email"));

// Set the layout as the root layout of the panel
panel.setLayout(form);
```

The resulting layout will look as follows.

Figure 5.4. A Panel Layout



CSS Style Rules

```
.i-panel {}
.i-panel-caption {}
.i-panel-nocaption {}
.i-panel-content {}
.i-panel-deco {}
```

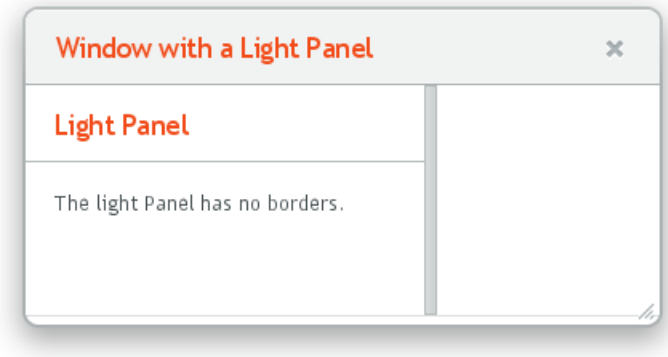
The entire panel has `i-panel` style. A panel consists of three parts: the caption, content, and bottom decorations (shadow). These can be styled with `i-panel-caption`, `i-panel-content`, and `i-panel-deco`, respectively. If the panel has no caption, the caption element will have the style `i-panel-nocaption`.

The *light style* for the **Panel** is a predefined style that has now borders or border decorations for the panel. You enable it simply by adding the `light` style name for the panel, as is done in the example below.

The light style is typical when using a **Panel** as the root layout of a window or some similar layout, as in the example below.

```
// Have a window with a SplitPanel.
final Window window = new Window("Window with a Light Panel");
window.setWidth("400px");
window.setHeight("200px");
final SplitPanel splitter = new SplitPanel(SplitPanel.ORIENTATION_HORIZONTAL);
window.setLayout(splitter);

// Create a panel with a caption.
final Panel light = new Panel("Light Panel");
light.setSizeFull();

// The "light" style is a predefined style without borders.
light.addStyleName("light");

light.addComponent(new Label("The light Panel has no borders."));
light.getLayout().setMargin(true);

// The Panel will act as a "caption" of the left panel in SplitPanel.
splitter.addComponent(light);
splitter.setSplitPosition(250, Sizeable.UNITS_PIXELS);

main.addWindow(window);
```

Figure 5.5. A Panel with Light Style



5.2.4. TabSheet

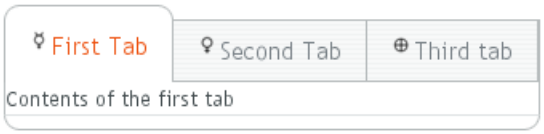
The **TabSheet** is a multicomponent container that allows switching between the components with "tabs". The tabs are organized as a tab bar at the top of the tab sheet. Clicking on a tab opens its contained component in the main display area of the layout.

New tabs can be added simply with the `addComponent()` method, but doing so leaves them without a caption. You can set the caption with `setTabCaption()` or simply use the `addTab()` method to create tabs and give them a caption. In addition to a caption, tabs can contain an icon, which you can define either in the `addtab()` call or set later with `setTabIcon()`.

The following example demonstrates the creation of a simple tab sheet, where each the tabs shows a different **Label** component. The tabs have an icon, which are loaded as Java class loader resources from the WAR package of the application.

```
final TabSheet tabsheet = new TabSheet();
tabsheet.addStyleName("tabsheetexample");

// Make the tabsheet to shrink to fit the contents.
tabsheet.setSizeUndefined();

tabsheet.addTab(new Label("Contents of the first tab"),
                "First Tab",
                new ClassResource("images/Mercury_small.png", this));
tabsheet.addTab(new Label("Contents of the second tab"),
                "Second Tab",
                new ClassResource("images/Venus_small.png", this));
tabsheet.addTab(new Label("Contents of the third tab"),
                "Third tab",
                new ClassResource("images/Earth_small.png", this));
```

Figure 5.6. A Simple TabSheet Layout



The `hideTabs()` method allows hiding the tab bar entirely. This can be useful in tabbed document interfaces (TDI) when there is only one tab. An individual tab can be made invisible by making its component invisible with `setVisible(false)`. A tab can be disabled by disabling its component with `setEnabled(false)`. A tab can be selected programmatically with `setSelectedTab()`.

Clicking on a tab selects it. This fires a **TabSheet.SelectedTabChangeEvent**, which can be handled with the **TabSheet.SelectedTabChangeListener**. The source component of the event, which you can retrieve with `getSource()` method of the event, will be the **TabSheet** component. You can find out the currently selected component with `getSelectedTab()`.

The example below demonstrates handling **TabSheet** related events and enabling and disabling tabs. The sort of logic used in the example is useful in sequential user interfaces, often called *wizards*, where the user goes through the tabs one by one, but can return back if needed.

```
import com.itmill.toolkit.ui.*;
import com.itmill.toolkit.ui.Button.ClickEvent;
import com.itmill.toolkit.ui.TabSheet.SelectedTabChangeEvent;

public class TabSheetExample extends CustomComponent
        implements Button.ClickListener, TabSheet.SelectedTabChangeListener {
    TabSheet tabsheet = new TabSheet();
    Button tab1 = new Button("Push this button");
    Label  tab2 = new Label("Contents of Second Tab");
    Label  tab3 = new Label("Contents of Third Tab");

    TabSheetExample () {
        setCompositionRoot (tabsheet);

        /* Listen for changes in tab selection. */
        tabsheet.addListener(this);

        /* First tab contains a button, for which we listen button click events. */
        tab1.addListener(this);
        tabsheet.addTab(tab1, "First Tab", null);

        /* A tab that is initially invisible. */
        tab2.setVisible(false);
        tabsheet.addTab(tab2, "Second Tab", null);
```

```
        /* A tab that is initially disabled. */
        tab3.setEnabled(false);
        tabsheet.addTab(tab3, "Third tab", null);
    }

    public void buttonClick(ClickEvent event) {
        /* Enable the invisible and disabled tabs. */
        tab2.setVisible(true);
        tab3.setEnabled(true);

        /* Change selection automatically to second tab. */
        tabsheet.setSelectedTab(tab2);
    }

    public void selectedTabChange(SelectedTabChangeEvent event) {
        /* Cast to a TabSheet. This isn't really necessary in this example,
         * as we have only one TabSheet component, but would be useful if
         * there were multiple TabSheets. */
        TabSheet source = (TabSheet) event.getSource();
        if (source == tabsheet) {
            /* If the first tab was selected. */
            if (source.getSelectedTab() == tab1) {
                tab2.setVisible(false);
                tab3.setEnabled(false);
            }
        }
    }
}
```

Figure 5.7. A TabSheet with Hidden and Disabled Tabs

5.3. Layout Formatting

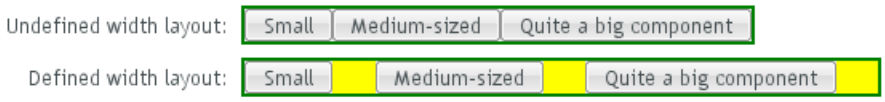
While the formatting of layouts is mainly done with style sheets, just as with other components, style sheets are not ideal or even possible to use in some situations. For example, CSS does not allow defining the spacing of table cells, which is done with the `cellspacing` attribute in HTML.

Moreover, as many layout sizes are calculated dynamically in the Client-Side Engine of IT Mill Toolkit, some CSS settings can fail altogether.

5.3.1. Layout Size

The size of a layout component can be specified with the `setWidth()` and `setHeight()` methods defined in the **Sizeable** interface, just like for any component. It can also be undefined, in which case the layout shrinks to fit the component(s) inside it. Section 4.18.1, “Sizing Components through **Sizeable** interface” gives details on the interface.

Figure 5.8. HorizontalLayout with Undefined vs Defined size



Many layout components take 100% width by default, while have the height undefined.

The sizes of components inside a layout can also be defined as a percentage of the space available in the layout, for example with `setWidth("100%");` or with the (most commonly used method) `setFullSize()` that sets 100% size in both directions. If you use a percentage in a **HorizontalLayout**, **VerticalLayout**, or **GridLayout**, you will also have to set the component as *expanding*, as noted below.

Warning: *A layout that contains components with percentual size must have a defined size*! If a layout has undefined size and component has, say, 100% size, the component would fill the space given by the layout, while the layout would shrink to fit the space taken by the component, which is a paradox. This requirement holds for height and width separately. The debug mode allows detecting such invalid cases; see Section 9.1.1, “Debug Mode”.

For example:

```
// This takes 100% width but has undefined height.
VerticalLayout layout = new VerticalLayout();

// A button that takes all the space available in the layout.
Button button = new Button("100%x100% button");
button.setSizeFull();
layout.addComponent(button);

// We must set the layout to a defined height vertically, in this
// case 100% of its parent layout, which also must not have
// undefined size.
layout.setHeight("100%");
```

The default layout of **Window** and **Panel** is **VerticalLayout** with undefined height. If you insert enough components in such a layout, it will grow outside the bottom of the view area and scrollbars will appear in the browser. If you want to have your application to use all the browser view, nothing more or less, you should use `setFullSize()` for the root layout.

```
// Create the main window.
Window main = new Window("Main Window");
setMainWindow(main);

// Use full size.
main.getLayout().setSizeFull();
```

Expanding Components

If you set a **HorizontalLayout** to a defined size horizontally or a **VerticalLayout** vertically, and there is space left over from the contained components, the extra space is distributed equally between the component cells. The components are aligned within these cells, according to their alignment setting, top left by default, as in the example below.

Often, you don't want such empty space, but want to have one or more components to take all the leftover space. You need to set such a component to 100% size and use `setExpandRatio()`. If there is just one such expanding component in the layout, the ratio parameter is irrelevant.

If you set multiple components as expanding, the expand ratio dictates how large proportion of the available space (overall or excess depending on whether the components are sized as a percentage or not) each component takes. In the example below, the buttons have 1:2:3 ratio for the expansion.

GridLayout has corresponding method for both of its directions, `setRowExpandRatio()` and `setColumnExpandRatio()`.

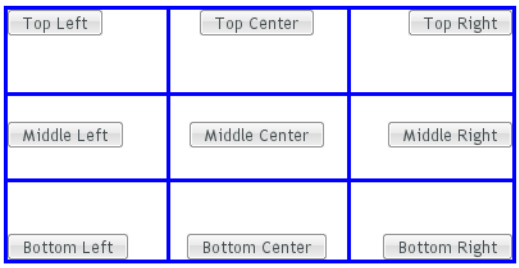
Expansion is dealt in detail in the documentation of the layout components that support it. See Section 5.2.1, "**VerticalLayout** and **HorizontalLayout**" and Section 5.2.2, "**GridLayout**" for details on components with relative sizes.

5.3.2. Layout Cell Alignment

You can set the alignment of the component inside a specific layout cell with the `setComponentAlignment()` method. The method takes as its parameters the component contained in the cell to be formatted, and the horizontal and vertical alignment.

Figure 5.9, "Cell Alignments" below illustrates the alignment of components within a **GridLayout**.

Figure 5.9. Cell Alignments



The easiest way to set alignments is to use the constants defined in the **Alignment** class. Let us look how the buttons in the top row of the above **GridLayout** are aligned with constants:

```
// Create a grid layout
final GridLayout grid = new GridLayout(3, 3);

grid.setWidth(400, Sizeable.UNITS_PIXELS);
grid.setHeight(200, Sizeable.UNITS_PIXELS);

Button topleft = new Button("Top Left");
grid.addComponent(topleft, 0, 0);
grid.setComponentAlignment(topleft, Alignment.TOP_LEFT);

Button topcenter = new Button("Top Center");
grid.addComponent(topcenter, 1, 0);
grid.setComponentAlignment(topcenter, Alignment.TOP_CENTER);

Button topright = new Button("Top Right");
grid.addComponent(topright, 2, 0);
grid.setComponentAlignment(topright, Alignment.TOP_RIGHT);
...
```

The following table lists all the alignment constants by their respective locations:

Table 5.1. Alignment Constants

Alignment.TOP_LEFT	**Alignment.TOP_CENTER**	**Alignment.TOP_RIGHT**
Alignment.MIDDLE_LEFT	**Alignment.MIDDLE_CENTER**	**Alignment.MIDDLE_RIGHT**
Alignment.BOTTOM_LEFT	**Alignment.BOTTOM_CENTER**	**Alignment.BOTTOM_RIGHT**

Another way to specify the alignments is to create an **Alignment** object and specify the horizontal and vertical alignment with separate constants. You can specify either of the directions, in which case the other alignment direction is not modified, or both with a bitmask operation between the two directions.

```
Button middleleft = new Button("Middle Left");
grid.addComponent(middleleft, 0, 1);
grid.setComponentAlignment(middleleft, new Alignment(Bits.ALIGNMENT_VERTICAL_CENTER |
                                                     Bits.ALIGNMENT_LEFT));

Button middlecenter = new Button("Middle Center");
grid.addComponent(middlecenter, 1, 1);
grid.setComponentAlignment(middlecenter, new Alignment(Bits.ALIGNMENT_VERTICAL_CENTER |
                                                    Bits.ALIGNMENT_HORIZONTAL_CENTER));

Button middleright = new Button("Middle Right");
grid.addComponent(middleright, 2, 1);
grid.setComponentAlignment(middleright, new Alignment(Bits.ALIGNMENT_VERTICAL_CENTER |
                                                      Bits.ALIGNMENT_RIGHT));
```

Obviously, you may combine only one vertical bitmask with one horizontal bitmask, though you may leave either one out. The following table lists the available alignment bitmask constants:

Table 5.2. Alignment Bitmasks

Horizontal	**Bits.ALIGNMENT_LEFT**
	Bits.ALIGNMENT_HORIZONTAL_CENTER
	Bits.ALIGNMENT_RIGHT
Vertical	**Bits.ALIGNMENT_TOP**
	Bits.ALIGNMENT_VERTICAL_CENTER
	Bits.ALIGNMENT_BOTTOM

You can determine the current alignment of a component with `getComponentAlignment()`, which returns an **Alignment** object. The class provides a number of getter methods for decoding the alignment, which you can also get as a bitmask value.

5.3.3. Layout Cell Spacing

The **VerticalLayout**, **HorizontalLayout**, and **GridLayout** layouts offer a `setSpacing()` method for enabling space between the cells in the layout. Enabling the spacing adds a spacing style for all cells except the first.

To enable spacing, simply call `setSpacing(true)` for the layout as follows:

```
HorizontalLayout layout2 = new HorizontalLayout();
layout2.addStyleName("spacingexample");
layout2.setSpacing(true);
layout2.addComponent(new Button("Component 1"));
layout2.addComponent(new Button("Component 2"));
layout2.addComponent(new Button("Component 3"));
```

```
VerticalLayout layout4 = new VerticalLayout();
layout4.addStyleName("spacingexample");
layout4.setSpacing(true);
layout4.addComponent(new Button("Component 1"));
layout4.addComponent(new Button("Component 2"));
layout4.addComponent(new Button("Component 3"));
```

Enabling the spacing adds spacing style names to all the cells except the first one (on left or top), thereby allowing setting of amount of spacing between the cells. Spacing can be horizontal (for **HorizontalLayout**) or vertical (for **VerticalLayout**), or both for **GridLayout**. The name of the spacing style is the base name of the component style name plus "`-spacing-on`" for horizontal and vertical spacing, respectively, as shown in the following table:

Table 5.3. Spacing Style Names

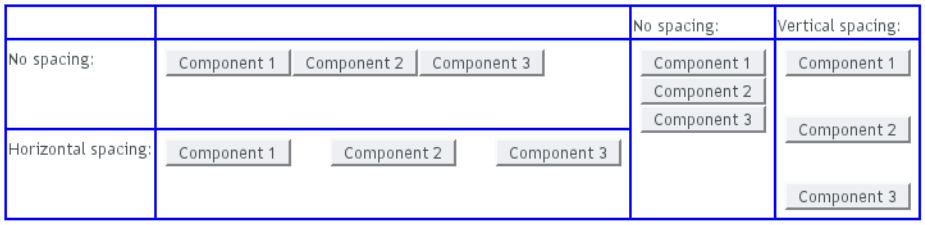
VerticalLayout	`i-orderedlayout-spacing-on`
HorizontalLayout	`i-orderedlayout-spacing-on`
GridLayout	`i-gridlayout-spacing-on`

Below we specify the exact amount of spacing for the code example given above, for the elements with the "`spacingexample`" style name:

```
/* Set horizontal cell spacing in specific layout with "spacingexample" style. */
.i-orderedlayout-spacingexample .i-orderedlayout-spacing-on {
    padding-left: 30px;
}

/* Set vertical cell spacing in specific layout with "spacingexample" style. */
.i-orderedlayout-spacingexample .i-orderedlayout-spacing-on {
    margin-top: 30px;

/* Set vertical and horizontal cell spacing in specific gridlayout with "spacingexample"
 style. */
.i-gridlayout-spacingexample .i-gridlayout-spacing-on {
    margin-top: 30px;
    margin-left: 50px;
}
```

The resulting layouts will look as shown in Figure 5.10, “Layout Spacings” below, which also shows the layouts with no spacing.

Figure 5.10. Layout Spacings



Note

Spacing is unrelated to "cell spacing" in HTML tables. While many layout components are implemented with HTML tables in the browser, this implementation is not guaranteed to stay the same and at least **Vertical-**/**HorizontalLayout** could be implemented with `<div>` elements as well.

In fact, as GWT compiles widgets separately for different browsers, the implementation could even vary between browsers.

5.3.4. Layout Margins

By default, layout components do not have any margin around them. You can add margin with CSS directly to the layout component. Below we set margins for a specific layout component:

```
layout1.addStyleName("marginexample1");
```

```
.i-orderedlayout-marginexample1 .i-orderedlayout-margin { padding-left:   200px; }
.i-orderedlayout-marginexample1 .i-orderedlayout-margin { padding-right:  100px; }
.i-orderedlayout-marginexample1 .i-orderedlayout-margin { padding-top:    50px;  }
.i-orderedlayout-marginexample1 .i-orderedlayout-margin { padding-bottom: 25px;  }
```

In addition to pure CSS method, you can enable margin around the layout with `setMargin(true)`. The margin element has some default margin widths, but you can adjust the widths in CSS if you need to.

Let us consider the following example, where we enable the margin on all sides of the layout:

```
// Create a layout
HorizontalLayout layout2 = new HorizontalLayout();
containinglayout.addComponent(new Label("Layout with margin on all sides:"));
containinglayout.addComponent(layout2);

// Set style name for the layout to allow styling it
layout2.addStyleName("marginexample");

// Have margin on all sides around the layout
layout2.setMargin(true);

// Put something inside the layout
layout2.addComponent(new Label("Cell 1"));
layout2.addComponent(new Label("Cell 2"));
layout2.addComponent(new Label("Cell 3"));
```

You can enable the margins only for specific sides. The margins are specified for the `setMargin()` method in clockwise order for top, right, bottom, and left margin. The following would enable the top and left margins:

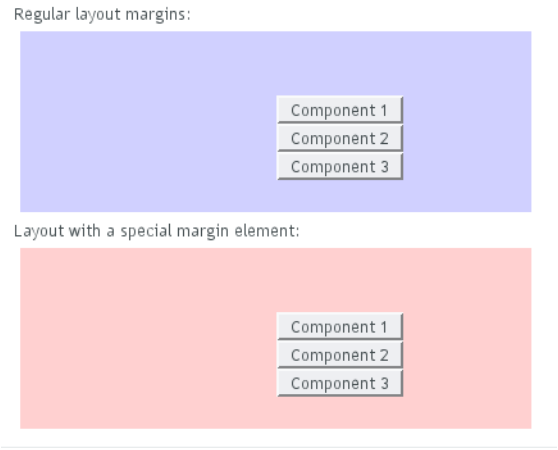
```
layout2.setMargin(true, false, false, true);
```

You can specify the actual margin widths in the CSS if you are not satisfied with the default widths:

```
.i-orderedlayout-marginexample .i-orderedlayout-margin-left   {padding-left:   200px;}
.i-orderedlayout-marginexample .i-orderedlayout-margin-right  {padding-right:  100px;}
.i-orderedlayout-marginexample .i-orderedlayout-margin-top    {padding-top:    50px; }
.i-orderedlayout-marginexample .i-orderedlayout-margin-bottom {padding-bottom: 25px; }
```

The resulting margins are shown in Figure 5.11, “Layout Margins” below. The two ways produce identical margins.

Figure 5.11. Layout Margins



CSS Style Rules

The CSS style names for the margin widths for `setMargin()` consist of the specific layout name plus `-margin-left` and so on. Below, the style rules are given for **VerticalLayout**:

```
.i-orderedlayout-margin-left   {padding-left:   ___px;}
.i-orderedlayout-margin-right  {padding-right:  ___px;}
.i-orderedlayout-margin-top    {padding-top:    ___px;}
.i-orderedlayout-margin-bottom {padding-bottom: ___px;}
```

5.4. Custom Layouts

While it is possible to create almost any typical layout with the standard layout components, it is sometimes best to separate the layout completely from code. With the **CustomLayout** component, you can write your layout as a template in XHTML that provides locations of any contained components. The layout template is included in a theme. This separation allows the layout to be designed separately from code, for example using WYSIWYG web designer tools such as Adobe Dreamweaver.

A template is a HTML file located under `layouts` folder under a theme folder under the `WebContent/ITMILL/themes/` folder, for example, `WebContent/ITMILL/themes/`*`themename/layouts/mylayout.html`*. (Notice that the root path `WebContent/ITMILL/themes/` for themes is fixed.) A template can also be provided dynamically from an **InputStream**, as explained below. A template includes `<div>` elements with a *`location`* attribute that defines the location identifier. All custom layout HTML-files must be saved using UTF-8 character encoding.

```
<table width="100%" height="100%">
  <tr height="100%">
    <td>
      <table align="center">
        <tr>
          <td align="right">User name:</td>
          <td><div location="username"></div></td>
        </tr>
        <tr>
          <td align="right">Password:</td>
          <td><div location="password"></div></td>
        </tr>
      </table>
    </td>
```

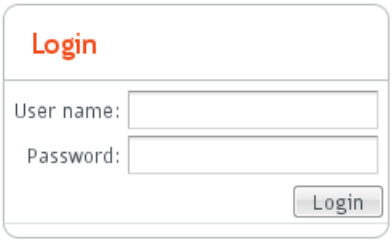
```
  </tr>
  <tr>
    <td align="right" colspan="2"><div location="okbutton"></div></td>
  </tr>
</table>
```

The client-side engine of IT Mill Toolkit will replace contents of the location elements with the components. The components are bound to the location elements by the location identifier given to `addComponent()`, as shown in the example below.

```
// Have a Panel where to put the custom layout.
final Panel panel = new Panel("Login");
panel.setSizeUndefined();
main.addComponent(panel);

// Create the custom layout from the "layoutname.html" template.
final CustomLayout custom = new CustomLayout("layoutname");
custom.addStyleName("customlayoutexample");

// Use it as the layout of the Panel.
panel.setLayout(custom);

// Create a few components and bind them to the location tags
// in the custom layout.
TextField username = new TextField();
custom.addComponent(username, "username");

TextField password = new TextField();
custom.addComponent(password, "password");

final Button ok = new Button("Login");
custom.addComponent(ok, "okbutton");
```

The resulting layout is shown below in Figure 5.12, “Example of a Custom Layout Component”.

Figure 5.12. Example of a Custom Layout Component



You can use `addComponent()` also to replace an existing component in the location given in the second parameter.

In addition to a static template file, you can provide a template dynamically with the **CustomLayout** constructor that accepts an **InputStream** as the template source. For example:

```
new CustomLayout(new ByteArrayInputStream("<b>Template</b>".getBytes()));
```

or

```
new CustomLayout(new FileInputStream(file));
```

Chapter 6. Themes

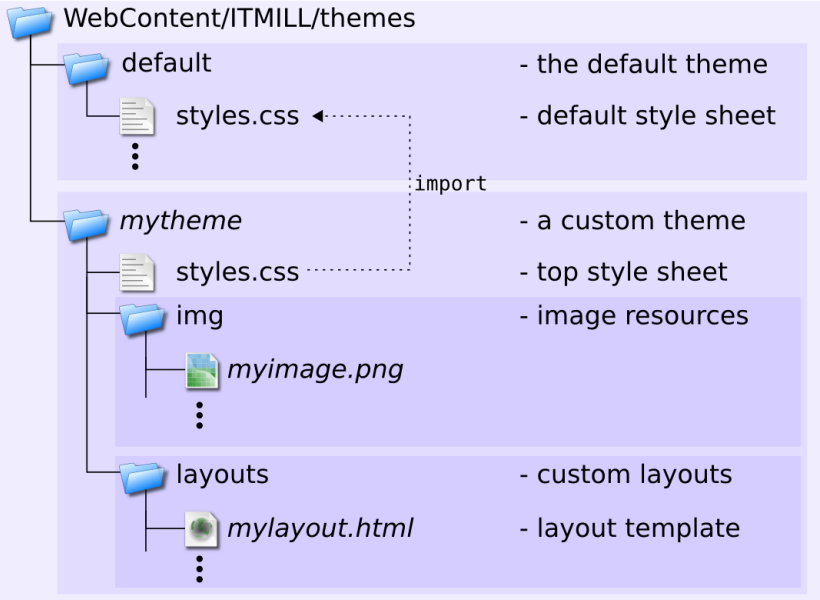
This chapter provides details about using and creating *themes* that control the visual look of web applications using Cascading Style Sheets (CSS) and other theme resources. We provide an introduction to CSS, especially concerning the styling of HTML by element classes.

6.1. Overview

IT Mill Toolkit separates the appearance of the user interface from its logic using *themes*. Themes can include CSS style sheets, custom HTML layouts, and any necessary graphics used by them. Theme resources can also be accessed from an application with **ThemeResource** objects.

You place custom themes under the `WebContents/ITMILL/themes/` folder of the web application. This location is fixed -- the folder "ITMILL" specifies that these are (static) resources specific to IT Mill Toolkit. The folder should normally contain also the built-in default theme, although you can let it be loaded dynamically from the Toolkit JAR (even though that is somewhat inefficient). Figure 6.1, “Theme Contents” illustrates the contents of a theme.

Figure 6.1. Theme Contents



The name of a theme folder defines the name of the theme. The name is used in the `setTheme()` call. A theme must contain the `styles.css` stylesheet, but other contents have free naming. We suggest a convention for naming the folders as `img` for images, `layouts` for custom layouts, and `css` for additional stylesheets.

Custom themes must inherit the default theme (unless you just copy the default theme and use it as a template for efficiency reasons). See the section called “Default Theme” and Section 6.3.3, “Theme Inheritance” for details on inheriting the default theme.

You use a theme with a simple `setTheme()` method call for the **Application** object as follows:

```
public class MyApplication extends com.itmill.toolkit.Application {
    public void init() {
```

```
        setTheme("demo");
        ...
    }
}
```

An application can use different themes for different users and switch between themes during execution. For smaller changes, a theme can contain alternate styles for user interface components, which can be changed as needed.

In addition to style sheets, a theme can contain HTML templates for custom layouts used with **CustomLayout**. See Section 5.4, “Custom Layouts” for details.

Resources provided in a theme can also be accessed using the **ThemeResource** class, as described in Section 3.5.4, “Theme Resources”. This allows using theme resources, such as images, for example in **Embedded** objects and other objects that allow inclusion of images using resources.

6.2. Introduction to Cascading Style Sheets

Cascading Style Sheets or CSS is a technique to separate the appearance of a web page from the content represented in HTML or XHTML. Let us give a short introduction to Cascading Style Sheets and look how they are relevant to software development with IT Mill Toolkit.

6.2.1. Basic CSS Rules

A style sheet is a file that contains a set of *rules*. Each rule consists of one or more *selectors*, separated with commas, and a *declaration block* enclosed in curly braces. A declaration block contains a list of *property* statements. Each property has a label and a value, separated with a colon. A property statement ends with a semicolon.

Let us look at an example:

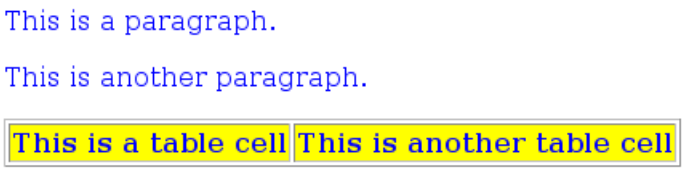
```
p, td {
  color: blue;
}

td {
  background: yellow;
  font-weight: bold;
}
```

In the example above, `p` and `td` are element type selectors that match with `<p>` and `<td>` elements in HTML, respectively. The first rule matches with both elements, while the second matches only with `<td>` elements. Let us assume that you have saved the above style sheet with the name `mystylesheet.css` and consider the following HTML file located in the same folder.

```
<html>
    <head>
          <link rel="stylesheet" href="mystylesheet.css" type="text/css" />
    </head>
    <body>
        <p>This is a paragraph</p>
        <p>This is another paragraph</p>
        <table>
            <tr>
                <td>This is a table cell</td>
                <td>This is another table cell</td>
            </tr>
        </table>
    </body>
</html>
```

The `<link>` element defines the style sheet to use. The HTML elements that match the above rules are emphasized. When the page is displayed in the browser, it will look as shown in the figure below.

Figure 6.2. Simple Styling by Element Type



CSS has an *inheritance* mechanism where contained elements inherit the properties of their parent elements. For example, let us change the above example and define it instead as follows:

```
table {
    color: blue;
    background: yellow;
}
```

All elements contained in the `<table>` element would have the same properties. For example, the text in the contained `<td>` elements would be in blue color.

Each HTML element type accepts a certain set of properties. The `<div>` elements are generic elements that can be used to create almost any layout and formatting that can be created with a specific HTML element type. IT Mill Toolkit uses `<div>` elements extensively, especially for layouts.

Matching elements by their type is, however, rarely if ever used in style sheets for IT Mill Toolkit components or Google Web Toolkit widgets.

6.2.2. Matching by Element Class

Matching HTML elements by the *class* attribute of the elements is the most relevant form of matching with IT Mill Toolkit. It is also possible to match with the *identifier* of a HTML element.

The class of an HTML element is defined with the `class` attribute as follows:

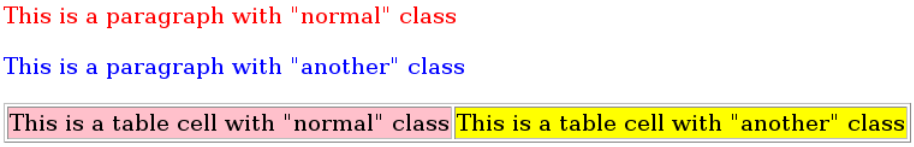
```
<html>
    <body>
        <p class="normal">This is the first paragraph</p>

        <p class="another">This is the second paragraph</p>

        <table>
            <tr>
                <td class="normal">This is a table cell</td>
                <td class="another">This is another table cell</td>
            </tr>
        </table>
    </body>
</html>
```

The class attributes of HTML elements can be matched in CSS rules with a selector notation where the class name is written after a period following the element name. This gives us full control of matching elements by their type and class.

```
p.normal   {color: red;}
p.another  {color: blue;}
td.normal  {background: pink;}
td.another {background: yellow;}
```

The page would look as shown below:

Figure 6.3. Matching HTML Element Type and Class

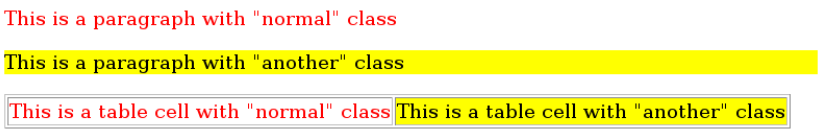


We can also match solely by the class by using the universal selector `*` for the element name, for example `*.normal`. The universal selector can also be left out altogether so that we use just the class name following the period, for example `.normal`.

```
.normal {
    color: red;
}

.another {
    blackground: yellow;
}
```

In this case, the rule will match with all elements of the same class regardless of the element type. The result is shown in Figure 6.4, “Matching Only HTML Element Class”. This example illustrates a technique to make style sheets compatible regardless of the exact HTML element used in drawing a component.

Figure 6.4. Matching Only HTML Element Class



To assure compatibility, we recommend that you use only matching based on the element classes and *do not* match for specific HTML element types in CSS rules, because either IT Mill Toolkit or GWT may use different HTML elements to render some components in the future. For example, IT Mill Toolkit Release 4 used `<div>` elements extensively for layout components. However, Release 5 uses GWT to render the components, and GWT uses `<table>` element to implement most layouts. Similarly, Release 4 used `<div>` element also for buttons, but in Release 5 GWT uses the `<button>` element. IT Mill has little control over how GWT renders its components, so we can not guarantee compatibility in different versions of GWT. However, both `<div>` and `<table>` as well as `<tr>` and `<td>` elements accept most of the same properties, so matching only the class hierarchy of the elements should be compatible in most cases.

6.2.3. Matching by Descendant Relationship

CSS allows matching HTML by their containment relationship. For example, consider the following HTML fragment:

```
<body>
    <p class="mytext">Here is some text inside a div element</p>
    <table class="mytable">
        <tr>
            <td class="mytext">Here is text inside a table and inside a div element.</td>

        </tr>
    </table>
</body>
```

Matching by the class name `.mytext` alone would match both the `<p>` and `<td>` elements. If we want to match only the table cell, we could use the following selector:

```
.mytable .mytext {color: blue;}
```

To match, a class listed in a rule does not have to be an immediate descendant of the previous class, but just a descedant. For example, the selector "`.i-panel .i-button`" would match all elements with class `.i-button` somewhere inside an element with class `.i-panel`.

Let us give an example with a real case. Consider the following IT Mill Toolkit component.

```
public class LoginBox extends CustomComponent {
    Panel         panel  = new Panel("Log In");

    public LoginBox () {
        setCompositionRoot(panel);

        panel.addComponent(new TextField("Username:"));
        panel.addComponent(new TextField("Password:"));
        panel.addComponent(new Button("Login"));
    }
}
```

The component will look by default as shown in the following figure.

Figure 6.5. Themeing Login Box Example with Default Theme

Now, let us look at the HTML structure of the component. The following listing assumes that the application contains only the above component in the main window of the application.

```
<body>
   <div id="itmtk-ajax-window">
    <div>
       <div class="i-orderedlayout">
        <div>
           <div class="i-panel">
            <div class="i-panel-caption">Log In</div>
            <div class="i-panel-content">
               <div class="i-orderedlayout">
                <div>
                   <div>
                    <div class="i-caption"><span>Username:</span></div>
                   </div>
                   <input type="text" class="i-textfield"/>
                </div>
                <div>
                   <div>
                    <div class="i-caption"><span>Password:</span></div>
                   </div>
                   <input type="password" class="i-textfield"/>
                </div>
                <div><button type="button" class="i-button">Login</button></div>
```

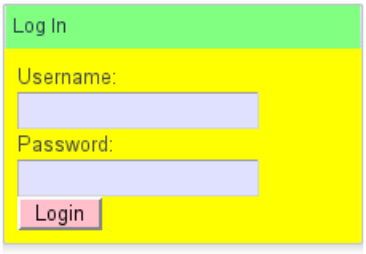
```
                </div>
              </div>
              <div class="i-panel-deco"/>
              </div>
             </div>
          </div>
         </div>
      </div>
 </body>
```

Now, consider the following theme where we set the backgrounds of various elements.

```
.i-panel .i-panel-caption {
 background: #80ff80; /* pale green */
}

.i-panel .i-panel-content {
 background: yellow;
}

.i-panel .i-textfield {
 background: #e0e0ff; /* pale blue */
}

.i-panel .i-button {
 background: pink;
}
```

The coloring has changed as shown in the following figure.

Figure 6.6. Themeing Login Box Example with Custom Theme



An element can have multiple classes separated with a space. With multiple classes, a CSS rule matches an element if any of the classes match. This feature is used in many IT Mill Toolkit components to allow matching based on the state of the component. For example, when the mouse is over a **Link** component, `over` class is added to the component. Most of such styling is a feature of Google Web Toolkit.

6.2.4. Notes on Compatibility

CSS was first proposed in 1994. The specification of CSS is maintained by the CSS Working Group of World Wide Web Consortium (W3C). Its versions are specified as *levels* that build upon the earlier version. CSS Level 1 was published in 1996, Level 2 in 1998. Development of CSS Level 3 was started in 1998 and is still under way.

While the support for CSS has been universal in all graphical web browsers since at least 1995, the support has been very incomplete at times and there still exists an unfortunate number of incompatibilities between browsers. While we have tried to take these incompatibilities into account in the default themes in IT Mill Toolkit, you need to consider them while developing custom themes.

Compatibility issues are detailed in various CSS handbooks.

6.3. Creating and Using Themes

Custom themes are placed in `ITMILL/themes` folder of the web application (in the `WebContent` directory) as illustrated in Figure 6.1, “Theme Contents”. This location is fixed. You need to have a theme folder for each theme you use in your application, although applications rarely need more than a single theme. For example, if you want to define a theme with the name `mytheme`, you will place it in folder `ITMILL/themes/mytheme`.

A custom theme must also inherit the default theme, as shown in the example below:

```
@import "../default/styles.css";

body {
    background: yellow;
}
```

See the section called “Default Theme” and Section 6.3.3, “Theme Inheritance” below for details on inheriting the default theme and theme inheritance generally.

6.3.1. Styling Standard Components

Each user interface component in IT Mill Toolkit has a set of style classes that you can use to control the appearance of the component. Some components have additional elements that also allow styling.

The following table lists the style classes of all client-side components of IT Mill Toolkit. Notice that a single server-side component can have multiple client-side implementations. For example, a **Button** can be rendered on the client side either as a regular button or a check box, depending on the *`switchMode`* attribute of the button. For details regarding the mapping to client-side components, see Section 8.4, “Defining a Widget Set”. Each client-side component type has its own style class and a number of additional classes that depend on the client-side state of the component. For example, a text field will have `i-textfield-focus` class when mouse pointer hovers over the component. This state is purely on the client-side and is not passed to the server.

Some client-side components can be shared by different server-side components. There is also the **IUnknownComponent**, which is a component that indicates an internal error in a situation where the server asked to render a component which is not available on the client-side.

Table 6.1. Default CSS Style Names of IT Mill Toolkit Components

Server-Side Component	Client-Side Widget	CSS Class Name	State Indicators	
Button	IButton	i-button		
	ICheckBox			
CustomComponent	ICustomComponent	i-customcomponent		
CustomLayout	ICustomLayout			
DateField	IDateField	i-datefield		
	ICalendar	i-datefield-entrycalendar		
	IDateFieldCalendar	i-datefield-calendar		
	IPopupCalendar	i-datefield-calendar		
	ITextualDate		i-readonly, i-textfield-error	
Embedded	IEmbedded	-		
Form	IForm	i-form		
FormLayout	IFormLayout	-		
GridLayout	IGridLayout	-		
Label	ILabel	i-label		
Link	ILink	i-link	readonly, enabled	
OptionGroup	IOptionGroup	i-select-optiongroup		
HorizontalLayout	IOrderedLayout	i-orderedlayout		
	VerticalLayout	IOrderedLayout	i-orderedlayout	
		Panel	IPanel	i-panel (i-panel-caption, i-panel-content, i-panel-deco)
Select				
	IListSelect	i-listselect		
IFilterSelect	i-filterselect			
Slider	ISlider	i-slider		
SplitPanel	ISplitPanel	-		
	ISplitPanelHorizontal	-		
	ISplitPanelVertical	-		
Table	IScrollTable	i-table		
	ITablePaging	i-table (i-table-tbody)		
TabSheet	ITabSheet	i-tabsheet (i-tabsheet-content, i-tablsheet-tabs)		

Server-Side Component	Client-Side Widget	CSS Class Name	State Indicators
TextField	ITextField	i-textfield	i-textfield-focus
	ITextArea		
	IPasswordField		
Tree	ITree	i-tree (i-tree-node-selected)	
TwinColSelect	ITwinColSelect	i-select-twincol (i-select-twincol-selections, i-select-twincol-buttons, i-select-twincol-deco)	
Upload	IUpload	-	
Window	IWindow	i-window	
-	CalendarEntry	-	
-	CalendarPanel	i-datefield-calendarpanel	
-	ContextMenu	i-contextmenu	
-	IUnknownComponent	itmtk-unknown (itmtk-unknown-caption)	
-	IView	-	
-	Menubar	gwt-MenuBar	
-	MenuItem	gwt-MenuItem	
-	Time	i-datefield-time (i-select)	

Style names of sub-components are shown in parentheses.

Default Theme

The default theme is provided in the `ITMILL/themes/default/styles.css` stylesheet in the IT Mill Toolkit library JAR. This stylesheet is a compilation of the separate stylesheets for each component in the corresponding subdirectory.

Notice that the default theme included in the IT Mill Toolkit library JAR is served dynamically from the JAR by the servlet. Serving the theme statically by the web server is much more efficient. You only need to extract the `ITMILL/` directory from the JAR under your `WebContent` directory. Just make sure to update it if you upgrade to a newer version of IT Mill Toolkit.

Creation of a default theme of custom GWT widgets is detailed in Section 8.2.3, “Styling GWT Widgets”.

6.3.2. Using Themes

Using a theme is simple, you only need to set the theme with `setTheme()`.

Defining the appearance of a user interface component is fairly simple. First, you create a component and add a custom style name for it with `addStyleName()`. Then you write the CSS element that defines the formatting for the component.

6.3.3. Theme Inheritance

When you define your own theme, you will need to inherit the default theme (unless you just copy the default theme).

Inheritance in CSS is done with the `@import` statement. In the typical case where you define your own theme, you inherit the default theme as follows:

```
@import "../default/styles.css";

body {
    background: yellow;
}
```

You can even create a deep hierarchy of themes by inheritance. Such a solution is often useful if you have some overall theme for your application and a slightly modified theme for different user classes. You can even make it possible for each user to have his or her own theme.

For example, let us assume that we have the base theme of an application with the name **myapp** and a specific **myapp-student** theme for users with the student role. The stylesheet of the base theme would be located in `themes/myapp/styles.css`. We can then "inherit" it in `themes/myapp-student/styles.css` with a simple `@import` statement:

```
@import "../myapp/styles.css";

body {
    background: green;
}
```

This would make the page look just as with the base theme, except for the green background. You could use the theme inheritance as follows:

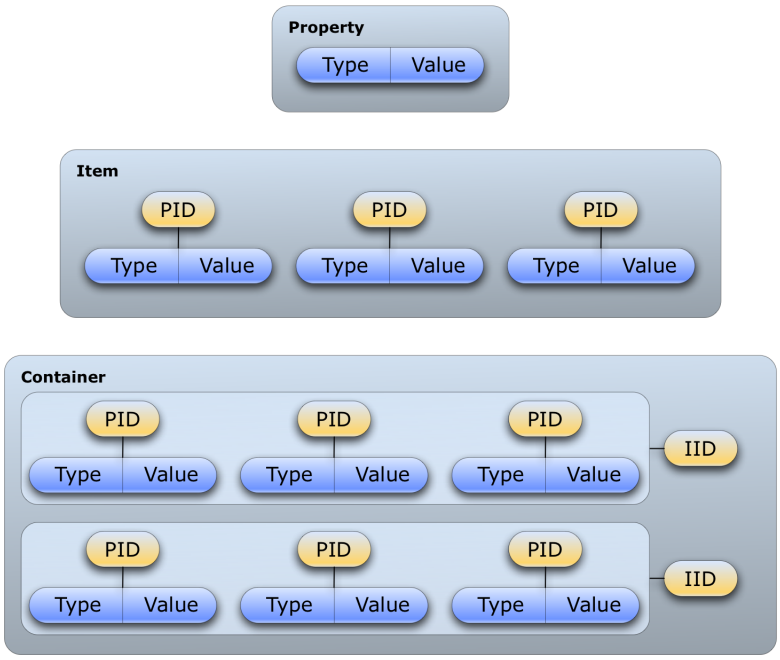
```
public class MyApplication extends com.itmill.toolkit.Application {

    public void init() {
        setTheme("myapp");
        ...
    }

    public void login(User user) {
        if (user.role == User.ROLE_STUDENT)
            setTheme("myapp-student");
        ...
    }

    public void logout() {
        setTheme("myapp");
        ...
    }
}
```

Chapter 7. Data Model

7.1. Overview

IT Mill Toolkit Data Model is one of the core concepts of the library. Let us revisit the model-view-controller design pattern. The model consists of application data and business logic, which acts on the data. The user interface generates events, which are processed by the controller, which controls the user interface and the data. To allow the view (user interface components) to access the application data directly, without need to annoy the controller with every mundane change, we need a standard data interface. Application data needs a common interface so that the data can be accessed by the view and the controller alike. In the Toolkit, we have solved this with the Data Model.

Figure 7.1. IT Mill Toolkit Data Model



At the heart of Data Model is the *property* that consists of a value and its data type. A property is always typed and the type can be any object type in Java. Properties are in themselves unnamed objects. Properties are collected in an *item*, which associates the properties with names, the *Property Identifiers* or *PID*s. Items can be contained in containers and are identified with *Item Identifiers* or *IID*s.

The Data Model is realized as a set of interface classes in the fittingly named package **com.itmill.toolkit.data**. The package contains interfaces **Property**, **Item**, and **Container**, along with a number of more specialized interfaces and classes.

Notice that the Data Model does not define data representation, but only interfaces. This leaves the representation fully to the implementation of the containers. The representation can be almost anything, such as a Java object structure, a filesystem, or a database query.

The Data Model is used heavily in UI components of the Toolkit. A key feature of all UI components is that they can either maintain their data by themselves or be bound to an external data source. For example, many UI components, such as **Button**, **Label**, or **TextField** have a single property which they control.

You can access this property through the **Property** interface inherited by the components. By default, the property is contained within the component, but you can bind the the components to external data sources with the `setPropertyDataSource()` method of the **com.itmill.toolkit.ui.AbstractField** class inherited by such components.

Many UI components are actually both containers and properties. This is especially true for selectable components (that implement **Select**), because they are containers that contain selectable items. Their property is the currently selected item. For more details on components, see Chapter 4, *User Interface Components*.

By implementing a container interface, you can bind UI components directly to data. As containers can be unordered, ordered, indexed, or hierarchical, they can interface practically any kind of data representation. The Toolkit includes data connectors for some common data sources, such as the filesystem.

The Data Model has several important features, such as support for change notification, transactions, validation, and lazy loading. These features are discussed in detail below.

7.2. Properties

IT Mill Toolkit data model is one of the core concepts in the library and Property-interface is the base of that model. Property provides standardized API for a single data object that can be read (get) and written (set). A property is always typed, but can optionally support data type conversions. Optionally properties can provide value change events for following the state changes.

The most important function of the Property as well as other data models is to connect classes implementing the interface directly to editor and viewer classes. Typically this is used to connect different data sources to UI components for editing and viewing their contents.

Properties can be utilized either by implementing the interface or by using some of the existing property implementations. IT Mill Toolkit includes Property interface implementations for arbitrary function pairs or Bean-properties as well as simple object properties.

Many of the UI components also implement Property interface and allow setting of other components as their data-source. These UI-components include TextField, DateField, Select, Table, Button, Label and Tree.

7.3. Holding properties in Items

Item is an object that contains a set of named properties. Each property is identified by a property identifier (PID) and a reference to the property can be queried from the **Item**. **Item** defines inner interfaces for maintaining the item property set and listening changes in the item property set.

Items generally represent objects in the object-oriented model, but with the exception that they are configurable and provide an event mechanism. The simplest way of utilizing **Item** interface is to use existing Item implementations. Provided utility classes include a configurable property set, a bean-to-item adapter and a Form UI component.

7.4. Collecting items in Containers

Container is the most advanced of the data model supported by IT Mill Toolkit. It provides a very flexible way of managing a set of items that share common properties. Each item is identified by an item id. Properties can be requested from container with item and property ids. Another way of accessing properties is to first request an item from container and then request its properties from it.

Container interface was designed with flexibility and efficiency in mind. It contains inner interfaces for ordering the items sequentially, indexing the items and accessing them hierarchically. Those ordering models provide the basis for the **Table**, **Tree**, and **Select** UI components. As with other data models, the containers support events for notifying about the changes.

A set of utilities for converting between container models by adding external indexing or hierarchy into existing containers. In memory containers implementing indexed and hierarchical models provide easy to use tools for setting up in memory data storages. There is even a hierarchical container for direct file system access.

As the items in a **Container** are not indexed, iterating over the items has to be done using an **Iterator**. The `getItemIds()` method of **Container** returns a **Collection** of item identifiers over which you can iterate. The following example demonstrates a typical case where you iterate over the values of check boxes in a column of a **Table** component. The context of the example is the example used in Section 4.10, "**Table**".

```
/* Collect the results of the iteration into this string. */
String items = "";

/* Iterate over the item identifiers of the table. */
for (Iterator i = table.getItemIds().iterator(); i.hasNext();) {
    /* Get the current item identifier, which is an integer. */
    int iid = (Integer) i.next();

    /* Now get the actual item from the table. */
    Item item = table.getItem(iid);

    /* And now we can get to the actual checkbox object. */
    Button button = (Button) (item.getItemProperty("ismember").getValue());

    /* If the checkbox is selected. */
    if ((Boolean)button.getValue() == true) {
        /* Do something with the selected item; collect the first names in a string. */
        items += item.getItemProperty("First Name").getValue() + " ";
    }
}

/* Do something with the results; display the selected items. */
layout.addComponent (new Label("Selected items: " + items));
```

Notice that the `getItemIds()` returns an *unmodifiable collection*, so the **Container** may not be modified during iteration. You can not, for example, remove items from the **Container** during iteration. The modification includes modification in another thread. If the **Container** is modified during iteration, a **ConcurrentModificationException** is thrown and the iterator may be left in an undefined state.

Chapter 8. Developing Custom Components

This chapter gives details on how to create custom client-side components as Google Web Toolkit (GWT) widgets and how to integrate them with IT Mill Toolkit. The client-side implementations of all standard user interface components in IT Mill Toolkit use the same client-side interfaces and patterns.

Google Web Toolkit is intended for developing browser-based user interfaces using the Java language, which is compiled into JavaScript. Knowledge of such client-side technologies is usually not needed with IT Mill Toolkit, as its repertoire of user interface components should be sufficient for most applications. The easiest way to create custom components in IT Mill Toolkit is to make composite components with the **CustomComponent** class. See Section 4.17, “Custom Composite Components” for more details on the composite components. In some cases, however, you may need to either make modifications to existing components or create new or integrate existing GWT widgets with your application.

If you need more background on the architecture, Section 2.3, “Client-Side Engine” gives an introduction to the architecture of the IT Mill Toolkit Client-Side Engine. If you are new to Google Web Toolkit, Section 2.2.2, “Google Web Toolkit” gives an introduction to GWT and its role in the architecture of IT Mill Toolkit.

On Terminology

Google Web Toolkit uses the term *widget* for user interface components. In this book, we use the term widget to refer to client-side components made with Google Web Toolkit, while using the term *component* in a general sense and also in the special sense for server-side components.

8.1. Overview

Google Web Toolkit (GWT) is an integral part of IT Mill Toolkit since Release 5. All rendering of user interface components in a web browser is programmed with GWT. Using custom GWT widgets is easy in IT Mill Toolkit. This chapter gives an introduction to GWT widgets and details on how to integrate them with IT Mill Toolkit.

On the client side, in the web browser, you have the IT Mill Toolkit Client-Side Engine. It uses the GWT framework, and both are compiled into a JavaScript runtime component. The client-side engine is contained in the com.itmill.toolkit.terminal.gwt.client package and the client-side implementations of various user interface components are in the com.itmill.toolkit.terminal.gwt.client.ui package. You can find the source code for these packages in the IT Mill Toolkit installation package. You make custom components by inheriting GWT widget classes. To integrate them with IT Mill Toolkit, you have to implement the **Paintable** interface of the Client-Side Engine that provides the AJAX communications with the server-side application. To enable the custom widgets, you also need to implement a *widget set*. A widget set is a factory class that can instantiate your widgets. It needs to inherit the **DefaultWidgetSet** that acts as the factory for the standard widgets. You can also define stylesheets for custom widgets. A client-side module is defined in a GWT Module Descriptor.

To summarize, to implement a client-side widget that is integrated with IT Mill Toolkit, you need the following:

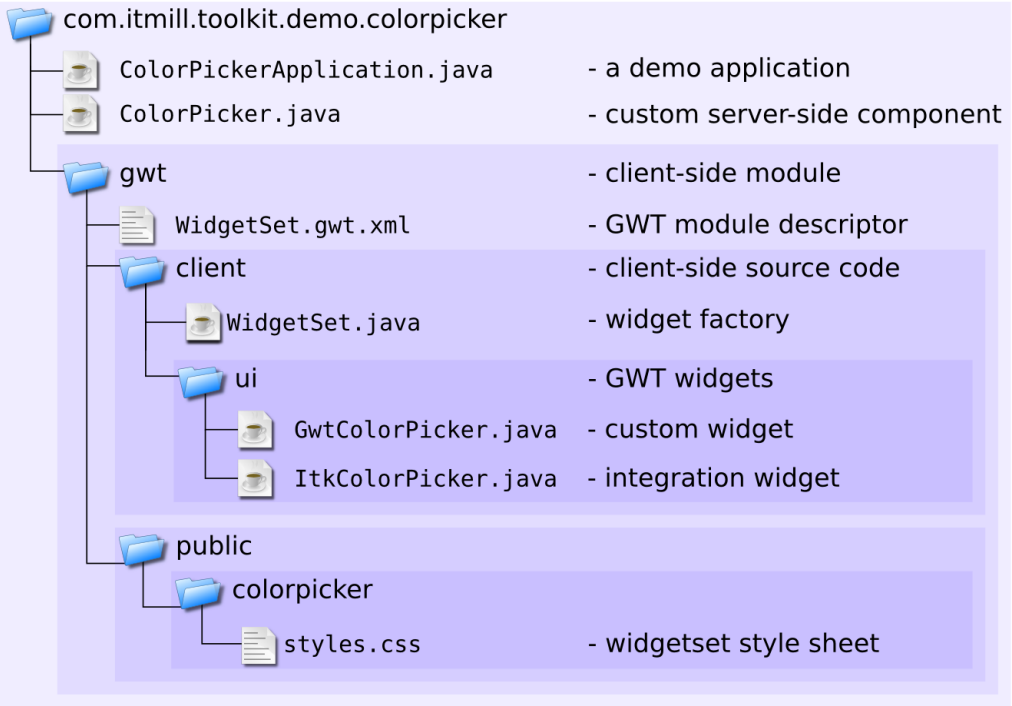
- A GWT widget that implements the **Paintable** interface of the IT Mill Toolkit Client-Side Engine
- A widget factory (a "widget set") that can create the custom widget or widgets

- Default CSS style sheet for the widget set (optional)
- A GWT Module Descriptor (`.gwt.xml`) that describes the entry point and style sheet

On the server side, you need to implement a server-side component that manages serialialization and deserialization of its attributes with the client-side widget. A server-side component usually inherits the **AbstractComponent** or **AbstractField** class and implements either the `paintContent()` or the more generic `paint()` method to serialize its data to the client. These methods "paint" the component by generating a UIDL element that is sent to the client. The UIDL element contains all the relevant information about the component, and you can easily add your own attributes to it.

Figure 8.1, “Color Picker Module” below illustrates the folder hierarchy of the Color Picker example used in this chapter. The example is available in the demo application of IT Mill Toolkit with URL `/colorpicker/`. You can find the full source code of the application in the source module for the demos in the installation package.

Figure 8.1. Color Picker Module



The `ColorPickerApplication.java` application provides an example of using the **ColorPicker** custom component. To allow accessing the application, it must be defined in the deployment descriptor `web.xml`. See Section 3.7.3, “Deployment Descriptor `web.xml`” for details. The source code for the server-side component is located in the same folder.

A client-side widget set must be developed within a single source module tree. This is because GWT Compiler takes as its argument the root folder of the source code, in the Color Picker example the `colorpicker.gwt.client` module, and compiles all the contained Java source files into JavaScript. The path to the source files, the entry point class, and the style sheet are specified in the `WidgetSet.gwt.xml` descriptor for the GWT Compiler. The `WidgetSet.java` provides source code for the entry point, which is a factory class for creating the custom widget objects. The actual custom widget is split into two classes: **GwtColorPicker**, a pure GWT widget, and **IColorPicker** that provides

the integration with IT Mill Toolkit. The default style sheet for the widget set is provided in `gwt/public/colorpicker/styles.css`.

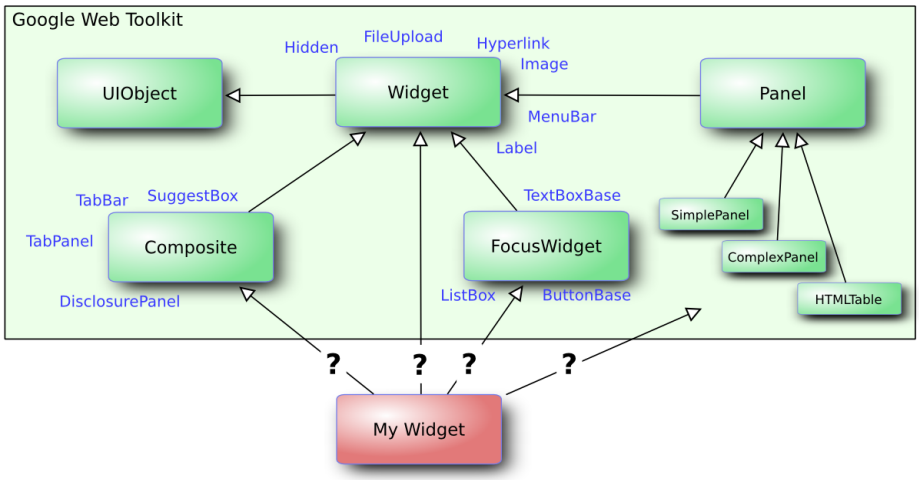
8.2. Google Web Toolkit Widgets

Let us take a look into how custom GWT widgets are created. The authoritative sources for developing with GWT are the *Google Web Toolkit Developer Guide* and *Google Web Toolkit Class Reference*.

Google Web Toolkit offers a variety of ways for creating custom widgets. The easiest way is to create *composite* widgets by grouping existing basic widgets and adding some interaction logic to them. You can also develop widgets using the lower-level Java interfaces used by the standard GWT widgets or the really low-level JavaScript interfaces.

A custom GWT widget needs to find its place in the GWT class hierarchy. Figure 8.2, “GWT Widget Base Class Hierarchy” below illustrates the abstract base classes for GWT widgets.

Figure 8.2. GWT Widget Base Class Hierarchy



Each of the base classes offers various services for different types of widgets. Many custom widgets, such as the Color Picker example below, extend the **Composite** class to combine the widget from existing GWT widgets. Other base classes offer various features useful for different kinds of widgets. You can also opt to extend an existing GWT widget, as we have done for most of the standard user interface components of IT Mill Toolkit, or extend an IT Mill Toolkit widget.

8.2.1. Extending an IT Mill Toolkit Widget

Extending an IT Mill Toolkit widget is an easy way to add features, such as advanced client-side validation, to existing standard components. If the extended widget does not require any additional parameters, which is usual in client-side validation, you may not even need to define a server-side counterpart for your widget. A server-side component can be mapped to multiple client-side components depending on its parameters. The mapping is defined in the widget factory, i.e., the class inheriting **DefaultWidgetSet**. For details on how to implement a widget factory, see Section 8.4, “Defining a Widget Set”.

8.2.2. Example: A Color Picker GWT Widget

In the following example, we implement a composite GWT widget built from **HorizontalPanel**, **Grid**, **Button**, and **Label** widgets. This widget does not yet have any integration with the server side code, which

will be shown later in this chapter. The source code is available in the source folder for the demo application in IT Mill Toolkit installation folder, under package com.itmill.toolkit.demo.colorpicker.

```
package com.itmill.toolkit.demo.colorpicker.gwt.client.ui;

import com.google.gwt.user.client.DOM;
import com.google.gwt.user.client.Element;
import com.google.gwt.user.client.ui.*;

/**
 * A regular GWT component without integration with IT Mill Toolkit.
 **/
public class GwtColorPicker extends Composite implements ClickListener {

    /** Currently selected color name to give client-side feedback to the user. */
    protected Label currentcolor = new Label();

    public GwtColorPicker() {
        // Create a 4x4 grid of buttons with names for 16 colors
        Grid grid = new Grid(4,4);
        String[] colors = new String[] {"aqua", "black", "blue", "fuchsia",
                "gray", "green", "lime", "maroon", "navy", "olive",
                "purple", "red", "silver", "teal", "white", "yellow"};
        int colornum = 0;
        for (int i=0; i<4; i++)
            for (int j=0; j<4; j++, colornum++) {
                // Create a button for each color
                Button button = new Button(colors[colornum]);
                button.addClickListener(this);

                // Put the button in the Grid layout
                grid.setWidget(i, j, button);

                // Set the button background colors.
              DOM.setStyleAttribute(button.getElement(), "background", colors[colornum]);


                // For dark colors, the button label must be in white.
                if ("black navy maroon blue purple".indexOf(colors[colornum]) != -1)
                    DOM.setStyleAttribute(button.getElement(), "color", "white");
            }

        // Create a panel with the color grid and currently selected color indicator
        HorizontalPanel panel = new HorizontalPanel();
        panel.add(grid);
        panel.add(currentcolor);

        // Set the class of the color selection feedback box to allow CSS styling.
        // We need to obtain the DOM element for the current color label.
        // This assumes that the <td> element of the HorizontalPanel is
        // the parent of the label element. Notice that the element has no parent
        // before the widget has been added to the horizontal panel.
        Element panelcell = DOM.getParent(currentcolor.getElement());
        DOM.setElementProperty(panelcell, "className", "colorpicker-currentcolorbox");

        // Set initial color. This will be overridden with the value read from server.
        setColor("white");

        // Composite GWT widgets must call initWidget().
        initWidget(panel);
    }

    /** Handles click on a color button. */
    public void onClick(Widget sender) {
        // Use the button label as the color name to set
        setColor(((Button) sender).getText());
    }
```

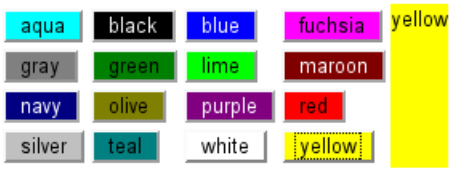
```
    /** Sets the currently selected color. */
    public void setColor(String newcolor) {
        // Give client-side feedback by changing the color name in the label
        currentcolor.setText(newcolor);

        // Obtain the DOM elements. This assumes that the <td> element
        // of the HorizontalPanel is the parent of the label element.
        Element nameelement = currentcolor.getElement();
        Element cell = DOM.getParent(nameelement);

        // Give feedback by changing the background color
        DOM.setStyleAttribute(cell,        "background", newcolor);
        DOM.setStyleAttribute(nameelement, "background", newcolor);
        if ("black navy maroon blue purple".indexOf(newcolor) != -1)
            DOM.setStyleAttribute(nameelement, "color", "white");
        else
            DOM.setStyleAttribute(nameelement, "color", "black");
    }
}
```

This example demonstrates one reason for making a custom widget: it provides client-side feedback to the user in a way that would not be possible or at least practical from server-side code. Server-side code can only select a static CSS style or a theme, while on the client-side we can manipulate styles of HTML elements very flexibly. Notice that manipulation of the DOM tree depends somewhat on the browser. In this example, the manipulation should be rather compatible, but in some cases there could be problems. Standard GWT and IT Mill Toolkit widgets handle many of such compatibility issues, but when doing such low-level operations as DOM manipulation, you may need to consider them.

The structure of the DOM tree depends on how GWT renders its widgets for a specific browser. It is also not guaranteed that the rendering does not change in future releases of GWT. You should therefore make as few assumptions regarding the DOM structure as possible. Unfortunately, GWT does not provide a way to set the style of, for example, cells of layout elements. The above example therefore assumes that the **Grid** is a table and the `<button>` elements are inside `<td>` elements of the table. See Section 8.2.3, “Styling GWT Widgets” below for more details on compatibility.

The widget will look as follows:

Figure 8.3. Color Picker Widget Without Styling



As you can notice, the widget will look rather uninviting without CSS styling. We will next look how to define a default style for a GWT widget.

8.2.3. Styling GWT Widgets

GWT renders its widgets in the DOM tree of the web browser as HTML elements. Therefore, their style can be defined with Cascading Style Sheets (CSS) just as in HTML. GWT Compiler supports packaging style sheets from the source package tree. The style sheet is defined in the `.gwt.xml` descriptor file (see Section 8.4.1, “GWT Module Descriptor” for details).

```
<!-- WidgetSet default theme -->
<stylesheet src="colorpicker/styles.css"/>
```

The style sheet path is relative to the `public` folder under the folder containing the `.gwt.xml` file.

Let us define the `colorpicker/styles.css` as follows.

```
/* Set style for the color picker table.
   This assumes that the Grid layout is rendered as a HTML <table>.*/
table.example-colorpicker {
    border-collapse: collapse;
    border: 0px;
}

/* Set color picker button style.
   This does not make assumptions about the HTML element tree as it only uses
   the class attributes of the elements.*/
.example-colorpicker .gwt-Button {
    height: 60px;
    width: 60px;
    border: none;
    padding: 0px;
}

/* Set style for the right-hand box that shows the currently selected color.
   While this may work for other implementations of the HorizontalPanel as well,
   it somewhat assumes that the layout is rendered as a table where cells
   are <td> elements. */
.colorpicker-currentcolorbox {
    width: 240px;
    text-align: center;
    vertical-align: middle !important; /* Must be !important to override GWT styling. */
}
```

The stylesheet makes some assumptions regarding the HTML element structure. First, it assumes that the **Grid** layout is a table. Second, the custom class name, `colorpicker-currentcolorbox`, of the right-hand **HorizontalPanel** cell was inserted in the DOM representation of the widget in the GwtColorPicker implementation. Styling a button makes less assumptions. Using only class names instead of specific element names may make a stylesheet more compatible if the HTML representation is different in different browsers or changes in the future.

Figure 8.4. Color Picker Widget With Styling

8.3. Integrating a GWT Widget

Integration of GWT widgets with IT Mill Toolkit can be done in two basic ways: by modifying the original class or by inheriting it and adding the integration code in the subclass. The latter way is actually the way

the standard client-side components in IT Mill Toolkit are done: they simply inherit the corresponding standard GWT widgets. For example, **IButton** inherits GWT **Button**.

The integration code has the following tasks:

- Manage CSS style class
- Receive component state from server
- Send state changes caused by user interaction to server

The integration is broken down in the following sections into server-client deserialization done in `updateFromUIDL()` and client-server serialization done with `updateVariable()`. The complete example of the integration of the Color Picker widget is given at the end of this section.

Naming Conventions

While the use of IT Mill Toolkit does not require the use of any particular naming conventions for GWT widgets, some notes regarding naming may be necessary. While Java name spaces make it possible to use identical class names in the same context, it may be useful to try to make them more distinctive to avoid any inconvenience. GWT uses simple names for its standard widgets, such as **Button**. The standard components of IT Mill Toolkit use identical or similar names, but that does not cause any inconvenience, because the GWT widgets and server-side components of IT Mill Toolkit are never used in the same context. For the client-side components of IT Mill Toolkit, we use the "`I`" prefix, for example **IButton**. In the Color Picker example, we use **GwtColorPicker** for the GWT widget and **IColorPicker** for the integration implementation. You may wish to follow similar conventions.

8.3.1. Deserialization of Component State from Server

To receive data from the server, a widget must implement the **Paintable** interface and its `updateFromUIDL()` method. The idea is that the method "paints" the user interface description by manipulating the HTML tree on the browser. Typically, when using composite GWT components, most of the DOM tree manipulation is done by standard GWT widgets.

An implementation of the `updateFromUIDL()` method must include some routine tasks:

- Call `updateComponent()` and return if it succeeds
- Manage the component identifier
- Manage a reference to the **ApplicationConnection** object. The widget needs to know it to be able to initiate a server request when a browser event occurs.

The latter two of these tasks are not needed if the widget does not handle any user input that needs to be sent to server.

The following excerpt provides a skeleton for the `updateFromUIDL()` method and shows how the component identifier and connection object reference are managed by a widget.

```
String uidlId;
ApplicationConnection client;

...

public void updateFromUIDL(UIDL uidl, ApplicationConnection client) {
    if (client.updateComponent(this, uidl, true))
```

```
            return;

        this.client = client;
        uidlId = uidl.getId();

        ...
    }
```

The `updateComponent()` call has several functions important for different kinds of components. It updates various default attributes, such as *`disabled`*, *`readonly`*, *`invisible`*, and (CSS) *`style`* attributes. If the *`manageCaption`* argument is `true`, the call will also update the caption of the component. By default, the caption is managed by the parent layout of the component. Components, such as a **Button**, that manage the caption themselves, do not need management of the caption.

The `updateComponent()` is also part of the transmutation mechanism that allows a single server-side component to have alternative client-side implementations, based on its parameters. For example, the **Button** server-side component can manifest either as a clickable **IButton** or as a switchable **ICheckBox** widget on the client-side. If the parameters are changed, the client-side widget can be replaced with another dynamically. Determination of the correct implementation is done in a **WidgetSet**. If `updateComponent()` returns true, the client-side engine can attempt to replace the implementation. For more details on the transmutation mechanism, see Section 8.4, “Defining a Widget Set”.

The component identifier is used when the component needs to serialize its updated state to server. The reference to the application connection manager is needed to make the server request. If a component does not have any state changes that need to be sent to the server, management of the variables is not needed. See Section 8.3.2, “Serialization of Component State to Server” below for further details.

The design of the client-side framework of IT Mill Toolkit, because the **Paintable** is an interface and can not store any references. Having an API layer between GWT and custom widgets would be a much more complicated solution.

8.3.2. Serialization of Component State to Server

User input is handled in GWT widgets with events.

User input is passed to the server using the `updateVariable()` method. If the *`immediate`* parameter is *`false`*, the value is simply added to a queue to be sent to the server at next AJAX request. If the argument is *`true`*, the AJAX request is made immediately, and will include all queued updates to variables.

```
if (uidl_id == null || client == null)
        return;

client.updateVariable(uidl_id, "myvariable", newvalue, immediate);
```

The *`client`* of the above example is a reference to the **ApplicationConnection** object that manages server requests. The *`uidl_id`* argument is the UIDL identifier obtained during a `updateFromUIDL()` call with `uidl.getId()` method.

The `updateVariable()` method has several varieties to send variables of different types.

Table 8.1. UIDL Variable Types

Type	Description	UIDL Type
`String`	String object.	s
`int`	Native integer value.	i
`long`	Native long integer value.	l
`float`	Native single-precision floating-point value.	f
`double`	Native double-precision floating-point value.	d
`boolean`	Native boolean value.	b
`Object[]`	Array of object data. The `toString()` method is used to serialize each of the objects. The content strings are escaped with `escapeString()`, to allow characters such as quotes.	a

The *`immediate`* argument is described below.

This serialization mechanism is intended to be as simple as possible in most cases, when the user input is typically just one state variable, while also allowing the serialization of more complex data, if necessary.

Immediateness

Server-side components that inherit **AbstractComponent** have an *`immediate`* attribute, set with `setImmediate()`. This attribute dictates whether a component makes a server request immediately when its state changes, or only afterwards. For example, there is no need to send the contents of a "Username" **TextField** before the "Login" button has been clicked. On the other hand, the server can set the **TextField** as immediate to receive changes for example when the component loses focus.

Most widgets should support immediateness by receiving the *`immediate`* attribute from the UIDL message that renders the widget. The following example is extracted from the **ITextField** implementation.

```
// Store the immediate attribute in a member variable
private boolean immediate = false;
...

public void updateFromUIDL(UIDL uidl, ApplicationConnection client) {
    if(client.updateComponent(this, uidl, true))
        return;

    // Receive and store the immediate attribute
    immediate  = uidl.getBooleanAttribute("immediate");
    ...
}

public void onChange(Widget sender) {
    if(client != null && id != null) {
        // Use the stored immediate attribute to say whether or not
        // make the server request immediately.
        client.updateVariable(id, "text", getText() , immediate);
    }
}
```

In some widgets, the *`immediate`* attribute would have little meaning, and in fact an accidental *`false`* value would cause undesired behaviour. For example, a button is always expected to send a request to the server when it is clicked. Such widgets can simply use *`true`* for the *`immediate`* argument in `updateVariable()`. For example, **IButton** does as follows:

```
public void onClick(Widget sender) {
    if (id == null || client == null)
```

```
            return;
        client.updateVariable(id, "state", true, /* always immediate */ true);
    }
```

8.3.3. Example: Integrating the Color Picker Widget

Below is a complete example of an integration component for the Color Picker example. It demonstrates all the basic tasks needed for the integration of a GWT widget with its server-side counterpart component.

```
import com.itmill.toolkit.terminal.gwt.client.ApplicationConnection;
import com.itmill.toolkit.terminal.gwt.client.Paintable;
import com.itmill.toolkit.terminal.gwt.client.UIDL;

public class IColorPicker extends GwtColorPicker implements Paintable {

    /** Set the CSS class name to allow styling. */
    public static final String CLASSNAME = "example-colorpicker";

    /** Component identifier in UIDL communications. */
    String uidlId;

    /** Reference to the server connection object. */
    ApplicationConnection client;

    /**
     * The constructor should first call super() to initialize the component and
     * then handle any initialization relevant to IT Mill Toolkit.
     */
    public IColorPicker() {
        // The superclass has a lot of relevant initialization
        super();

        // This method call of the Paintable interface sets the component
        // style name in DOM tree
        setStyleName(CLASSNAME);
    }

    /**
     * This method must be implemented to update the client-side component from
     * UIDL data received from server.
     *
     * This method is called when the page is loaded for the first time, and
     * every time UI changes in the component are received from the server.
     */
    public void updateFromUIDL(UIDL uidl, ApplicationConnection client) {
        // This call should be made first. Ensure correct implementation,
        // and let the containing layout manage caption, etc.
        if (client.updateComponent(this, uidl, true))
            return;

        // Save reference to server connection object to be able to send
        // user interaction later
        this.client = client;

        // Save the UIDL identifier for the component
        uidlId = uidl.getId();

        // Get value received from server and actualize it in the GWT component
        setColor(uidl.getStringVariable("colorname"));
    }

    /** Override the method to communicate the new value to server. */
    public void setColor(String newcolor) {
        // Ignore if no change
        if (newcolor.equals(currentcolor.getText()))
            return;
```

```
        // Let the original implementation to do whatever it needs to do
        super.setColor(newcolor);

        // Updating the state to the server can not be done before
        // the server connection is known, i.e., before updateFromUIDL()
        // has been called.
        if (uidlId == null || client == null)
            return;

        // Communicate the user interaction parameters to server. This call will
        // initiate an AJAX request to the server.
        client.updateVariable(uidlId, "colorname", newcolor, true);
    }
}
```

8.4. Defining a Widget Set

The client-side components, or in GWT terminology, widgets, must be made usable in the client-side GWT application by defining a widget set factory that can create the widgets by their UIDL tag name. (Actually, such a widget set factory *is* the client-side application.)

A widget set factory needs to inherit the default factory **DefaultWidgetSet** and implement the `createWidget()` and `resolveWidgetType()` methods. The methods must call their default implementation to allow creation of the standard widgets.

The following example shows how to define a widget set factory class for the Color Picker example. The tag name of the widget was defined in the server-side implementation of the widget (see Section 8.3.3, “Example: Integrating the Color Picker Widget”) as `colorpicker`. The `resolveWidgetType()` must resolve this name to the class object of the **IColorPicker** integration class, which is later passed to the `createWidget()` method for creating an instance of the **IColorPicker** class.

```
import com.itmill.toolkit.demo.colorpicker.gwt.client.ui.IColorPicker;
import com.itmill.toolkit.terminal.gwt.client.DefaultWidgetSet;
import com.itmill.toolkit.terminal.gwt.client.Paintable;
import com.itmill.toolkit.terminal.gwt.client.UIDL;

public class ColorPickerWidgetSet extends DefaultWidgetSet {
    /** Resolves UIDL tag name to widget class. */
    protected Class resolveWidgetType(UIDL uidl) {
        final String tag = uidl.getTag();
        if ("colorpicker".equals(tag))
            return IColorPicker.class;

        // Let the DefaultWidgetSet handle resolution of default widgets
        return super.resolveWidgetType(uidl);
    }

    /** Creates a widget instance according to its class object. */
    public Paintable createWidget(UIDL uidl) {
        final Class type = resolveWidgetType(uidl);
        if (IColorPicker.class == type)
            return new IColorPicker();

        // Let the DefaultWidgetSet handle creation of default widgets
        return super.createWidget(uidl);
    }
}
```

The default widgets in IT Mill Toolkit actually use more than just the tag name to resolve the actual widget class. For example, the **Button** server-side component, which has tag name `button`, can be resolved to

either an **IButton** or **ICheckBox** widget, depending on the *`switch`* (*`switchMode`*) attribute. IT Mill Toolkit Client-Side Engine can actually replace the client-side object of the parameters change.

8.4.1. GWT Module Descriptor

A widget set is actually a GWT application and needs to be defined in the GWT module descriptor as the entry point of the application. A GWT module descriptor is an XML file with extension `.gwt.xml`.

The following example shows the GWT module descriptor of the Color Picker application. The client-side entry point will be the **WidgetSet** class. We also define the default stylesheet for the color picker widget, as described above in Section 8.2.3, “Styling GWT Widgets”.

```
<module>
    <!-- Inherit the NoEntry version to avoid multiple entrypoints -->
    <inherits name="com.itmill.toolkit.terminal.gwt.DefaultWidgetSetNoEntry" />

    <!-- WidgetSet default theme -->
    <stylesheet src="colorpicker/styles.css"/>

    <!-- Entry point -->
    <entry-point class="com.itmill.toolkit.demo.colorpicker.gwt.client.WidgetSet"/>

</module>
```

For more information about the GWT Module XML Format, please see Google Web Toolkit Developer Guide.

8.5. Server-Side Components

Server-side components provide the API for user applications to build their user interface. Many applications do not ever need to bother with the client-side implementation of the standard components, but those that use their own GWT widgets need to have corresponding server-side components.

A server-side component has two basic tasks: it has to be able to serialize its state variables to the corresponding client-side component, and deserialize any user input received from the client. Many of these tasks are taken care of by the component framework.

Component Tag Name

Server-side components are identified with a unique UIDL tag name, which must be returned by the `getTag()` method. The tag should follow XML rules for element names, that is, only characters a-z, A-Z, 0-9, and _, and not begin with a number. Actually, as IT Mill Toolkit Release 5 uses a JSON notation for serialization, the tag syntax is more relaxed, but we nevertheless recommend using a stricter syntax. UIDL is detailed in Chapter 10, *User Interface Definition Language (UIDL)* together with lists of reserved tags. The server-side implementation of the Color Picker component defines the tag as follows:

```
public String getTag() {
    return "colorpicker";
}
```

On the client side, this tag is mapped to a GWT widget. The mapping from server-side to client-side components is actually one to many; a server-side component can manifest as several client-side components, depending on its parameters. For example, a server-side **Button** can manifest either as an **IButton** or **ICheckBox** in client, depending on the *switchMode* attribute. For the client side, see Section 8.2, “Google Web Toolkit Widgets” above.

The implementation of the server-side component is broken down into server-client serialization and client-server deserialization in the following sections. We will also present the complete example of the server-side implementation of the Color Picker component below.

8.5.1. Server-Client Serialization

The server-side implementation of a component must be able to serialize its data into a UIDL message that is sent to the client. You need to override the `paintContent()` method, defined in **AbstractComponent**. You should call the superclass to allow it to paint its data as well.

The data is serialized with the variants of the `addAttribute()` and `addVariable()` methods for different basic data types.

The UIDL API offered in **PaintTarget** is covered in Section 10.1, “API for Painting Components”.

8.5.2. Client-Server Deserialization

The server-side component must be able to receive state changes from the client-side widget. This is done by overriding the `changeVariables()` method, defined in **AbstractComponent**. A component should always call the superclass implementation in the beginning to allow it handle its variables.

The variables are given as objects in the *`variables`* map, with the same key with which they were serialized on the client-side. The object type is likewise the same as given for the particular variable in `updateVariable()` in the client-side.

```
@Override
public void changeVariables(Object source, Map variables) {
    // Let superclass read any common variables.
    super.changeVariables(source, variables);

    // Sets the currently selected color
    if (variables.containsKey("colorname") && !isReadOnly()) {
        final String newValue = (String) variables.get("colorname");
        // Changing the property of the component will
        // trigger a ValueChangeEvent
        setValue(newValue, true);
    }
}
```

The above example handles variable changes for a field component inheriting **AbstractField**. Fields have their value as the value property of the object. Setting the value with `setValue()`, as above, will trigger a **ValueChangeEvent**, which the user of the component can catch with a **ValueChangeListener**.

Contained components, such as components inside a layout, are deserialized by referencing them by their *paintable identifier* or *PID*.

8.5.3. Extending Standard Components

Extending standard components is one way to develop new components that have some additional features.

Every component needs to have a unique UIDL tag that is used to create and communicate with widgets on the client-side. The tag is normally unique for server-side components. The minimal requirement for the server-side component is that you reimplement the `getId()` method that provides the tag.

If your extension component contains any specific state variables, you need to handle their serialization in **paintContent()** and deserialization in **changeVariables()** and call the superclass implementation in the

beginning. See Section 8.5.1, “Server-Client Serialization” Section 8.5.2, “Client-Server Deserialization” above for details.

The client-side implementation goes also much like for regular custom widgets.

8.5.4. Example: Color Picker Server-Side Component

The following example provides the complete server-side **ColorPicker** component for the Color Picker example. It has only one state variable: the currently selected color, which is stored as the property of the component. Implementation of the **Property** interface is provided in the **AbstractField** superclass of the component. The UIDL tag name for the component is `colorpicker` and the state is communicated through the *`colorname`* variable.

```
package com.itmill.toolkit.demo.colorpicker;

import java.util.Map;
import com.itmill.toolkit.terminal.PaintException;
import com.itmill.toolkit.terminal.PaintTarget;
import com.itmill.toolkit.ui.*;

public class ColorPicker extends AbstractField {

    public ColorPicker() {
        super();
        setValue(new String("white"));
    }

    /** The property value of the field is an Integer. */
    public Class getType() {
        return String.class;
    }

    /** Tag is the UIDL element name for client-server communications. */
    public String getTag() {
        return "colorpicker";
    }

    /** Set the currently selected color. */
    public void setColor(String newcolor) {
        // Sets the color name as the property of the component.
        // Setting the property will automatically cause repainting of
        // the component with paintContent().
        setValue(newcolor);
    }

    /** Retrieve the currently selected color. */
    public String getColor() {
        return (String) getValue();
    }

    /** Paint (serialize) the component for the client. */
    public void paintContent(PaintTarget target) throws PaintException {
        // Superclass writes any common attributes in the paint target.
        super.paintContent(target);

        // Add the currently selected color as a variable in the paint target.
        target.addVariable(this, "colorname", getColor());
    }

    /** Deserialize changes received from client. */
    public void changeVariables(Object source, Map variables) {
        // Sets the currently selected color
        if (variables.containsKey("colorname") && !isReadOnly()) {
            String newValue = (String) variables.get("colorname");
```

```
            // Changing the property of the component will
            // trigger a ValueChangeEvent
            setValue(newValue,true);
        }
    }
}
```

8.6. Using a Custom Component

A custom component is used like any other IT Mill Toolkit component. You will, however, need to compile the client-side widget set with the GWT Compiler.

8.6.1. Example: Color Picker Application

The following server-side example application shows how to use the Color Picker custom widget. The example includes also server-side feedback of the user input and changing the color selection to show that the communication of the component state works in both directions.

```
package com.itmill.toolkit.demo.colorpicker;

import com.itmill.toolkit.data.Property.ValueChangeEvent;
import com.itmill.toolkit.data.Property.ValueChangeListener;
import com.itmill.toolkit.ui.*;
import com.itmill.toolkit.ui.Button.ClickEvent;

/**
 * Demonstration application that shows how to use a simple
 * custom client-side GWT component, the ColorPicker.
 */
public class ColorPickerApplication extends com.itmill.toolkit.Application {
    Window main = new Window("Color Picker Demo");

    /* The custom component. */
    ColorPicker colorselector = new ColorPicker();

    /* Another component. */
    Label colorname;

    public void init() {
        setMainWindow(main);
        setTheme("demo");

        // Listen for value change events in the custom component,
        // triggered when user clicks a button to select another color.
        colorselector.addListener(new ValueChangeListener() {
            public void valueChange(ValueChangeEvent event) {
                // Provide some server-side feedback
                colorname.setValue("Selected color: " + colorselector.getColor());
            }
        });
        main.addComponent(colorselector);

        // Add another component to give feedback from server-side code
        colorname = new Label("Selected color: "+colorselector.getColor());
        main.addComponent(colorname);

        // Server-side manipulation of the component state
        Button button = new Button("Set to white");
        button.addListener(new Button.ClickListener() {
            public void buttonClick(ClickEvent event) {
                colorselector.setColor("white");
            }
        });
```

```
            main.addComponent(button);
        }
    }
```

8.6.2. Web Application Deployment

Deployment of web applications that include custom components is almost identical to the normal case where you use only the default widget set of IT Mill Toolkit. The default case is documented in Section 3.7.3, "Deployment Descriptor `web.xml`". You only need to specify the widget set for the application in the `WebContent/WEB-INF/web.xml` deployment descriptor.

The following deployment descriptor specifies the Color Picker Application detailed in the previous section.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_ID" version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
    <display-name>myproject</display-name>

    <servlet>
        <servlet-name>ColorPickerServlet</servlet-name>

<servlet-class>com.itmill.toolkit.terminal.gwt.server.ApplicationServlet</servlet-class>

        <init-param>
            <param-name>application</param-name>

<param-value>com.itmill.toolkit.demo.colorpicker.ColorPickerApplication</param-value>
        </init-param>
        <init-param>
            <param-name>widgetset</param-name>

<param-value>com.itmill.toolkit.demo.colorpicker.gwt.ColorPickerWidgetSet</param-value>
        </init-param>
    </servlet>

    <servlet-mapping>
        <servlet-name>ColorPickerServlet</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

Project specific parameter are emphasized. Notice that the widget set name is not a file name, but the base name for the `ColorPickerWidgetSet.gwt.xml` module descriptor.

As the project context root in the above example is `myproject` and the *`<url-pattern>`* is `/*`, the URL for the application will be `/myproject/`.

8.7. GWT Widget Development

Development of new GWT widgets includes management of the source code tree, running and debugging the application with the GWT Hosted Mode Browser, and compiling the widgets and the IT Mill Toolkit Client-Side Engine to JavaScript with the GWT Compiler.

You can use any IDE for developing GWT components for IT Mill Toolkit. The examples given in this book are for the Eclipse IDE. It allows easy launching of the GWT Hosted Mode Browser, debugging, and running an external compiler for GWT widget sets.

8.7.1. Creating a Widget Project in Eclipse

Creation of an IT Mill Toolkit project that uses the default widget set was covered in Section 1.6, “Your First Project with IT Mill Toolkit”. Developing custom widgets creates a number of additional requirements for a project. Let us review the steps required for creating a project. Details for each step are given in the subsequent sections.

1. Create a new project in the Eclipse IDE. (Section 1.6.1)
2. Import IT Mill Toolkit library JAR into the project. (Section 1.6.2)
3. Import GWT directory into the project. (Section 8.7.2 below)
4. Write the source code in a Java module. (Section 8.7.3 below)
5. Write the `web.xml` Deployment Descriptor for the web application.
   - Define the custom widget set to use instead of the default widget set. (Section 8.6.2 above)
6. Compile the widget set to JavaScript runtime with GWT Compiler. (Section 8.7.4 below)
7. Configure the project in Apache Tomcat (or some other web container) and start the server. (Section 1.6.6 below)
8. Either:
   a. Open a web browser to use the web application.
   b. Debug the widgets with Hosted Mode Browser. (Section 8.7.6)

The contents of a ready widget development project are described in Section 8.7.5, “Ready to Run”.

8.7.2. Importing GWT Installation Package

You will need to include the Google Web Toolkit in your project to develop custom widgets. The installation directory of IT Mill Toolkit includes full GWT installation in the `gwt` subdirectory. The package includes precompiled libraries and applications for the specific platform of the installation. To use the libraries, you need to configure them in the classpath of your project as described below.

You need to copy the `gwt` directory to your project. You can either copy it with system tools or, if you are using Eclipse, import the directory. Importing the directory is done as follows.

1. Right-click on the project folder in **Project Explorer** and select **Import** → **Import...**.
2. From the Import dialog, select **General** → **File System** and click **Next**.
3. Click **Browse** button of the "**From directory**" field and browse to the `gwt` directory under the IT Mill Toolkit installation directory. Click **Ok** in the file selection dialog.
4. Select the `gwt` entry in the list box for importing.
5. In the "**Into folder**" field, enter `myproject/gwt`. (If you do not set this, all the contents of the `gwt` directory will be imported directly below the root directory of the project which is undesirable.)
6. Click **Finish**.

If you copied the directory with system tools, remember to select your project and press F5 to refresh the project.

GWT libraries must be included in the classpath of the project. Right-click on the project folder in the **Project Explorer** in Eclipse and select **Properties**. Select **Java Build Path** → **Libraries**.

8.7.3. Creating a GWT Module

This section gives details on writing an application module that includes custom widgets.

Creating the Source Folder

While the source files can be placed in any directory in ordinary projects, usually in the `src` directory directly under the project root, the widget build script described below in Section 8.7.4, "Compiling GWT Widget Sets" as well as the GWT Hosted Mode Browser assume that source files are located under the `WebContent/WEB-INF/src` folder. The source folder has to be created and designated as a source folder for the project.

1. Right-click on the `WebContent/WEB-INF` folder and select **New** → **Folder**.
2. In the **New Folder** dialog, enter `src` as the **Folder name** and click **Finish**.
3. Right-click on the `src` folder and select **Build Path** → **Use as Source Folder**.

The folders designated as source folders are moved under the **Java Resources** folder in the **Project Explorer** of Eclipse. This is only a display feature; the source directory remains in its original location in the filesystem.

Creating Source Files

In Eclipse, you can insert a folder inside a source package in **File** → **New** → **Folder**.

Importing the ColorPicker Demo

If you want to use the Color Picker application as an application skeleton, you need to import it under the source folder.

1. Right-click on the source folder and select **Import**.
2. In the **Import** dialog, select **General** → **File System** and click **Next**.
3. Browse to `WebContent/WEB-INF/src/com/itmill/toolkit/demo/colorpicker/` and click **Ok** button in the **Import from directory** dialog.
4. In the **Into folder** field, enter `myproject/WebContent/WEB-INF/src/com/itmill/toolkit/demo/colorpicker`.
5. Check the `colorpicker` entry in the list box.
6. Click **Finish**.

This will import the directory as com.itmill.toolkit.demo.colorpicker package. If you want to use it as a skeleton for your own project, you should refactor it to some other name. Notice that you will need to refactor the package and application name manually in the `web.xml` and `.gwt.xml` descriptor files.

8.7.4. Compiling GWT Widget Sets

When running an application in a regular web browser, you need to compile the IT Mill Toolkit Client-Side Engine and your custom widget set to JavaScript. This is done with the GWT Compiler. IT Mill Toolkit installation package includes an Ant build script `build-widgetset.xml` in the `WebContent/doc/example-source/` directory. To compile the Color Picker widget set example, just change to the directory and enter:

```
$ ant -f build-widgetset.xml
```

We advice that you copy the build script to your project and use it as a template. Just set the paths in the "configure" target and the widget set class name in the "compile-widgetset" target to suit your project.

Alternatively, you can launch the build script from Eclipse, by right-clicking the script in Package Explorer and selecting **Run As** → **Ant Build**. Progress of the compilation is shown in the **Console** window.

After compilation, *refresh the project by selecting it and pressing **F5***. This makes Eclipse scan new content and become aware of the output of the compilation in the `WebContent/ITMILL/widgetsets/` directory. If the project is not refreshed, the JavaScript runtime is not included in the web application and running the application will result in an error message such as the following:

```
Requested resource
[ITMILL/widgetsets/com.itmill.toolkit.demo.colorpicker.gwt.ColorPickerWidgetSet/
com.itmill.toolkit.demo.colorpicker.gwt.ColorPickerWidgetSet.nocache.js] not found
from filesystem or through class loader. Add widgetset and/or theme JAR to your
classpath or add files to WebContent/ITMILL folder.
```

Compilation with GWT is required also initially when using the Hosted Mode Browser described in Section 8.7.6, “Hosted Mode Browser”. The compilation with the GWT Compiler must be done at least once, as it provides files that are used also by the Hosted Mode Browser, even though the browser runs the GWT application in Java Virtual Machine instead of JavaScript.

Warning

Because GWT supports a slightly reduced version of Java, GWT compilation can produce errors that do not occur with the Java compiler integrated in the Eclipse IDE.

Compiling a Custom Widget Set

If you wish to use the build script to compile your own widget sets, open it in an editor. The build script contains some instructions in the beginning of the file. You can use the *`compile-my-widgetset`* target as a template for your own widget sets.

```
<!-- NOTE: Modify this example to compile your own widgetset  -->
<target name="compile-widgetset" depends="init">
  <echo>Compiling com.itmill.toolkit.demo.colorpicker.gwt.ColorPickerWidgetSet.</echo>
  <echo>Modify this example ant-script to compile your own widgetsets.</echo>
  <java classname="com.google.gwt.dev.GWTCompiler"
        failonerror="yes" fork="yes" maxmemory="256m">
      <arg value="-out" />
      <arg value="${client-side-destination}" />

      <!-- Define your GWT widget set class here. -->
      <arg value="com.itmill.toolkit.demo.colorpicker.gwt.ColorPickerWidgetSet" />

      <jvmarg value="-Xss1024k"/>
      <jvmarg value="-Djava.awt.headless=true"/>
      <classpath>
        <path refid="compile.classpath"/>
      </classpath>
```

```
  </java>
</target>
```

Replace the target name with your desired target name and the widget set class name with your own class name. The `-Xss` parameter may be necessary if you experience stack overflow errors with the default stack size. The `-Djava.awt.headless=true` is relevant in Linux/UNIX platforms to avoid some X11 warnings.

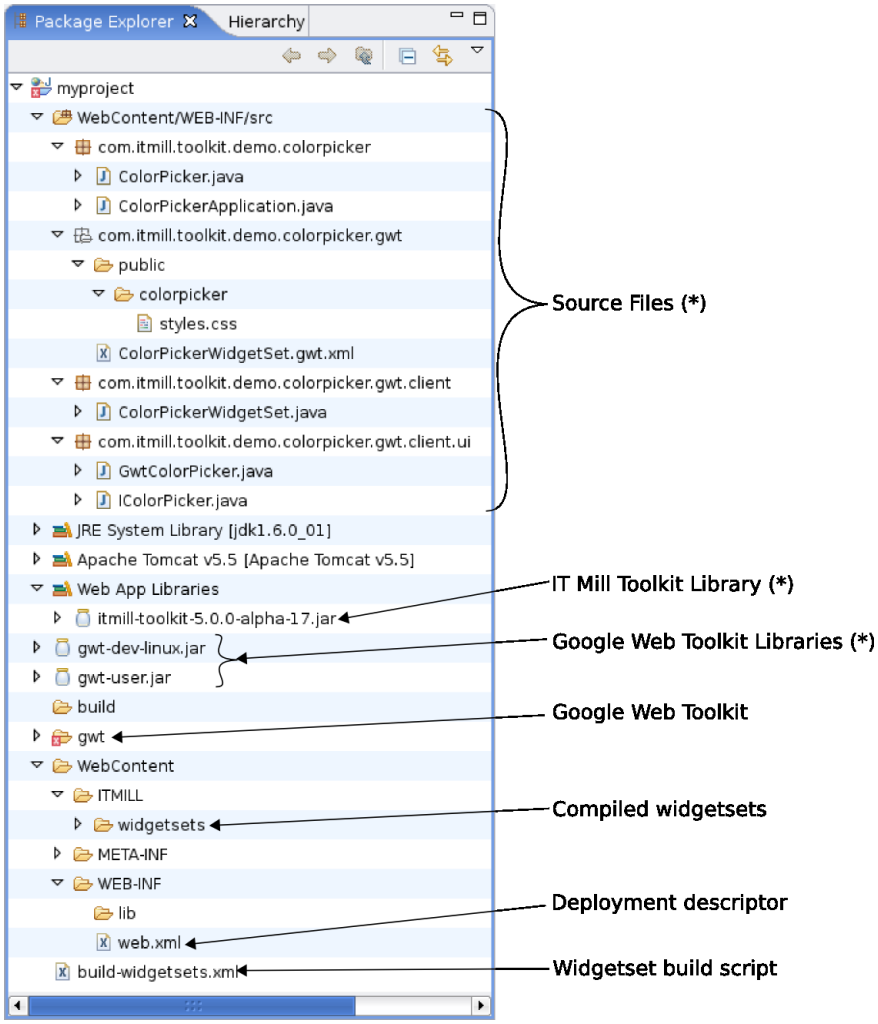
You can now compile the widget set with the following command:

```
$ ant -f build-widgetset.xml
```

8.7.5. Ready to Run

Figure 8.5, “Annotated Project Contents” shows the contents of a ready project.

Figure 8.5. Annotated Project Contents



Notice that the Package Explorer does not correspond with the file system contents. Eclipse displays the items marked with asterisk (*) in a logical location, instead of the physical location in the file system.

You can either run the application in web mode, as introduced in Section 1.6.7, or debug it with the GWT Hosted Mode Browser, as detailed in the next section.

8.7.6. Hosted Mode Browser

The Hosted Mode Browser does not work in Linux since IT Mill Toolkit 5.3.0. The Out of Process Hosted Mode, described in Section 8.7.7, "Out of Process Hosted Mode (OOPHM)", is an experimental alternative for the Hosted Mode Browser. It is platform-independent and works also on Linux.

The GWT Hosted Mode Browser allows easy debugging of GWT applications. The GWT application is actually not compiled into JavaScript, as is done in the deployment phase, but executed as a Java application. This makes it possible to debug the application with, for example, the Eclipse IDE.

Figure 8.6. Hosted Mode Browser

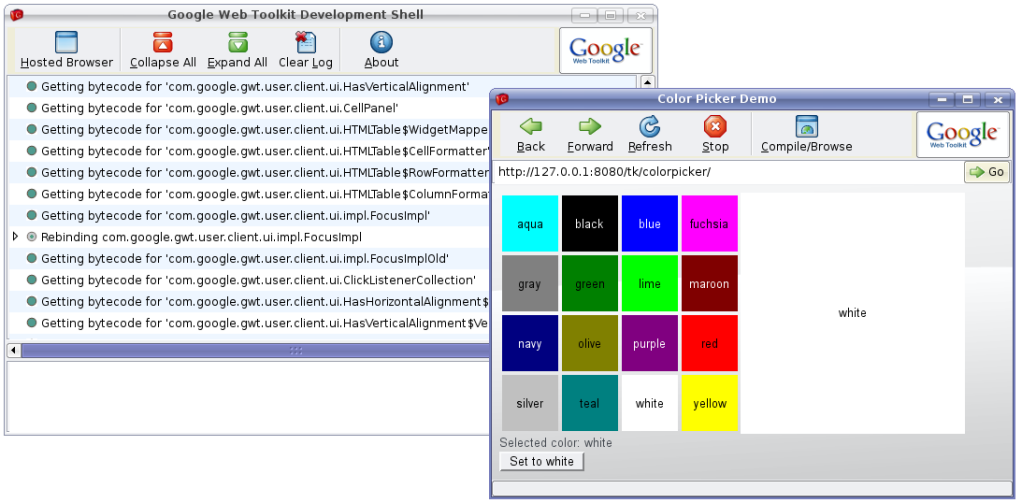


Figure 8.6, "Hosted Mode Browser" shows the hosted mode browser in action. On the left, you have the GWT Development Shell window. It displays compilation information and possible errors that occur during compilation. You can open a new browser window by clicking **Hosted Browser**.

The browser window has a **Compile/Browse** button, which runs the GWT Compiler to produce the JavaScript runtime and opens a regular web browser to run the application in Web Mode. Notice that even though it is possible to recompile the program with the button, GWT Compiler must be run before launching the Hosted Mode Browser, as described in Section 8.7.4, "Compiling GWT Widget Sets".

Because GWT supports a slightly reduced version of Java, GWT compilation can produce errors that do not occur with the Java compiler integrated in the Eclipse IDE. Such errors will show up in the GWT Development Shell window.

While the Hosted Mode Browser is a fast and easy way to debug applications, it does not allow inspecting the HTML or DOM tree or network traffic like Firebug does in Mozilla Firefox.

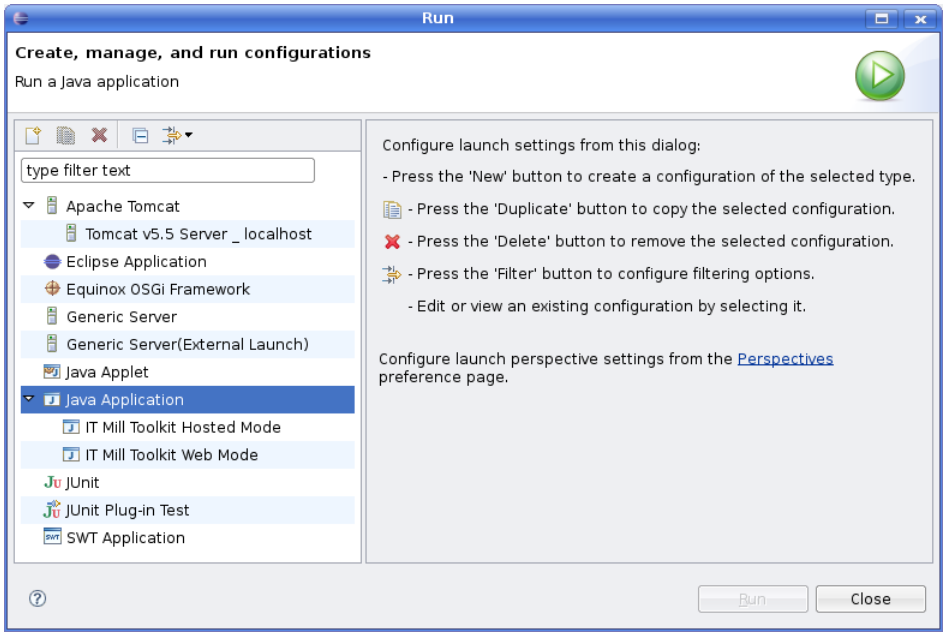
Configuring Hosted Mode Launching in Eclipse

This section gives details on configuring a launcher for the Hosted Mode Browser in the Eclipse IDE. We use the QuickStart installation of IT Mill Toolkit covered in Section 1.5, "QuickStart with Eclipse" as an example project. The project includes source code for the Color Picker demo application.

1. Select from menu **Run** → **Debug...** and the **Debug** configuration window will open. Notice that it is not purposeful to run the Hosted Mode Browser in the "**Run**" mode, because its entire purpose is to allow debugging.

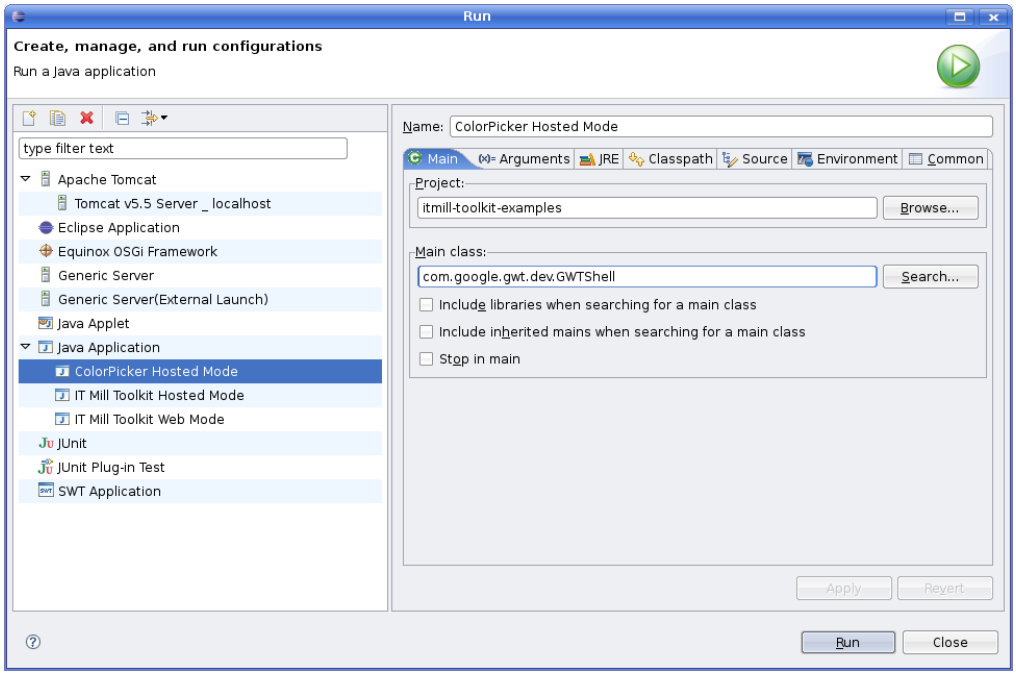
2. Select the **Java Application** folder and click on the **New** button to create a new launch configuration.

Figure 8.7. Creating New Launch Configuration



3. Click on the created launch configuration to open it on the right-side panel. In the **Main** tab, give the launch configuration a name. Define the **Main class** as **com.google.gwt.dev.GWTShell**.

Figure 8.8. Naming Launch Configuration



4. Switch to the **Arguments** tab and enter arguments for the Hosted Mode Browsed Java application.

   a. In the **Program arguments** field, enter:

   ```
   -noserver -whitelist "127.0.0.1  ^http[:][/][/]127[.]0[.]0[.]1[:]8080"
   -out WebContent/ITMILL/widgetsets http://127.0.0.1:8080/myproject
   ```

   The browser application, **GWTShell**, takes as its arguments the following parameters:

   -noserver — Prevents an embedded web server from starting, thereby allowing to use an already running server.

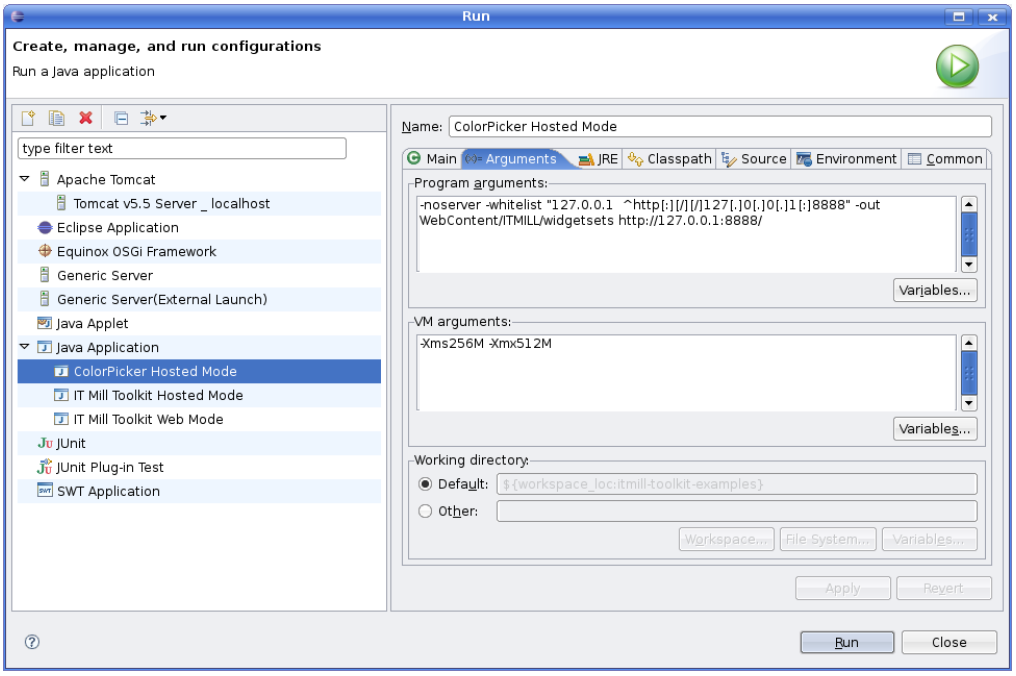
   -whitelist — Adds a regular expression to the list of allowed URL patterns for the web browser. Modify the port number from the 8080 given above as necessary.

   -out — Output directory for compiling widgets with GWT Compiler. The directory must be `WebContent/ITMILL/widgetsets`. You can compile the widgets either from the Hosted Mode Browser or externally as explained later in this chapter.

   URL — The URL to connect to. This must be the same as the whitelist entry given above. The port number must correspond to the port of the running web server. The Jetty web server included in IT Mill Toolkit will run in port 8888 by default. In contrast, Apache Tomcat installed under Eclipse will run in port 8080 by default.

   b. In the **VM arguments** field enter, for example, `-Xms256M -Xmx512M` to give the hosted mode browser more memory than the default amount. On Mac, add also `-XstartOnFirstThread`.

Figure 8.9. GWTShell Arguments



5. In the **Classpath** tab, you will by default have *`itmill-toolkit-examples`*, which contains the default classpath entries for the project. If the classpath entries for the project are sufficient, this should be enough.

6. Click **Apply** to save the launch configuration.

7. Click **Debug** to launch the Hosted Mode Browser using the launch configuration.

See the following section for details on debugging with the Hosted Mode Browser.

Debugging with Hosted Mode Browser

The purpose of the hosted mode browser is to allow debugging client-side GWT applications, or in our case, GWT widgets. Below is a checklist for important requirements for launching the Hosted Mode Browser:

- GWT is installed under the project folder.
- GWT libraries are included in the project classpath.
- Widget sets have been compiled with GWT Compiler.
- `web.xml` descriptor is configured.
- Web server is running and listening to the correct port.
- Hosted Mode Browser launcher is configured.

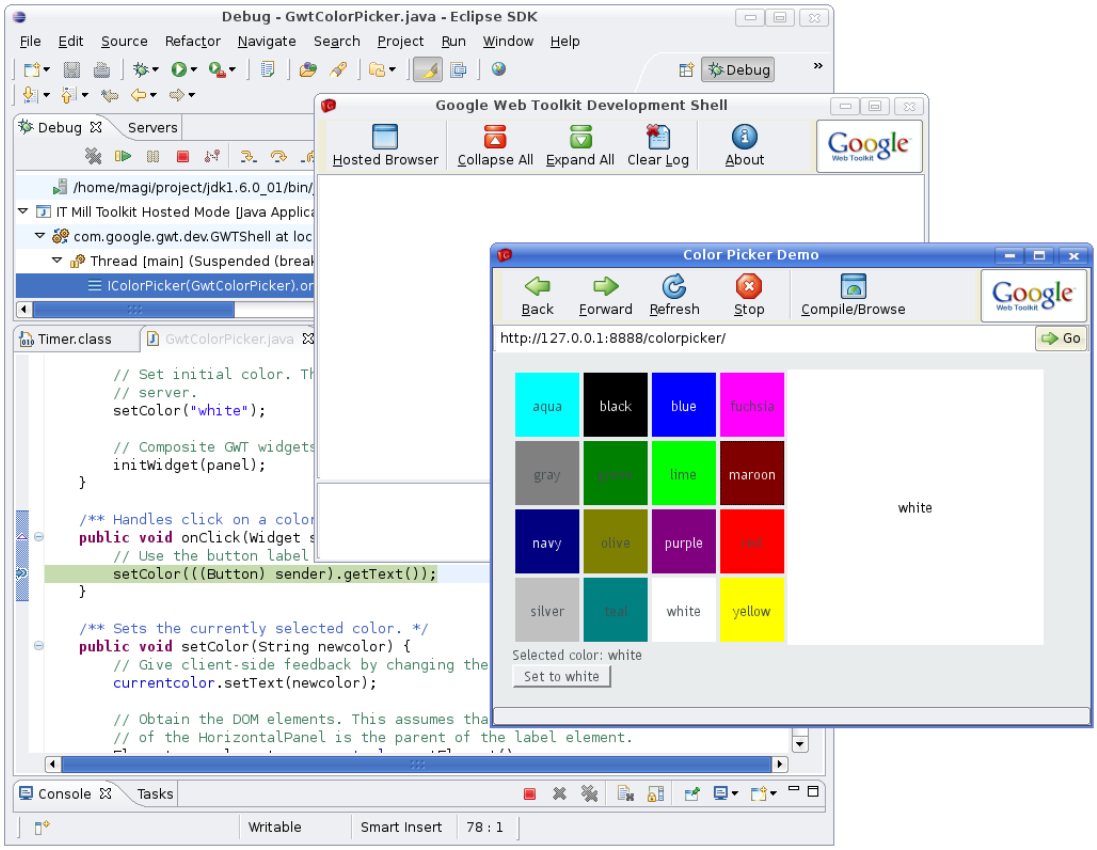
Once everything is ready to start debugging, just open a source file, for example, the **com.itmill.toolkit.demo.colorpicker.gwt.client.ui.GwtColorPicker** class. Find the `onClick()` method. At

the line containing the `setColor()` call, right-click on the leftmost bar in the editor and select **Toggle Breakpoint** from the popup menu. A small magnifying glass will appear in the bar to indicate the breakpoint.

Figure 8.10. Setting a Breakpoint

Select from menu **Run** → **Debug...** and the **Debug** configuration window will open. Notice that it is not purposeful to run the Hosted Mode Browser in the "**Run**" mode, because its entire purpose is to allow debugging.

Figure 8.11. Debugging with Hosted Mode Browser



Starting demo applications under the Hosted Mode Browser can take considerable time! This is especially true for the Reservation and Color Picker applications, which require compilation of custom widget sets. During this time, the Hosted Mode Browser is unresponsive and does not update its window. Compiling widgets can take 5-30 seconds, depending on the hardware.

Please refer to Eclipse IDE documentation for further instructions on using the debugger.

8.7.7. Out of Process Hosted Mode (OOPHM)

The Out of Process Hosted Mode of GWT is an experimental new way for debugging GWT applications in a regular web browser. This allows using other browser debugging tools, such as Firebug, while debugging in hosted mode.

The Hosted Mode Browser does not work in Linux since IT Mill Toolkit 5.3.0, so the OOPHM is the only way to debug client-side code in Linux.

The OOPHM installation package of IT Mill Toolkit is a platform-independent package available separately from the platform specific packages. Use of OOPHM requires (see more detailed notes further below):

1. Install OOPHM plugin from `gwt/plugins` in your browser
2. Compile custom widget sets with the GWT Compiler provided in `gwt-dev-oophm.jar` instead of the platform-dependent library.
3. Launch Hosted Mode debugging with the `gwt-dev-oophm.jar` in class path.

If you try debugging the demo applications in the IT Mill Toolkit installation package, just install the plugin (Step 1), launch the server in Web Mode, and then launch the Hosted Mode in debug mode (Step 3) with the included launch configuration.

The OOPHM plugin is available for Mozilla Firefox, Internet Explorer, and WebKit based browsers. You should install the plugin from the browser by opening the plugin file for your browser in the `gwt/plugins` directory. The Firefox plugin directory contains two plugins; you should normally use the `oophm-xpcom.xpi` plugin.

The installation package contains the built-in default widget set compiled with the OOPHM, but if you have your own widget sets (which is usually the reason why you want to use client-side debugging in the first place), you need to compile them. If you have compiled them previously with a regular installation of IT Mill Toolkit, you need to recompile them with the GWT Compiler provided in the `gwt-dev-oophm.jar` library. Compiling GWT widget sets is covered in Section 8.7.4, “Compiling GWT Widget Sets”. The compilation of OOPHM widget sets uses a large amount of stack memory, so if the JVM default is too small, you should set it explicitly in `compile-widgetset.xml` with the following parameter for the Java process (currently included in the example build script):

```
<jvmarg value="-Xss1024k"/>
```

Launching the debugging is done just as described in Section 8.7.6, “Hosted Mode Browser” for the regular Hosted Mode Browser, except that you must include the `gwt-dev-oophm.jar` library in the class path instead of the platform specific library. Launching the application with the debug configuration will contact the plugin in your browser and automatically opens the configured page.

Chapter 9. Advanced Web Application Topics

9.1. Debug and Production Mode

IT Mill Toolkit applications can be run in two modes: *debug mode* and *production mode*. The debug mode, which is on by default, enables a number of built-in debug features for the developers. The features include:

- Debug Window for accessing debug functionalities
- Display debug information in the Debug Window and server console.
- **Analyze layouting** button that analyzes the layout for possible problems.

Starting from IT Mill Toolkit version 5.3.0, all applications are by default run in the debug mode. The production mode can be enabled (and debug mode thereby disabled) by adding a `productionMode=true` parameter to the servlet context in the `web.xml` deployment descriptor:

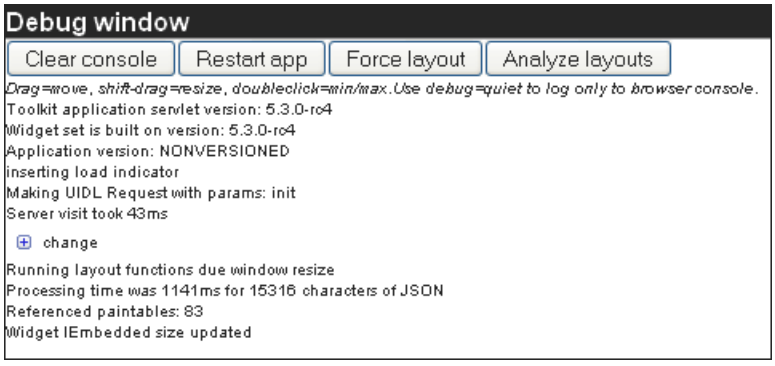
```
<context-param>
  <param-name>productionMode</param-name>
  <param-value>true</param-value>
  <description>IT Mill Toolkit production mode</description>
</context-param>
```

Enabling the production mode disables the debug features, thereby preventing users from easily inspecting the inner workings of the application from the browser.

9.1.1. Debug Mode

Running an application in the debug mode enables the client-side Debug Window in the browser. You can open the Debug Window by adding "`?debug`" to the application URL, e.g., `http://localhost:8080/myapp/?debug`. The Debug Window, shown in Figure 9.1, “Debug Window”, consists of buttons controlling the debugging features and a scrollable log of debug messages.

Figure 9.1. Debug Window



Clear console Clears the log in the Debug Window.

Restart app Restarts the application.

Force layout	Causes all currently visible layouts to recalculate their appearance. Layout components in IT Mill Toolkit 5.3.0 and later calculate the space required by all child components, so the layout appearance must be recalculated whenever the size of a child component is changed. In normal applications, this is done automatically, but when you do themeing or alter the CSS with Firebug, you may need to force all layouts to recalculate themselves, taking into account the recently made changes.
Analyze layouts	This is described in the following section.

If you use the Firebug plugin in Mozilla Firefox, the log messages will also be printed to the Firebug console. In such a case, you may want to enable client-side debugging without showing the Debug Window with "`?debug=quiet`" in the URL. In the quiet debug mode, log messages will only be printed to the Firebug console.

9.1.2. Analyzing Layouts

The **Analyze layouts** button analyzes the currently visible layouts and makes a report of possible layout related problems. All detected layout problems are displayed in the log and also printed to the console.

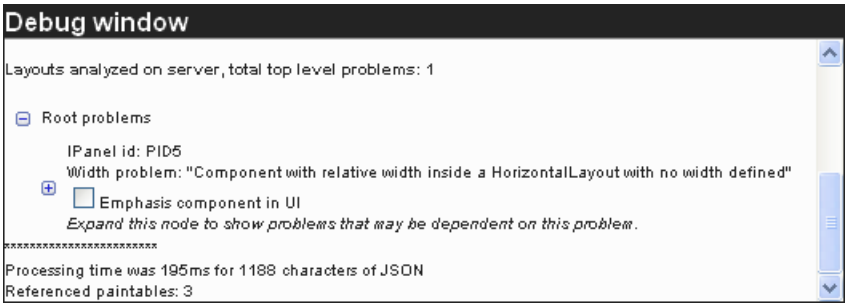
The most common layout problem is caused by placing a component that has a relative size inside a container (layout) that has undefined size, e.g., adding a 100% wide **Panel** inside a **HorizontalLayout** with no width specification. In such a case, the error will look as shown below:

```
IT Mill Toolkit DEBUG
- Window/1a8bd74 "My window" (width: MAIN WINDOW)
  - HorizontalLayout/1cf243b (width: UNDEFINED)
    - Panel/12e43f1 "My panel" (width: RELATIVE, 100.0 %)
Layout problem detected: Component with relative width inside a HorizontalLayout with no
 width defined
Relative sizes were replaced by undefined sizes, components may not render as expected.
```

This particular error tells that the **Panel** "My panel" is 100% wide while the width of the containing **HorizontalLayout** is undefined. The components will be rendered as if the the width of the contained **Panel** was undefined, which might not be what the developer wanted. There are two possible fixes for this case: if the **Panel** should fill the main window horizontally, set a width for the **HorizontalLayout** (e.g. 100% wide), or set the width of the **Panel** to "undefined" to render the it as it is currently rendered but avoiding the warning message.

The same error is shown in the Debug Window in a slightly different form and with an additional feature (see Figure 9.2, “Debug Window Showing the Result of **Analyze layouts**.”). Checking the **Emphasize component in UI** box will turn the background of the component that caused a warning red, making it easy for the developer to figure out which component each warning relates to. The messages will also be displayed hierarchically, as a warning from a containing component often causes more warnings from its child components. A good rule of thumb is to work on the upper-level problems first and only after that worry about the warnings from the children.

Figure 9.2. Debug Window Showing the Result of Analyze layouts.



9.1.3. Custom Layouts

CustomLayout components can not be analyzed in the same way as other layouts. For custom layouts, the **Analyze layouts** button analyzes all contained relative-sized components and checks if any relative dimension is calculated to zero so that the component will be invisible. The error log will display a warning for each of these invisible components. It would not be meaningful to emphasize the component itself as it is not visible, so when you select such an error, the parent layout of the component is emphasized if possible.

9.1.4. Debug Functions for Component Developers

You can take advantage of the debug mode when developing client-side components. The static function `ApplicationConnection.getConsole()` will return a reference to a **Console** object which contains logging methods such as `log(String msg)` and `error(String msg)`. These functions will print messages to the Debug Window and Firebug console in the same way as other debugging functionalities of IT Mill Toolkit do. No messages will be printed if the Debug Window is not open or if the application is running in production mode.

9.2. Special Characteristics of AJAX Applications

This section is intended for people familiar with the development of either traditional web applications or desktop applications, who are entering AJAX enabled web application development. AJAX application development has a few special characteristics with respect to other types of applications. Possibly the most important one is how the display is managed in the web browser.

The web was originally not built for applications, but for hypertext pages that you can view with a browser. The purpose of web pages is to provide *content* for the user. Application software has a somewhat different purpose; usually to allow you to work on some data or content, much of which is not ever intended to be accessible through a web browser as web pages. As the web is inherently page-based, conventional web applications had to work with page requests and output HTML as response. JavaScript and AJAX have made it possible to let go of the pages.

Pages are largely an unknown concept to conventional desktop applications. At most, desktop applications can open multiple windows, but usually they work with a single main window, with an occasional dialog window here and there. Same goes usually for web applications developed with IT Mill Toolkit: an application typically runs on a single page, changing the layout as needed and popping up dialog boxes.

Not having to load pages and use hyperlinks to communicate all user interaction is a relief for application development. However, they are an important feature that ordinary desktop applications lack. They allow referencing different functionalities of an application or resources managed by the application. They are also important for integration with external applications.

Certain resources can be identified through a *URI* or *Universal Resource Identifier*. A URI can easily be passed around or stored as a bookmark. We will see in Section 9.3.1, “URI Handlers” how you can retrieve the URI of a page request. Similarly, a page request can have query parameters, which can be handled as detailed in Section 9.3.2, “Parameter Handlers”.

Using URIs or request parameters to access functionalities or content is not as straight-forward as in conventional page-based web applications. IT Mill Toolkit, just as any other AJAX framework, uses browser cookies not just for tracking users but also for tracking the application state. Cookies are unique in a browser, so any two windows share the same cookies and therefore also the state. The advantage is that you can close your browser and open it again and the application will be in the state where you left off (except for components such as text fields which did not have the immediate attribute enabled). The disadvantage is that there is no good way to distinguish between the windows, so there can usually be only a single application window. Even if there were several, you would have trouble with synchronization of application data between windows. Many conventional page-based web applications simply ignore out-of-sync situations, but such situations are risky for application platforms that are intended to be stable. Therefore it is safer to work with a single browser window. If you wish to have multiple windows in your application, you can create them inside the main window as **Window** objects. A URI can be used to fetch resources that have no particular state or to provide an entry point to the application.

9.3. Resources

In addition to high-level resource classes described in Section 3.5, “Referencing Resources”, IT Mill Toolkit provides low-level facilities for retrieving the URI and other parameters of HTTP requests. In the following, we will look into low-level interfaces for handling URIs and parameters to provide resources and functionalities.

Notice that using URI or parameter handlers to create "pages" is not meaningful in IT Mill Toolkit or in AJAX applications generally. See Section 9.2, “Special Characteristics of AJAX Applications” for reasons.

9.3.1. URI Handlers

The URI parameter for the application is useful mainly for two purposes: for providing some special functionality according to the URI or for providing dynamic content. Dynamic content can also be provided with **StreamResource**.

You can retrieve the URI for the HTTP request made for your application by implementing the **com.itmill.toolkit.terminal.URIHandler** interface. The handler class needs to be registered in the main window object of your application with the `addURIHandler()` method. You then get the URI by implementing the `handleURI()` method. The method gets two parameters: a context and a URI relative to the context. The context is the base URI for your application.

```
public void init() {
    final Window main = new Window("Hello window");
    setMainWindow(main);

    URIHandler uriHandler = new URIHandler() {
        public DownloadStream handleURI(URL context, String relativeUri) {
            // Do something here
            System.out.println("handleURI=" + relativeUri);
            return null; // Should be null unless providing dynamic data.
        }
    };
    main.addURIHandler(uriHandler);

}
```

If you have multiple URI handlers attached to a window, they are executed after one another. The URI handlers should return *null*, unless you wish to provide dynamic content with the call. Other URI handlers attached to the window will not be executed after some handler returns non-null data. The combined parameter and URI handler example below shows how to create dynamic content with a URI handler.

Notice that if you do provide dynamic content with a URI handler, the dynamic content is returned in the HTTP response. If the handler makes any changes to the UI state of the application, these changes are not rendered in the browser, as they are usually returned in the HTTP response made by the Application object and now the custom URI handler overrides the default behaviour. If your client-side code makes a server call that does update the UI state, the client-side must initiate an update from the server. For example, ff you have an integration situation where you make a JavaScript call to the server, handle it the request with a URI handler, and the server state changes as a side-effect, you can use the `itmill.forceSync()` method to force the update.

9.3.2. Parameter Handlers

You can retrieve the parameters passed to your application by implementing the **com.itmill.toolkit.terminal.ParameterHandler** interface. The handler class needs to be registered in the main window object of your application with the `addParameterHandler()` method. You then get the parameters in the `handleParameters()` method. The parameters are passes as a map from string key to a vector of string values.

```
class MyParameterHandler implements ParameterHandler {
    public void handleParameters(Map parameters) {
        // Print out the parameters to standard output
        for (Iterator it = parameters.keySet().iterator(); it.hasNext();) {
            String key   = (String) it.next();
            String value = ((String[]) parameters.get(key))[0];
            System.out.println("Key: "+key+", value: "+value);
        }
    }
}
```

The parameter handler is not called if there are no parameters. Parameter handler is called before the URI handler, so if you handle both, you might typically want to just store the URI parameters in the parameter handler and do actual processing in URI handler. This allows you, for example, to create dynamic resources based on the URI parameters.

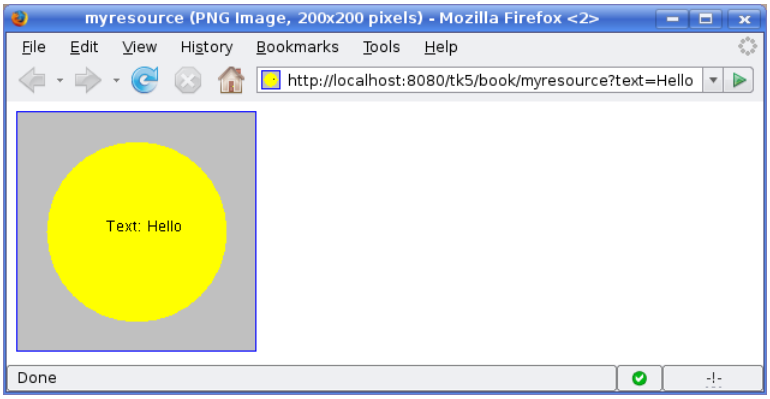
```
import java.awt.*;
import java.awt.image.BufferedImage;
import java.io.*;
import java.net.URL;
import java.util.Map;
import javax.imageio.ImageIO;
import com.itmill.toolkit.terminal.*;

/**
 * Demonstrates handling URI parameters and the URI itself to create a dynamic
 * resource.
 */
public class MyDynamicResource implements URIHandler, ParameterHandler {
    String textToDisplay = "- no text given -";

    /**
     * Handle the URL parameters and store them for the URI handler to use.
     */
    public void handleParameters(Map parameters) {
        // Get and store the passed HTTP parameter.
        if (parameters.containsKey("text"))
            textToDisplay = ((String[])parameters.get("text"))[0];
    }
```

```
    /**
     * Provides the dynamic resource if the URI matches the resource URI. The
     * matching URI is "/myresource" under the application URI context.
     *
     * Returns null if the URI does not match. Otherwise returns a download
     * stream that contains the response from the server.
     */
    public DownloadStream handleURI(URL context, String relativeUri) {
        // Catch the given URI that identifies the resource, otherwise let other
        // URI handlers or the Application to handle the response.
        if (!relativeUri.startsWith("myresource"))
            return null;

        // Create an image and draw some background on it.
        BufferedImage image = new BufferedImage (200, 200, BufferedImage.TYPE_INT_RGB);
        Graphics drawable = image.getGraphics();
        drawable.setColor(Color.lightGray);
        drawable.fillRect(0,0,200,200);
        drawable.setColor(Color.yellow);
        drawable.fillOval(25,25,150,150);
        drawable.setColor(Color.blue);
        drawable.drawRect(0,0,199,199);

        // Use the parameter to create dynamic content.
        drawable.setColor(Color.black);
        drawable.drawString("Text: "+textToDisplay, 75, 100);

        try {
            // Write the image to a buffer.
            ByteArrayOutputStream imagebuffer = new ByteArrayOutputStream();
            ImageIO.write(image, "png", imagebuffer);

            // Return a stream from the buffer.
            ByteArrayInputStream istream = new
ByteArrayInputStream(imagebuffer.toByteArray());
            return new DownloadStream (istream,null,null);
        } catch (IOException e) {
            return null;
        }
    }
}
```

When you use the dynamic resource class in your application, you obviously need to provide the same instance of the class as both types of handler:

```
MyDynamicResource myresource = new MyDynamicResource();
mainWindow.addParameterHandler(myresource);
mainWindow.addURIHandler(myresource);
```

Figure 9.3. Dynamic Resource with URI Parameters



9.4. Shortcut Keys

Shortcut keys can be defined as *actions* using the **ShortcutAction** class. To handle key presses, you need to define an action handler by implementing the **Handler** interface. The interface has two methods that you need to implement: `getActions()` and `handleAction()`.

The `getActions()` interface method must return an array of **Action** objects for the component specified with the second parameter for the method, the *`sender`* of an action. For a keyboard shortcut, you use a **ShortcutAction**. The implementation of the method should look somewhat as follows:

```
public Action[] getActions(Object target, Object sender) {
    Action[] actions = new Action[1];

    // Set the action for the requested component
    if (sender == ok) {
        // Bind the unmodified Enter key to the Ok button.
        actions[0] = new ShortcutAction("Default key",
                                        ShortcutAction.KeyCode.ENTER, null);
    } else if (sender == cancel) {
        // Bind "C" key modified with Alt to the Cancel button.
        actions[0] = new ShortcutAction("Alt+C",
                                        ShortcutAction.KeyCode.C, new int[] {
                                               ShortcutAction.ModifierKey.ALT});

    } else
        return null;
    return actions;
}
```

The method takes a symbolic caption for the action; this is largely irrelevant for shortcut actions. The second parameter is the keycode, as defined in **ShortcutAction.KeyCode** interface. Currently, the following keycodes are allowed:

Keys A to Z	Normal letter keys
F1 to F12	Function keys
BACKSPACE, DELETE, ENTER, ESCAPE, INSERT, TAB	Control keys
NUM0 to NUM9	Number pad keys
ARROW_DOWN, ARROW_UP, ARROW_LEFT, ARROW_RIGHT	Arrow keys

HOME, END, PAGE_UP, PAGE_DOWN	Other movement keys

The third parameter is an array of modifier keys, as defined in the **ShortcutAction.ModifierKey** interface. The following modifier keys are allowed: *ALT*, *CTRL*, and *SHIFT*. The modifier keys can be combined; for example, the following defines shortcut key combination **Ctrl-Shift-S**:

```
ShortcutAction("Ctrl+Shift+S",
               ShortcutAction.KeyCode.S, new int[] {
                      ShortcutAction.ModifierKey.CTRL,
                      ShortcutAction.ModifierKey.SHIFT});
```

The following example demonstrates the definition of a default button for a user interface, as well as a normal shortcut key, **Alt-C** for clicking the **Cancel** button.

```
import com.itmill.toolkit.event.Action;
import com.itmill.toolkit.event.ShortcutAction;
import com.itmill.toolkit.event.Action.Handler;
import com.itmill.toolkit.ui.*;

public class DefaultButtonExample extends CustomComponent implements Handler {
    // Define and create user interface components
    Panel            panel      = new Panel("Login");
    VerticalLayout   formlayout = new VerticalLayout();
    TextField        username   = new TextField("Username");
    TextField        password   = new TextField("Password");
    HorizontalLayout buttons    = new HorizontalLayout();

    // Create buttons and define their listener methods. Here we use parameterless
    // methods so that we can use same methods for both click events and keyboard
    // actions.
    Button        ok         = new Button("OK", this, "okHandler");
    Button        cancel     = new Button("Cancel", this, "cancelHandler");

    public DefaultButtonExample() {
        // Set up the user interface
        setCompositionRoot(panel);
        panel.addComponent(formlayout);
        formlayout.addComponent(username);
        formlayout.addComponent(password);
        formlayout.setStyle("form");
        formlayout.addComponent(buttons);
        buttons.addComponent(ok);
        buttons.addComponent(cancel);

        // Set focus to username
        username.focus();

        // Set this object as the action handler for actions related to the Ok
        // and Cancel buttons.
        ok.addActionHandler(this);
        cancel.addActionHandler(this);
    }

    /**
     * Retrieve actions for a specific component. This method will be called for each
     * object that has a handler; in this example the Ok and Cancel buttons.
     **/
    public Action[] getActions(Object target, Object sender) {
        Action[] actions = new Action[1];

        // Set the action for the requested component
        if (sender == ok) {
            // Bind the unmodified Enter key to the Ok button.
            actions[0] = new ShortcutAction("Default key",
                                            ShortcutAction.KeyCode.ENTER, null);
```

```
            } else if (sender == cancel) {
                // Bind "C" key modified with Alt to the Cancel button.
                actions[0] = new ShortcutAction("Alt+C",
                                                ShortcutAction.KeyCode.C, new int[] {
                                                       ShortcutAction.ModifierKey.ALT});
            } else
                return null;
            return actions;
        }

        /**
         * Handle actions received from keyboard. This simply directs the actions to
         * the same listener methods that are called with ButtonClick events.
         **/
        public void handleAction(Action action, Object sender, Object target) {
            if (target == ok)
                this.okHandler();
            if (target == cancel)
                this.cancelHandler();
        }

        public void okHandler() {
            // Do something: report the click
            formlayout.addComponent(new Label("OK clicked"));
        }

        public void cancelHandler() {
            // Do something: report the click
            formlayout.addComponent(new Label("Cancel clicked"));
        }
    }
```

Notice that the keyboard actions are handled from the entire page. This can cause problems if you have components that require a certain key. For example, multi-line **TextField** requires the **Enter** key. There is currently no way to filter the shortcut actions out while the focus is inside some specific component, so you need to avoid such conflicts.

9.5. Printing

IT Mill Toolkit does not currently have any special support for printing. Printing on the server-side is anyhow largely independent from the web UI of an application. You just have to take care that the printing does not block server requests, possibly by running printing in another thread.

For client-side printing, most browsers support printing the web page. The `print()` method in JavaScript opens the print window of the browser. You can easily make a HTML button or link that prints the current page by placing the custom HTML code inside a **Label**.

```
main.addComponent(new Label("<input type='button' onClick='print()' value='Click to
Print'/>", Label.CONTENT_XHTML));
```

This button would print the current page. Often you want to be able to print a report or receipt and it should not have any UI components. In such a case you could offer it as a PDF resource, or you could open a new window as is done below and automatically launch printing.

```
// A button to open the printer-friendly page.
Button printButton = new Button("Click to Print");
main.addComponent(printButton);
printButton.addListener(new Button.ClickListener() {
    public void buttonClick(ClickEvent event) {
        // Create a window that contains stuff you want to print.
        Window printWindow = new Window("Window to Print");

        // Have some content to print.
```

```
        printWindow.addComponent(new Label("Here's some dynamic content."));

        // To execute the print() JavaScript, we need to run it
        // from a custom layout.
        CustomLayout scriptLayout = new CustomLayout("printpage");
        printWindow.addComponent (scriptLayout);

        // Add the printing window as an application-level window.
        main.getApplication().addWindow(printWindow);

        // Open the printing window as a new browser window
        main.open(new ExternalResource(printWindow.getURL()), "_new");
    }
});
```

How the browser opens the window, as an actual window or just a tab, depends on the browser. Notice that above we create a new window object each time the print button is clicked, which leads to unused window objects. If this is a real problem, you might want to reuse the same window object or clean up the old ones - it's ok because the user doesn't interact with them later anyhow.

You will also need a custom layout that contains the `print()` JavaScript function that launches the printing. Notice that the custom layout *must* contain also another element (below a `<div>`) in addition to the `<script>` element.

```
<div>This is some static content.</div>

<script type='text/javascript'>
    print();
</script>
```

Printing as PDF would not require creating a **Window** object, but you would need to provide the content as a static or a dynamic resource for the `open()` method. Printing a PDF file would obviously require a PDF viewer cabability (such as Adobe Reader) in the browser.

Chapter 10. User Interface Definition Language (UIDL)

User Interface Definition Language (UIDL) is a language for serializing user interface contents and changes in responses from web server to a browser. The idea is that the server-side components "paint" themselves to the screen (a web page) with the language. The UIDL messages are parsed in the browser and translated to GWT widgets.

The UIDL is used through both server-side and client-side APIs. The server-side API consists of the **PaintTarget** interface, described below in Section 10.1, “API for Painting Components”. The client-side interface depends on the implementation of the client-side engine. In IT Mill Toolkit Release 5, the client-side engine uses the Google Web Toolkit framework. Painting the user interface with a GWT widget is described in Section 8.2, “Google Web Toolkit Widgets”.

UIDL supports painting either the entire user interface or just fragments of it. When the application is started by opening the page in a web browser, the entire user interface is painted. If a user interface component changes, only the changes are painted.

Since IT Mill Toolkit Release 5, the UIDL communications are currently done using JSON (JavaScript Object Notation), which is a lightweight data interchange format that is especially efficient for interfacing with JavaScript-based AJAX code in the browser. The use of JSON as the interchange format is largely transparent; IT Mill Toolkit version 4 and the older versions used an XML-based UIDL representation with the same API. Nevertheless, the UIDL API uses XML concepts such as attributes and elements. Below, we show examples of a **Button** component in both XML and JSON notation.

With XML notation:

```
<button id="PID2" immediate="true" caption="My Button" focusid="1">
    <boolean id="v1" name="state" value="false"></boolean>
</button>
```

With JSON notation:

```
["button",
 {"id": "PID2",
  "immediate":true,
  "caption": "My Button",
  "focusid":1,
  "v":{"state":false}
 }
]
```

Components are identified with a *PID* or *paintable identifier* in the `id` attribute. Each component instance has its individual PID, which is usually an automatically generated string, but can be set manually with `setDebugId()` method.

Section 10.2, “JSON Rendering” below gives further details on JSON. For more information about handling UIDL messages in the client-side components, see Chapter 8, *Developing Custom Components*.

You can track and debug UIDL communications easily with the Firebug extension for Mozilla Firefox, as illustrated in Section 10.2, “JSON Rendering” below.

10.1. API for Painting Components

Serialization or "painting" of user interface components from server to the client-side engine running in the browser is done through the **PaintTarget** interface. In IT Mill Toolkit Release 5, the only implementation of the interface is the **JsonPaintTarget**, detailed in Section 10.2, “JSON Rendering” below.

The abstract **AbstractComponent** class allows easy painting of user interface components by managing many basic tasks, such as attributes common for all components. Components that inherit the class need to implement the abstract `getTag()` method that returns the UIDL tag of the component. For example, the implementation for the **Button** component is as follows:

```
public String getTag() {
    return "button";
}
```

AbstractComponent implements the `paint()` method of the **Paintable** interface to handle basic tasks in painting, and provides `paintContent()` method for components to paint their special contents. The method gets the **PaintTarget** interface as its parameter. The method should call the default implementation to paint any common attributes.

```
/* Paint (serialize) the component for the client. */
public void paintContent(PaintTarget target) throws PaintException {
    // Superclass writes any common attributes in the paint target.
    super.paintContent(target);

    // Set any values as variables of the paint target.
    target.addVariable(this, "colorname", getColor());
}
```

Serialized data can be attributes or variables, serialized with the `addAttribute()` and `addVariable()` methods, respectively. You must always serialize the attributes first and the variables only after that.

The API provides a number of variations of the methods for serializing different basic data types. The methods support the native Java data types and strings of the **String** class. `addVariable()` also supports vectors of strings.

Contained components are serialized by calling the `paint()` method of a sub-component, which will call the `paintContent()` for the sub-component, allowing the serialization of user interfaces recursively. The `paint()` method is declared in the server-side **Paintable** interface and implemented in the abstract base classes, **AbstractComponent** and **AbstractComponentContainer** (for layouts).

Layout components have to serialize the essential attributes and variables they need, but not the contained components. The **AbstractComponentContainer** and **AbstractLayout** baseclasses manage the recursive painting of all the contained components in layouts.

The **AbstractField** provides an even higher-level base class for user interface components. The field components hold a value or a *property*, and implement the **Property** interface to access this property. For example the property of a **Button** is a **Boolean** value.

```
public void paintContent(PaintTarget target) throws PaintException {
    super.paintContent(target);

    // Serialize the switchMode as an attribute
    if (isSwitchMode())
        target.addAttribute("type", "switch");

    // Get the state of the Button safely
    boolean state;
    try {
        state = ((Boolean) getValue()).booleanValue();
```

```
            } catch (NullPointerException e) {
                state = false;
            }
            target.addVariable(this, "state", state);

    }
```

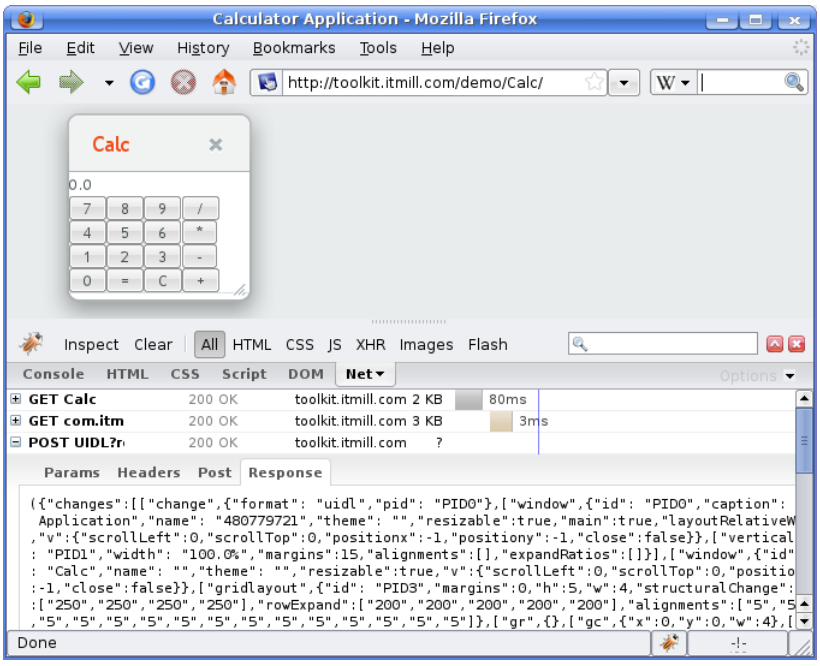
10.2. JSON Rendering

IT Mill Toolkit 5 uses JSON, a lightweight data-interchange format, to communicate UI rendering with the browser, because it is very fast to parse compared to XML. JSON messages are essentially JavaScript statements that can be directly evaluated by the browser. The client-side engine of IT Mill Toolkit parses and evaluates the UIDL messages with the JSON library that comes with the Google Web Toolkit.

Section 2.2.3, “JSON” gave a general introduction to JSON as part of the architecture of IT Mill Toolkit. In this section, we look into the technical details of the format. The technical details of the JSON messages are useful mainly for debugging purposes, for example using the Firebug plugin for Mozilla Firefox.

To view a UIDL message, open the Firebug panel in Firefox, select **Net** tab, select a "`POST UIDL`" request, open the **Response** tab, and click **Load Response**. This displays the entire UIDL message, as shown in Figure 10.1, “Debugging UIDL Messages with Firebug” below.

Figure 10.1. Debugging UIDL Messages with Firebug



JSON messages are represented as nested lists and associative arrays (objects with named properties) in JavaScript syntax. At the top level, we can find an associative array with the following fields:

changes	Changes to the UI caused by the request.
meta	Meta-information regarding the response and the application state.
resources	Information about application resources.

locales	Locale-specific data for locale-dependent components, such as names of months and weekdays.

The "`changes`" field contains the actual UI changes as a list of components. Components that can contain other components are represented in a recursive list structure.

A component is represented as a list that first contains the UIDL tag of the component, which identifies its class, followed by data fields. The basic representation of component data as attributes and variables is defined in the base classes of the framework. Attributes are represented as an associative array and variables as a separate associative array inside the special "`v`" attribute. For example, a **Button** component is communicated with a JSON representation such as the following:

```
["button",
 {"id": "PID5",
  "immediate": true,
  "caption": "7",
  "v":{"state":false}}
]
```

A component can give its data also in additional fields in the list instead of the attributes or variables, as is done for the **Label** component:

```
["label",
 {"id": "PID4",
  "width": "100.0%"},
 "Some text here"]
```

The meta-information field can contain certain types of information, which are not displayed in the UI, but used by the client-side engine. The `repaintAll` parameter tells that the changes include the entire window contents, not just partial changes. Other data includes redirection details for expired sessions.

Bibliography

[biblio.gwt-guide] *Google Web Toolkit Developer Guide*.

`http://code.google.com/webtoolkit/documentation/com.google.gwt.doc.DeveloperGuide.html`