Hands On: Creating a Basic Java Web App

CREATING OUR FIRST JAVA WEB APP AND SERVLET

Sarah Holderness
PLURALSIGHT AUTHOR
@dr_holderness

Creating Our First Java Web App and Servlet

Overview

The main components of a Java MVC web app:

- Servlets (Controller)
- Java Server Pages (View)
- Database (Model)

Demo

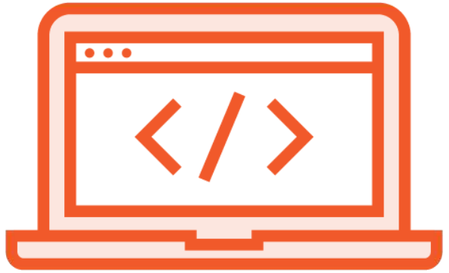
Book Store

Add New Book

List of Books

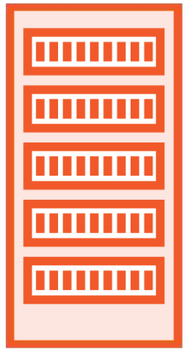
Title	Author	Price
1984	George Orwell	$5.00
To Kill a Mockingbird	Harper Lee	$5.00
Ready Player One	Ernest Cline	$7.00

Environment Setup



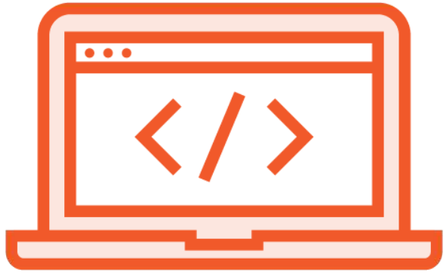
ECLIPSE IDE & JDK

Install the Java Development Kit and we'll be using the Eclipse IDE.



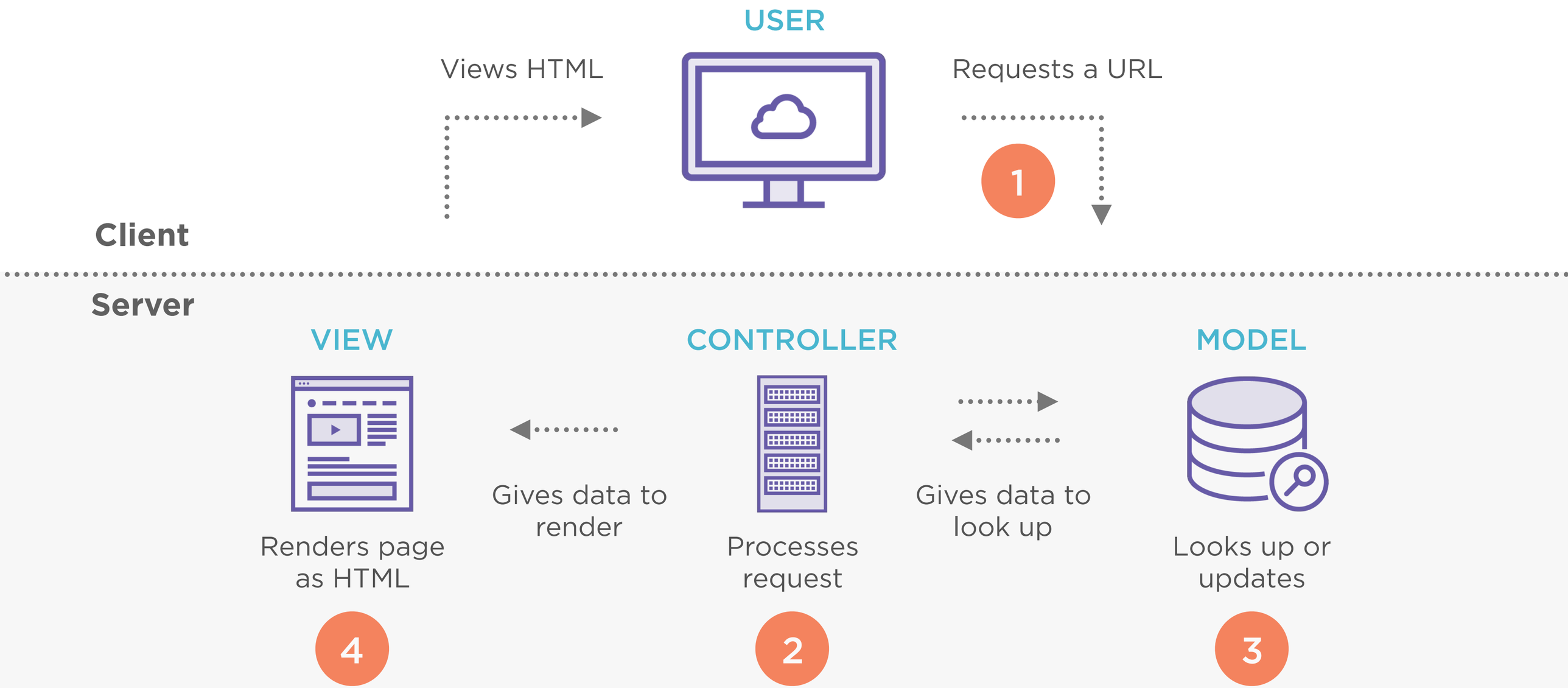
TOMCAT SERVER

We'll be using Tomcat with Eclipse. But you can use any Java compatible server.

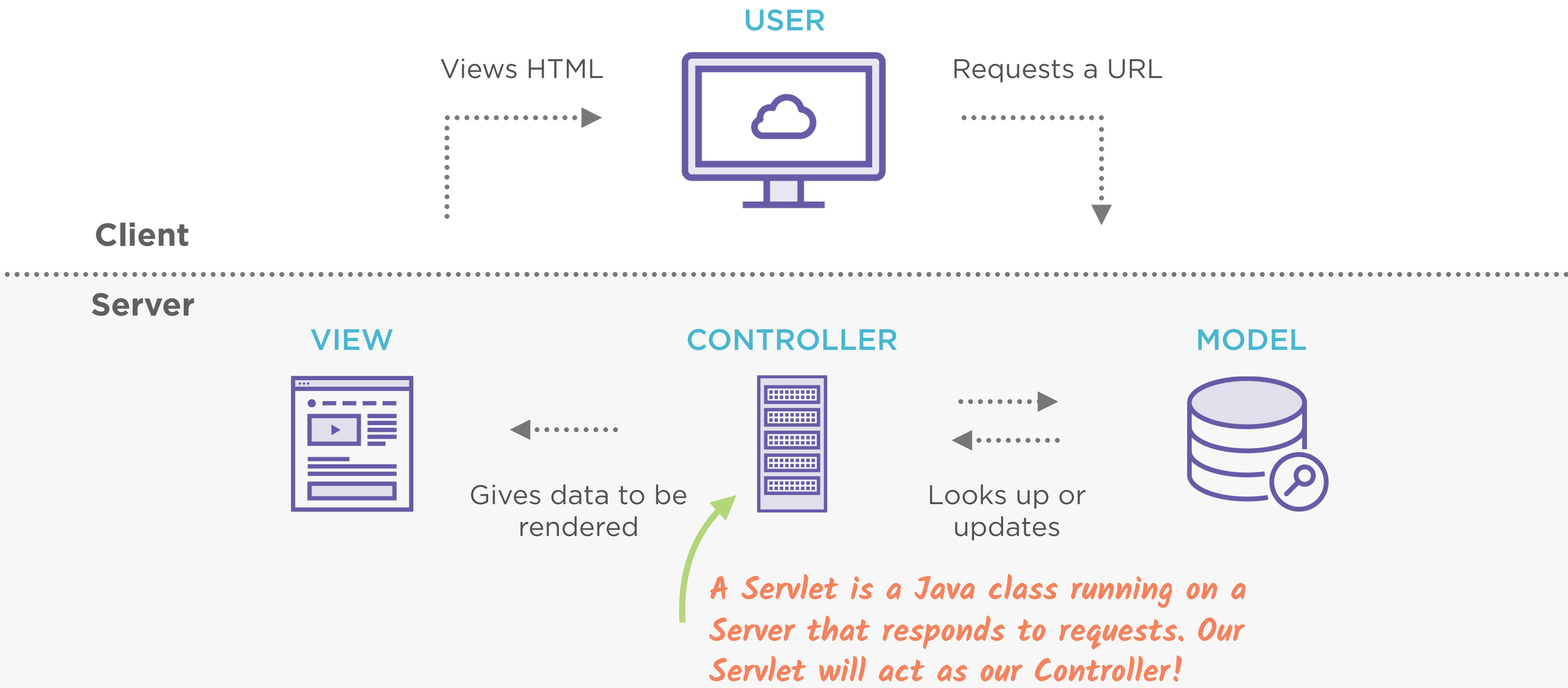


ECLIPSE PROJECT

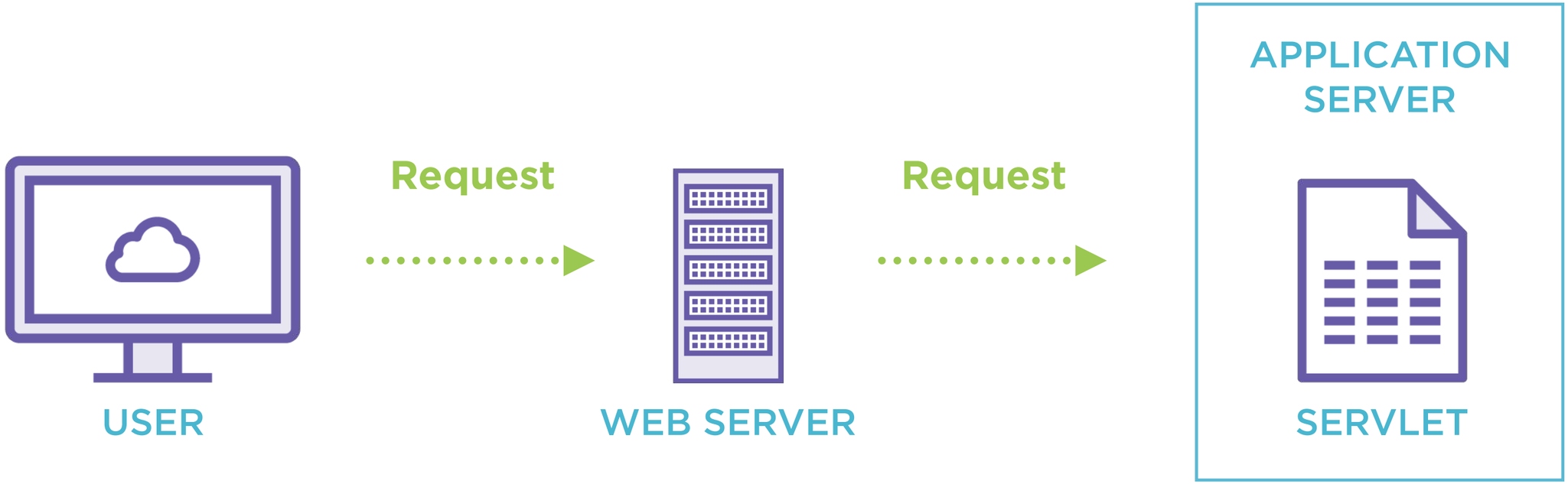
How Data Moves Through a Java Web App



How Data Moves Through a Java Web App



What is a Servlet?



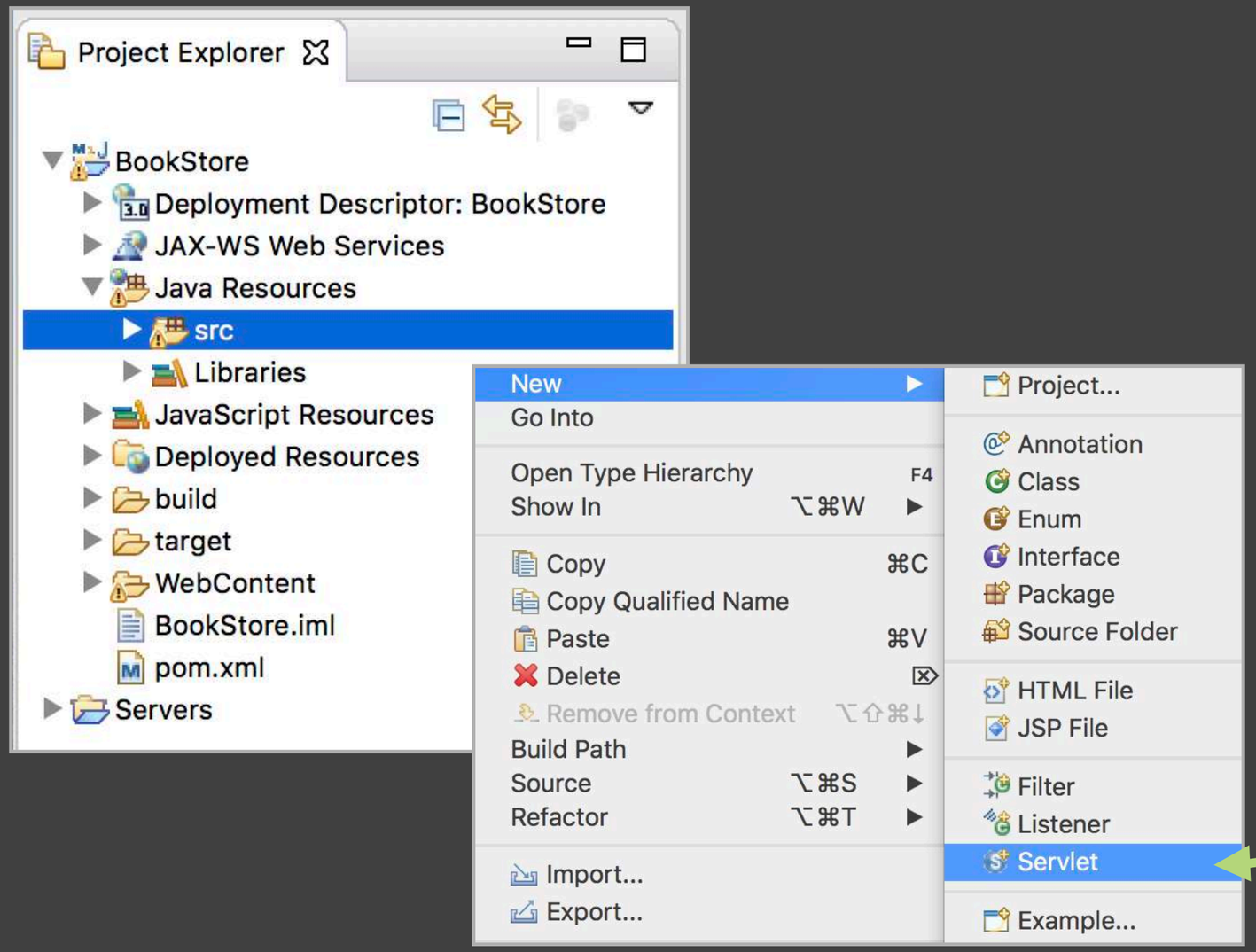
The server will run a special application server that can handle Java Servlets (e.g. Tomcat).

A Servlet is Java class running on an application server that can respond to requests.

The `HttpServlet` Class

We'll use the `HttpServlet` class, which provides methods, such as `doGet` and `doPost`, for handling HTTP-specific services.

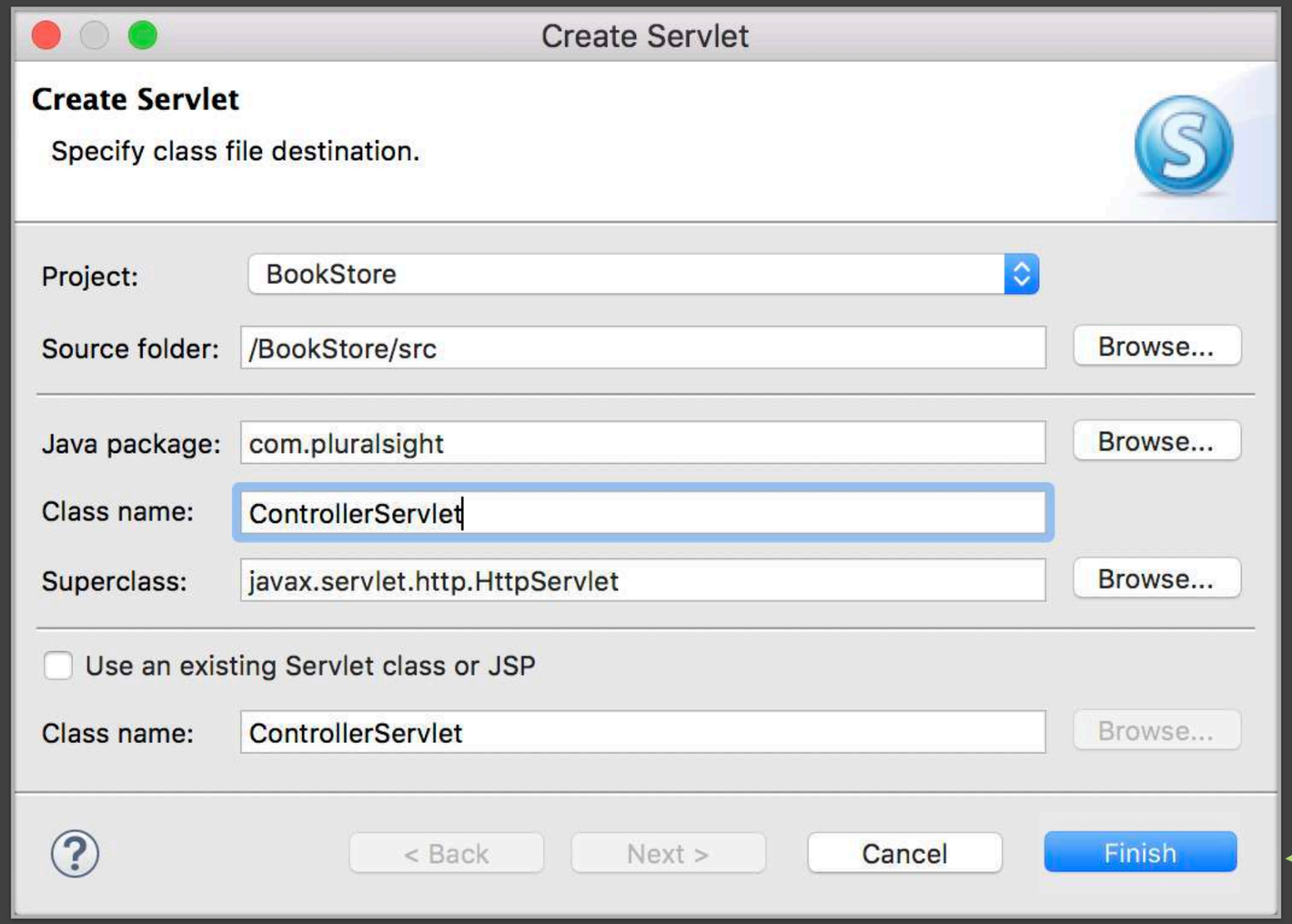
Creating Our First Servlet



To create a new Servlet class, first right click the src folder under Java Resources.

Then select New Servlet.

Creating Our First Servlet



Then type in the package as `com.pluralsight` *(or whatever you want). The reverse of your domain is standard.*

And the class name, ours will be `ControllerServlet`*.*

Then click Finish.

Creating Our First Servlet

ControllerServlet.java

Our new class will auto-generate some code.

```
…import…

@WebServlet("/ControllerServlet")
public class ControllerServlet extends HttpServlet {
…

   protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException  {
      // TODO Auto-generated method stub
      response.getWriter().append("Served at: ").append(request.getContextPath());
   }
}
```

We will focus on the `doGet()` *method first. We can delete the line of code that was generated in there.*

Creating Our First Servlet

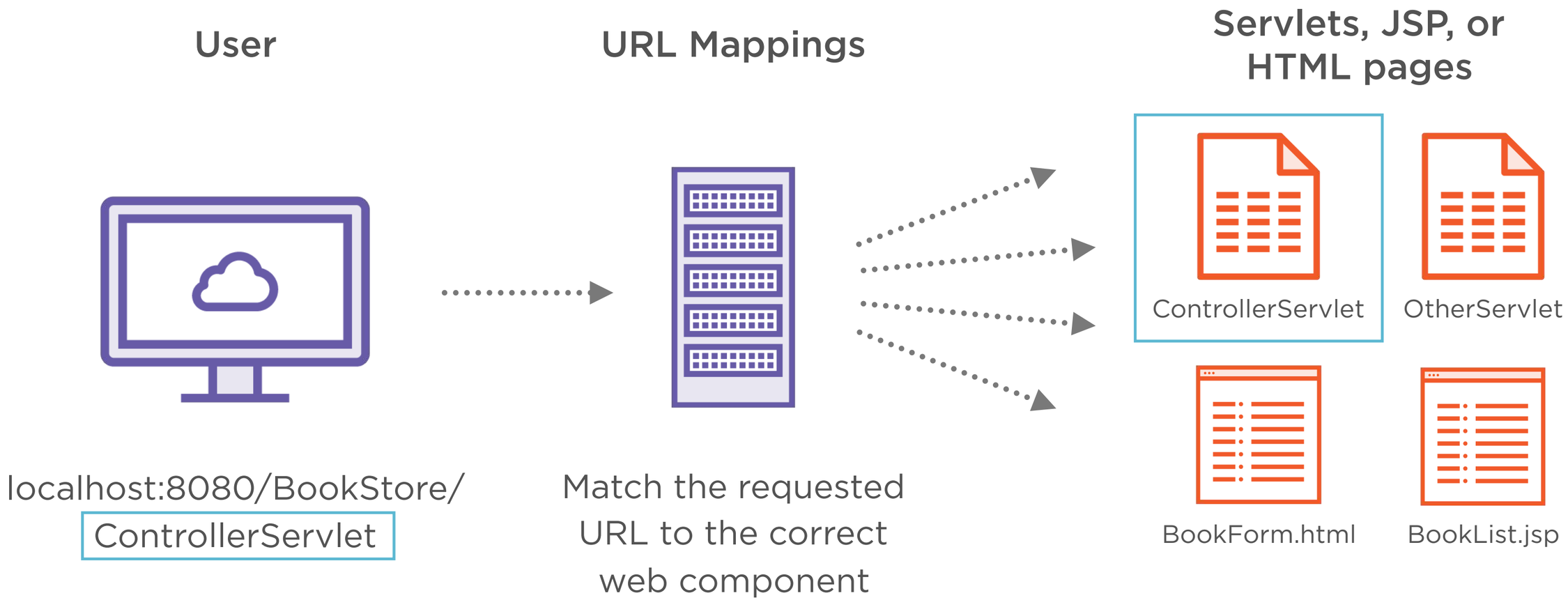
ControllerServlet.java

```
…import…

@WebServlet("/ControllerServlet")
public class ControllerServlet extends HttpServlet {
…

   protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException  {

      PrintWriter output = response.getWriter();
      output.println("Welcome to ControllerServlet!");
   }
}
```

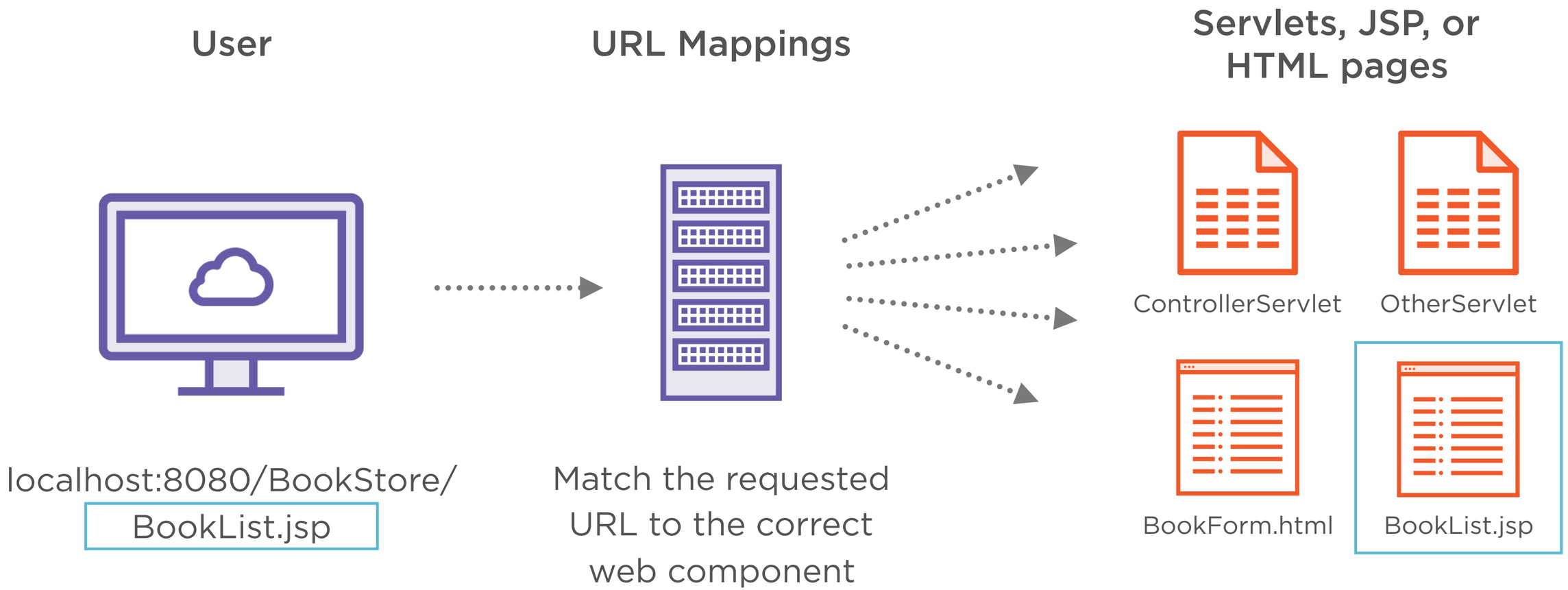
We'll use `PrintWriter` *to print some text to the screen.*

URL Mappings

URL Mappings - How Do We Connect to Servlets?



URL Mappings - How Do We Connect to Servlets?

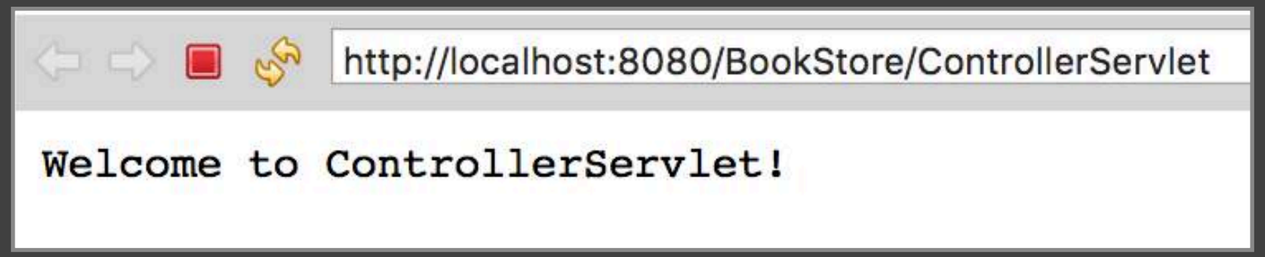


@WebServlet Notation Defines a URL Mapping

ControllerServlet.java

```
… import …

@WebServlet("/ControllerServlet")
public class ControllerServlet extends HttpServlet {

    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response) … {
     PrintWriter output = response.getWriter();
     output.println("Welcome to ControllerServlet!");
   }
   ...
}
```

The @WebServlet annotation is followed by the URL patterns that this servlet will be available at.

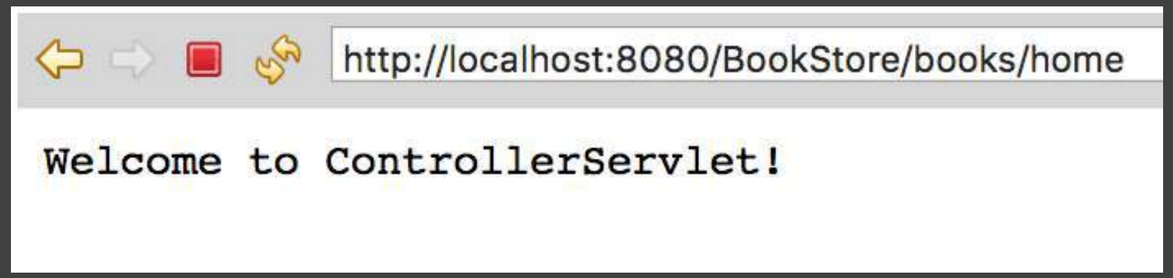


@WebServlet Notation Defines a URL Mapping

ControllerServlet.java

```
…

@WebServlet("/books/*")
public class ControllerServlet extends HttpServlet {

    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response) … {
     PrintWriter output = response.getWriter();
     output.println("Welcome to ControllerServlet!");
   }
   ...
}
```

*By using the * wildcard, anything that comes after /books/ will map to this ControllerServlet class.*

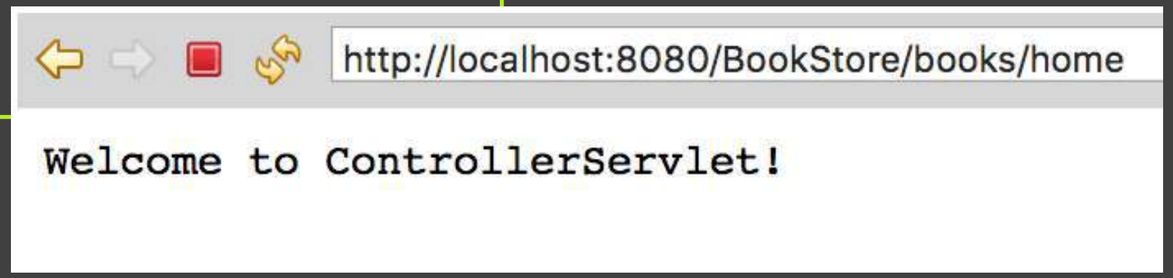


The `web.xml` File also Defines URL Mappings

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="…" xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="…" …>
  <display-name>BookStore</display-name>

<servlet>
   <servlet-name>ControllerServlet</servlet-name>
   <servlet-class>com.pluralsight.ControllerServlet</servlet-class>
</servlet>

<servlet-mapping>
   <servlet-name>ControllerServlet</servlet-name>
   <url-pattern>/books/*</url-pattern>
</servlet-mapping>

</web-app>
```

This will define the same exact URL mapping as the WebServlet notation we just defined.



The `web.xml` File also Defines URL Mappings

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="…" xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="…" …>
  <display-name>BookStore</display-name>

<servlet>
   <servlet-name>ControllerServlet</servlet-name>
   <servlet-class>com.pluralsight.ControllerServlet</servlet-class>
</servlet>

<servlet-mapping>
   <servlet-name>ControllerServlet</servlet-name>
   <url-pattern>/books/*</url-pattern>
</servlet-mapping>

</web-app>
```

In the <servlet> tag we can define the servlet name and the full class name.

In the <servlet-mapping> tag we reference servlet-name we just defined. And then we provide the url pattern that leads to our Servlet class.

Getting Data into a Servlet from Query Parameters

Getting Data in a Servlet from Query Params

ControllerServlet.java

```
…

@WebServlet("/books/*")
public class ControllerServlet extends HttpServlet {
…
   protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException  {

      PrintWriter output = response.getWriter();

      String title = request.getParameter("title");

      output.println("Book Title = " + title);
   }
}
```

We can get the query string parameters with the request's getParameter() method with the key "title" as a parameter.

http://localhost:8080/BookStore/books?title=1984

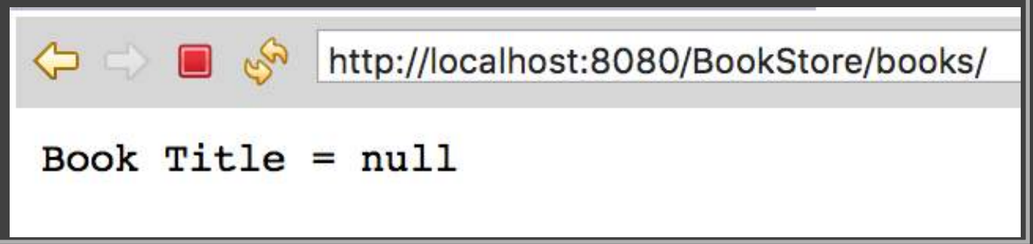
```
Book Title = 1984
```

Getting Data in a Servlet from Query Params

ControllerServlet.java

```
…

@WebServlet("/books/*")
public class ControllerServlet extends HttpServlet {
…
   protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException  {

      PrintWriter output = response.getWriter();

      String title = request.getParameter("title");

      output.println("Book Title = " + title);
   }
}
```

If there are no query params, the String title is null.



Getting Data into a Servlet from a Form

```html
<form action="/books">
    <p><label>Title:</label>
    <input type="text" name="title" /></p>

    <p><label>Author:</label>
    <input type="text" name="author" /></p>

    <p><input type="submit" value="Submit"></p>
</form>
```

Book Title:

Author:

Submit

HTML Forms Review

`<form action="/books">` The opening form element defines our form. The action defines the action to take when the form is submitted.

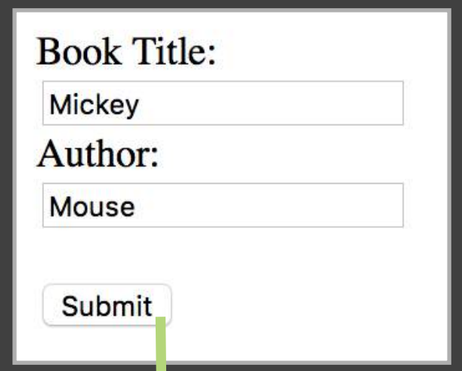
`<input type="text">` defines a one-line input field for text input.

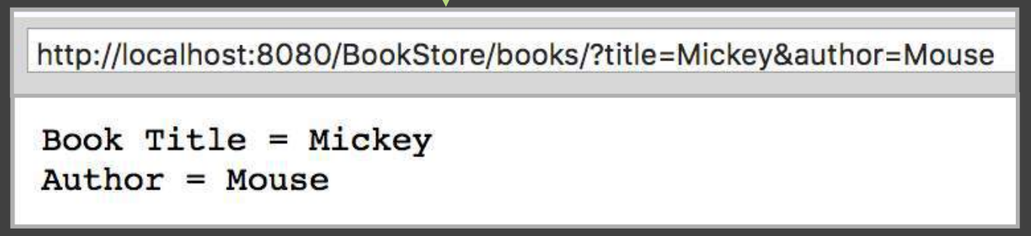
`<input type="submit">` defines a button for submitting our form to a form-handler.

Getting Data in a Servlet from a Form

ControllerServlet.java

```
…
   protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException  {

      PrintWriter output = response.getWriter();

      String title = request.getParameter("title");
      String author = request.getParameter("author");

      output.println("Book Title = " + title);
      output.println("Author = " + author);
   }
}
```





```
<form method="post" action="/books">
    <p><label>Title:</label>
    <input type="text" name="title" /></p>

    <p><label>Author:</label>
    <input type="text" name="author" /></p>

    <p><input type="submit" value="Submit"></p>
</form>
```

Get vs Post

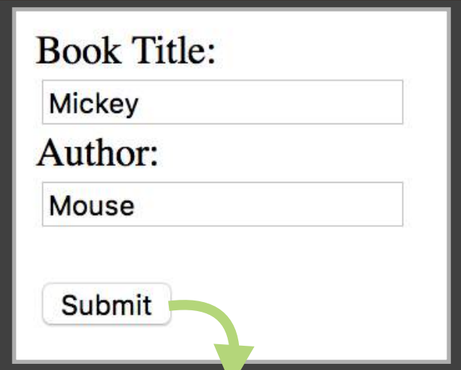
A form should use a GET request if it is requesting or searching for data from the server.

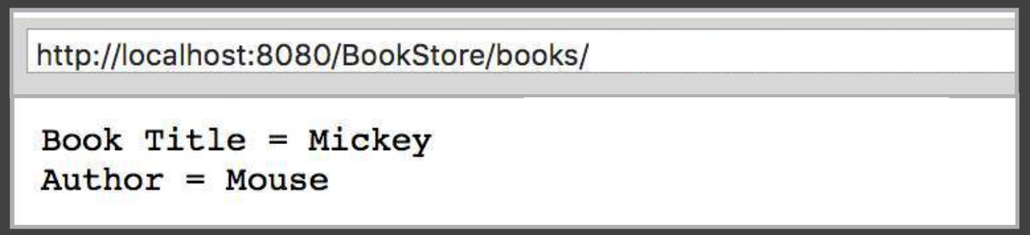
If the form is going to update data on the server the request should be a POST.

Moving the Contents of doGet() to doPost()

ControllerServlet.java

```
…
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {}

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException  {

      PrintWriter output = response.getWriter();

      String title = request.getParameter("title");
      String author = request.getParameter("author");
      output.println("Book Title = " + title);
      output.println("Author = " + author);
    }
}
```





In This Demo - Sending Data to a Servlet With a Form

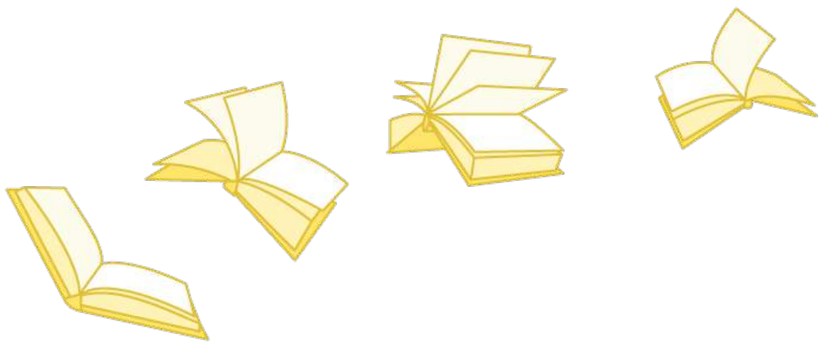
Book Title:

Author:

Price:

Submit

Submitting values for title, author, and price to a Servlet.



Summary

- The main components of a Java MVC web app
- Introduction to Servlets
- URL Mappings
- Passing Data to a Servlets

Hands On: Creating a Basic Java Web App

CREATING OUR FIRST JSP PAGE

Sarah Holderness
PLURALSIGHT AUTHOR
@dr_holderness

Creating Our First JSP Page

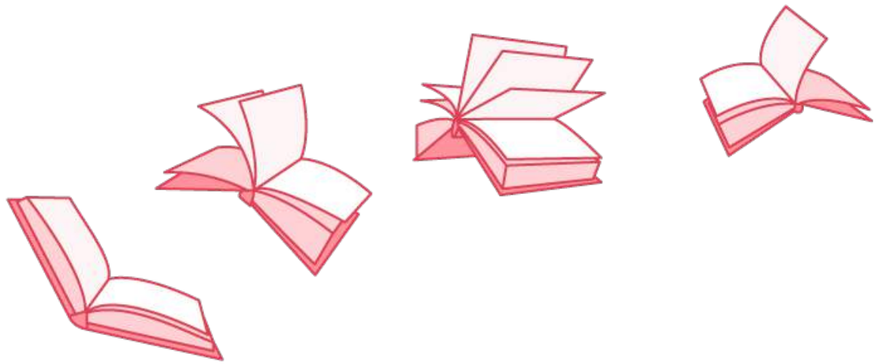
Overview

The main components of a Java MVC web app:

- ☑ Servlets (Controller)
- ☐ Java Server Pages (View)
- ☐ Database (Model)

Displaying Our Books

How can we display dynamic data like our list of books?



http://localhost:8080/BookStore/books/*

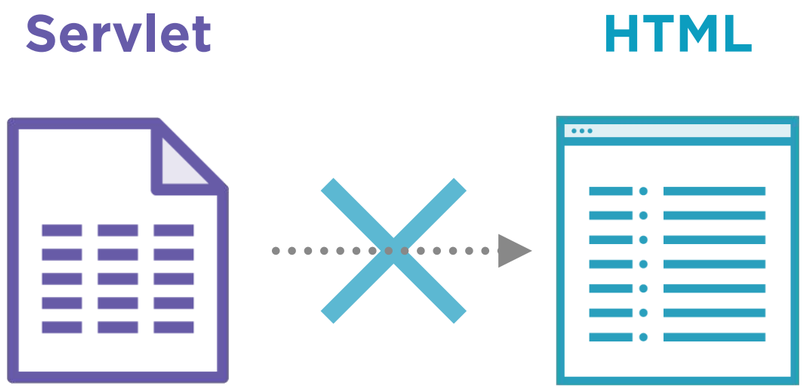
Book Store

Add New Book

List of Books

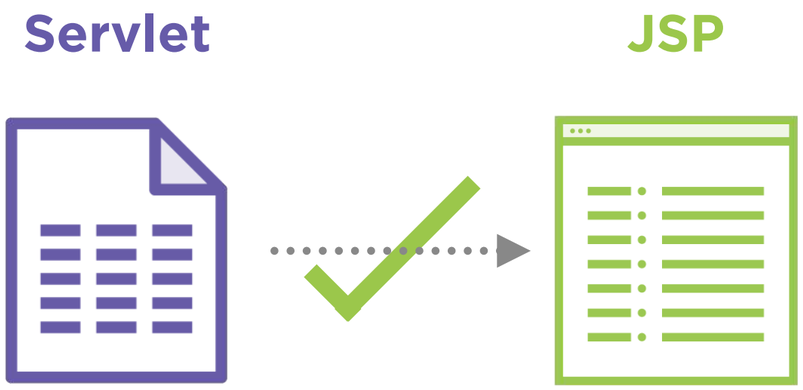
Title	Author	Price
To Kill a Mockingbird	Harper Lee	5.5
1984	George Orwell	4.5
Frankenstein	Mary Shelly	4.0

How Do We Pass Dynamic Data to an HTML Page?



HTML

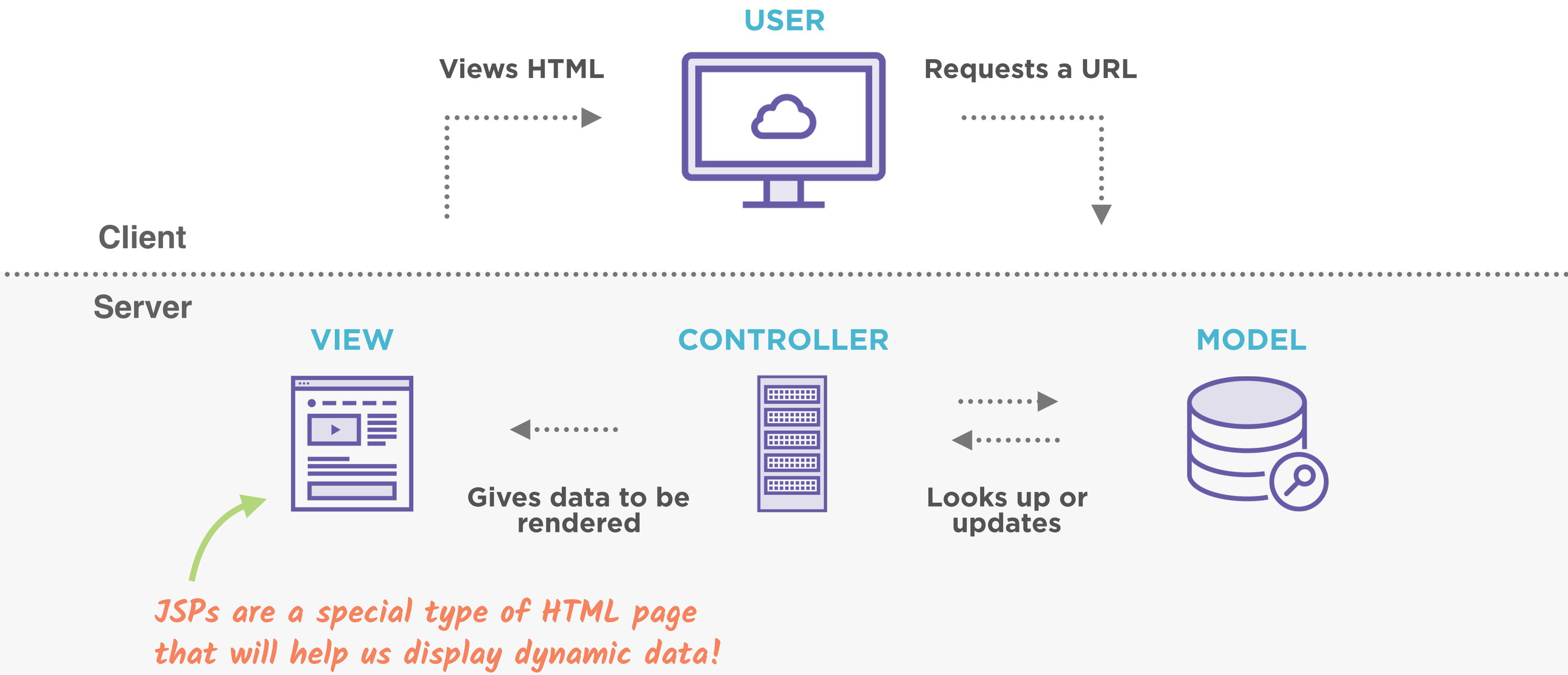
A Servlet can't pass Java objects to HTML.



JSP

A Servlet can pass Java objects to a Java Server Page (JSP). JSPs are special HTML pages that can write Java code.

The Java MVC Design



```
<%@ page language="java" contentType="text/html;%>

<!DOCTYPE …>
<html>
…
</html>
```

What is a JSP File?

All JSP pages have a page directive, the `<%@ page` part. It is used to define attributes that apply to an entire JSP page.

For example, right now we're only concerned with `language="java"` which sets the scripting language for the entire page.

```
…
<html>…
<body>
    <% java.util.Date today = new java.util.Date(); %>
    <%= today.toString() %>
</body>
</html>
```

Scriptlet tag

Expression tag

No semicolon ; needed inside expression tag

Running Java in a JSP File

A scriptlet tag allows you to write Java code inside the JSP page.

An expression tag allows you to output the values of variables or methods.

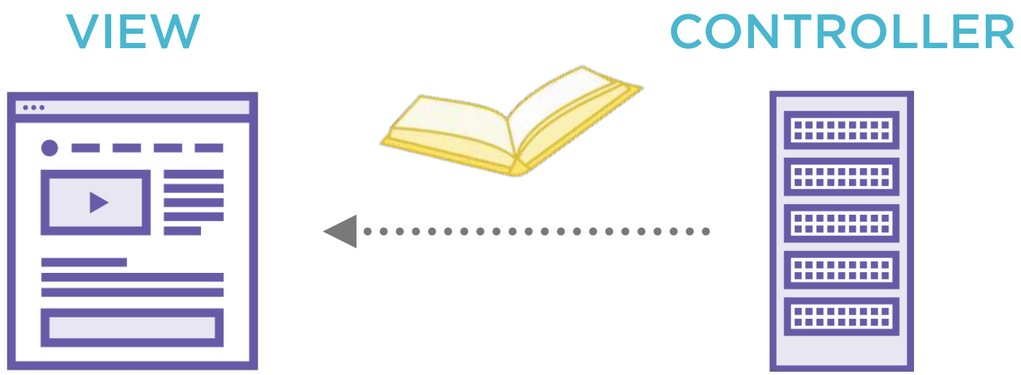
```
<%@ page  language="java" contentType="text/html;%>
<%@ page  import="java.util.Date, java.other.libs.*"%>

<html>
…
</html>
```

Importing Java Libraries in JSP Pages

We can get use the `@ page` directive, similar to how we set the page language to be Java. We can import libraries by listing them separated by commas.

Passing Data to a JSP Page

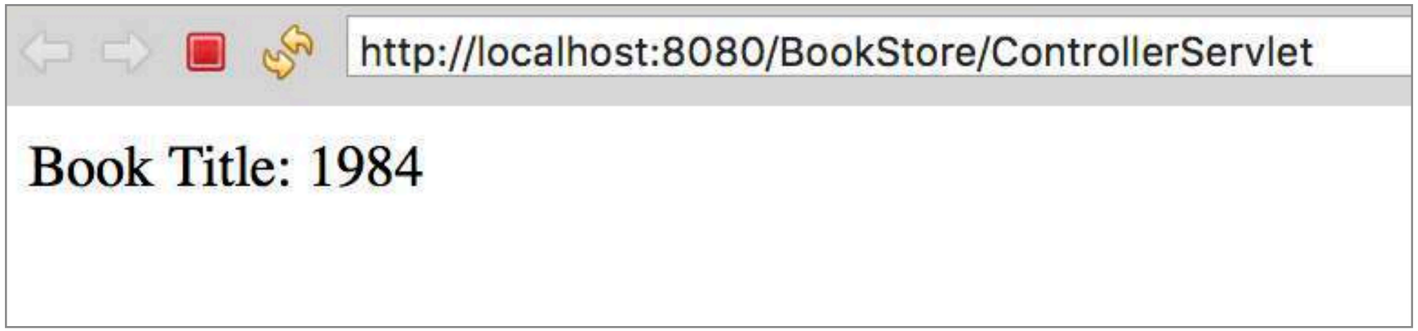
Step 1: Pass 1 Book to our JSP Page



Display the passed in book in BookList.jsp

Pass 1 Book from ControllerServlet.java to BookList.jsp

Result:



Passing a Book to BookList.jsp

ControllerServlet.java

```
package com.pluralsight;
import …

public class ControllerServlet extends HttpServlet {
    …
    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response) throws… {

        request.setAttribute("book_title","1984");

    }
}
```

First, we're going to set an attribute on the request with the key "book_title" and the value "1984".

Forwarding a Request with **RequestDispatcher**

ControllerServlet.java

```
public class ControllerServlet extends HttpServlet {
    …
    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response) throws… {
        request.setAttribute("book_title","1984");
        RequestDispatcher dispatcher = request
                .getRequestDispatcher("BookList.jsp");
        dispatcher.forward(request, response);
    }
}
```

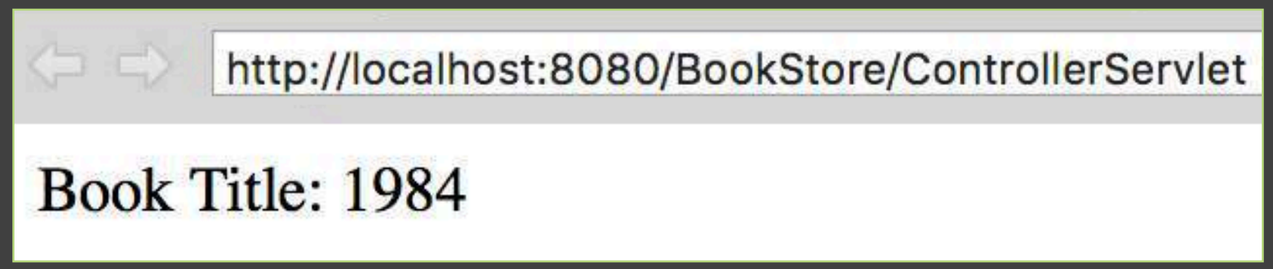
Then we want to create a RequestDispatcher that points to our BookList.jsp page. And then forward our request to that page.

Displaying the Book in BookList.jsp

BookList.jsp

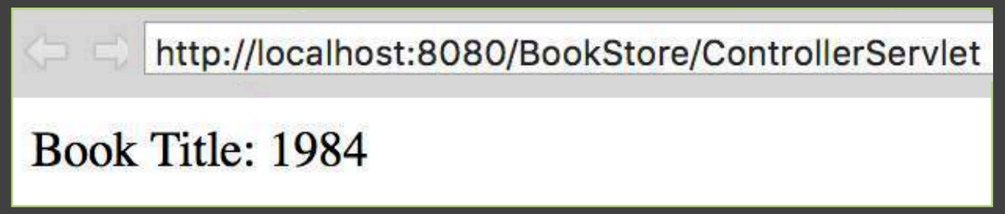
```
…
<body>
<%
  String book_title =
  (String)request.getAttribute("book_title");
%>

<%=  "Book Title: " + book_title%>
</body>
</html>
```

Since we passed in an attribute with the key "book_title" we can get that attribute in this page.



```
…<html>…<body>
<%
  String book_title =
  (String)request.getAttribute("book_title");
%>
<p> "Book Title: " + ${book_title} </p>
</body></html>
```

We can access the attribute book_title from the request directly using EL.



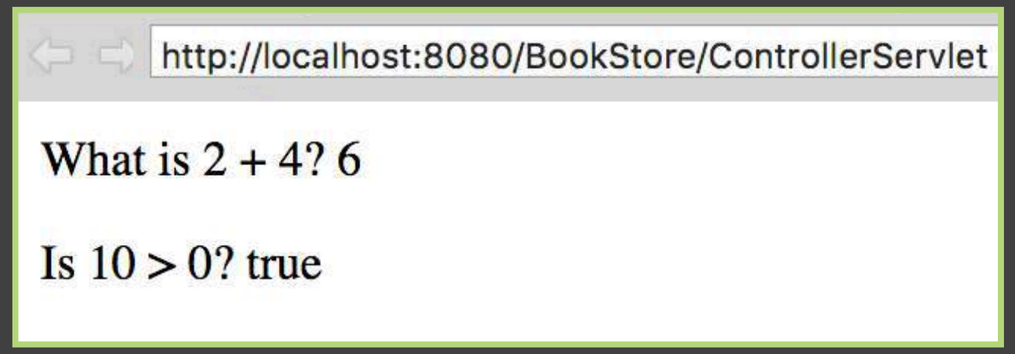
The JSP Expression Language (EL)

Allows us to access implicit objects like the request object and its attributes with shorter syntax.

The EL syntax is **${** *expression language code* **}**.

To access the "book_title" attribute we set in the ControllerServlet we type **${book_title}**.

```
…<html>…<body>

<p> "What is 2 + 4? " + ${2 + 4} </p>
<p> "Is 10 > 0? " + ${10 > 0} </p>

</body></html>
```



The JSP Expression Language (EL)

Also allows arithmetic or logical expressions with shorter notation.

To evaluate 2 + 4, we type **`${2 + 4}`**.

To evaluate 10 > 0, we type **`${10 > 0}`**.

Displaying a List of Book Titles

We Want to Display a Table of Books

http://localhost:8080/BookStore/books/*

Book Store

Add New Book

List of Books

Title	Author	Price
To Kill a Mockingbird	Harper Lee	5.5
1984	George Orwell	4.5
Frankenstein	Mary Shelly	4.0

We're going to display a table of books with their Title, Author, and Price.

Creating a List of Book Titles

ControllerServlet.java

Let's say we have a List of Book Titles in our Servlet.
How can we display them all in our JSP page?

```
… public class ControllerServlet extends HttpServlet {
  private static final long serialVersionUID = 1L;
   private ArrayList<String> bookTitles = new ArrayList<String>();

   public ControllerServlet() {
       // Add books to our ArrayList
       bookTitles.add("To Kill a Mockingbird");
       bookTitles.add("1984");
       bookTitles.add("Frankenstein");
   }
```

Passing a List of Book Titles to BookList.jsp

ControllerServlet.java

```
… protected void doGet(HttpServletRequest request,
                       HttpServletResponse response) … {

    request.setAttribute("book_titles", bookTitles);
  }

…
```

Setting the attribute on the request to be the `ArrayList bookTitles` ***we just created.***

Forwarding the Request to BookList.jsp

ControllerServlet.java

```
… protected void doGet(HttpServletRequest request,
                       HttpServletResponse response) … {

    request.setAttribute("book_titles", bookTitles);
    RequestDispatcher dispatcher =
                 request.getRequestDispatcher("BookList.jsp");
    dispatcher.forward(request, response);
…
```

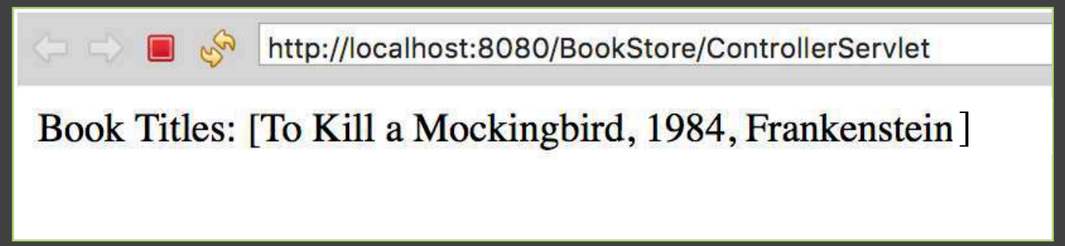
Displaying a List of Book Titles

We want to display each book title on a separate line - to do this we'll need a loop.

BookList.jsp

```
…<html>…<body>

<p>  "Book Titles: " + ${book_titles} </p>
</body></html>
```

Using JSP EL to access the request attribute directly.



The Long Way to Display Books in a For Loop

BookList.jsp

```
…<html>…<body>
<%
  ArrayList<String> titles =
  (ArrayList<String>)request.getAttribute("book_titles");

  for (int i = 0; i<titles.size(); i++) {
%>
    <p>Book: <%=titles.get(i) %></p>
<% } %>
</body></html>
```

Starting the for loop

HTML and a JSP expression tag to display each book

Ending the for loop

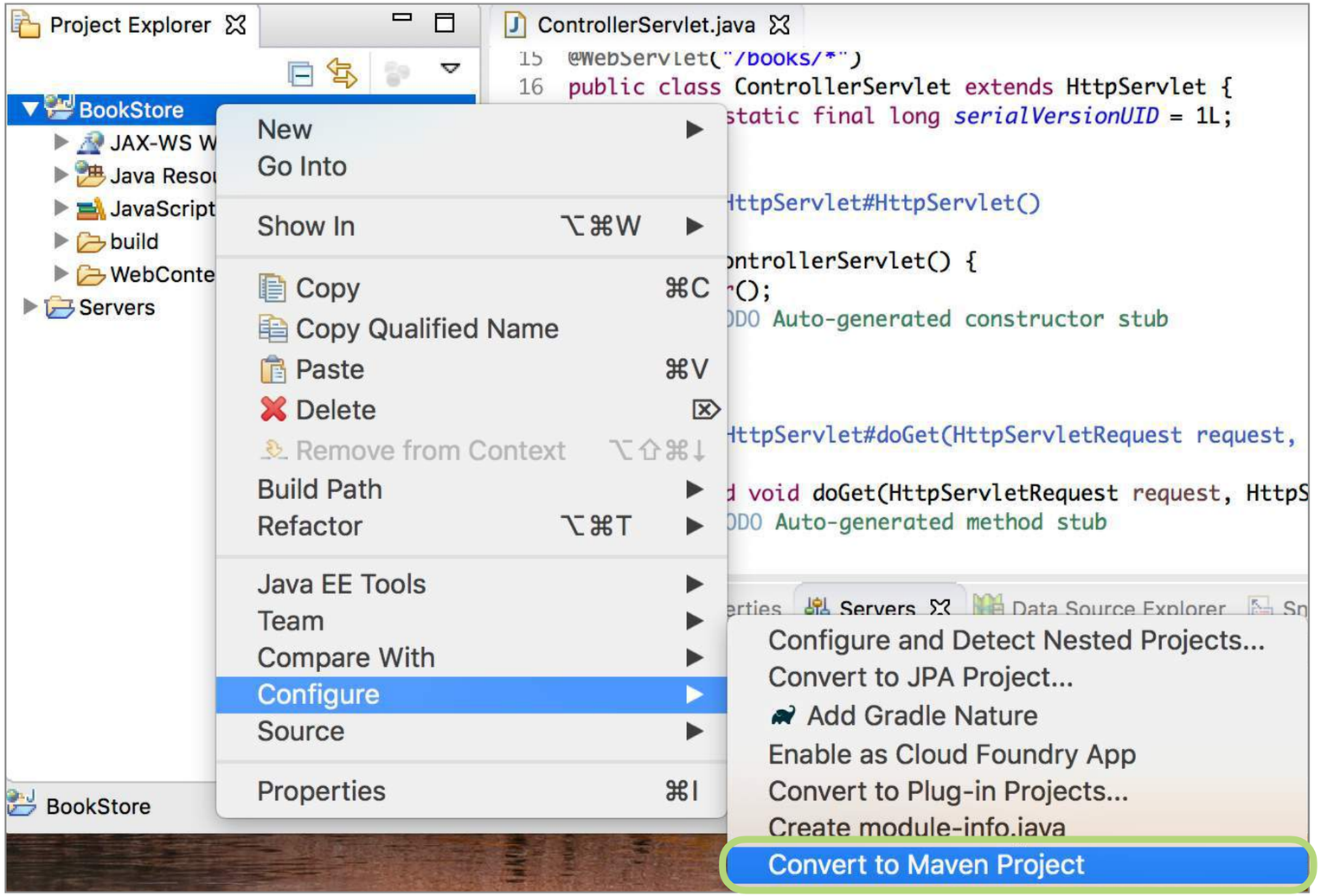
This would be easier with EL and a for-each loop using the JSP Standard Tag Library!

Book: 1984

Book: To Kill a Mockingbird

Book: Frankenstein

Using Maven to Add Dependencies



We want for-each loops, to use those we need the JSP Standard Tag Library (JSTL).

We can use Maven to add JSTL as a dependency!

This will add a pom.xml file to our project.

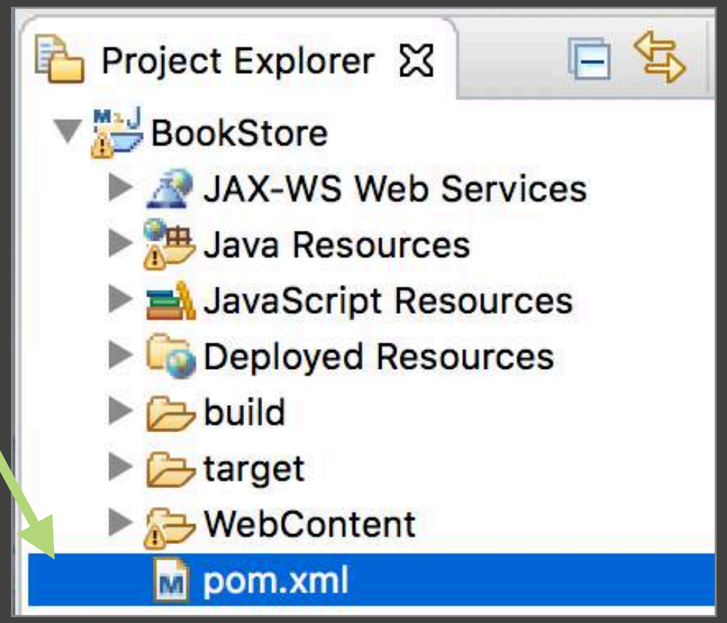
Using the JSP Standard Tag Library (JSTL)

Step 1 - We need to include JSTL in our Maven pom.xml file as a dependency. To get this dependency code we can search for JSTL on mvnrepository.com.

pom.xml

```
<project …> …
  <dependencies>…
  <dependency>
    <groupId>jstl</groupId>
    <artifactId>jstl-impl</artifactId>
    <version>1.2</version>
  </dependency>
</dependencies> …
```

The Maven pom.xml file



Using the JSP Standard Tag Library (JSTL)

Step 2 - We need to define a tag lib directive that will allow our page to use the JSTL core library with a custom tag that looks like HTML.

BookList.jsp

```
<%@ page language="java" …%>
<%@ taglib uri = "http://java.sun.com/jsp/jstl/core" prefix = "c" %>
…
<html>
…
```

Displaying a Book in a Foreach Loop

BookList.jsp

```
…<html>…<body>

<c:forEach items="${book_titles}" var="item">
    <p>Book: ${item} </p>
</c:forEach>

</body>
</html>
```

Collection of items

Name of the variable for the current item in the iteration

Display each item

Ending the forEach loop

Book: 1984

Book: To Kill a Mockingbird

Book: Frankenstein

Summary

- Introduction to JSP Pages
- Running Java in a JSP File
- Passing Data to a JSP File
- Using the JSP Expression Language (EL)
- Adding Maven to our Project
- Using JSTL to Display a List of Book Titles

Hands On: Creating a Basic Java Web App

CREATING A JAVA BOOK CLASS

Sarah Holderness
PLURALSIGHT AUTHOR
@dr_holderness

Creating A Java Book Class

Overview

The main components of a Java MVC web app:

- ☑ Servlets (Controller)
- ☐ Java Server Pages (View)
- ☐ Database (Model)

We still have work to do here!

We Want to Display Books and Their Properties

http://localhost:8080/BookStore/books/*

Book Store

Add New Book

List of Books

Title	Author	Price
To Kill a Mockingbird	Harper Lee	5.5
1984	George Orwell	4.5
Frankenstein	Mary Shelly	4.0

If we had a Book class we could group the title, author, and price properties together.

Then we could send a list of Book Objects to the BookList.jsp page.

Displaying a List of Books and Their Properties

BookList.jsp

```
…<html>…<body>

<c:forEach items="${book_list}" var="item">
    <p>Book: ${item} </p>
</c:forEach>
</body>
</html>
```

Right now we have this.

Book: (To Kill a Mockingbird, Harper Lee, 5.5)

Book: (1984, George Orwell, 4.5)

Book: (Frankenstein, Mary Shelly, 4.0)

We want to add each property to a Table Column.

Creating a Table of Books and Their Properties

BookList.jsp

```
… <html>…<body> …
<% … %>
    <div class="container">
        <h1>Book Store</h1>
        <table>
            <caption>List of Books</caption>
            <tr>
                <th>Title</th>
                <th>Author</th>
                <th>Price</th>
            </tr>
…
```

We'll display a Book Store Header.

We'll also create a table where the first row is the column headings.

A Row for each Book and its Properties

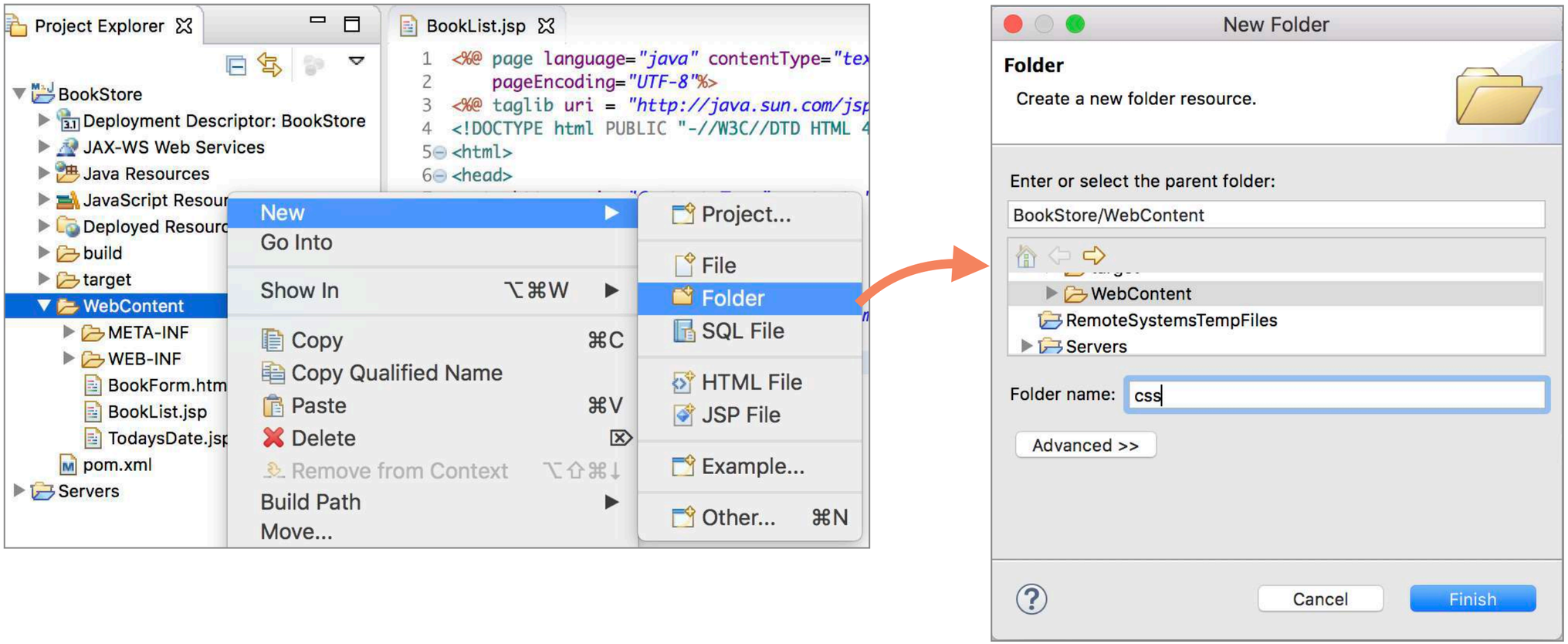
BookList.jsp

```
…
        <table>
            …
        <c:forEach items="${book_list}" var="item">
            <tr>
                <td> ${item.getTitle()  }  </td>
                <td> ${item.getAuthor() } </td>
                <td> ${item.getPrice()  } </td>
            </tr>
        </c:forEach>
…
```

We keep our forEach loop so we can create a row for each Book.

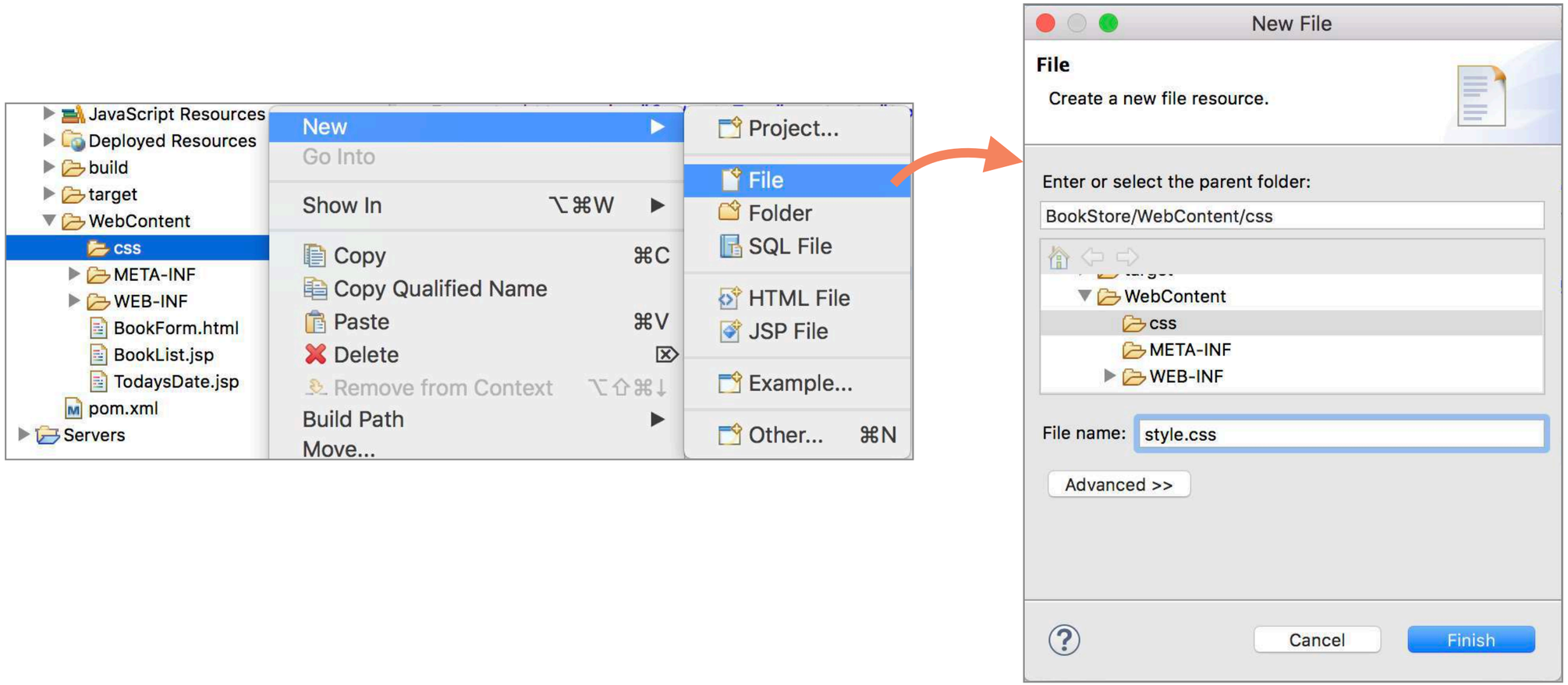
The cells will then display the title, author, and price from each Book object.

Adding a CSS File



First, we'll add a sub-folder in WebContent called css.

Adding a CSS File



Then, we'll add a style.css file.

The Resulting Table of Books and their Properties

http://localhost:8080/BookStore/books/*

Book Store

List of Books

Title	Author	Price
To Kill a Mockingbird	Harper Lee	5.5
1984	George Orwell	4.5
Frankenstein	Mary Shelly	4.0

Routing to Pages in ControllerServlet

Routing to Different Pages from ControllerServlet

http://localhost:8080/BookStore/books/*

Book Store

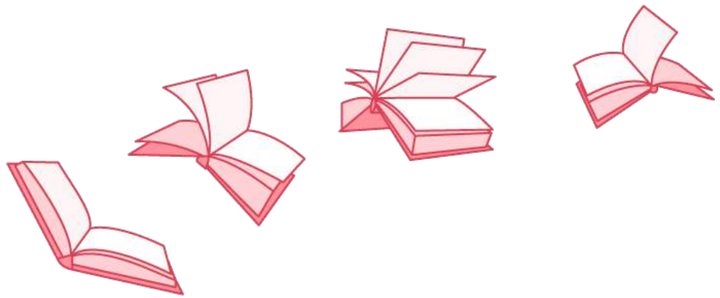
Add New Book

List of Books

Title	Author	Price
To Kill a Mockingbird	Harper Lee	5.5
1984	George Orwell	4.5
Frankenstein	Mary Shelly	4.0

We want to show the BookList.jsp page by default, but also add a link to add a new book - that will take us to our BookForm.

Then when we submit the Form, we'll come back to this page and see the book in the list.



Creating Links in BookList.jsp

BookList.jsp

```
…<html>…<body>…
<% … %>
   <div class="container">
        <div class="links">
            <h1><a href="list">Book Store</a></h1>
            <h2><a href="new">Add New Book</a></h2>
        </div>
        <table>
             <capture>List of Books</capture>
             <tr>
                  …
             </tr>
…
```

We'll create add navigation links to the top of BookList.jsp and BookForm.html.

We'll use relative URLs to "list" *and* "new" *and route here in our ControllerServlet later.*

Updating the Form's **action** attribute

BookForm.html

```
…<html>…<body>…

    <div class="container">
        <div class="links">
            <h1><a href="list">Book Store</a></h1>
            <h2><a href="new">Add New Book</a></h2>
        </div>
        <form name="book_form" method="post" action="insert">
        <caption><h2>New Book Form</h2></caption>
            <p><label>Title:</label>
            <input type="text" name="booktitle" /></p>…
```

We'll use a relative URL to `"insert"` *and create an action for that in our ControllerServlet later.*

3 Different Actions

http://localhost:8080/BookStore/books/*

Book Store

Add New Book

List of Books

Title	Author	Price
To Kill a Mockingbird	Harper Lee	5.5
1984	George Orwell	4.5
Frankenstein	Mary Shelly	4.0

http://localhost:8080/BookStore/BookForm.jsp

Book Store

Add New Book

New Book Form

Title:

Author:

Price:

Submit

(3) Insert New Book and Forward to...

(1) List All Books

(2) Show Book Form

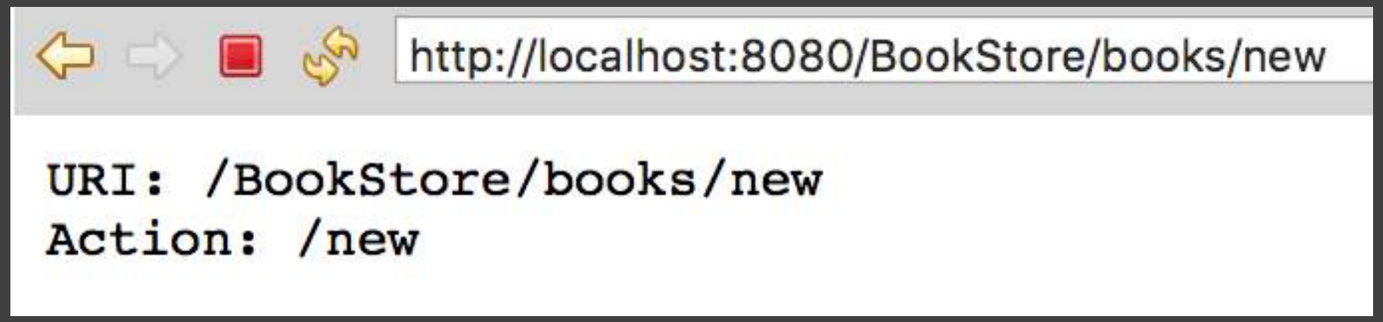
Routing to Different Pages in **ControllerServlet.java**

```
… protected void doGet(HttpServletRequest request,
                       HttpServletResponse response) throws… {
    String uri = request.getRequestURI();
```

`getRequestURI()` *gives the complete URI.*

```
    String action = request.getPathInfo();
```

`getPathInfo()` *gives the extra path information after the URI, used to access your Servlet.*




```
} …
```

Routing to Different Pages in **ControllerServlet.java**

```
… protected void doGet(HttpServletRequest request,
                       HttpServletResponse response) throws… {
    String action = request.getPathInfo();
     if (action.equals("/new")) {
        This is where we will forward to the BookForm.html page.
     }
     else {
        This is where we will forward to the BookList.jsp page.
     }
  } …
```

Routing to Different Pages in **ControllerServlet.java**

```
… protected void doPost(HttpServletRequest request,
                        HttpServletResponse response)throws… {
    String action = request.getPathInfo();
     if (action.equals("/insert")) {
        This is where we add a new book to our ArrayList with the submitted information.
     }
  } …
```

Creating Action Methods in **ControllerServlet.java**

```
… protected void doGet(HttpServletRequest request,
                       HttpServletResponse response) throws… {
    String action = request.getPathInfo();
     if (action.equals("/new")) {
        addBook(request, response);
     }
     else {
        listBooks(request, response);
     }
  } …
```

We'll add a separate method for each action case.

Routing to Different Pages in **ControllerServlet.java**

```
… private void addBook(HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException {

    RequestDispatcher dispatcher =
                       request.getRequestDispatcher("/BookForm.html");
    dispatcher.forward(request, response);
} …
```

addBook() simply forwards to the BookForm page

Routing to Different Pages in **ControllerServlet.java**

```
… private void listBooks(HttpServletRequest request,

          throws ClassNotFoundException, SQLException,
                 ServletException, IOException {

      request.setAttribute("book_list", bookList);
      RequestDispatcher dispatcher =
                       request.getRequestDispatcher("/BookList.jsp");
      dispatcher.forward(request, response);
  } …
```

listBooks() also sets the bookList as an attribute

Routing to Different Pages in **ControllerServlet.java**

```
… protected void doPost(HttpServletRequest request,
                        HttpServletResponse response) throws… {
    String action = request.getPathInfo();
     if (action.equals("/insert")) {
        insertBook(request, response);
     }
  } …
```

In doPost(), if the action is insert, call insertBook().

Routing to Different Pages in **ControllerServlet.java**

```
… private void insertBook(HttpServletRequest request,
                          HttpServletResponse response)
        throws ClassNotFoundException, SQLException,
               ServletException, IOException {
    String title = request.getParameter("booktitle");
    String author = request.getParameter("bookauthor");
    String priceString = request.getParameter("bookprice");

    Book newBook = new Book(title, author, Float.parseFloat(priceString));
    bookList.add(newBook);

    response.sendRedirect("list");
} …
```

First, we get the submitted parameters

Then create and add a new Book to the list.

Finally, redirect back to the Book List page.

Summary

- Adding a Book Class
- Displaying Books in a Table
- Adding CSS
- Routing to Different Pages from a Servlet

Hands On: Creating a Basic Java Web App

CONNECTING TO OUR MODEL

Sarah Holderness
PLURALSIGHT AUTHOR
@dr_holderness

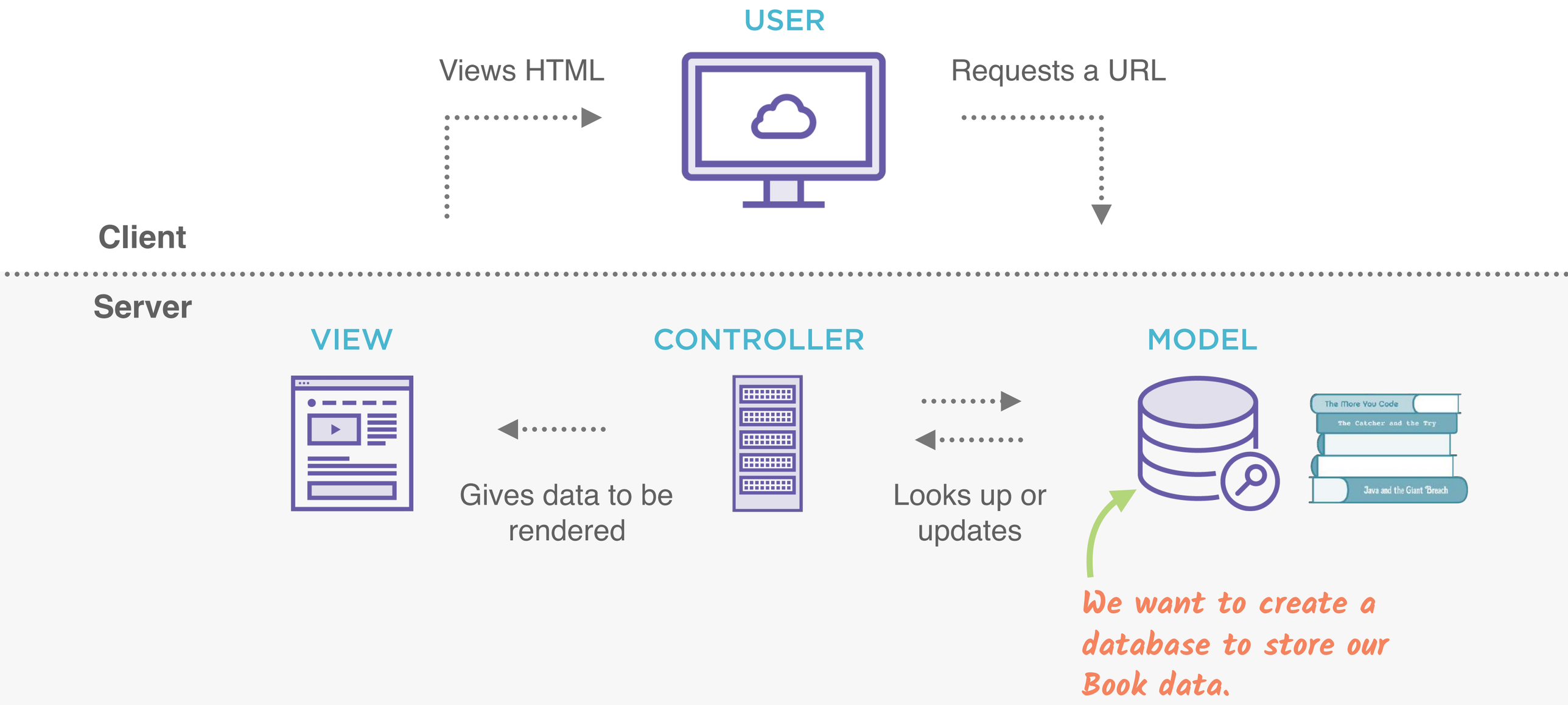
Connecting to Our Model

Overview

The main components of a Java MVC web app:

- ☑ Servlets (Controller)
- ☑ Java Server Pages (View)
- ☐ Database (Model)

The Java MVC Design



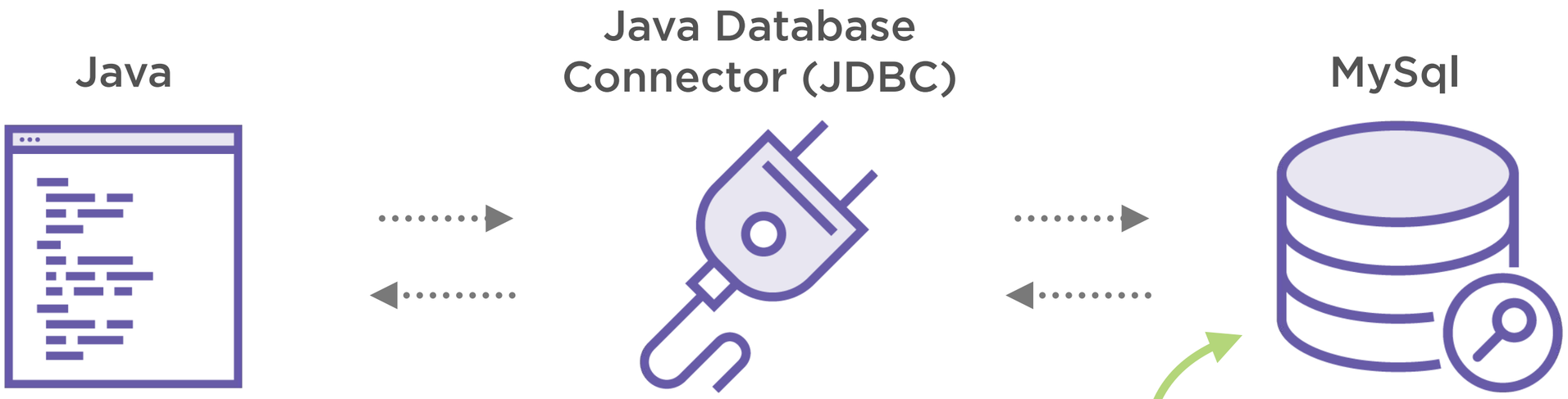
Environment Setup

```
mysql> SELECT * FROM book;

+----+-------+----------------+-------+
| id | title | author         | price |
+----+-------+----------------+-------+
|  1 | 1984  | George Orwell  |  5.00 |
+----+-------+----------------+-------+
```

MYSQL DATABASE

Install MySql locally and create a Book Database and table.

We Need a Java Database Connector



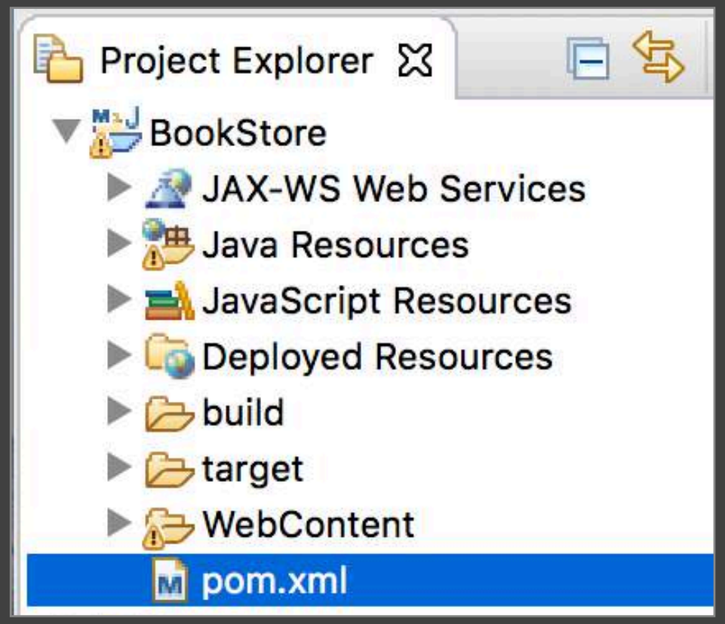
We need to get a MySQL to JDBC Connector library.

We can use Maven to manage dependencies like this one.

pom.xml

```
<project …> …
  <dependencies>…
   <dependency>
     <groupId>mysql</groupId>
     <artifactId>mysql-connector-java</artifactId>
     <version>6.0.6</version>
   </dependency>
</dependencies> …
```

The Maven pom.xml file



Adding Dependencies to the **pom.xml** File

If you search for mysql connector java on mvnrepository.com it will show you how to fill out the Maven dependency section. Then just copy and paste.

Maven | Gradle | SBT | Ivy | Grape | Leiningen | Buildr

```
<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
<dependency>
   <groupId>mysql</groupId>
   <artifactId>mysql-connector-java</artifactId>
   <version>6.0.6</version>
</dependency>
```

A Book Data Access Object (BookDAO) Class

BookDAO.java

```
… public class BookDAO {
    private String jdbcURL = "jdbc:mysql://localhost:3306/book_store";
    private String jdbcUsername = "root";
    private String jdbcPassword = "";
    private Connection jdbcConnection;

    public void connect()  {

    }
    public void disconnect()  {

    }
}
```

Our database connection variables.

The methods we'll create to connect and disconnect from the MySQL database.

Connecting to the Database

BookDAO.java

```
… public class BookDAO {
    private String jdbcURL = "jdbc:mysql://localhost:3306/book_store";
    private String jdbcUsername = "root";
    private String jdbcPassword = "";
    private Connection jdbcConnection;

    public void connect()  {
        jdbcConnection = DriverManager.getConnection(
                                jdbcURL, jdbcUsername, jdbcPassword);

        System.out.println("Connection Established to MySQL Database");
    }
}
```

The DriverManager *class will get a* Connection *using the URL, username, and password.*

Connecting to the Database

BookDAO.java

```
… public void connect()  {
      try {
         if (jdbcConnection == null || jdbcConnection.isClosed()) {
               jdbcConnection = DriverManager.getConnection(
                                 jdbcURL, jdbcUsername, jdbcPassword);

               System.out.println("MySQL Connection Established");
         }
      } catch (SQLException e) {
       e.printStackTrace();
    }
  }
```

Only need to connect if not already connected.

Wrapping in a `try/catch` *block because these SQL methods can throw* `SQLExceptions`.

Disconnecting from the Database

BookDAO.java

```
…
  public void disconnect()  {
      jdbcConnection.close();
  }
```

Simply close the Connection *to disconnect.*

Disconnecting from the Database

BookDAO.java

```
…
  public void disconnect()  {
    try {
      if (jdbcConnection != null && !jdbcConnection.isClosed()) {
          jdbcConnection.close();
      }
    } catch (SQLException e) {
      e.printStackTrace();
    }
  }
```

Only disconnect if already connected.

Wrapping in a `try/catch` *block because these SQL methods can throw* `SQLExceptions`*.*

Testing the BookDAO Connection Methods

ControllerServlet.java

```
… public class ControllerServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    private ArrayList<Book> bookList = new ArrayList<Book>();
    private BookDAO bookDAO;

    public ControllerServlet() {
        bookDAO = new BookDAO();
        bookDAO.connect();
        bookDAO.disconnect();

        bookList.add(new Book("To Kill a Mockingbird", "Harper Lee", 5.00f));
        bookList.add(new Book("1984", "George Orwell", 5.00f));
        bookList.add(new Book("Frankenstein", "Mary Shelly", 5.00f));
    }…
```

Create a BookDAO ***Class Variable***

Initialize the BookDAO ***and test*** connect() ***and*** disconnect().

Querying our Database Table

Using Raw SQL to Get All of the Books

Title	Author	Price
1984	George Orwell	4.50
To Kill a Mockingbird	Harper Lee	5.50
...	...	...


```
SELECT * FROM book
```

```
+-------+--------+-------+
| Title | Author | Price |
+-------+--------+-------+
| 1984  | 4.50   | 4.50  |
| Blah  | 5.50   | 5.50  |
```

`SELECT *` *returns all of the columns and will return all of the book rows which is what we want*

Querying Our MySql Database with Java

We can use a `Statement` object (from the java.sql library).

```
Statement statement = jdbcConnection.createStatement();
```

`jdbcConnection` *is our* `Connection` *object to our MySql database*

Querying Our MySql Database with Java

We can use a `Statement` object (from the java.sql library). with the `executeQuery()` method.

Our query will return all of the book rows and we can navigate the resulting rows with our `ResultSet`.

```
Statement statement = jdbcConnection.createStatement();
ResultSet resultSet = statement.executeQuery("SELECT * FROM book");
```

A `ResultSet` exposes the results from a query

We pass our SQL query as a `String` to the `executeQuery()` method

```
ResultSet resultSet = statement.executeQuery("SELECT * FROM book");
```

ResultSet's initial position (i.e. -1)

Title	Author	Price
1984	George Orwell	4.50
To Kill a Mockingbird	Harper Lee	5.50
...	...	...

How a **ResultSet** Works

A `ResultSet` stores query result records in rows and have methods to access and iterate through the records.

```
ResultSet resultSet = statement.executeQuery("SELECT * FROM book");
```

After calling **resultSet.next()**

Title	Author	Price
1984	George Orwell	4.50
To Kill a Mockingbird	Harper Lee	5.50

```
String title = resultSet.getString("title");
```

title *is now "1984"*

How a **ResultSet** Works

A `ResultSet` **lets you access one row at a time with the** `next()` **method.**

A Method to List All Books in the Database

BookDAO.java

```
… public ArrayList<Book> listAllBooks() {
      connect();

  }…
```

First we need to connect to the Database.

A Method to List All Books in the Database

BookDAO.java

```
… public ArrayList<Book> listAllBooks() {
      connect();
      ArrayList<Book> bookList = new ArrayList<>();
```

Then we want to create an ArrayList ***of type*** Book ***to store our results.***

```
      return bookList;
  }…
```

A Method to List All Books in the Database

BookDAO.java

```
… public ArrayList<Book> listAllBooks() {
      connect();
      ArrayList<Book> bookList = new ArrayList<>();
      Statement statement = jdbcConnection.createStatement();
      ResultSet resultSet = statement.executeQuery("SELECT * FROM book");
```

We can create a `Statement` *to execute our query, and store the results in a* `ResultSet`.

```
      return bookList;
  }…
```

A Method to List All Books in the Database

BookDAO.java

```
… public ArrayList<Book> listAllBooks() {
      connect();
      ArrayList<Book> bookList = new ArrayList<>();
      Statement statement = jdbcConnection.createStatement();
      ResultSet resultSet = statement.executeQuery("SELECT * FROM book");

      while (resultSet.next()) {

      return bookList;
  }…
```

Our loop will go through each item in the `ResultSet` ***until*** `next()` ***returns*** `false`.

A Method to List All Books in the Database

BookDAO.java

```
… public ArrayList<Book> listAllBooks() {
      connect();
      ArrayList<Book> bookList = new ArrayList<>();
      Statement statement = jdbcConnection.createStatement();
      ResultSet resultSet = statement.executeQuery("SELECT * FROM book");

      while (resultSet.next()) {
         String title = resultSet.getString("title");
         String author = resultSet.getString("author");
         float price = resultSet.getFloat("price");


      }
      return bookList;
  }…
```

Getting the current book row's properties.

A Method to List All Books in the Database

BookDAO.java

```
… public ArrayList<Book> listAllBooks() {
      connect();
      ArrayList<Book> bookList = new ArrayList<>();
      Statement statement = jdbcConnection.createStatement();
      ResultSet resultSet = statement.executeQuery("SELECT * FROM book");

      while (resultSet.next()) {
          String title = resultSet.getString("title");
          String author = resultSet.getString("author");
          float price = resultSet.getFloat("price");

          Book book = new Book(title, author, price);
          bookList.add(book);
      }
      return bookList;
  }…
```

Creating a Book object and adding it to the resulting list.

A Method to List All Books in the Database

BookDAO.java

```
… public ArrayList<Book> listAllBooks() {
      connect();
      ArrayList<Book> bookList = new ArrayList<>();
      Statement statement = jdbcConnection.createStatement();
      ResultSet resultSet = statement.executeQuery("SELECT * FROM book");
      while (resultSet.next()) {
          String title = resultSet.getString("title");
          String author = resultSet.getString("author");
          float price = resultSet.getFloat("price");
          Book book = new Book(title, author, price);
          bookList.add(book);
      }
      resultSet.close();
      statement.close();
      disconnect();
      return bookList;
  }…
```

To cleanup, we want to close the ResultSet, close the Statement, and disconnect from the database.

A Method to List All Books in the Database

BookDAO.java

```
… public ArrayList<Book> listAllBooks() {
      connect();
      ArrayList<Book> bookList = new ArrayList<>();
      Statement statement = jdbcConnection.createStatement();
      ResultSet resultSet = statement.executeQuery("SELECT * FROM book");
      while (resultSet.next()) {
          String title = resultSet.getString("title");
          String author = resultSet.getString("author");
          float price = resultSet.getFloat("price");
          Book book = new Book(title, author, price);
          bookList.add(book);
      }
      resultSet.close();
      statement.close();
      disconnect();
      return bookList;
  }…
```

We also want to wrap all of this in a `try/catch` *block, since some of the methods throw a* `SQLException`.

Calling our BookDAO listAllBooks() Method

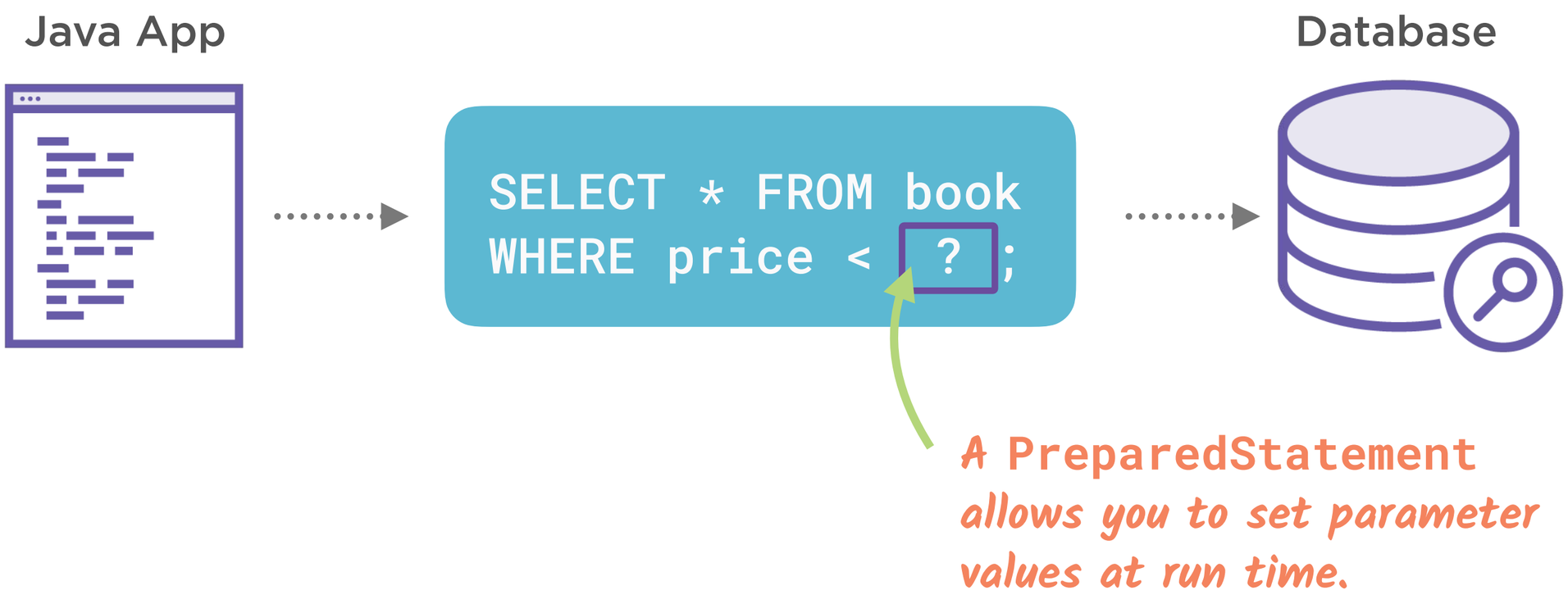
ControllerServlet.java

```
… private void listBooks(HttpServletRequest request,
                         HttpServletResponse response)
          throws ServletException, IOException {

     request.setAttribute("book_list", bookList);

     ArrayList<Book> bookList = bookDAO.listAllBooks();
     request.setAttribute("book_list", bookList);

     RequestDispatcher dispatcher =
                         request.getRequestDispatcher("/BookList.jsp");
     dispatcher.forward(request, response);
  }…
```

Passing the current books in the database to BookList.jsp.

Using a Prepared Statement

A `PreparedStatement` Can Set Parameter Values



Adding a Parameter to a PreparedStatement

```
String sql = "SELECT * FROM book WHERE price < ?";
```

Create a query String with a ? for each parameter.

```
PreparedStatement statement = jdbcConnection.prepareStatement(sql);
```

Create the PreparedStatement with the query.

Setting the parameter requires a type, here it's Float.

```
statement.setFloat(1, 5.00f);
```

The parameter position in the query.

The value.

A Method to Insert a Book Into the Database

BookDAO.java

```
… public boolean insertBook(Book book) {
        connect();

  }…
```

First we need to connect to the Database.

A Method to Insert a Book Into the Database

BookDAO.java

```
… public boolean insertBook(Book book) {
        connect();
        String sql = "INSERT INTO book (title, author, price) VALUES (?, ?, ?)";
        PreparedStatement statement = jdbcConnection.prepareStatement(sql);
```

We can create a `PreparedStatement` *to execute our query and insert parameters.*

```
    }…
```

A Method to Insert a Book Into the Database

BookDAO.java

```
… public boolean insertBook(Book book) {
        connect();
        String sql = "INSERT INTO book (title, author, price) VALUES (?, ?, ?)";

        PreparedStatement statement = jdbcConnection.prepareStatement(sql);
        statement.setString(1, book.getTitle());
        statement.setString(2, book.getAuthor());
        statement.setFloat(3, book.getPrice());

    }…
```

Setting each parameter to the Book *object's properties.*

A Method to Insert a Book Into the Database

BookDAO.java

```
… public boolean insertBook(Book book) {
        connect();
        String sql = "INSERT INTO book (title, author, price) VALUES (?, ?, ?)";

        PreparedStatement statement = jdbcConnection.prepareStatement(sql);
        statement.setString(1, book.getTitle());
        statement.setString(2, book.getAuthor());
        statement.setFloat(3, book.getPrice());

        statement.executeUpdate();

  }…
```

Inserting the values into the database.

A Method to Insert a Book Into the Database

BookDAO.java

```
… public boolean insertBook(Book book) {
        connect();
        String sql = "INSERT INTO book (title, author, price) VALUES (?, ?, ?)";

        PreparedStatement statement = jdbcConnection.prepareStatement(sql);
        statement.setString(1, book.getTitle());
        statement.setString(2, book.getAuthor());
        statement.setFloat(3, book.getPrice());

        boolean rowInserted = statement.executeUpdate() > 0;

  }…
```

We'll check if the execution was successful to be able to return a boolean.

A Method to Insert a Book Into the Database

BookDAO.java

```
… public boolean insertBook(Book book) {
        connect();
        String sql = "INSERT INTO book (title, author, price) VALUES (?, ?, ?)";

        PreparedStatement statement = jdbcConnection.prepareStatement(sql);
        statement.setString(1, book.getTitle());
        statement.setString(2, book.getAuthor());
        statement.setFloat(3, book.getPrice());

        boolean rowInserted = statement.executeUpdate() > 0;

        statement.close();
        disconnect();
        return rowInserted;
  }…
```

Cleaning up by closing the statement and disconnecting.

A Method to Insert a Book Into the Database

BookDAO.java

```
… public boolean insertBook(Book book) {
        connect();
        String sql = "INSERT INTO book (title, author, price) VALUES (?, ?, ?)";
        PreparedStatement statement = jdbcConnection.prepareStatement(sql);
        statement.setString(1, book.getTitle());
        statement.setString(2, book.getAuthor());
        statement.setFloat(3, book.getPrice());

        boolean rowInserted = statement.executeUpdate() > 0;

        statement.close();
        disconnect();
        return rowInserted;
  }…
```

We want to wrap all of this in a `try/catch` *block, since some of the methods throw a* `SQLException`.

Calling our BookDAO insertBook() Method

ControllerServlet.java

```
… private void insertBook(HttpServletRequest request,
                          HttpServletResponse response) throws … {
    String title = request.getParameter("booktitle");
    String author = request.getParameter("bookauthor");
    String priceString = request.getParameter("bookprice");

    Book newBook = new Book(title, author, Float.parseFloat(priceString));
    bookList.add(newBook);

    bookDAO.insertBook(newBook);
    response.sendRedirect("list");
  }…
```

Inserting the submitted Book object into the database.

Summary

- Use the Java Database Connector to connect to a database
- Perform a query and display the results from a result set
- Use Prepared Statements