Beginning Java MVC 1.0

Model View Controller Development to Build Web, Cloud, and Microservices Applications

Peter Späth

Apress®

Beginning Java MVC 1.0

Model View Controller Development to Build Web, Cloud, and Microservices Applications

Peter Späth

Apress®

Beginning Java MVC 1.0: Model View Controller Development to Build Web, Cloud, and Microservices Applications

Peter Späth
Leipzig, Sachsen, Germany

ISBN-13 (pbk): 978-1-4842-6279-5 ISBN-13 (electronic): 978-1-4842-6280-1
https://doi.org/10.1007/978-1-4842-6280-1

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Steve Anglin
Development Editor: Matthew Moodie
Coordinating Editor: Mark Powers

Cover designed by eStudioCalamar

Cover image by Janko Ferlic on Unsplash (www.unsplash.com)

Distributed to the book trade worldwide by Apress Media, LLC, 1 New York Plaza, New York, NY 10004, U.S.A. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484262795. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

To Nicole

Table of Contents

About the Author

Peter Späth graduated in 2002 as a physicist and soon afterward became an IT consultant, mainly for Java-related projects. In 2016, he decided to concentrate on writing books on various aspects, but with a main focus on software development. With two books about graphics and sound processing, three books on Android app development, and a beginner's book on Jakarta EE development, the author continues his effort in writing software development-related literature.

About the Technical Reviewer

Luciano Manelli was born in Taranto, Italy, where he currently resides with his family. He graduated in Electronic Engineering at the Polytechnic of Bari at 24 years of age and then served as an officer in the Navy. In 2012, he earned a PhD in computer science from the IT department, University of Bari - Aldo Moro. His PhD focused on grid computing and formal methods, and he published the results in international publications. He is a professionally certified engineer and an innovation manager, and in 2014, he began working for the Port Network Authority of the Ionian Sea – Port of Taranto, after working for 13 years for InfoCamere SCpA as a software developer. He has worked mainly in the design, analysis, and development of large software systems; research and development; testing; and production with roles of increasing responsibility in several areas over the years. Luciano has developed a great capability to make decisions in technical and business contexts and is mainly interested in project management and business process management. In his current position, he deals with port community systems and digital innovation.

Additionally, he has written several IT books and is a contract professor at the Polytechnic of Bari and at the University of Bari - Aldo Moro. You can find out more at his LinkedIn page: `it.linkedin.com/in/lucianomanelli`.

Introduction

Starting at the very infancy of software creation, developers tried to modularize their applications in order to streamline their projects and increase the maintainability of the software they created. Soon, a very basic segregation scheme was identified: One part of the software must deal with data and persistence, another part must deal with presenting the data to the user, and one last part must handle data input and frontend view propagation.

This segregation scheme showed up in so many projects that it was promoted to a common software design pattern, called Model-View-Controller, or MVC for short. Its power also manifested in its versatility, even with big paradigm changes, like the onset of the Internet age. With database products for the model layer, browsers for the view layer, and some kind of user input processing for the controller layer, the pattern's accuracy and applicability to the majority of software projects became even more apparent with web applications.

Interestingly, even though most web application frameworks under the hood apply some kind of MVC layer demarcation, Java Server products up to JEE 7 did not include a dedicated MVC framework. With JSR-371 (Java Specification Request number 371) only recently and starting with JEE 8/Jakarta EE 8, an MVC specification entered the Java Enterprise application realm, which is one of the reasons this book was born. It does not describe all MVC Frameworks that you can add to Java EE/Jakarta EE as an external library. There are just too many of them and you can learn about them by looking at each library's documentation. Instead, we talk about the genuine Java MVC library as described by JSR-371.

The target version of Java MVC is 1.0, and we use a Jakarta EE version 8.0 compliant server to run Java MVC on it.

The Book's Targeted Audience

The book is for beginning or advanced enterprise software developers with knowledge of Java Standard Edition version 8 or later and some experience in Jakarta EE (or JEE) development. It is also assumed that the reader is able to use the online API references,

as this book is not a reference in the sense that all API classes and methods are listed. Instead, it presents techniques and technologies that help professional Java Enterprise level developers leverage web application programming by including Java MVC in their software.

The book uses the Linux operating system as the development platform, although the code can be run on other platforms (Windows and macOS) without complex adaptions. This book also does not talk about hardware issues (in case you don't use a laptop, a PC, or a server).

The readers will in the end be able to develop and run Java MVC programs of mid- to high-level complexity.

Sources

All sources shown or referred to in this book can be accessed via the Download Source Code button located at www.apress.com/9781484262795.

How to Read This Book

You can read this book sequentially from the beginning to the end, or you can read chapters on an ad hoc basis if your work demands special attention on a certain topic.

CHAPTER 1

About MVC: Model, View, Controller

MVC is a software design pattern. It describes the separation of software into three elements:

- **Model**: Manages the data of an application. This is to be understood in a narrow sense. Of course, any part of a less than trivial application deals with the application's data in one way or another, but the model from MVC corresponds to data items viewable to the user and possibly subject to change by user interactions. The model is agnostic to the way the data is represented to the user or any application workflow, so it can be said that the model is the central part of a MVC application. It is not surprising that developing a model is among the first steps of any MVC software project.
- **View**: Describes the presentation of the data and control elements (inputs, buttons, check boxes, menus, and so on) to the user. A view may provide different modes, like paged or non-paged tables, a formatted list or a link list, and so on. A view also may use different technologies, like a GUI component installed on the user's PC, an app on a mobile phone, or a web page to be viewed in a browser.
- **Controller**: Handles user input and prepares the data set necessary for the view part to do its work. While a view shows model items, the view never has to know how data is stored and retrieved from some persistent storage (database). This is the controller's responsibility. Because the user input determines what an application has to do next, the controller also contains the application logic. Any calculation and data transformation happens in the control part of MVC.

P. Späth, *Beginning Java MVC 1.0*, https://doi.org/10.1007/978-1-4842-6280-1_1

For example, consider a book club application. In this case, the model consists of elements such as books (including rental status), book storage location (building, room, or shelf), and member. For search application modules, you normally define lists of books, users, and so on, as model values.

The view part of the book club application will contain pages that show books, show members, show book locations, enable members to rent books, add club members, show book and member lists, as well as various search functionalities, and so on. Technically, this will often go hand in hand with a templating engine that defines placeholders for model elements, shortcuts for loops (for tables and lists), and other view elements like menus and buttons.

The controller handles the data the user enters. If, for example, the view currently shows a search page for books and the user enters a book's name and clicks on the Search button, the controller is informed as to which button was clicked. The controller then reads the request parameters (the book's name in this case) and possibly some model values (for example, the username and whether the user is logged in), queries the database, builds a result list, creates a model from this list, and finally decides which view page to show next.

There exists some fluffiness concerning the implementation details. This comes from the technical details of the data flow between view elements and model elements. MVC makes no assumption about *when* updates to view elements and model elements actually happen and which procedure is chosen to keep them synchronized. This is why, for MVC, you find many different diagrams in the literature.

For Java MVC, we can narrow our ideas about MVC to the following—a model (stored in memory) defines the application's state; a view shows model values and sends user interactions to a controller; and the controller prepares model data, handles user input and accordingly changes model values, and then decides which view page to show next. This kind of MVC model is depicted in Figure 1-1.

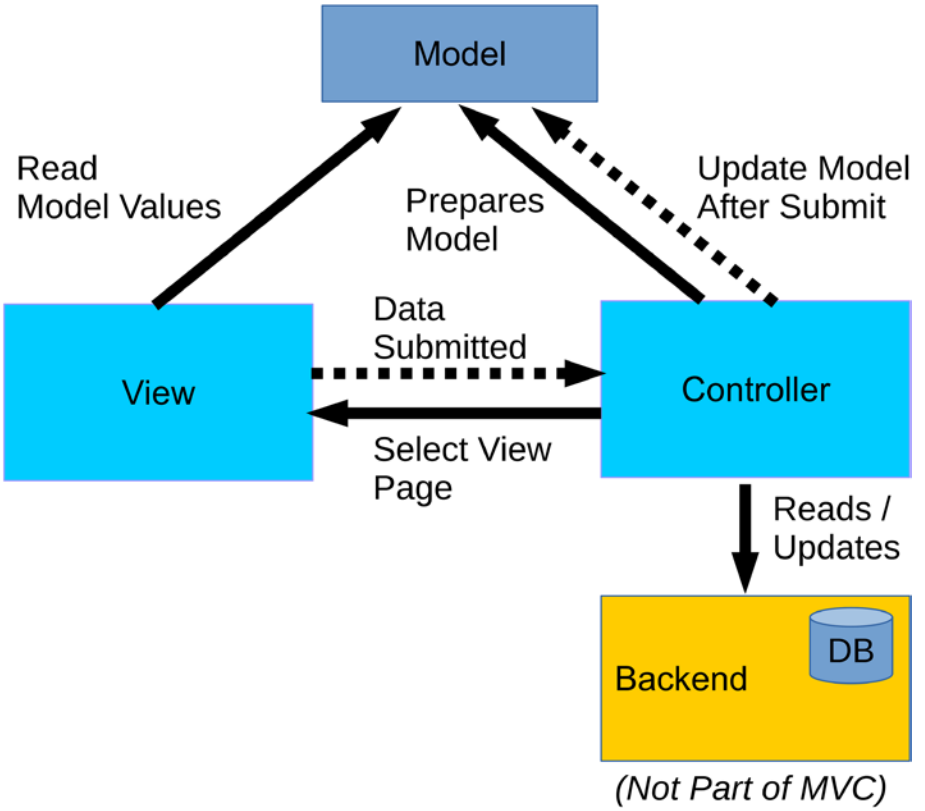

Figure 1-1. *The Java MVC design pattern*

The History of MVC

The advent of MVC dates back to the 1970s. It was introduced into the computer language Smalltalk as a programming concept. At that time, it did not have a name. Only later, in the late 1980s, was the moniker MVC explicitly used. It appeared in an article in the periodical *Journal of Object Technology*.

MVC steadily became more and more widespread, and its ideas were so widely adopted that variants evolved from MVC. We don't talk about these variants in this book, but a short list includes:

- **PAC (Presentation-Abstraction-Control) and HMVC (Hierarchical MVC).** This is a variation of MVC, where submodules have their own MVC-like structure and only later is a view page constructed from them.

- **MVA (Model-View-Adapter).** In this pattern, the view and the model are separated and only the controller (called an adapter in this case) mediates between the model and the view. The view has no direct access to model values.
- **MVP (Model-View-Presenter).** In MVP, the view contains logic to inform the controller (called a presenter in this case) about view-related data changes. The presenter then performs some activities and eventually calls back to the view in order to inform the user about data changes.
- **MVVM (Model-View-View-Model).** In MVVM, some automatism is introduced, which translates model values to view elements and vice versa.

The real power of MVC was revealed in the 1990s with the rise of the Internet. Although some technical details changed—such as the exact technical characteristics of the data flow and the point in time when data traverses the layer boundaries—the idea remained the same: a model holds the application state, a view presents the browser pages, and a controller handles the interaction between the browser and the model, and decides which view page to show.

Various MVC web frameworks were invented; `https://en.wikipedia.org/wiki/Comparison\_of\_web\_frameworks` shows you a comprehensive list (further down on the page, MVC capabilities are also listed).

MVC in Web Applications

Web applications impose some restrictions if we try to let them work the MVC way. The most important distinction comes from the stateless nature of the HTTP protocol, which is used for communication between the view (browser window) and the controller (HTTP server). In fact, the way web application frameworks handle the HTTP protocol leads to decisive differences between the different MVC implementations.

In more detail, important questions concerning MVC for web applications are as follows:

- **Sessions:** We already pointed out the stateless nature of HTTP. So, if the browser sends a request, maybe because the user entered some string into a text field and then pressed the Submit button, how would the server know which user is performing the request? This usually gets handled by a session, which is identified by a session ID transmitted as a cookie, request, or POST parameter. Sessions are transparently handled by the framework, so you don't have to create and maintain sessions from inside the application's code.
- **Accessing model values from the view:** With web applications, some kind of templating engine usually handles the view generation. There, we could have expressions like `${user.firstName}` to read the contents of a model entry.
- **Transmitted data extent:** If data is submitted from the web page to the server, we basically have two options. First, the complete form could be transmitted. Second, only the data that changed could be sent to the server. The latter reduces network traffic, but requires some script logic (JavaScript) to perform the data collection on the web page.
- **Updating the view:** With web applications, the way a view is updated is crucial. Either the complete page is loaded after the controller works a request, or only those parts of a web page that actually need an update are transmitted from the server to the browser. Again, the latter method reduces network traffic.

From these points, you can see that programming a MVC framework for web applications is not an utterly trivial task. This is also why there are quite a large number of different MVC frameworks you can use for web applications. In the rest of the book, I will show you why choosing Java MVC is not the worst thing you can do if you need MVC software for your Java platform.

MVC for Java

In the Java ecosystem, a framework named Struts entered the software world around 2000. It is a MVC framework aimed at web applications and integrating with Java EE/Jakarta EE and Tomcat (a server product boiled down to web functionalities). It has been used in many software projects and is still being used, albeit it is not part of the Java EE/Jakarta EE specification. Instead, Java EE/Jakarta EE names JSF (Java Server Faces) as the dedicated web framework. JSF, in contrast to MVC, uses a component-oriented approach for creating web applications.

JSF works out-of-the-box for any Java EE/Jakarta EE 8 or later product. Up to version 7, if you wanted to use MVC, Struts was one of the prominent frameworks you could use. However, in order for Struts to work, an external library had to be added to the application, and Struts always felt like an extension and not so much like something that seamlessly integrated with Java EE/Jakarta EE.

With Java EE 8/Jakarta EE 8, the MVC world reentered the game in form of a Java MVC specification. It is still kind of a second-class citizen in the Java EE/Jakarta EE world, but there are reasons to favor MVC over JSF. We talk about the merits and disadvantages of MVC over other frameworks like JSF at the end of this chapter.

Finally, Java MVC (JSR-371)

The latest Java EE/Jakarta EE MVC implementation operates under the name *Java MVC* and is governed by JSR-371. It is the first MVC framework available for Java EE/Jakarta EE servers version 8 or higher. In fact, the JSR describes an interface. For Java MVC to actually work, you need to add an implementation library.

Note We use Eclipse Krazo as the Java MVC implementation library. See
https://projects.eclipse.org/proposals/eclipse-krazo

or

https://projects.eclipse.org/projects/ee4j.krazo

We will later see how to install Eclipse Krazo for your web application.

Java MVC is a lean and clever extension of the REST technology JAX-RS included within Java EE/Jakarta EE. This relationship gives Java MVC a modern touch and allows for a concise and highly comprehensive programming style.

We already learned that MVC allows for some fluffiness concerning the implementation details. Figure 1-1 describes how Java MVC works quite well: A request for a first page in the browser window routes to the controller, which prepares model values (with or without querying some backend for additional data). The controller then decides which view page (browser page) to show next (maybe a login page). The view can access model values. With a data set entered by the user and submitted to the controller, the controller takes request parameters (for example, the login name and password), possibly queries the backend (the user database), updates the model, and finally selects a new view page (for example, a welcome page after successful authentication).

But there is an additional feature that seamlessly integrates with Java MVC. Instead of always loading a complete new page after each HTTP request, you can decide to let parts of your web application use AJAX for more fine-grained frontend-backend communication. Because we use Java MVC in a Java EE/Jakarta EE 8 (or later) environment, we can use JAX-RS for that aim out-of-the-box.

Why MVC

With so many web frontend technologies out there, it is not easy to decide which to use for your project. The new Java MVC certainly is an option and it might very well suit your needs. In order to help you make a decision, here is a list of pros and cons of Java MVC.

Cons:

- MVC seems to be a old-fashioned design pattern. Although this is true, it also has been proven to work well for many projects, and Java MVC allows developers to mix in more modern web development techniques.
- MVC forces the developer to be aware of HTTP internals. MVC is also said to be an action-based design pattern. Actions in a web environment mean HTTP requests and responses. MVC doesn't really hide the internals of the HTTP communication like other frameworks do.

- MVC does not introduce two-way data bindings like other frameworks do. With two-way data bindings, a change in a frontend input field immediately reflects in the model value changes. Instead, in a MVC controller, you have to explicitly implement the update of model values.

Pros:

- Since it's closer to the HTTP communication internals compared to other frameworks, despite introducing some complexity, this introduces less invasive memory management. If you look at JSF, a complete component tree (and component data tree) is built with each browser request. In contrast, a MVC application can be tailored with an extremely small memory footprint.
- Java MVC is part of the Java EE/Jakarta EE 8 specification. This helps to more reliably handle maintenance.
- If you are used to Struts or similar frontend frameworks, switching to Java MVC feels more natural compared to switching to other products with other frontend design patterns.

Where Is Hello World?

In many software-related development instruction books, you find a really simple "Hello World" example in one of the first chapters. For Jakarta EE, this means we must provide a shortcut way to do the following:

- Write a short program that does something simple, like output the string `"Hello World"`.
- Build a deployable artifact from the string (for example, a `.war` file).
- Run a Jakarta EE server.
- Deploy the application (the `.war` file) on the server.
- Connect a client (for example, a browser) to the server.
- Observe the output.

This is a lot of stuff, so instead of building a quick-and-dirty setup to run such an example, I prefer to first talk about Java/Jakarta Enterprise Edition (Java/Jakarta EE) in general, then discuss the development workflow, and only after that, introduce a simple first project. This way, we can make sure your first Java MVC application is developed and runs correctly.

If you think a quick-and-dirty Hello World example will help you, the following paragraphs show you how to create one. Note that we won't use the development processes shown here in the rest of the book—this is simply a simplistic and fast, and maybe not-so-clean, approach. You can also skip this section safely, because we create a proper Hello World project in Chapter 4.

1. First make sure OpenJDK 8 is installed on your PC. Go to `https://jdk.java.net/java-se-ri/8-MR3` to download it. In the rest of this section, we call the OpenJDK 8 folder `OPENJDK8_DIR`.
2. Download and install GlassFish 5.1 from `https://projects.eclipse.org/projects/ee4j.glassfish/downloads` (choose the "Full Profile" variant). In the rest of this section, we call the GlassFish installation folder `GLASSFISH_INST_DIR`.
3. Inside the `GLASSFISH_INST_DIR/glassfish/config/asenv.conf` (Linux) or `GLASSFISH_INST_DIR/glassfish/config/asenv.bat` (Windows) file, add the following lines:

```
REM Windows:
REM Note, if the OPENJDK8_DIR contains spaces, wrap it
REM inside "..."
set AS_JAVA=OPENJDK8_DIR

# Linux:
AS_JAVA="OPENJDK8_DIR"
```

You must replace `OPENJDK8_DIR` with the installation folder of the OpenJDK 8 installation.

4. Start the GlassFish server:

```
REM Windows:
chdir GLASSFISH_INST_DIR
bin\asadmin start-domain
```

```
# Linux:
cd GLASSFISH_INST_DIR
bin/asadmin start-domain
```

You must replace `GLASSFISH_INST_DIR` with the installation folder of GlassFish.

5. Create a folder called `hello_world` anywhere on your file system. Its contents have to be (instructions follow):

```
build
  |- <empty>
src
  |- java
  |    |- book
  |         |- javamvc
  |              |- helloworld
  |                   |- App.java
  |                   |- RootRedirector.java
  |                   |- HelloWorldController.java
  |- webapp
  |    |- META-INF
  |    |    |- MANIFEST.MF
  |    |- WEB-INF
  |         |- lib
  |         |    |- activation-1.1.jar
  |         |    |- javaee-api-8.0.jar
  |         |    |- javax.mail-1.6.0.jar
  |         |    |- javax.mvc-api-1.0.0.jar
  |         |    |- jstl-1.2.jar
  |         |    |- krazo-core-1.1.0-M1.jar
  |         |    |- krazo-jersey-1.1.0-M1.jar
  |         |- views
  |         |    |- greeting.jsp
  |         |    |- index.jsp
  |         |- beans.xml
```

```
  |         |- glassfish-web.xml
make.bat
make.sh
```

6. Get the JARs for the `lib` folder from https://mvnrepository.com. Enter each name without the version and the `.jar` extension in the search field, select the version, and then get the JAR file.

7. The Java code reads as follows:

```
// App.java:
package book.javamvc.helloworld;

import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

@ApplicationPath("/mvc")
public class App extends Application {
}

// RootRedirector.java
package book.javamvc.helloworld;

import javax.servlet.FilterChain;
import javax.servlet.annotation.WebFilter;
import javax.servlet.http.HttpFilter;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

/**
 * Redirecting http://localhost:8080/HelloWorld/
 * This way we don't need a <welcome-file-list> in web.xml
 */
@WebFilter(urlPatterns = "/")
public class RootRedirector extends HttpFilter {
    @Override
    protected void doFilter(HttpServletRequest req,
           HttpServletResponse res,
```

```
            FilterChain chain) throws IOException {
        res.sendRedirect("mvc/hello");
    }
}

// HelloWorldController.java
package book.javamvc.helloworld;

import javax.inject.Inject;
import javax.mvc.Controller;
import javax.mvc.Models;
import javax.mvc.binding.MvcBinding;
import javax.ws.rs.FormParam;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.core.Response;

@Path("/hello")
@Controller
public class HelloWorldController {
    @Inject
    private Models models;

    @GET
    public String showIndex() {
        return "index.jsp";
    }

    @POST
    @Path("/greet")
    public Response greeting(@MvcBinding @FormParam("name")
           String name) {
        models.put("name", name);

        return Response.ok("greeting.jsp").build();
    }
}
```

8. As MANIFEST.MF, write the following:

```
Manifest-Version: 1.0
```

9. The view files read as follows:

```
<%-- index.jsp --%>
<%@ page contentType="text/html;charset=UTF-8"
    language="java" %>
<%@ taglib prefix="c"
    uri="http://java.sun.com/jsp/jstl/core" %>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Hello World</title>
</head>
<body>
  <form method="post"
      action="${mvc.uriBuilder('HelloWorldController#
             greeting').build()}">
    Enter your name: <input type="text" name="name"/>
    <input type="submit" value="Submit" />
  </form>
</body>
</html>
```

```
<%-- greeting.jsp --%>
<%@ page contentType="text/html;charset=UTF-8"
    language="java" %>
<%@ taglib prefix="c"
    uri="http://java.sun.com/jsp/jstl/core" %>
<html>
<head>
    <meta charset="UTF-8">
    <title>Hello World</title>
</head>
```

```
<body>
  Hello ${name}
</body>
</html>
```

(Remove the line break and the spaces after `HelloWorldController#`.)

10. As `beans.xml`, create an empty file (the file must exist, though!).

11. The contents of `glassfish-web.xml` reads as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<glassfish-web-app error-url="">
    <class-loader delegate="true"/>
</glassfish-web-app>
```

12. The Linux build file called `make.sh` reads as follows:

```
#!/bin/bash
JAVA_HOME=/path/to/your/openjdk-8

rm -rf build/*
cp -a src/webapp/* build
mkdir build/WEB-INF/classes

$JAVA_HOME/bin/javac \
    -cp src/webapp/WEB-INF/lib/javaee-api-8.0.jar:
        src/webapp/WEB-INF/lib/javax.mvc-api-1.0.0.jar \
    -d build/WEB-INF/classes \
    src/java/book/javamvc/helloworld/*

cd build
$JAVA_HOME/bin/jar cf ../HelloWorld.war *
cd ..
```

(Remove the line break and spaces after the `:`.)

13. The Windows build file `make.bat` reads as follows:

```
set JAVA_HOME=C:\dev\java-se-8u41-ri

mkdir build
CD build && RMDIR /S /Q .
CD ..
rmdir build

xcopy src\webapp build /s /e /i
mkdir build\WEB-INF\classes

%JAVA_HOME%\bin\javac ^
    -cp src\webapp\WEB-INF\lib\javaee-api-8.0.jar;
        src\webapp\WEB-INF\lib\javax.mvc-api-1.0.0.jar ^
    -d build\WEB-INF\classes ^

    src\java\book\javamvc\helloworld/*

cd build
%JAVA_HOME%\bin\jar cf ..\HelloWorld.war *
cd ..
```

(Remove the line break and spaces after the `;`.)

To build the application from inside the console, move into the `hello_world` folder and start the script:

```
# Linux
cd hello_world
./make.sh

rem Windows
chdir hello_world
make
```

Apart from some error messages for the Windows build script that you can safely ignore, you will end up with the `HelloWorld.war` web application in the main folder. From there, you can deploy the application via the following:

```
# Linux
GLASSFISH_INST_DIR/bin/asadmin deploy --force=true \
    HelloWorld.war
```