Mobile Front Controller

Developer's guide for software version 3.1

ERICSSON

TAKING YOU FORWARD

Copyright

Disclaimer

Trademark List

Abstract

This document describes how to build, configure and use Mobile Front Controller. The design of Mobile Front Controller is also described in this document.

Contents

1 Introduction

1.1 Scope

Mobile Front Controller (MFC) is a light-weight Java EE web application framework based on design patterns. MFC is used for creating internet web and mobile applications, i.e. applications with views for web browsing and mobile browsing that share UI logic. The framework is easy to use and does not require an XML file for navigation.

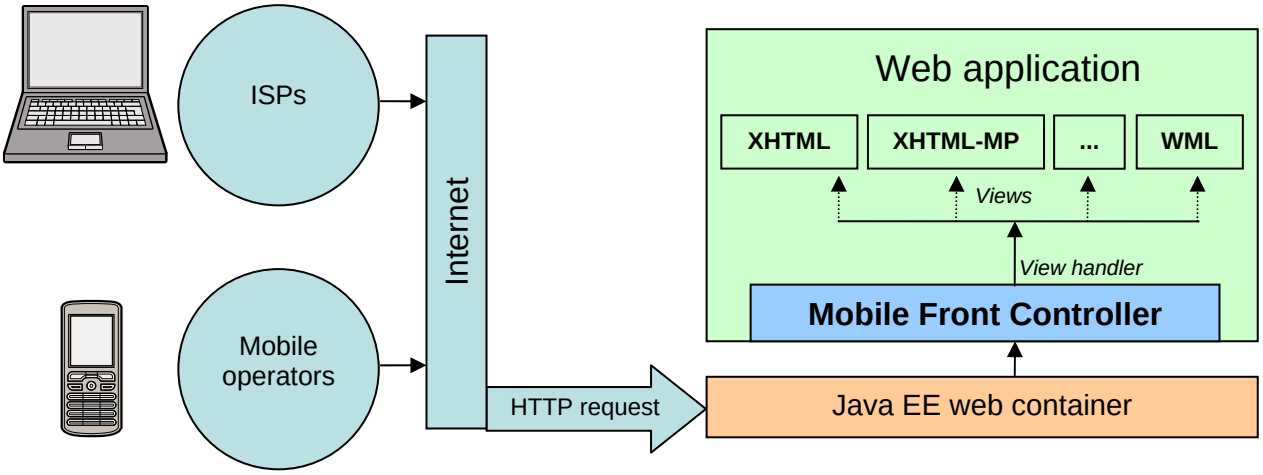

Figure 1 An overview of Mobile Front Controller used by a web application on a Java EE web container.

MFC is used on top of a Java EE web container, and does not require any other framework, such as JSF, Struts, etc.

MFC does the following:

- Detects and selects appropriate views based on HTTP request headers. A view is a subdirectory that, for example, corresponds to a markup language such as XHTML, XHTML Mobile Profile (MP) and WML (see Figure 1). The way MFC detects and selects views is customizable using view handlers.
- Shares UI logic between different views, for example, web and mobile browsing. This is done using action commands, which are classes that contain an execute method that is executed when a URL with, for example, the URL pattern `*.do` is called (for example `http://localhost:8080/mfc-basic-demo/xhtml/Print.do`).

MFC contains Ant scripts for building the library (.jar file) and creating the corresponding Javadoc, which is packaged into a .zip file.

1.2 Purpose

The purpose of MFC is to make it easier to start creating web applications that share UI logic between views, for example, different markup languages such as XHTML, XHTML-MP and WML.

1.3 Target group

The target group of MFC is Java developers that want to create applications for web browsing and mobile browsing.

1.4 Document disposition

This document contains the following major sections:

- Download and installation.
- A user guide that describes how to build, configure and use MFC.
- A guide that describes how MFC is designed.

The following layout convention is used:

- `Courier New, font size 8`
  This typeface is used for code-related text, parameters, catalog names and file names.
- **Bold text** is used for code-related text, parameters, etc. that requires extra attention.

2 Download and installation

This chapter describes how to download and install MFC and third-party products.

2.1 MFC

Download MFC from the Ericsson Mobility World Developer Program website, http://www.ericsson.com/mobilityworld.

Extract the zip file that contains MFC to, for example, `C:\MFC`, which will from now on be referred to as `<mfc>`. The package structure is described in `README.TXT`.

2.2 Third-party product information

MFC requires a series of freely downloadable third-party products (3PPs), marked with (**REQUIRED**). There are also some optional third-party products, which are marked with (**OPTIONAL**). Download and install the following 3PPs:

- Java SE Development Kit (JDK) 6 update 7. (**REQUIRED**). The JDK installation directory will from now on be referred to as `<jdk>`. Download from: http://java.sun.com/javase/downloads/.

- Ant 1.7.0. (**REQUIRED**).The Ant installation directory will from now on be referred to as `<ant>`. Download from: http://ant.apache.org/.

To build the MFC library, libraries from Tomcat **or** GlassFish are needed.

- Tomcat 6.0.18. The Tomcat installation directory will from now on be referred to as `<tomcat>`. Download from: http://tomcat.apache.org/download-60.cgi.

- GlassFish V2 UR2 b04. The GlassFish installation directory will from now on be referred to as `<glassfish>`. Download from: https://glassfish.dev.java.net/.

2.3 Operating system (OS) environment variables

2.3.1 Required OS environment variables

The environment variables that have to be set in order to perform different steps in this guide are shown in Table 1. See section 2.3.2 for information on how to set up environment variables.

Variable	Set to	Description
JAVA_HOME	<jdk>	Set JAVA_HOME operating system environment variable to the JDK installation directory. Required for Ant tasks.
TOMCAT_HOME	<tomcat>	Set TOMCAT_HOME operating system environment variable to the Tomcat installation directory.
AS_HOME	<glassfish>	Set AS_HOME operating system environment variable to the GlassFish installation directory.
PATH	%PATH%;%ANT_HOME%/bin	Add the bin directory of the Ant installation to the system path so that the ant command can be executed in any directory. Note that %PATH%; in the “Set to” column means that the old value should not be overwritten, but instead be appended to.
PATH	%PATH%;%JAVA_HOME%/bin	Add the bin directory of the JDK installation to the system path. Note that %PATH%; in the “Set to” column means that the old value should not be overwritten, but instead be appended to.

Table 1 Environment variables.

2.3.2 How to set a OS environment variable

This section describes the steps for setting an environment variable in Windows.

1 Open the system properties dialog. This is done by opening the Control Panel and double-clicking the System icon.

2 Choose the Advanced tab.

3 Click Environment Variables to edit the systems environment variables.

4 In the list, choose the variable to add/change and click New/Edit.

5 Enter the new variable value and click OK.

Adding directories to the `%PATH%` environment variable is done by appending a new entry to the existing one. For example, the new result can be:

```
%JAVA_HOME\bin;%PATH%
```

Figure 2 shows an example of what this can look like.

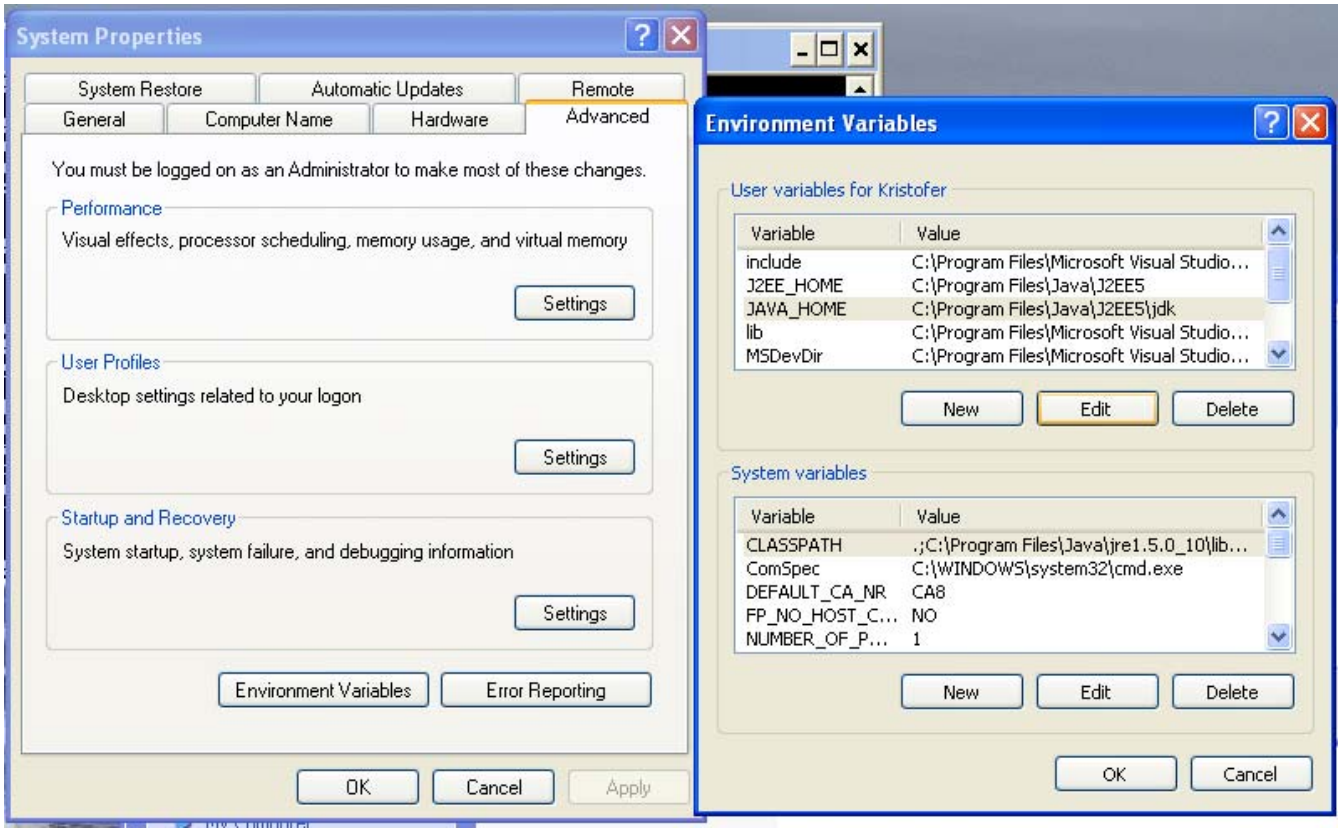

Figure 2 Setting environment variables in Windows.

3 User guide

This chapter describes how to build and use MFC. All of the installation and configuration in chapter 2 are assumed to have been performed.

3.1 Build MFC

The following steps are needed to build MFC:

1 Open a command prompt and change to the `<mfc>` directory.
2 Build the MFC library (.jar) by running `ant build`. The generated file is located in `<mfc>/dist`.

3.2 Use MFC in a web application

To use the MFC library in a web application, the generated library from `<mfc>/dist` needs to be copied to the library directory for the web application. For example, to `<application>/WEB-INF/lib`, where `<application>` is the root directory of the web application.

The web application needs to be configured, which is done by modifying `<application>/WEB-INF/web.xml`. The MFC servlet has to be added to the descriptor, shown in Figure 3. `*.do` can be changed to other URL patterns.

```
<servlet>
    <servlet-name>ControllerServlet</servlet-name>
    <servlet-
class>webframework.mobilefrontcontroller.controller.ControllerServlet</servlet-
class>
    <load-on-startup>3</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>ControllerServlet</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

Figure 3 Configuring the MFC servlet in web.xml.

3.3 Views and view handlers

The following sections describe views and view handlers, which are a part of the MFC library. There is also a section that describes how to create custom view handlers.

3.3.1 Views

A view is a subdirectory in a web application that is represented by the `View` class, which is shown in Figure 4. A `View` object is created by passing a directory path to its constructor. The `directoryPath` is the relative path from the application context root to the directory for the view.

```
package webframework.mobilefrontcontroller.views;

public final class View {
  private final String directoryPath;

  public View(String directoryPath) {
    if (directoryPath == null) {
      throw new IllegalArgumentException("null is not a valid value for a
directory!");
    }

    // Remove "/" in the beginning
    while (directoryPath.startsWith("/")) {
      directoryPath = directoryPath.substring(1);
    }
    // Remove "/" in the end
    while (directoryPath.endsWith("/")) {
      directoryPath = directoryPath.substring (0, directoryPath.length()-1);
    }
    this.directoryPath = directoryPath;
  }

  public String getDirectoryPath() {
    return directoryPath;
  }
}
```

Figure 4 The View class that represents a view.

3.3.2 View handlers

A view handler is a class that implements the `ViewHandler` interface (see Figure 5) and is used by MFC to detect and select a view. Section 3.5.3 describes how to configure the view handler that MFC should use.

```
public interface ViewHandler {
    void init(ServletConfig servletConfig);
    boolean isViewSelected(HttpServletRequest request);
    View selectView(HttpServletRequest request);
}
```

Figure 5 The ViewHandler interface that a view handler implements.

`init` is called during the MFC servlet initialization. `isViewSelected` is called before an action command is executed to check whether the request comes from a view or not. `selectView` is called to detect and return a view.

3.3.3 Default view handler

MFC contains a default view handler, `DefaultViewHandler` (see Figure 6), that has three different views: XHTML, XHTML-MP and WML. The views correspond to the following directories: `xhtml`, `xhtmlmp` and `wml`. The default view handler's `isViewSelected` returns `true` if the request comes from a subdirectory. `selectView` returns one of the following views:

- XHTML-MP view, if the `x-wap-profile` header exists and the `accept` header contains `xhtml+xml`.
- WML view, if the `x-wap-profile` header exists and the `accept` header contains `text/vnd.wap.wml`.
- XHTML view, if not one of the two views above is selected.

```
package webframework.mobilefrontcontroller.views;

import javax.servlet.ServletConfig;
import javax.servlet.http.HttpServletRequest;

public class DefaultViewHandler implements ViewHandler {
  protected View xhtml = null;
  protected View xhtmlmp = null;
  protected View wml = null;

  public void init(ServletConfig servletConfig) {
    xhtml = new View("xhtml");
    xhtmlmp = new View("xhtmlmp");
    wml = new View("wml");
  }

  public boolean isViewSelected(HttpServletRequest request) {
    String servletPath = request.getServletPath();
    return servletPath.substring(1).contains("/");
  }

  public View selectView(HttpServletRequest request) {
    if (request.getHeader("x-wap-profile") != null) {
      String accept = request.getHeader("accept");
      accept = accept != null ? accept : ""; // make sure that accept isn't null
      // XHTML-MP
      if (accept.indexOf("xhtml+xml") != -1) {
        return xhtmlmp;
      }
      // WML
      if (accept.indexOf("text/vnd.wap.wml") != -1) {
        return wml;
      }
    }
    // Default: XHTML
    return xhtml;
  }
}
```

Figure 6 The default view handler.

3.3.4 Create custom view handlers

To create a view handler that works in the same way as the default view handler but target other directories, the `DefaultViewHandler` class can be extended by overriding the `init` method, as shown in Figure 7.

```
package examples.mobilefrontcontroller.views;

import javax.servlet.ServletConfig;

public class CustomViewHandler extends DefaultViewHandler {
  @Override
  public void init(ServletConfig servletConfig) {
    // Set new directory paths (web, mobile, wap) for the views
    xhtml = new View("web");
    xhtmlmp = new View("mobile");
    wml = new View("wap");
  }
}
```

Figure 7 An example of a custom view handler.

To create a view handler that extends the default view handler, for example, by also detecting devices with smaller screen sizes, the WURFL library can be used. WURFL is an XML configuration file that contains information about capabilities and features of many mobile devices. The WURFL library can be downloaded from http://wurfl.sourceforge.net/. Figure 8 shows an example of a view handler that uses the WURFL library.

```
package examples.mobilefrontcontroller.views;

import javax.servlet.ServletConfig;
import javax.servlet.http.HttpServletRequest;
import net.sourceforge.wurfl.wurflapi.CapabilityMatrix;
import net.sourceforge.wurfl.wurflapi.ObjectsManager;
import net.sourceforge.wurfl.wurflapi.UAManager;
import webframework.mobilefrontcontroller.views.DefaultViewHandler;
import webframework.mobilefrontcontroller.views.View;

public final class WurflViewHandler extends DefaultViewHandler {
  private UAManager uam;
  private CapabilityMatrix cm;
  private View xhtmlmpsmall;

  @Override
  public void init(ServletConfig config) {
    // View for devices with smaller screens
    xhtmlmpsmall = new View("xhtmlmpsmall");

    ObjectsManager.getWurflInstance(
                    config.getServletContext().getRealPath("/") + "wurfl.xml");
    uam = ObjectsManager.getUAManagerInstance();
    cm = ObjectsManager.getCapabilityMatrixInstance();
  }

  @Override
  public View selectView(HttpServletRequest request) {
    if (request.getHeader("x-wap-profile") != null) {
      String accept = request.getHeader("accept");
      accept = accept != null ? accept : ""; // make sure that accept isn't null
      // XHTML-MP
      if (accept.indexOf("xhtml+xml") != -1) {
        String deviceId = uam.getDeviceIDFromUALoose(
                                              request.getHeader("user-agent"));
        int resolutionWidth = Integer.parseInt(
                      cm.getCapabilityForDevice(deviceId, "resolution_width"));
        if (resolutionWidth <= 120) {
          // width < 120
          return xhtmlmpsmall;
        } else {
          // default
          return xhtmlmp;
        }
      }
      // WML
      if (accept.indexOf("text/vnd.wap.wml") != -1) {
        return wml;
      }
    }
    // Default: XHTML
    return xhtml;
  }
}
```

Figure 8 An example of a custom view handler that uses the WURFL library.

3.4 Action commands

An action command is a class that implements the `webframework.mobilefrontcontroller.actions.ActionCommand` interface, shown in Figure 9.

```
public interface ActionCommand  {
    public String execute(HttpServletRequest request,
                          HttpServletResponse response);
}
```

Figure 9 The ActionCommand interface that all action commands implement.

An example of a URL to an action command is
`http://localhost:8080/application/xhtml/Test.do`.

The **`execute`** method is executed by MFC when calling the action command and thereafter the next view is shown. There are three ways to set the next view for an action command:

- Define the return value of the **`execute`** method to be a view page, for example, **`resultpage.jsp`**, as shown in Figure 10. MFC dispatches the view to this page by doing a forward. The dispatch type can be changed, as described in section 3.4.1.

```
package webframework.mobilefrontcontroller.actions;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class Test implements ActionCommand {
    public String execute(HttpServletRequest request, HttpServletResponse
response) {
        return "resultpage.jsp";
    }
}
```

Figure 10 A basic action command that returns the next view.

- Define the return value of the **`execute`** method to `null`. MFC does not dispatch to any view page, which means that the action command is responsible for changing the `HttpServletResponse` object. An example of this is shown in Figure 11.

```
package examples.mobilefrontcontroller.actions;

import webframework.mobilefrontcontroller.actions.ActionCommand;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class Output implements ActionCommand {
  public String execute(HttpServletRequest request,HttpServletResponse response) {
    PrintWriter writer = null;
      try {
        response.setContentType("text/plain");
        writer = response.getWriter();
        writer.println("Hello world!");
        writer.flush();
      } catch (IOException ex) {
          Logger.getLogger(getClass().getName()).log(Level.SEVERE, null, ex);
      } finally {
        if (writer != null)
          writer.close();
      }
      return null;
  }
}
```

Figure 11 A basic action command that returns null and outputs text using the response object.

- Define the request parameter **`nextpage`** when calling the action command, for example,
  `http://localhost:8080/application/xhtml/Test.do?`**`nextpage`**`=anotherpage.jsp`. The request parameter overrides the return value of the **`execute`** method.

3.4.1 Dispatch types

An action command can also implement the `webframework.mobilefrontcontroller.actions.DispatchActionCommand` interface (see Figure 12) that allows the dispatch type for an action command to be customized. The available dispatch types, which are listed in the `webframework.mobilefrontcontroller.actions.DispatchType` enum (see Figure 13), are:

- `FORWARD` (default):
  HTTP forward to the return value of the action command.
- `REDIRECT`:
  HTTP redirect to the return value of the action command.

Note: See chapter 5 for definitions of HTTP forward and HTTP redirect.

```
public interface DispatchActionCommand {
    public DispatchType getDispatchType();
}
```

Figure 12 The DispatchActionCommand interface is used to customize the dispatch type.

```
public enum DispatchType {
    FORWARD,
    REDIRECT;
}
```

Figure 13 The DispatchType enum lists the available dispatch types.

Figure 14 shows a basic action command that implements both the `ActionCommand` and `DispatchActionCommand` interfaces.

```
package examples.mobilefrontcontroller.actions;

import webframework.mobilefrontcontroller.actions.ActionCommand;
import webframework.mobilefrontcontroller.actions.DispatchActionCommand;
import webframework.mobilefrontcontroller.actions.DispatchType;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class Test implements ActionCommand, DispatchActionCommand {
    public String execute(HttpServletRequest request, HttpServletResponse
response) {
        return "resultpage.jsp";
    }

    public DispatchType getDispatchType() {
        return DispatchType.REDIRECT;
    }
}
```

Figure 14 A basic action command that does an HTTP redirect.

3.5 Configure MFC

The MFC configuration consists of three parts: action command mappings, action command package paths and parameters. These parts have default values that in most cases do not need any modifications. The default configuration for MFC is found in `<mfc>/config/mobilefrontcontroller.xml`, partly shown in Figure 15. The configuration file, together with its schema, is packaged into the MFC library when it is built.

```
<mobile-front-controller
  xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
  xsi:noNamespaceSchemaLocation="mobilefrontcontroller_1_0.xsd">
   <!-- Action commands mapped from name to class name -->
   <!--
   <action-command-mapping>
      <action-name>Test</action-name>
      <action-class>examples.mobilefrontcontroller.actions.Test</action-class>
   <action-command-mapping>
      <action-name>Print</action-name>
      <action-class>examples.mobilefrontcontroller.actions.Print</action-class>
   </action-command-mapping>
   -->

   <!—Action command package paths, for where to look up other ActionCommands. -->
   <!--
<action-command-package-path>examples.mobilefrontcontroller.actions</action-
command-package-path>
   <action-command-package-path>examples.otherpackage.actions</action-command-
package-path>
   -->

   <!-- ControllerServlet parameters -->
    <parameters>
        <!-- Specifies a custom view handler. If not defined, the default view
handler is used. -->
        <!--
        <view-
handler>examples.mobilefrontcontroller.views.CustomViewHandler</view-handler>
        -->
    </parameters>
</mobile-front-controller>
```

Figure 15 The default configuration for MFC.

The different kinds of configuration parameters are explained in section 3.5.1, 3.5.2 and 3.5.3. The default configuration can and should be overridden, as described in section 3.5.4

3.5.1 Specific action command mapping

Action command mapping is the part where action command names can be mapped against specific class names. For example:

Assume the following configuration:

```
<action-command-mapping>
  <action-name>Print</action-name>
  <action-class>example.specific.actions.Print</action-class>
</action-command-mapping>
```

When the action, for example, `http://localhost:8080/mfc-basic-demo/xhtml/Print.do` is called, the `execute` method in the `example.specific.actions.Print` class is executed.

Note: The default configuration defines no action command mappings.

3.5.2 Action command package path

Action command package path is the part where Java package paths can be listed. When an action command is executed and no corresponding action command mapping exists, MFC tries to find a class within each of the defined package paths that has the same name as the action command. For example,

Assume the following configuration:
`<action-command-package-path>example.default.actions</action-command-package-path>`

When the URL, for example, `http://localhost:8080/mfc-basic-demo/xhtml/Test.do` is called, the `execute` method in the `example.default.actions.Test` class is executed.

Note: The default configuration defines no package paths.

Note: By adding `<action-command-package-path></action-command-package-path>` to the configuration file, makes MFC look for action commands in the default (empty) package.

Note: If neither action command mappings nor action command package paths have been defined, MFC tries to load action commands without using a package path.

3.5.3 General parameters

The general parameters are:

- `view-handler`:
  The class name of a custom view handler that should be used by MFC. If this parameter is not defined, MFC uses the default view handler (see section 3.3).

  Example:

```
...
<parameters>
  <view-handler>examples.views.CustomViewHandler</view-handler>
</parameters>
...
```

3.5.4 Override configuration

The default configuration can be overridden by creating a customized configuration file, `mobilefrontcontroller.xml`, into the `WEB-INF` directory in the application, for example, `<application>/WEB-INF/`.

All of the settings in the configuration are optional and therefore it is not necessary for a customized configuration file to define all of the possible values.

3.6 Development tools

Besides containing a light-weight framework, MFC also provides a set of tools that can be used when developing web applications. The tools are:

- A filter that traces HTTP requests.
- A listener for logging web context events.
- A listener for logging session events.
- A listener for logging request events.

To use the tools, the following configuration is needed in the deployment descriptor of the web application (`<application>/WEB-INF/web.xml`):

```
<filter>
  <filter-name>HTTP request filter</filter-name>
  <filter-class>
    webframework.mobilefrontcontroller.tools.HTTPRequestFilter</filter-class>
  <init-param>
    <param-name>useFilter</param-name>
     <param-value>true</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>HTTP request filter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Figure 16 Configuration for the HTTP request filter.

```
<listener>
  <description>Web context logger listener</description>
  <listener-class>
   webframework.mobilefrontcontroller.tools.WebContextLoggerListener</listener-
class>
</listener>
```

Figure 17 Configuration for the web context listener.

```
<listener>
  <description>HTTP session logger listener</description>
  <listener-class>
    webframework.mobilefrontcontroller.tools.SessionLoggerListener</listener-
class>
</listener>
```

Figure 18 Configuration for the HTTP session listener.

```
<listener>
  <description>Request logger listener</description>
  <listener-class>
    webframework.mobilefrontcontroller.tools.RequestLoggerListener</listener-
class>
</listener>
```

Figure 19 Configuration for the request listener.

4 Design guide

MFC consists of the following parts:

- The controller servlet that is responsible for receiving HTTP requests, detecting appropriate views, executing action commands and dispatching to next views. The servlet is described in section 4.1
- The interfaces and the enum that make up the base for action commands. They are described in section 3.4.
- The class and the enum that are responsible for the configuration of MFC. They are described in section 3.5 and section 4.2.
- An overview of MFC and its parts is shown in Figure 20.

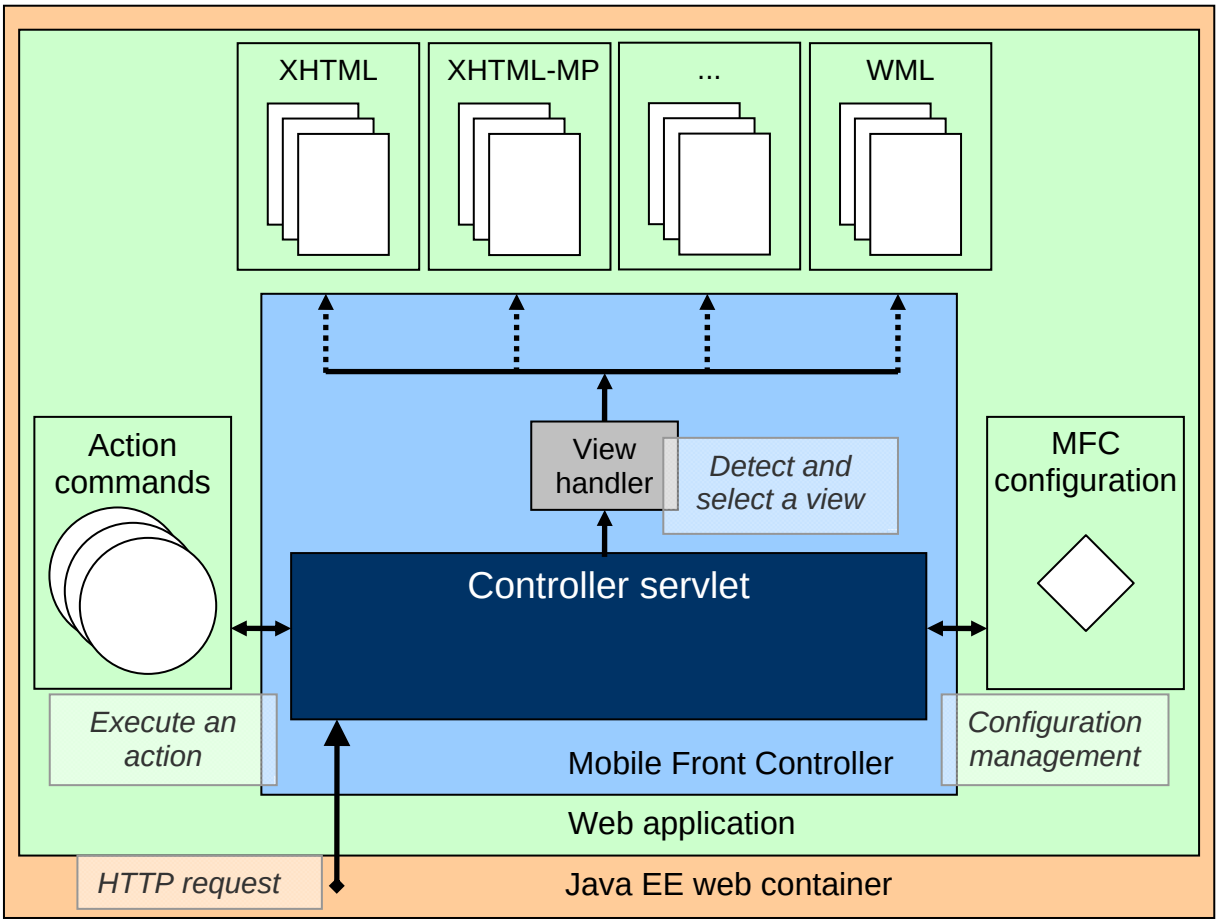

Figure 20 An overview of Mobile Front Controller that is used by a web application on a Java EE web container.

4.1 Controller servlet

The controller servlet handles action command calls, which are HTTP requests with, for example, the URL pattern `*.do` (assuming the servlet is correctly configured, as shown in Figure 3). The controller servlet class, `ControllerServlet`, is located in the following directory `<mfc>/src/webframework/mobilefrontcontroller/controller/`.

`ControllerServlet`, extends `javax.servlet.http.HttpServlet` and overrides the `init`, `doGet`, and `doPost` methods.

4.1.1 Servlet initialization

The `init` method initializes the servlet, loads MFC configuration, creates an instance of a view handler, and initializes the view handler as shown in Figure 21.

```
@Override
public void init(ServletConfig servletConfig) throws ServletException {
  super.init(servletConfig);
  loadConfiguration();

  // Load and create a view handler
  String viewHandlerClassName =
         configuration.getParameterValue(Parameter.VIEW_HANDLER_CLASS);
  viewHandler = createViewHandler(viewHandlerClassName);
  log("The view handler " +
      viewHandler.getClass().getName() + " was created successfully.");
  viewHandler.init(servletConfig);
}
```

Figure 21 Initialization of the controller servlet.

`loadConfiguration`, which is shown in Figure 22, loads configuration for MFC. The default configuration file, which is packaged into the MFC library, is loaded at first. Thereafter, MFC tries to load a customized configuration file found in the `WEB-INF` directory of the web application. If the customized configuration file is successfully loaded, the default configuration is overridden with the customized one.

```
private void loadConfiguration() {
  String filename = CONTROLLER_SERVLET_CONFIGURATION_FILE;
  try {
    // Loads configuration file that is located inside the jar file.
    configuration = new Configuration(
                      getClass().getClassLoader().getResourceAsStream(filename));
    log("Default configuration loaded from " + filename + " in library file.");
  } catch ( //... more code

  }

  // Loads configuration file from /WEB-INF/ and tries to override settings
  Configuration customConfiguration = null;
  filename = "WEB-INF/" + CONTROLLER_SERVLET_CONFIGURATION_FILE;
  File file = new File(getServletContext().getRealPath(filename));
  if (file.exists()) {
    try {
      customConfiguration = new Configuration(file);
      log("Custom configuration loaded from " + filename + ".");
    } catch ( //... more code

    }

    if (configuration != null) {
      configuration.overrideWith(customConfiguration);
      log("Default configuration overrided with custom configuration.");
    } else {
      configuration = customConfiguration;
    }
  }

  if (configuration != null) {
    if (configuration.getActionCommandMappings().isEmpty() &&
        configuration.getActionCommandPackagePaths().isEmpty()) {
      log("No action command mapping or action command package path exist in " +
          CONTROLLER_SERVLET_CONFIGURATION_FILE +
          "! Default package path is used for loading action commands.");
    }
  } else {
    log("Neither a default or a custom configuration has been loaded.");
    throw new IllegalStateException("Neither a default or a custom configuration
has been loaded.");
  }
}
```

Figure 22 Loads configuration for Mobile Front Controller.

4.1.2 Request processing

The method **`processRequest`**, which is shown in Figure 23, handles HTTP GET and POST requests that the servlet is responsible for. The method does the following:

1 Checks if the request if made from a view by using the `isViewSelected` method of the view handler.

2 If the request is not made from a view, the **`selectView`** method of the view handler is called to detect and return a view. A redirect is made to the view.

3 Invokes an action command by calling the **`invoke`** method, shown in Figure 24.

4 The **`nextViewPage`** variable gets the value that is returned by the `invoke` method and tells the servlet of the next view to dispatch to.

5 If the request parameter **`nextpage`** is set, **`nextViewPage`** is overridden by **`nextpage`**.

6 Finally, the dispatch type, which is determined by the **invoke** method, is used to decide how to dispatch to the next view.

```
private void processRequest(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
  // If a view is not selected, select a view.
  String nextpageParameter = request.getParameter("nextpage");
  if (!viewHandler.isViewSelected(request)) {
    View view = viewHandler.selectView(request);
    if (view == null)
      return;
    if (nextpageParameter != null) {
      response.sendRedirect(request.getContextPath() + "/" +
                            view.getDirectoryPath() + "/" + nextpageParameter);
    } else {
      response.sendRedirect(request.getContextPath() + "/" +
                            view.getDirectoryPath());
    }
    return;
  }

  // Invoke ActionCommand (and set the dispatch type)
  // nextViewPage is the page to dispatch to
  ActionCommandResult result = null;
  String nextViewPage = null;
  try {
    result = invoke(request, response);
    nextViewPage = result.getNextViewPage();
  } catch (ClassNotFoundException ex) {
    log(null, ex);
  } catch (InstantiationException ex) {
    log(null, ex);
  } catch (IllegalAccessException ex) {
    log(null, ex);
  }

  // If nextpage parameter was specified, it overrides the result page that
  // is provided by the ActionCommand or interpreted from the request
  if (nextpageParameter != null) {
    nextViewPage = nextpageParameter;
  }

  if (nextViewPage != null) {
    // Forward or redirect based on the
    // response type specified by the ActionCommand (default is to forward).
    switch (result.getDispatchType()) {
      default:
      case FORWARD:
        // Remove all subdirectories when doing a forward
        nextViewPage = removeSubdirectories(nextViewPage);
        forward(nextViewPage, request, response);
        break;
      case REDIRECT:
        response.sendRedirect(nextViewPage);
        break;
    }
  } else {
    // Do nothing. The action command is responsible for setting the response.
  }
}
```

Figure 23 Processes HTTP GET and POST requests.

The method **invoke** uses the **getActionCommand** method (see Figure 25) to create an action command. The `execute` method of the action command is executed and finally, the method returns the result in an `ActionCommandResult` class (see Figure 26).

`getActionCommand` is responsible for finding an action command class and creating a new instance of it. The order to find the action command class is:

1 Search for the action command name among the configured action command mappings (if there are any).
2 If the action command is not found, try to look up the class using the configured action command package paths.

```
private ActionCommandResult invoke(HttpServletRequest request,
                                   HttpServletResponse response)
   throws ClassNotFoundException, InstantiationException, IllegalAccessException {
  ActionCommand actionCommand = getActionCommand(request);
  DispatchType dispatchType = DispatchType.FORWARD; // Default value;
  if (actionCommand instanceof DispatchActionCommand) {
    dispatchType = ((DispatchActionCommand) actionCommand).getDispatchType();
    // Make sure that the dispatch type is set
    if (dispatchType == null) {
      log("The dispatch type in action command " +
          actionCommand.getClass().getName() +
          " cannot be null! It defaults to FORWARD.");
      dispatchType = DispatchType.FORWARD;
    }
  }

  return new ActionCommandResult(actionCommand.execute(request, response),
                                 dispatchType);
}
```

Figure 24 Invokes an action command.

```
private ActionCommand getActionCommand(HttpServletRequest req) throws
  ClassNotFoundException, InstantiationException, IllegalAccessException {
  String servletPath = req.getServletPath();
  int classNameStartIndex = servletPath.lastIndexOf("/") +1;
  int classNameEndIndex = servletPath.lastIndexOf(".");
  String actionName = null;
  if (classNameEndIndex > classNameStartIndex)
    actionName = servletPath.substring(classNameStartIndex, classNameEndIndex);
  else
    actionName = servletPath.substring(classNameStartIndex);

  Class<?> actionClass = null;
  if (!configuration.getActionCommandMappings().isEmpty() ||
      !configuration.getActionCommandPackagePaths().isEmpty()) {
    // Get class name if it is mapped against an action name.
    className = configuration.getActionCommand(actionName);

    if (className != null) {
      try {
        actionClass = Class.forName(className);
      } catch (ClassNotFoundException e) {
        throw new ClassNotFoundException("The action class " +
                                         className + " mapped to action name " +
                                         actionName + " could not be loaded.");
      }
    } else if (!configuration.getActionCommandPackagePaths().isEmpty()) {
      // If it's not mapped, try to load the class using package paths.
      for (String packagePath : configuration.getActionCommandPackagePaths()) {
        try {
          packagePath = packagePath.trim();
          if (!packagePath.equals(""))
            className = packagePath + "." + actionName;
          else
            className = actionName;
          actionClass = Class.forName(className);
          break;
        } catch (ClassNotFoundException e) {
          // Don't throw an exception or log since there might be several package
paths where the action command can reside in.
        }
      }
    }
  } else {
    // Use default package path if action command mappings or action command
package paths were not specified.
    try {
      actionClass = Class.forName(className);
    } catch (ClassNotFoundException e) {
      throw new ClassNotFoundException("The action class " + className +
                                       " was not found default package path.");
    }
  }

  // If the class could not be loaded, throw an exception!
  if (actionClass == null) {
    throw new ClassNotFoundException("Could not load action command " +
                                     actionName + "!");
  }

  return (ActionCommand)actionClass.newInstance();
}
```

Figure 25 Returns an action command.

```
private final class ActionCommandResult {
  private final String nextViewPage;
  private final DispatchType dispatchType;

  public ActionCommandResult(String nextViewPage, DispatchType dispatchType) {
    this.nextViewPage = nextViewPage;
    if (dispatchType != null)
      this.dispatchType = dispatchType;
    else
      this.dispatchType = DispatchType.FORWARD;
  }

  public String getNextViewPage() { return nextViewPage; }
  public DispatchType getDispatchType() { return dispatchType; }
}
```

Figure 26 The class that holds values set by the invoke method.

4.2 MFC configuration

The `Configuration` class is responsible for loading and managing configuration data. The class has two constructors: one that takes a `File` as an argument and another that takes an `InputStream`, shown in Figure 27. The first constructor is used when loading the customized configuration file and the other one is used to load the default configuration file, which is at runtime located in a .jar file.

```
public Configuration(File file) throws ParserConfigurationException, SAXException,
IOException, ConfigurationException {
    DocumentBuilder builder =
                        DocumentBuilderFactory.newInstance().newDocumentBuilder();
    Document doc = builder.parse(file);
    loadConfiguration(doc);
}

public Configuration(InputStream stream) throws ParserConfigurationException,
SAXException, IOException, ConfigurationException {
    DocumentBuilder builder =
                        DocumentBuilderFactory.newInstance().newDocumentBuilder();
    Document doc = builder.parse(stream);
    loadConfiguration(doc);
}
```

Figure 27 Constructors for the Configuration class, which are used to load configuration.

The constructors call the **`loadConfiguration`** method, shown in Figure 28. Document Object Model (DOM) is used to parse the configuration files.

```
private void loadConfiguration(Document doc) throws ConfigurationException {
    // Get the root element
    Element root =
           (Element) doc.getElementsByTagName("mobile-front-controller").item(0);
    if (root == null)
      throw new ConfigurationException("Root element mobile-front-controller was
not found!");

    // Get all specified ActionsCommands
    NodeList nodes = root.getElementsByTagName("action-command-mapping");
    for (int i = 0; i < nodes.getLength(); i++) {
      Element element = (Element) nodes.item(i);
      if (element != null) {
        // Insert <ActionCommand name, Class name>
        actionCommandMappings.put(
            getCharacterDataFromElement(
               (Element) element.getElementsByTagName("action-name").item(0)),
            getCharacterDataFromElement(
               (Element) element.getElementsByTagName("action-class").item(0)));
      }
    }

    // Get all specified packages actions commands
    nodes = root.getElementsByTagName("action-command-package-path");
    for (int i = 0; i < nodes.getLength(); i++) {
      // Insert <Package name>
      actionCommandPackagePaths.add(
              getCharacterDataFromElement((Element) nodes.item(i)));
    }

    // Get parameters for Front Controller
    Element element = (Element) root.getElementsByTagName("parameters").item(0);
    if (element != null) {
      for (Parameter parameter : Parameter.values()) {
        Element param =
              (Element) element.getElementsByTagName(parameter.getName()).item(0);
        if (param != null) {
          String paramValue = getCharacterDataFromElement(param);
          if (!paramValue.equals("")) {
            parameters.put(parameter, paramValue);
          }
        }
      }
    }
}
```

Figure 28 The loadConfiguration method that loads configuration data.

One instance of the Configuration class can override another one. This is done by calling the **`overrideWith`** method, shown in Figure 29.

```
public void overrideWith(Configuration configuration) {
  if (configuration != null) {
    // Override action command mappings
    if (!configuration.getActionCommandMappings().isEmpty()) {
      actionCommandMappings.clear(); // Remove existing mappings first!
      for (String actionCommandName :
           configuration.getActionCommandMappings().keySet()) {
        actionCommandMappings.put(actionCommandName,
                                  configuration.getActionCommand(actionCommandName));
      }
    }

    // Override action command package paths
    if (!configuration.getActionCommandPackagePaths().isEmpty()) {
      actionCommandPackagePaths.clear(); // Remove existing packages first!
      for (String packageName : configuration.getActionCommandPackagePaths()) {
        actionCommandPackagePaths.add(packageName);
      }
    }

    // Override configuration parameters
    for (Parameter parameter : Parameter.values()) {
      String paramValue = configuration.getParameterValue(parameter);
      if (paramValue != null && !paramValue.equals(""))
        parameters.put(parameter, paramValue);
    }
  }
}
```

Figure 29 The overrideWith method that allows one configuration to be overridden by another one.

4.3 Development tools

As described in section 3.6, there are tools included in MFC that can be useful when developing web applications. Figure 30 shows parts of the HTTP request filter.

```
//… more code here …
public final class HTTPRequestFilter implements Filter {
    private boolean useFilter = false;
    //… more code here …

    public void doFilter(ServletRequest request, ServletResponse response,
FilterChain chain) throws IOException, ServletException {
        // Only filter if the request is a HttpServletRequest.
        if (!(request instanceof HttpServletRequest) || !useFilter) {
            chain.doFilter(request, response);
            return;
        }

        HttpServletRequest httpRequest = (HttpServletRequest)request;
        String newLine = System.getProperty("line.separator");
        StringBuilder information = new StringBuilder();

        information.append(newLine);
        information.append("--- START HTTP request headers ---");
        information.append(newLine);
        information.append("getRemoteHost(): ");
        information.append(httpRequest.getRemoteHost());

        //… more code here …

        // Header names are Strings so cast warning can be suppressed.
        @SuppressWarnings("unchecked")
        Enumeration<String> requestHeaderNames =
                        (Enumeration<String>)httpRequest.getHeaderNames();
        if (requestHeaderNames != null) {
            while (requestHeaderNames.hasMoreElements()) {
                String headerName = requestHeaderNames.nextElement();
                // Headers are Objects so cast warning can be suppressed
                @SuppressWarnings("unchecked")
                Enumeration<Object> headers =
                          (Enumeration<Object>)httpRequest.getHeaders(headerName);
                if (headers != null) {
                    while (headers.hasMoreElements()) {
                        information.append(newLine);
                        Object header = headers.nextElement();
                        information.append(headerName);
                        information.append(": ");
                        information.append(header);
                    }
                }
            }
        }
        information.append(newLine);
        information.append("--- END HTTP request headers ---");
        information.append(newLine);

        // Log headers

Logger.getLogger(getClass().getName()).log(Level.INFO,information.toString());

        // Continue filter chain
        chain.doFilter(request, response);
    }

    //… more code here …

    public boolean isUseFilter() { return useFilter; }
    public void setUseFilter(boolean useFilter) { this.useFilter = useFilter; }
}
```

Figure 30 Parts of the HTTP request filter.

The filter implements the `javax.servlet.Filter` interface and logs many of its properties.

Parts of the web context listener are shown in Figure 31, the session listener in Figure 32 and the request listener in Figure 33.

```
//… more code here …
public final class WebContextLoggerListener implements ServletContextListener,
ServletContextAttributeListener {
    private final static String newLine = System.getProperty("line.separator");

    public void contextInitialized(ServletContextEvent event) {
        StringBuilder builder = new StringBuilder("contextInitialized: ");
        builder.append(newLine);
        builder.append(" getContextPath: ");
        builder.append(event.getServletContext().getContextPath());
        builder.append(newLine);
        builder.append(" getServletContextName: ");
        builder.append(event.getServletContext().getServletContextName());

        Logger.getLogger(getClass().getName()).log(Level.INFO,
builder.toString());
    }

    //… more code here …

    public void attributeAdded(ServletContextAttributeEvent event) {
        StringBuilder builder = new StringBuilder("attributeAdded: ");
        builder.append(newLine);
        builder.append(" getName: ");
        builder.append(event.getName());
        builder.append(newLine);
        builder.append(" getValue: ");
        builder.append(event.getValue());

        Logger.getLogger(getClass().getName()).log(Level.INFO,
builder.toString());
    }

    //… more code here …
}
```

Figure 31 Parts of the web context listener.

```
//… more code here …
public final class SessionLoggerListener implements HttpSessionListener,
HttpSessionActivationListener, HttpSessionAttributeListener {
    private final static String newLine = System.getProperty("line.separator");

    //… more code here …

    public void sessionCreated(HttpSessionEvent event) {
        StringBuilder builder = new StringBuilder("sessionCreated: ");
        builder.append(newLine);
        builder.append(" getId: ");
        builder.append(event.getSession().getId());

        Logger.getLogger(getClass().getName()).log(Level.INFO,
builder.toString());
    }

    //… more code here …

    public void sessionWillPassivate(HttpSessionEvent event) {
        StringBuilder builder = new StringBuilder("sessionWillPassivate: ");
        builder.append(newLine);
        builder.append(" getId: ");
        builder.append(event.getSession().getId());

        Logger.getLogger(getClass().getName()).log(Level.INFO,
builder.toString());
    }

    //… more code here …

    public void attributeAdded(HttpSessionBindingEvent event) {
        StringBuilder builder = new StringBuilder("attributeAdded: ");
        builder.append(newLine);
        builder.append(" getId: ");
        builder.append(event.getSession().getId());
        builder.append(newLine);
        builder.append(" getName: ");
        builder.append(event.getName());
        builder.append(newLine);
        builder.append(" getValue: ");
        builder.append(event.getValue());

        Logger.getLogger(getClass().getName()).log(Level.INFO,
builder.toString());
    }

    //… more code here …
}
```

Figure 32 Parts of the session listener.

```
//… more code here …
public final class RequestLoggerListener implements ServletRequestListener,
ServletRequestAttributeListener {
    private final static String newLine = System.getProperty("line.separator");
    private static int requestCount = 0;

    public void requestInitialized(ServletRequestEvent event) {
        requestCount = (requestCount % Integer.MAX_VALUE) + 1;
        StringBuilder builder = new StringBuilder("requestInitialized: ");
        builder.append(newLine);
        builder.append(" request count: ");
        builder.append(requestCount);

        ServletRequest request = event.getServletRequest();
        if (request instanceof HttpServletRequest) {
            builder.append(newLine);
            HttpServletRequest httpRequest = (HttpServletRequest)request;
            builder.append(" getRequestURL: ");
            builder.append(httpRequest.getRequestURL().toString());
        }

        // According to documentation, the key set is of type String
        @SuppressWarnings("unchecked")
        Set<String> paramSet = (Set<String>)request.getParameterMap().keySet();
        if (!paramSet.isEmpty()) {
            builder.append(newLine);
            builder.append(" getParameter...: ");
            for (String param : paramSet) {
                builder.append(newLine);
                builder.append("  ");
                builder.append(param);
                builder.append(": [ ");
                for (String value : request.getParameterValues(param)) {
                    builder.append(value);
                    builder.append(", ");
                }
                builder.append(" ]");
            }
        }

        Logger.getLogger(getClass().getName()).log(Level.INFO,
builder.toString());
    }

    //… more code here …

    public void attributeAdded(ServletRequestAttributeEvent event) {
        StringBuilder builder = new StringBuilder("attributeAdded: ");
        builder.append(newLine);
        builder.append(" getName: ");
        builder.append(event.getName());
        builder.append(newLine);
        builder.append(" getValue: ");
        builder.append(event.getValue());

        Logger.getLogger(getClass().getName()).log(Level.INFO,
builder.toString());
    }

    //… more code here …
}
```

Figure 33 Parts of the request listener.

5 Terminology

HTTP forward is the type of action that a servlet can perform at the end of processing a request. A forward is performed internally by the application (servlet). The browser is unaware that this has taken place and the URL remains intact.

HTTP redirect is the type of action that a servlet can perform at the end of processing a request. A redirect is a two step process, where the application instructs the browser to fetch a second URL, which differs from the original. A browser reload of the second URL does not repeat the original request, only the second one. Objects placed in the original request scope are not available to the second request.

WML (Wireless Markup Language) is a markup language based on XML for mobile devices. WML is a predecessor to other markup languages, such as XHTML-MP.

XHTML (Extensible Hypertext Markup Language) is a markup language that is similar to HTML. One of the differences to HTML is that XHTML conforms to XML syntax.

XHTML-MP (XHTML Mobile Profile) is a markup language designed for mobile phones and other resource-constrained devices.