Web-based information systems 1

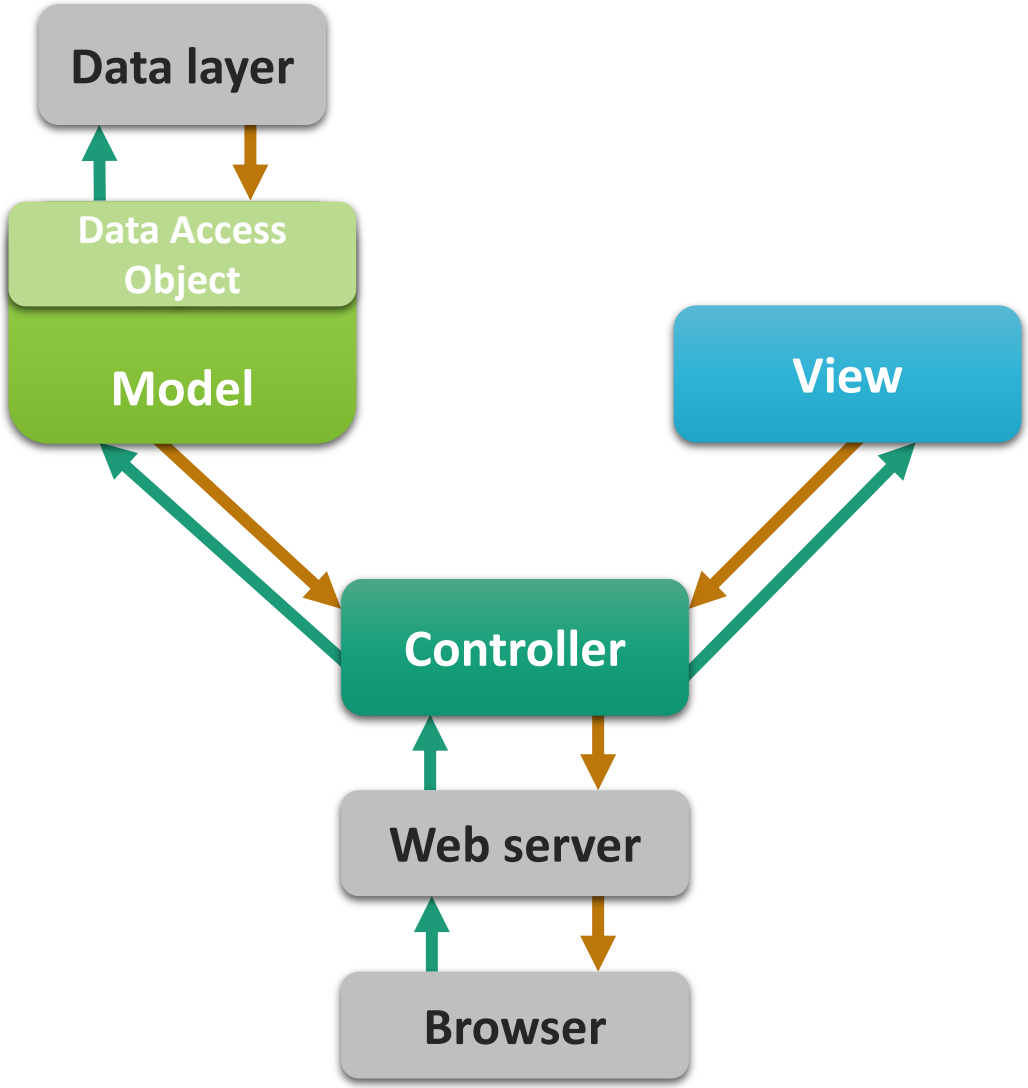
Implementing the MVC pattern

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



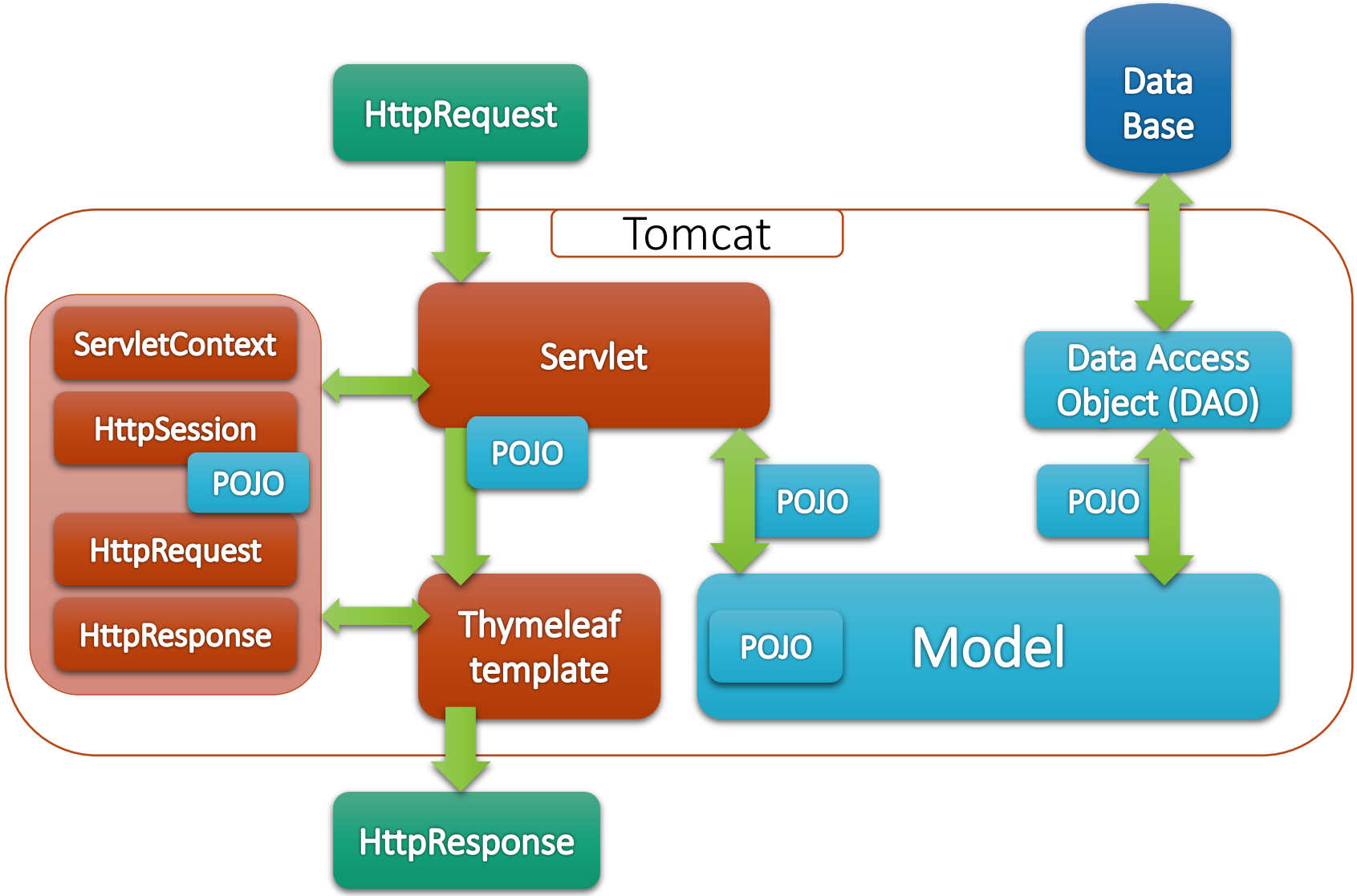
Application components (layers)

- 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

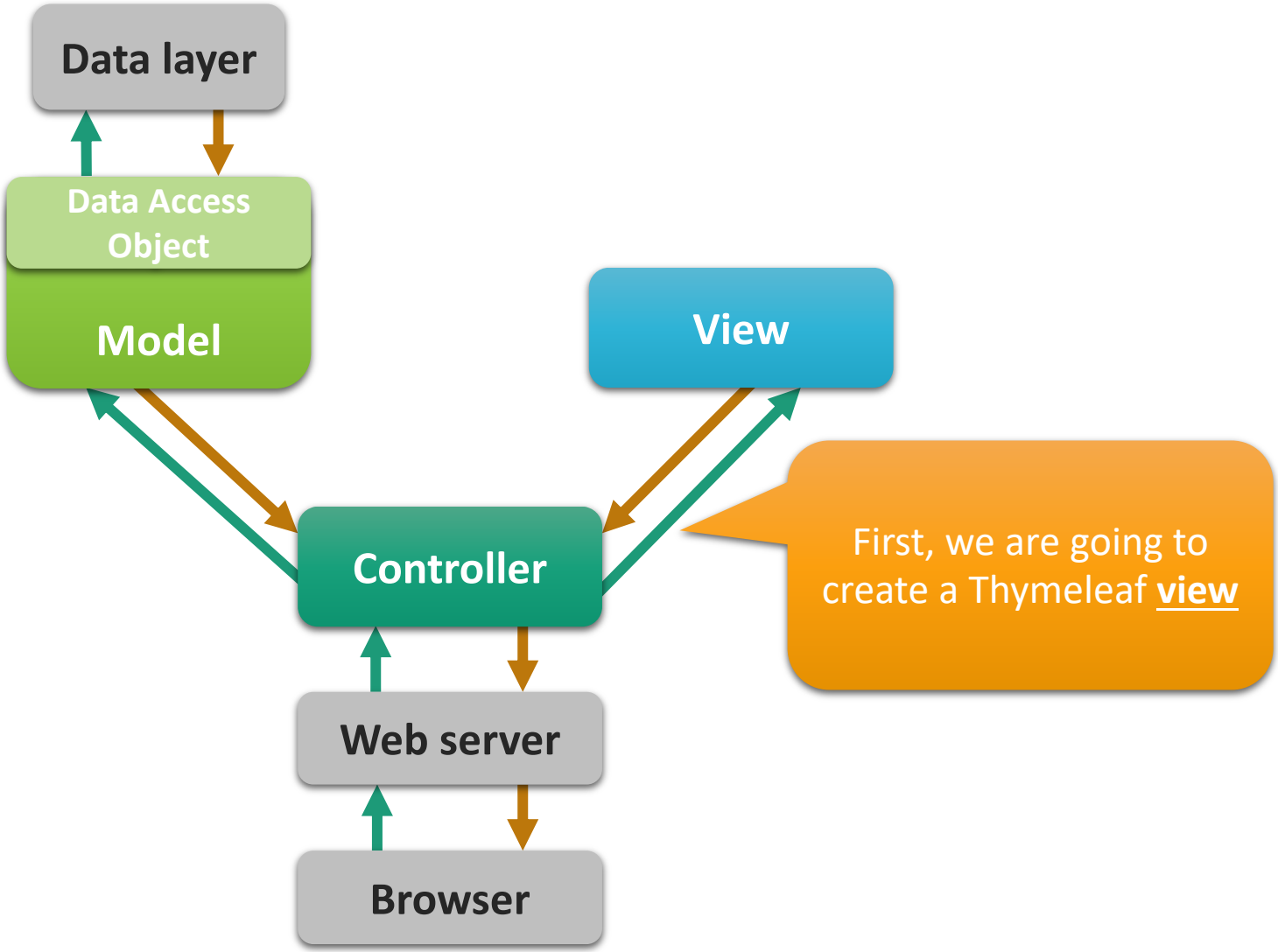
The goal: a phrase of the day

- Choose a topic: bullshit or management
- The tool will present a random phrase from the respective category

Java web app architecture



MVC design pattern



Create a Thymeleaf view

- Right-click on folder `webapp/templates`, select New → html
  - *You can also make your own Thymeleaf template!*
- Name it phrase_of_the_day.html and click OK
- The file will automatically open in an editor window, with some standard template code already present.

phrase_of_the_day.html

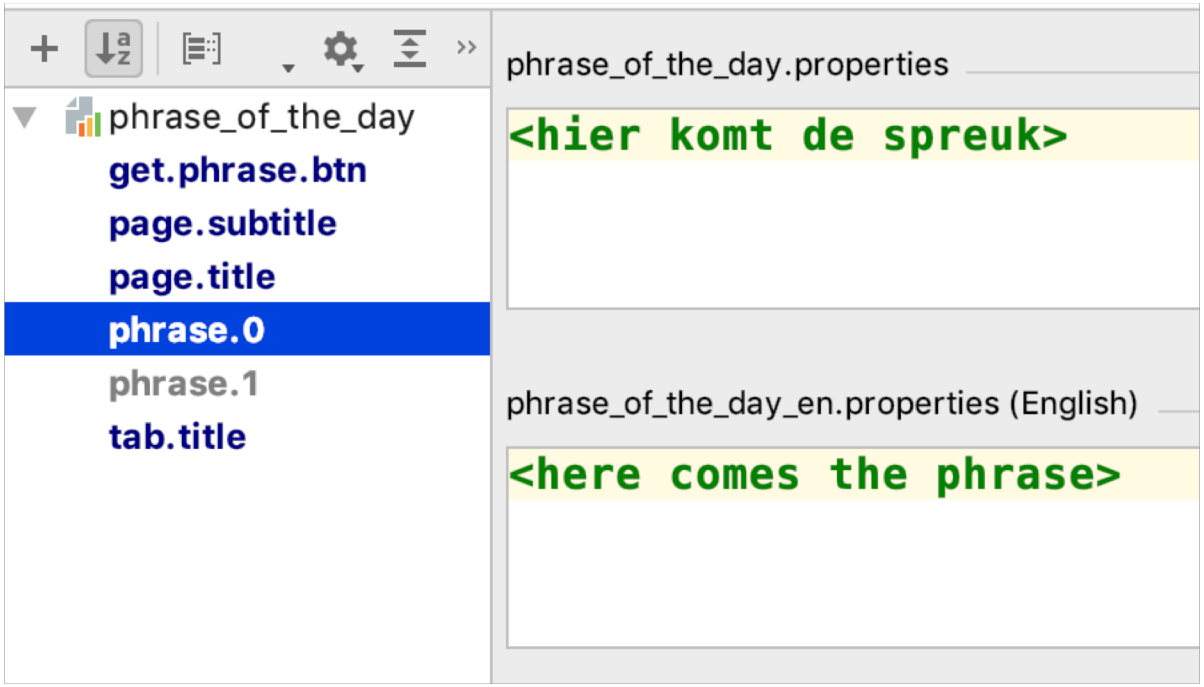
```
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml1-strict-thymeleaf-4.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:th="http://www.thymeleaf.org">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <title th:text="#{tab.title}">_offline_</title>
</head>
<body>
    <h3 th:text="#{page.title}">Offline title</h3>
    <h4 th:text="#{page.subtitle}">Offline subtitle</h4>

    <form action="give.phrase" method="GET">
        <select name="phrase_category">
            <option value="bullshit">Bullshit</option>
            <option value="management">Management</option>
        </select>
        <input type="submit" th:value="#{get.phrase.btn}" value="_offline_"/>
    </form>

    <h4 th:text="#{'phrase.' + ${phrase_num}}">_phrase placeholder_</h4>
</body>
</html>
```

View as static template in browser

- Within the editor pane, hold <shift> and click the icon of the browser you want to view it in
- Do not select “Run phrase_of_…” from the context menu because you will get a 404!

Create resource bundle

- The resource should have as base name `phrase_of_the_day` and should have a default and 'en' locale versions
- You also need entries for the used texts, of course



Create a ‘dummy’ model

- Create a new package called `model`
- Give it some dummy implementation (we’ll put some real code in there later on)

```
public class PhraseFactory {
    public static String getPhrase(String phraseType) {
        //we'll get to this later!
        //You thought I’d return some text here, didn’t you?
        return "1";
    }
}
```

Create a Servlet controller

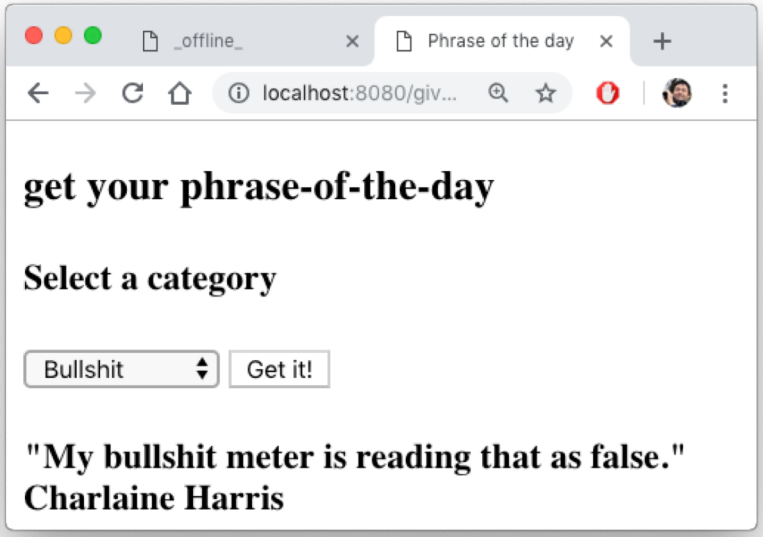
- Create a new Servlet (you should know how) and put the following code in there
  (listing is not complete; see repo for that)

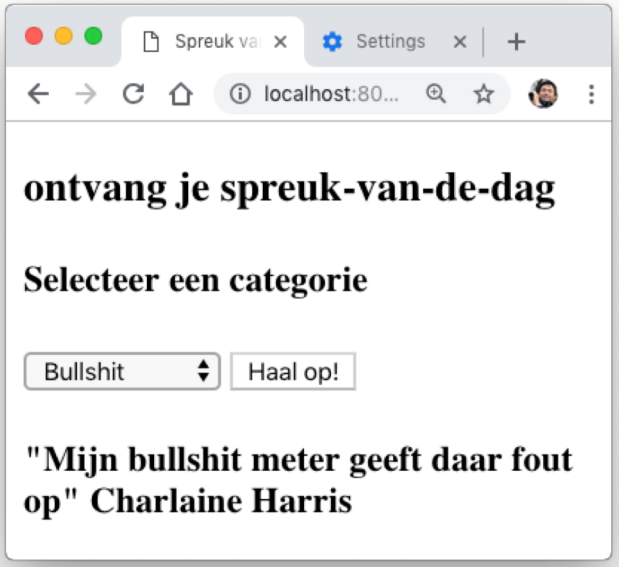
PhraseServlet.java

```
@WebServlet(name = "PhraseServlet", urlPatterns = "/give.phrase")
public class PhraseServlet extends HttpServlet {
    private TemplateEngine templateEngine;
    @Override
    public void init() throws ServletException { //same as WelcomeServlet  }

    protected void doGet(HttpServletRequest request,
            HttpServletResponse response)
            throws ServletException, IOException {
        String phraseType = request.getParameter("phrase_category");
        Locale locale = request.getLocale();
        WebContext ctx = new WebContext(request, response,
                request.getServletContext(), locale);
        if (phraseType != null) {
            String phrase = PhraseFactory.getPhrase(phraseType, locale);
            ctx.setVariable("phrase", phrase);
        } else {
            ctx.setVariable("phrase_num", "0");
        }
        templateEngine.process("phrase_of_the_day", ctx,
                               response.getWriter());
    }
}
```

Test first version of MVC

- Create a new run configuration called ‘phrase’ and point it to **give.phrase**
- Test the result; try the form submit; change your language; refresh





Extend the model

- Implement some nice behavior in PhraseFactory.java
- First the Junit5 test and the the implementation:

```
class PhraseFactoryTest {
    @Test
    void getPhrase() {
        for (int i=0; i<50; i++) {
            int random = Integer.parseInt(PhraseFactory.getPhrase("bullshit"));
            assertTrue(random >= 1 && random <= PhraseFactory.MAX_PHRASE_COUNT);
        }
    }
}
```

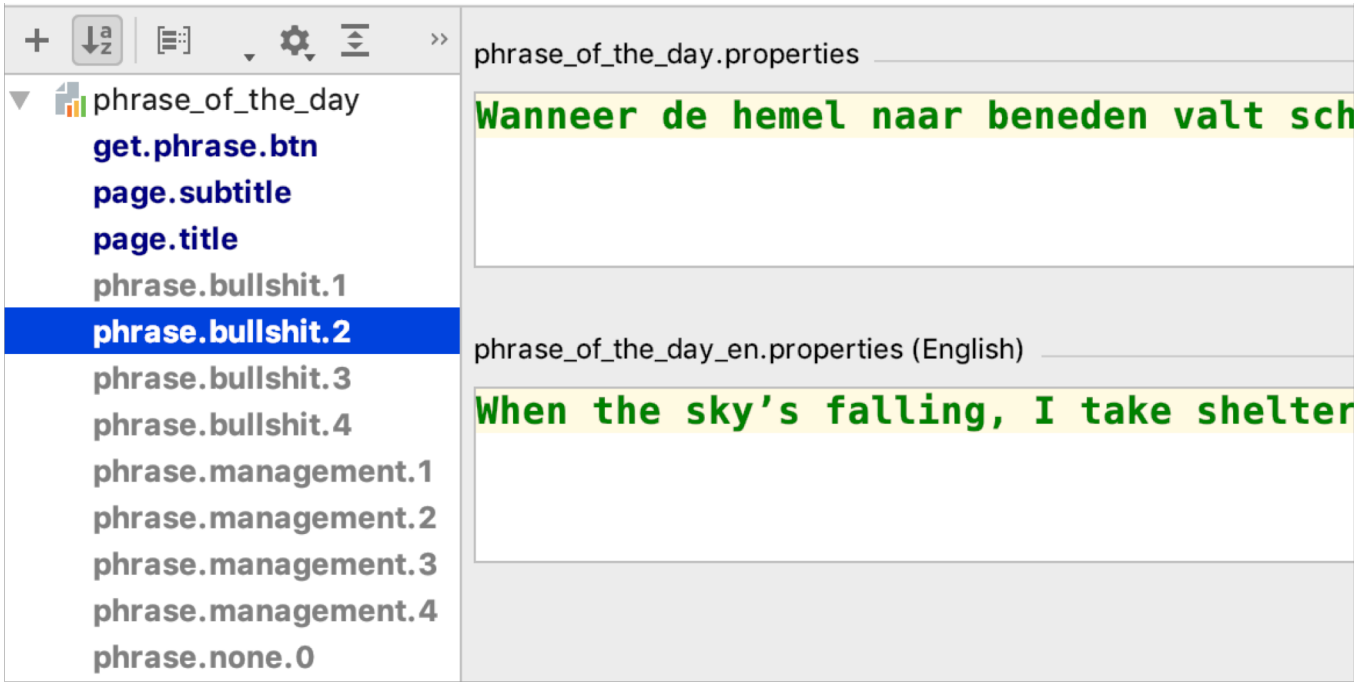
```
public class PhraseFactory {
    //I only have 4 phrases of each category
    public static int MAX_PHRASE_COUNT = 4;
    public static String getPhrase(String phraseType) {
        Random rand = new Random();
        int phraseIndex = rand.nextInt(MAX_PHRASE_COUNT) + 1;
        return Integer.toString(phraseIndex);
    }
}
```

Fetch the phrases; second version

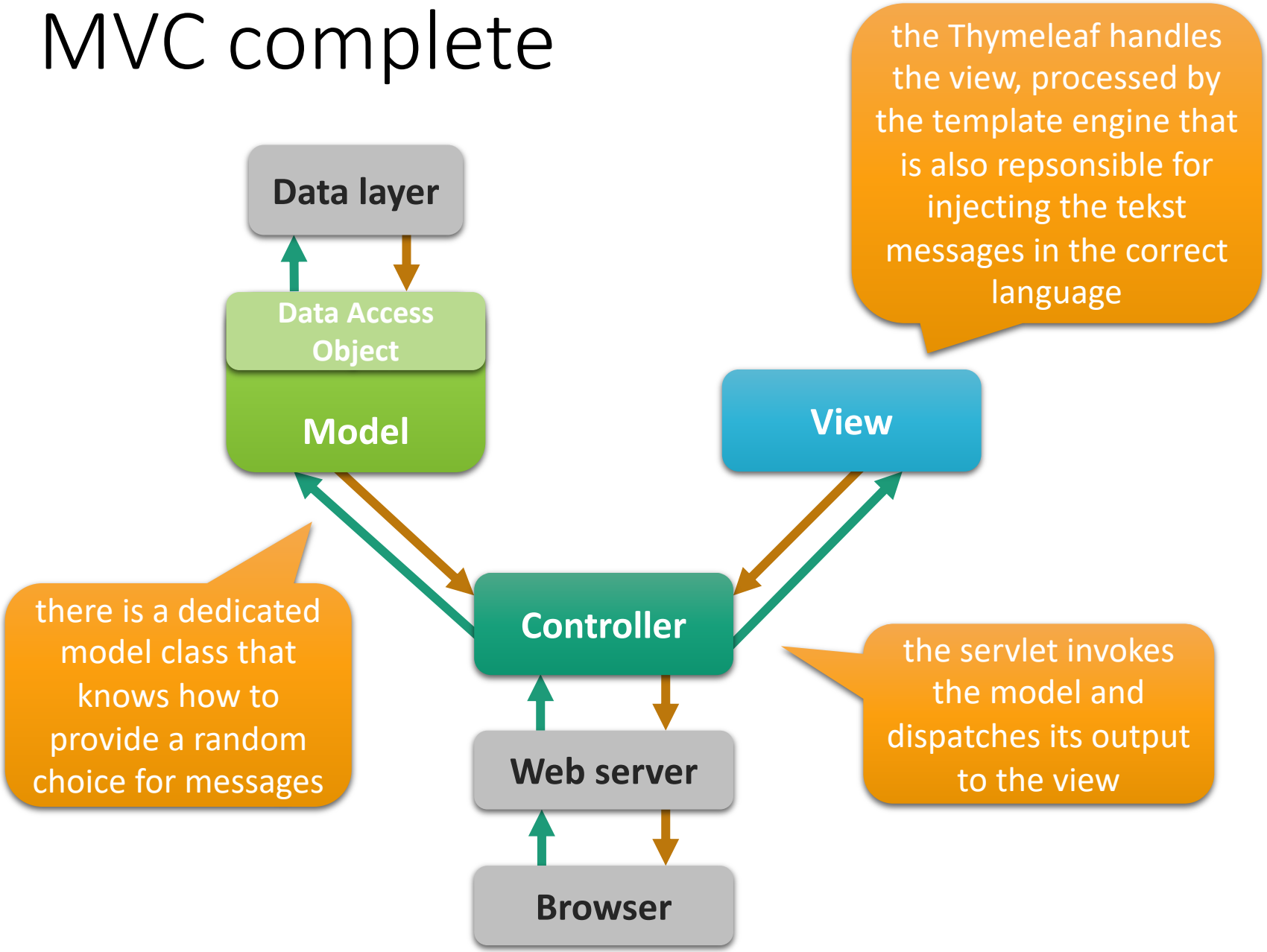
- The Thymeleaf template

```
<h4 th:text="#{'phrase.' + ${phrase_type} + '.' +
${phrase_num}}">_phrase placeholder_</h4>
```

- The resource bundle



MVC complete



Summary

- Today, you have seen the MVC pattern in action
  - model: plain old java objects (POJOs)
  - view: Thymeleaf
  - controller: servlet
- Thymeleaf has several techniques to deal with dynamic content generated by the model, the subject of the next presentation