Web-based information systems 1

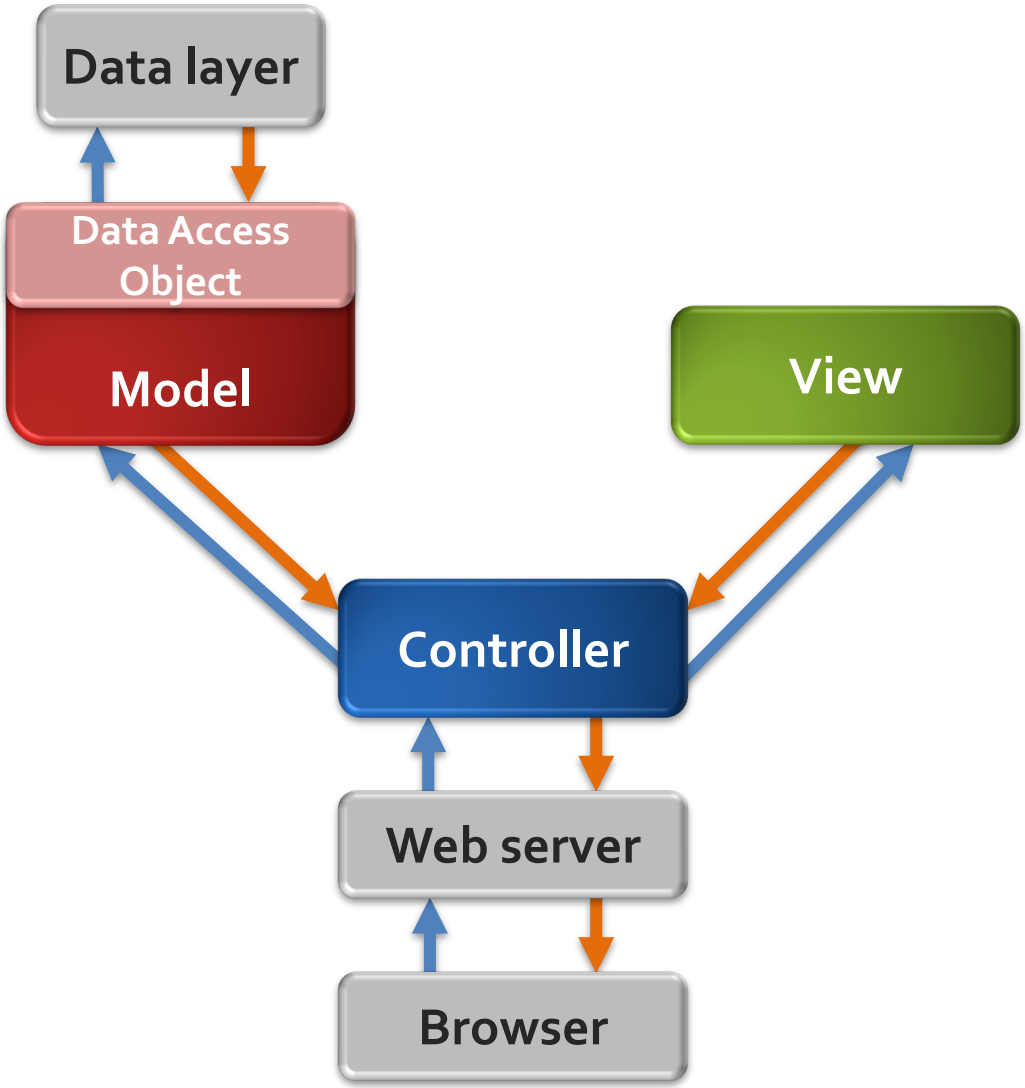
Implementing the MVC pattern in Java web applications

Michiel Noback (NOMI)
Institute for Life Sciences and Technology
Hanze University of Applied Sciences

Introduction

- The topic of this presentation is setting up a clean “MVC“ Java web application.
- This involves working with separated components for different responsibilities

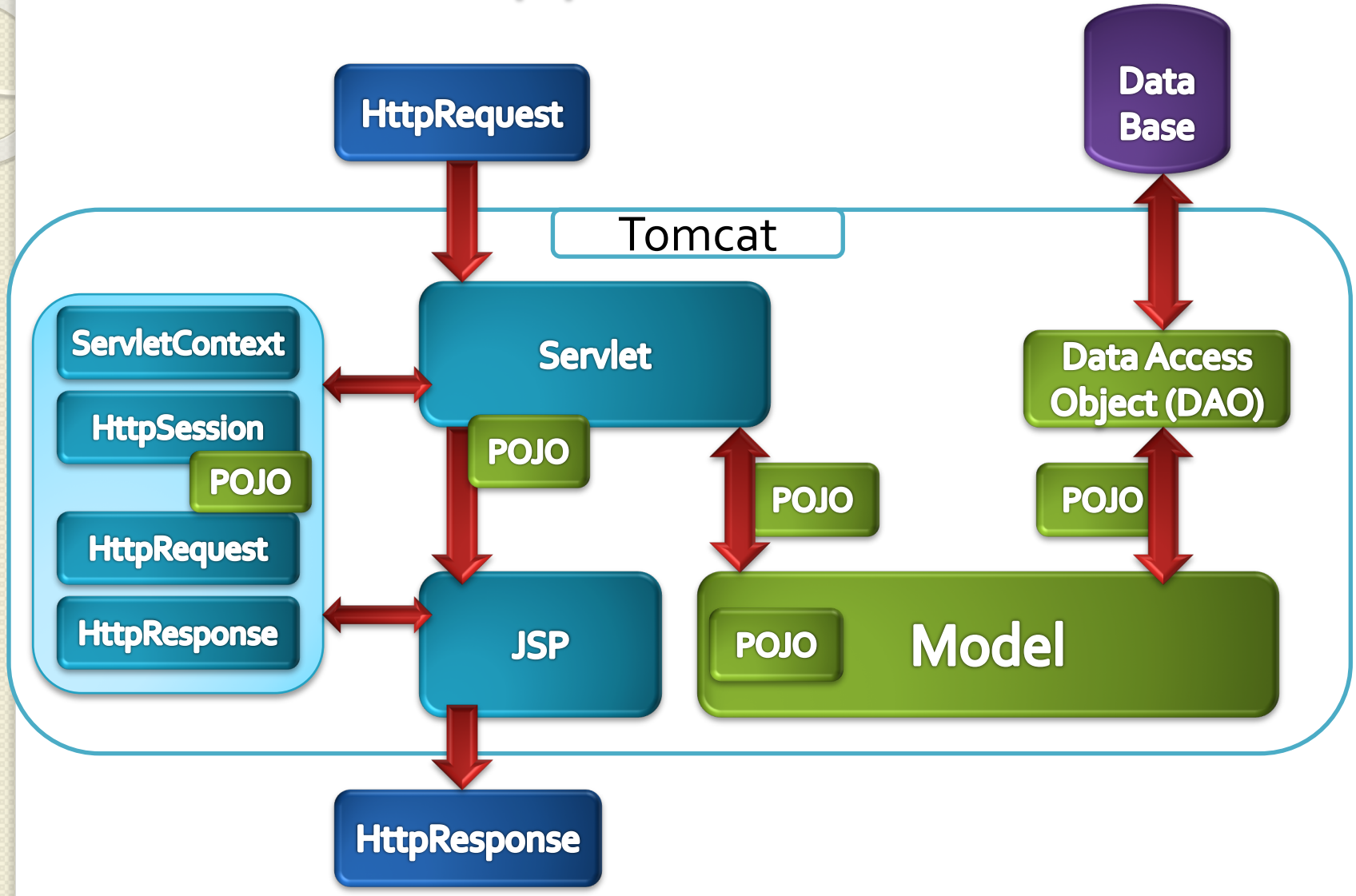
MVC: Model View Control



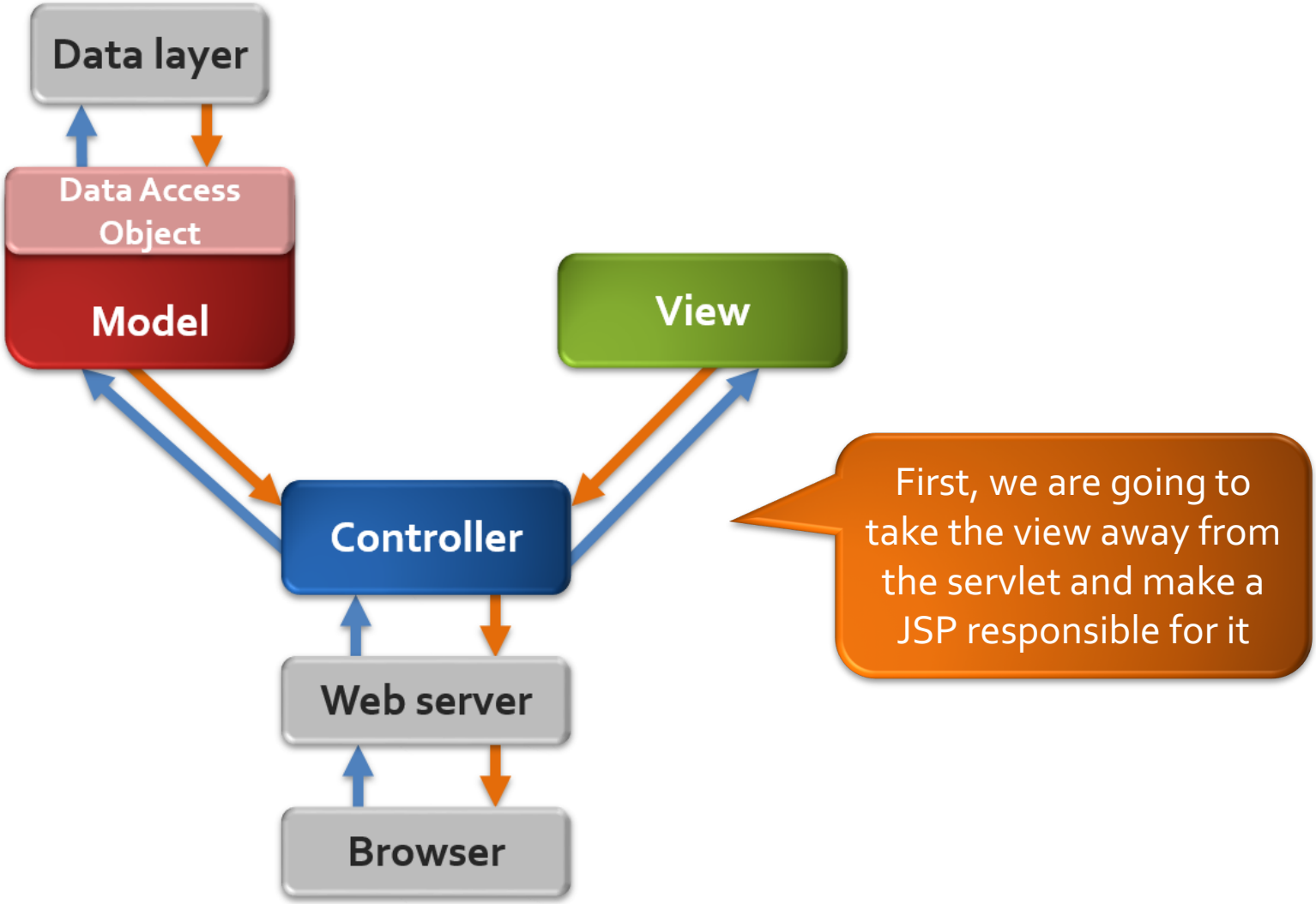
Application components (layers)

- These components are generally recognized in most applications, web-based or not
  - Model: the business-specific logic
  - View: what the user gets to see
  - Controller: receives input or requests (from the user) and determines what action should be taken
  - Data layer: all data is abstracted away behind data access object

Java web app architecture

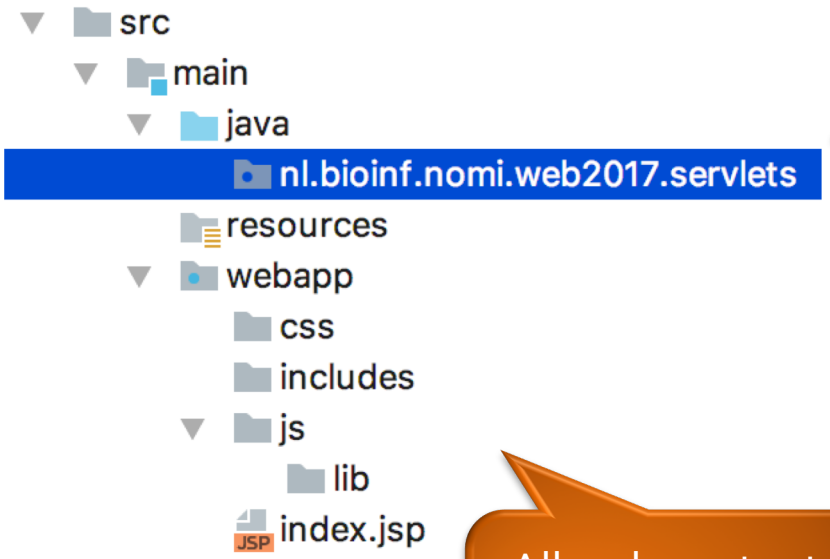


MVC design pattern



Your web project in IntelliJ

- This is how your web project will look like

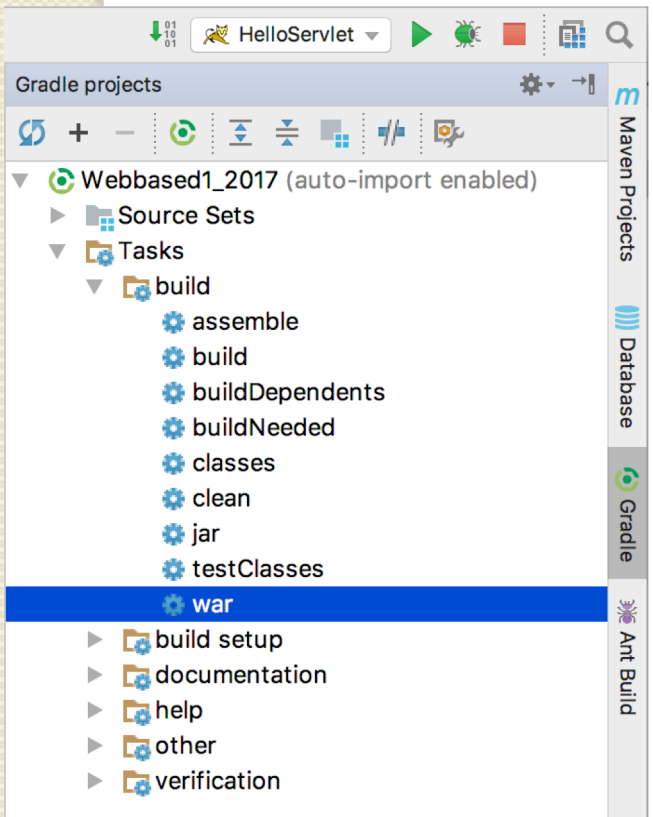


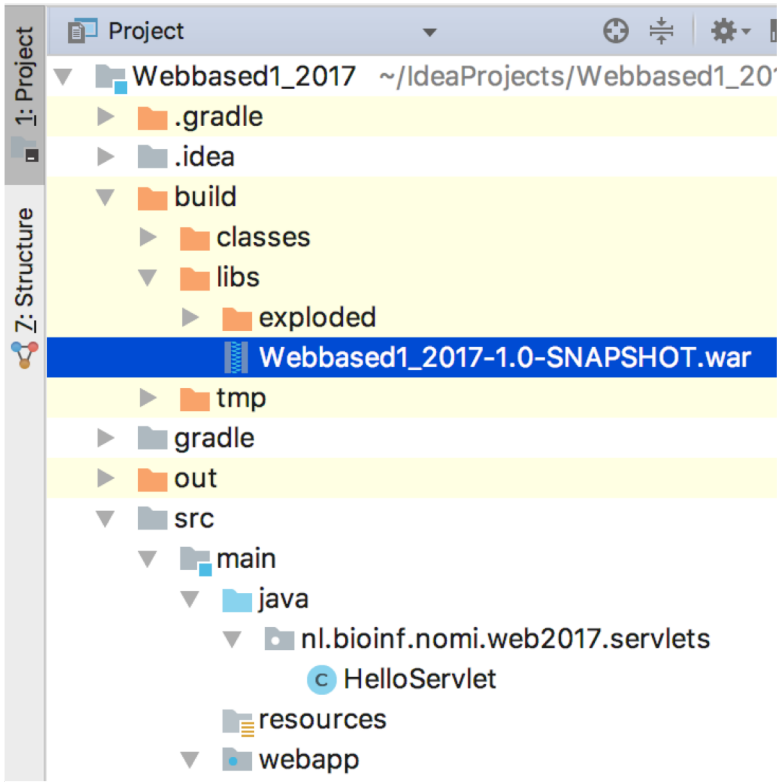
The nl.bioinf packages will hold your servlets and other Java classes before the web app is deployed. When you deploy your web app, classes will be compiled and placed under **webapp/WEB-INF,** and library jar files in a lib folder under this one. So far, WEB-INF does not exist yet.

All web content is organized under **webapp**

Deploying the web app (1)

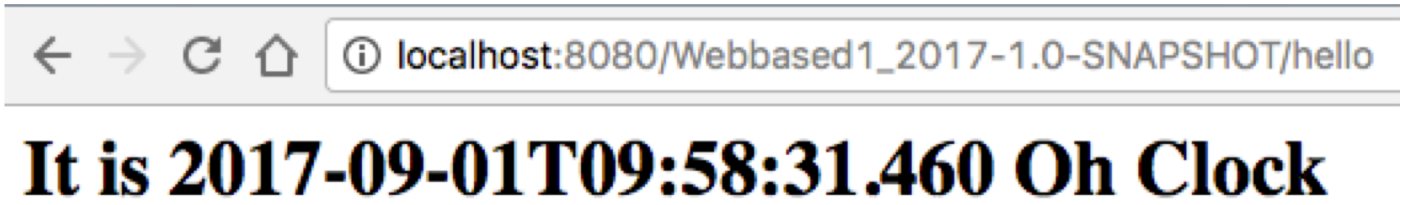
- When you select the 'war' task of the Gradle tool window, a war will be built.
- Double click it and have a look at the **build** folder



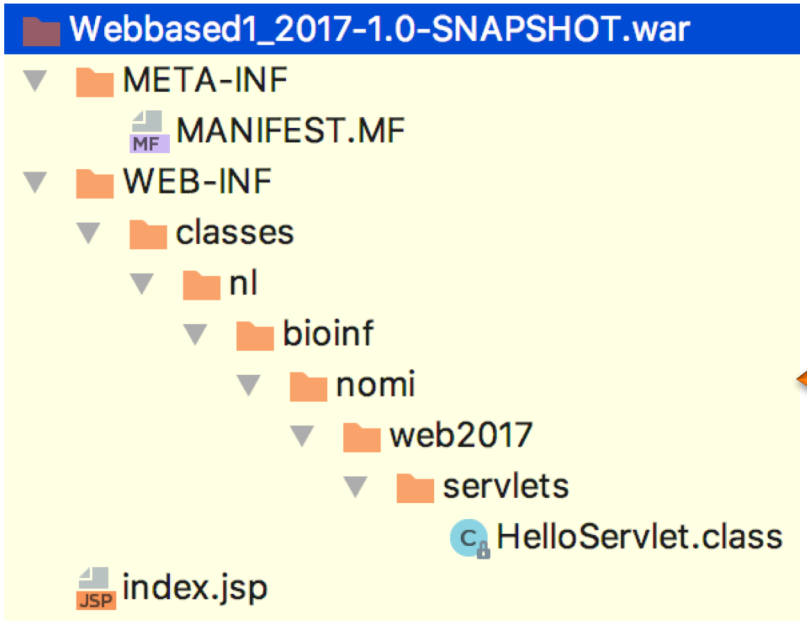


Deploying the web app (2)

- Copy the war archive to the `webapps` folder of your Tomcat installation
- Start up Tomcat using the `/bin/startup.sh` command
- Open a web browser and direct it the the correct location (the name of your war archive minus the extension plus the url of your servlet or jsp)!



Deploying the web app (3)



- The war file is just a zip. You can view its contents
- This shows the architecture of a deployed java web application

Remember your first servlet

- This is how your first servlet (HelloServlet.java) looked like: it is both controller and view
- Let's get the view out of it now

```
@WebServlet(name = "HelloServlet", urlPatterns = "/hello")
public class HelloServlet extends HttpServlet {

    protected void doGet(
            HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        PrintWriter out = response.getWriter();
        LocalDateTime currentTime = LocalDateTime.now();

        out.println("<html>");
        out.println("<head><title>My Hello World Servlet</title>" +
                "</head><body><h1>It is ");
        out.println(currentTime.toString());
        out.println(" O' Clock</h1></body></html>");
    }
}
```

This is as ugly as any old PHP script!

Creating a JSP view to go with the servlet controller

- Right-click on folder webapp, select New → JSP/JSPX
- Give a name (e.g. hello.jsp) and click OK
- The JSP will automatically open in an editor window, with some standard template code already present.

Creating a JSP view

```
<%@ page contentType="text/html;charset=UTF-8"
language="java" %>
<html>
<head>
    <title>My first View</title>
</head>
<body>
    <h1>[[my servlet message comes here]]</h1>
</body>
</html>
```

That's weird! It looks like a plain old HTML file, except for the first funny line.

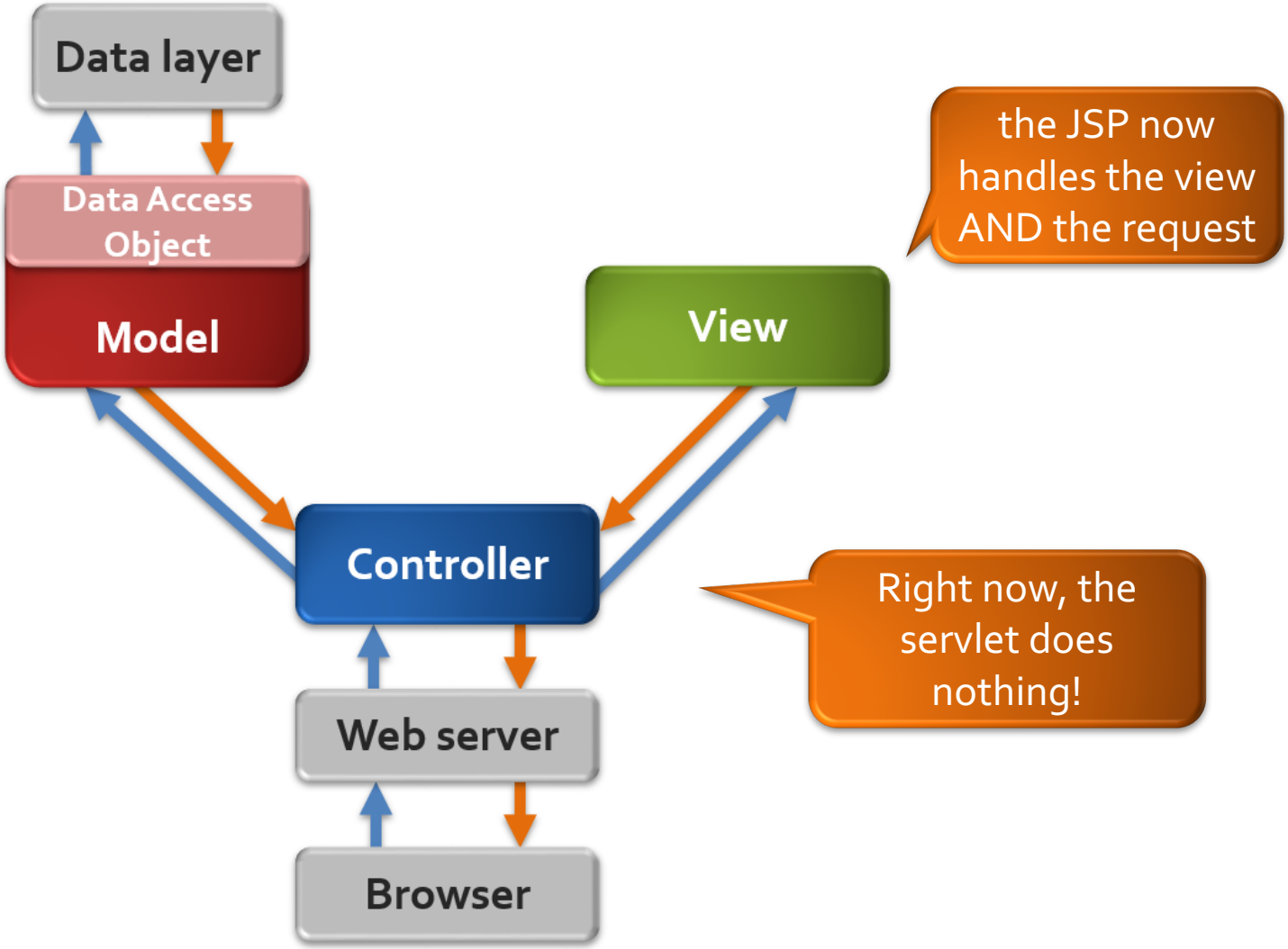
Creating a JSP view

- Run this file, just as you did with the servlet
- This is what your JSP view will yield (exact content may vary) when you have the internal browser selected

Making the view dynamic and MVC

- Now let's make the entire application MVC-true, and add some dynamic content to the JSP view

The current situation

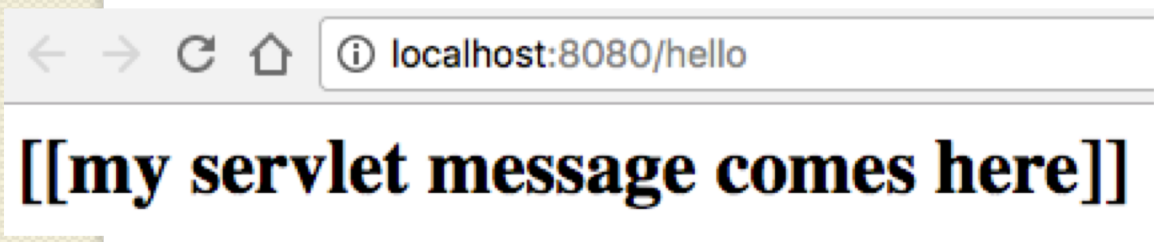


Refactor your servlet to only control

- Take the HelloServlet servlet, rip out the HTML and let it only redirect the request to the JSP

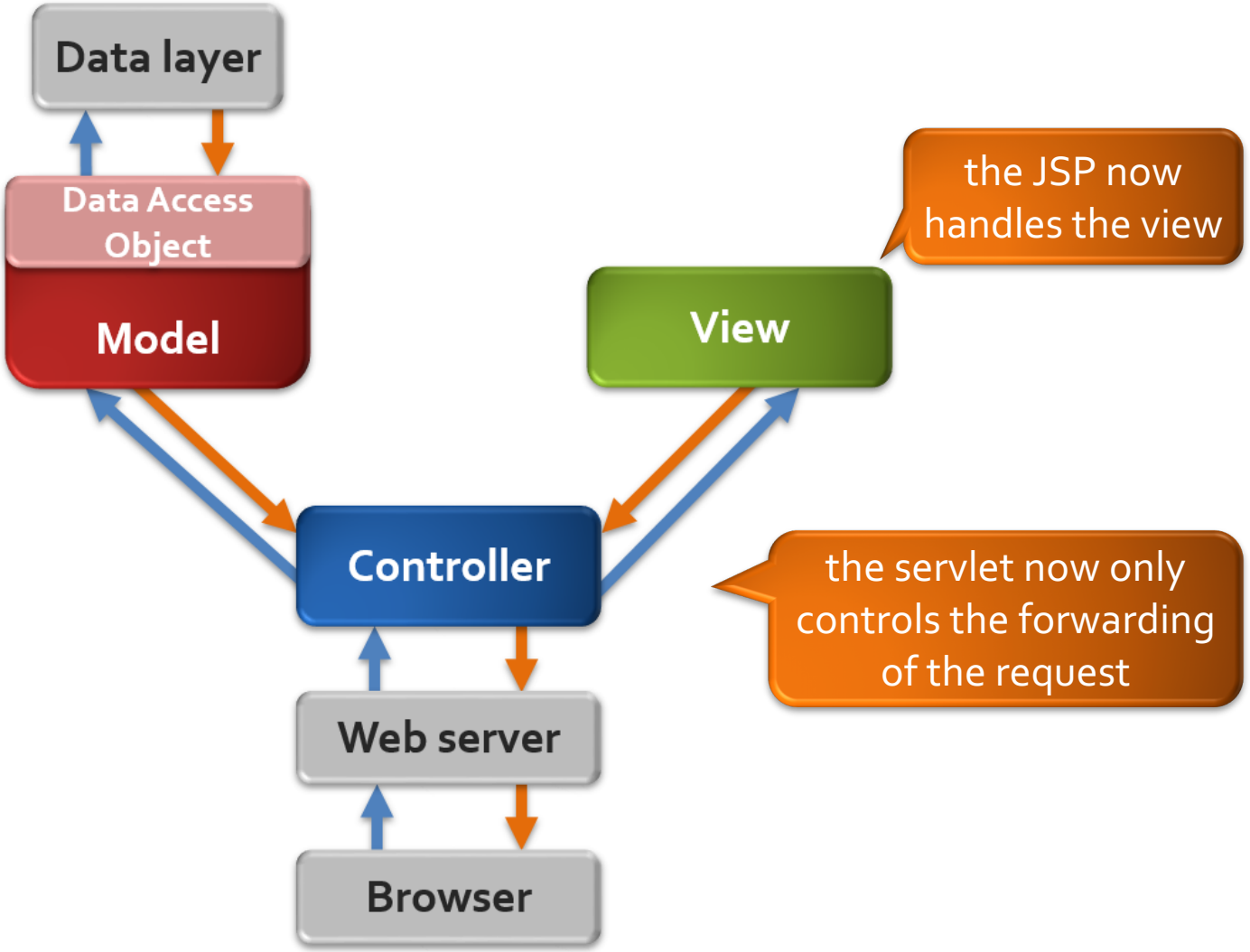
The servlet now does nothing itself; it dispatches the request directly to the jsp.

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ...{
    RequestDispatcher view = request.getRequestDispatcher("hello.jsp");
    view.forward(request, response);
}
```



Same view, different route!

MVC design pattern evolving



Have the servlet pass dynamic content

- Make the HelloWorld servlet attach some dynamic content to the request object

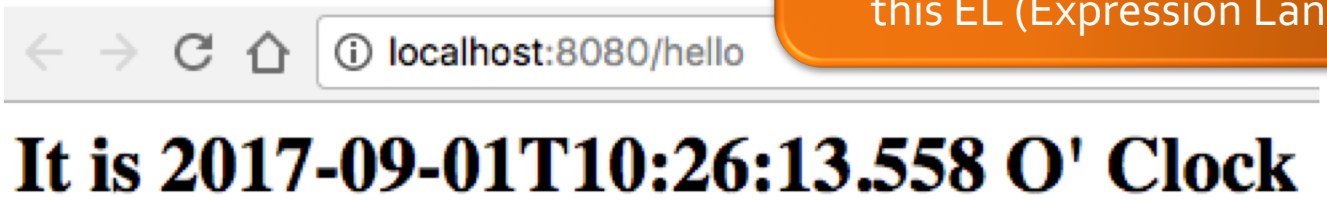
Before dispatching the request object, attach an attribute to it as a key-value pair

```
LocalDateTime currentTime = LocalDateTime.now();
String message = "It is " + currentTime.toString() + " O' Clock";
request.setAttribute("servletmessage", message);
RequestDispatcher view = request.getRequestDispatcher("hello.jsp");
view.forward(request, response);
```

Let the JSP catch the dynamic content

```
<body>
    <h1>${requestScope.servletmessage}</h1>
</body>
```

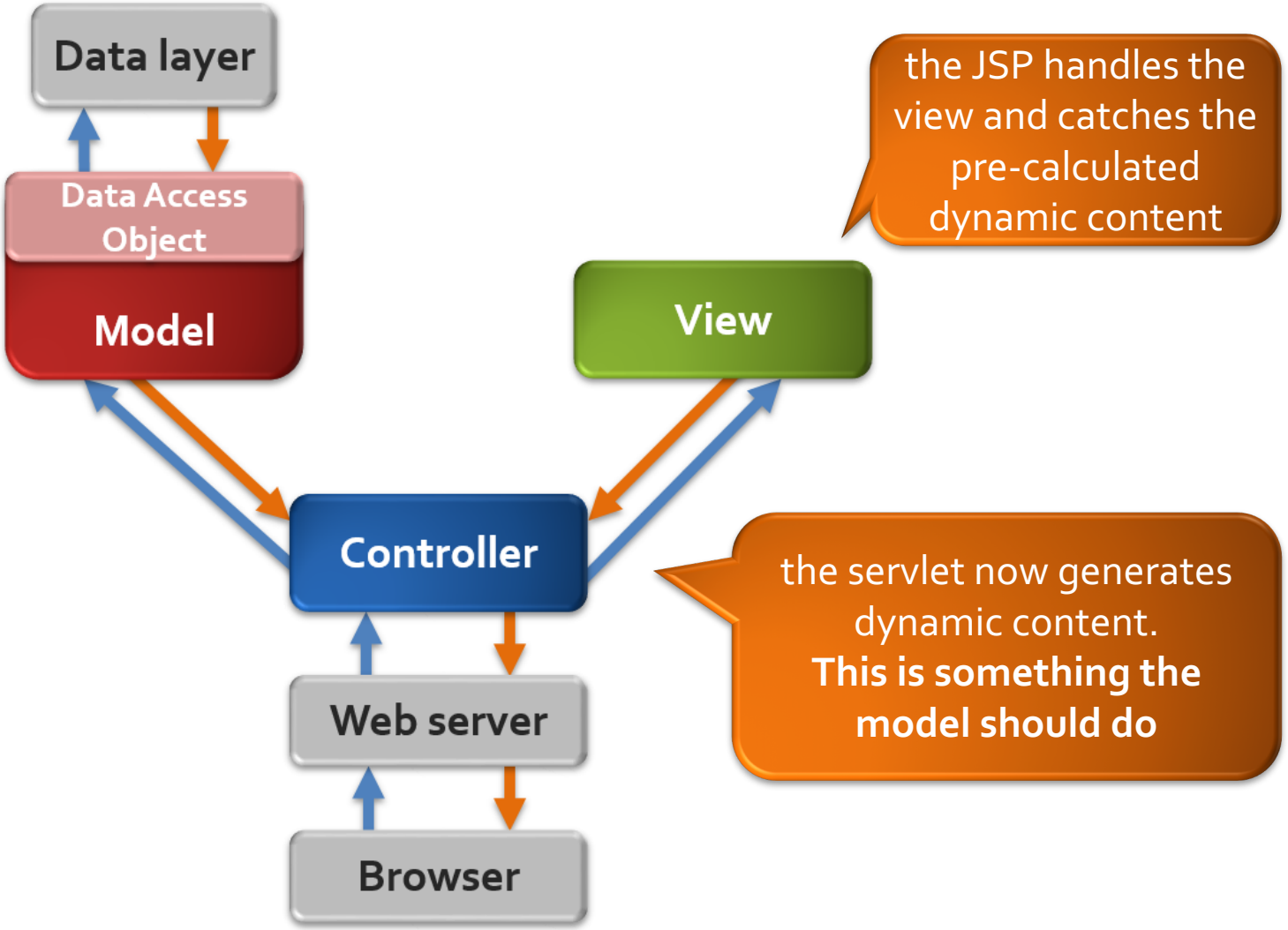
You attached dynamic content to the request object using the key "date". Now you can get it back in the JSP using this EL (Expression Language) code



requestScope is the "EL implicit object" referring to the HttpRequest object
servletmessage is the name of the variable that you attached to it using

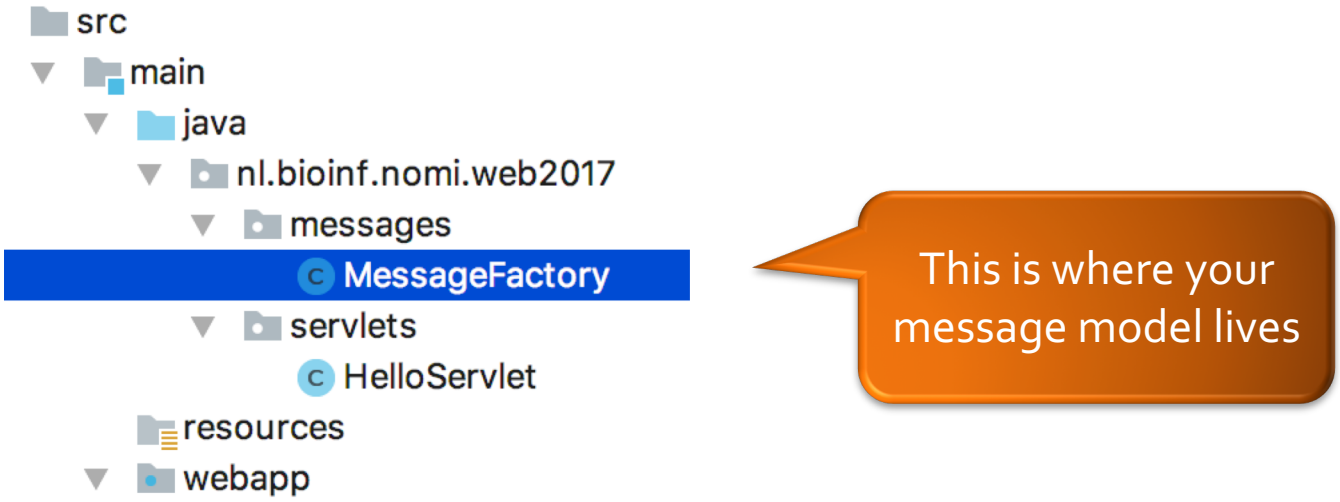
```
request.setAttribute("servletmessage", message);
```

MVC design pattern



Creating a model

- You decide you not only want to display the time, but also a more dedicated greeting, depending on the day of the week
- We'll now create a model to do just that
- First, create a regular Java class in a separate package



Creating a model

```
public class MessageFactory {
    public static String getMessage() {
        LocalDateTime currentTime = LocalDateTime.now();
        DayOfWeek dayOfWeek = currentTime.getDayOfWeek();
        String day = dayOfWeek.getDisplayName(TextStyle.FULL,
Locale.getDefault());
        String message = "Today is a " + day;

        switch (dayOfWeek.getValue()) {
            case 6: {
                return message += "; working on a " + day + "? Get a life!";
            }
            case 7: {
                return message += "; working on a " + day + "? You infidel!";
            }
            default: {
                return message += "; working like an office zombie...";
            }
        }
    }
}
```

Invoking the model

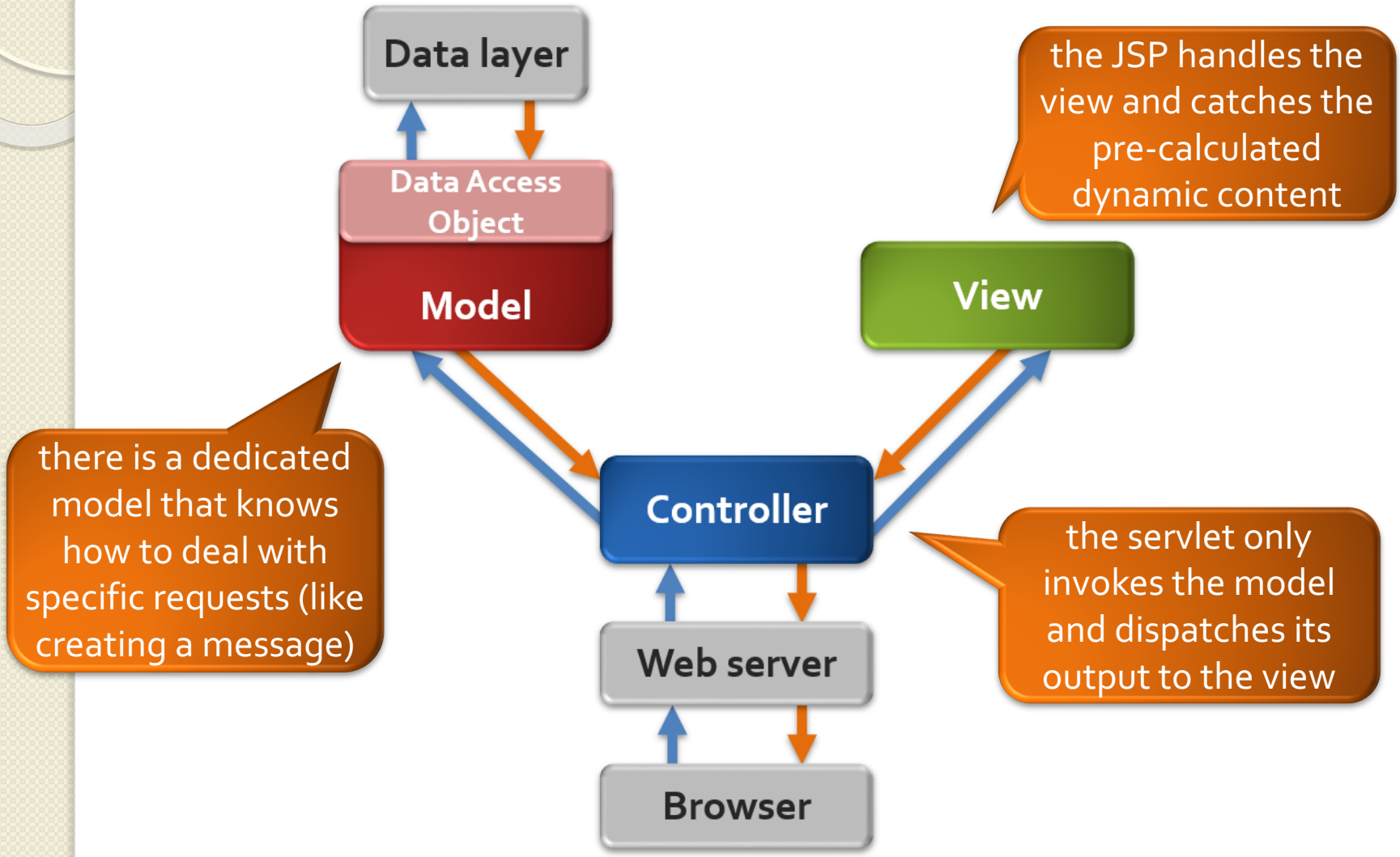
```
request.setAttribute("servletmessage", MessageFactory.getMessage());
```

Servlet: Invoke the model and pass the return value

JSP: Catch the message

```
<h1>${requestScope.servletmessage}</h1>
```

MVC complete



Summary

- Today, you have seen the MVC pattern in action
  - model: plain old java objects (POJOs)
  - view: JSP
  - controller: servlet
- JSP has several techniques to deal with dynamic content generated by the model:
  - Regular Java code (scriptlets)
  - EL (expression language)
  - JSP tags (html-*like* tags)
- We will investigate these in the next session, together with working with http sessions