SENG2050
Introduction to Web Engineering

Week 4

Assignment 2

- Reminder: Due 8th April 11:59 PM
- Please come to lab to get helps and hints

This Week

- MVC
- MVC in Practice
- JSP Implicit Objects & Directives

A - MVC

Separation of Concerns

- Everything should do one thing and one thing only

- Improves modularity -> good design

- Basic Ex. HTML, CSS, JS
  - HTML – Display Structure
  - CSS – Appearance
  - JS - Behaviour

So far …

- Java Servlets
  - Very good at handling a web request
  - Bad at generating HTML
  - Not the best place to implement business logic
- JSPs
  - Very good at dynamically generating HTML pages
  - Clunky to process request, implement business logic
- JavaBeans
  - Good at implementing business and application logic
  - Not good at anything else …

Overview

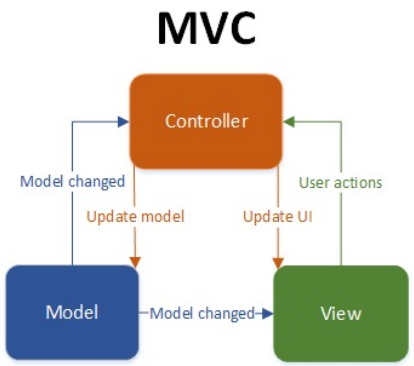
- Problem so far:
  - We still have a mix of Java code and HTML through out web applications
  - If a new HTML spec is released tomorrow you may need to **modify a lot of code to comply**
  - If your client would like you to create an Android application or a desktop application you will need to **re-write a lot of code**
  - If you client changes some of the specifications you may need to **modify a lot of code to comply**
    - Cinema has upgraded to a 10x7 grid of seats
    - Users may book 4 seats if 2 are for children
      - Users need to supply age
      - We need to validate age

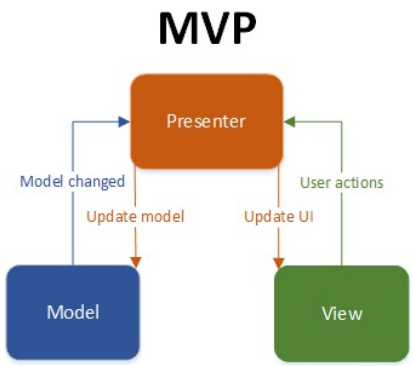
Solution

- Separation of concerns
  - Have presentation code separate from your business logic
  - i.e.
    - JSP is concerned with displaying data to the user and providing them controls (buttons, links, etc.)
    - Java classes have application logic
    - Java servlets react to user input (requests, forms, buttons, links) using logic defined in the above classes. Then decides what JSP to show.
- Model View Controller – MVC
  - Model – You data & business logic
  - View – The page presented to the user
  - Controller – The business logic reading/writing data and passing to view

Model View Controller – MVC

- Software user interface (UI) architecture pattern
- Can be used in any software that a user interacts with
  - Not just web applications
  - Different variations of MVC
- Introduced in the late 70s (78-80)
  - Improved and evolved over time
  - Still very relevant today
  - Still well known and supported
- Different flavors of MVC,
  - We will focus on a variety with “passive” models
    - Where a model doesn’t call controller methods

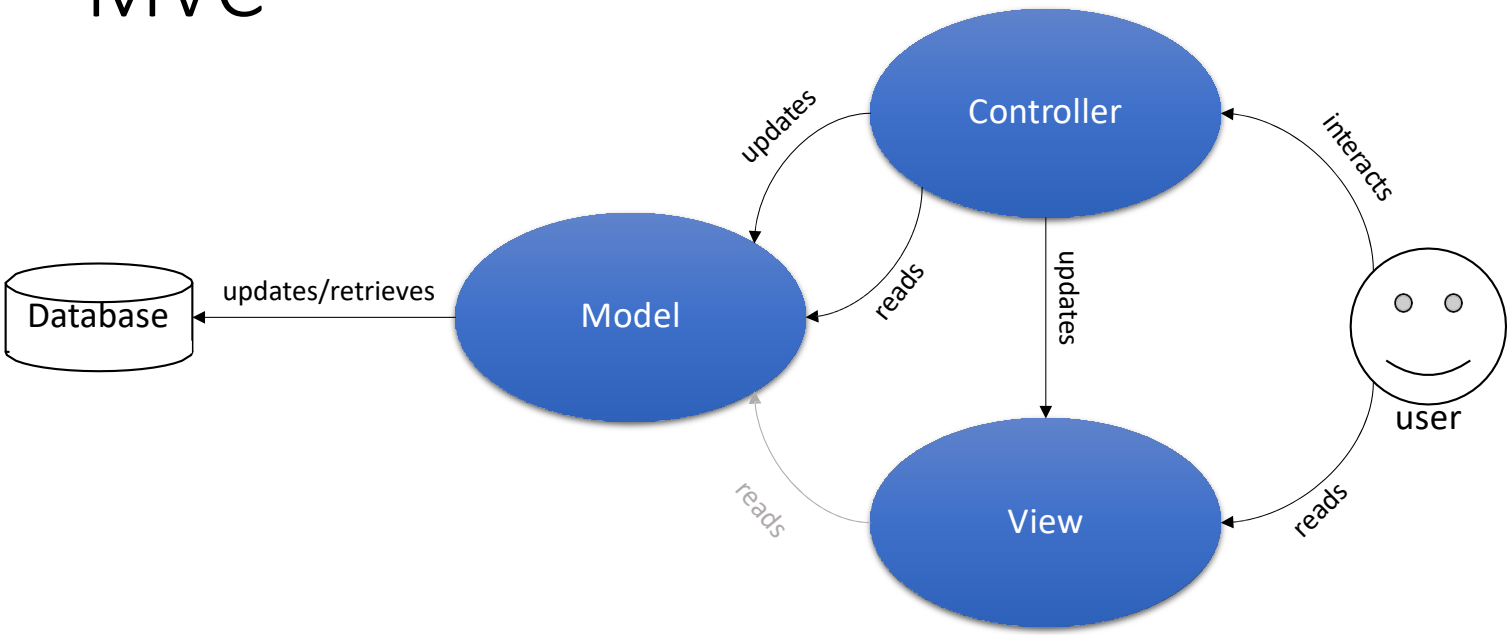




MVC

- Pattern used to organise code in such a way that we achieve a “separation of concerns”
- Consists of:
  - Models:
    - Classes that contain the application’s logic
    - Can represent objects in a business (seat, briefcase, student); or
    - Abstract concepts
    - Talks to data stores (database, text files, other data storage options)
  - Views
    - The page/screen that the user is presented with
    - Including all interactive content on the page (buttons, links, forms, etc.)
  - Controllers
    - Classes that process user input
    - Update models
    - And determine which view to show

MVC


Controller
updates
interacts
Database
updates/retrieves
Model
reads
updates
user
reads
View
reads

MVC – Models

- Classes that encapsulate the business/application logic
  - The actual behaviour/functionality of the application
  - *CinemaManager:*
    - *bookSeat(seatNumber, details)*
    - *isSeatAvailable(seatNumber)*
    - *canUserBook(username)*
  - *UserModel:*
    - *username*
    - *email*
- Interacts with data store (database, files, other classes that store data)
  - i.e. *CinemaManager* might have a *saveChanges()* method that saves the current state of the cinema to disk (it can decide how, database, file, etc.)
- Has no knowledge of how the data will be presented (**View**)
  - i.e. no HTML output because the model might be rendered in a desktop or an android application

MVC - Views

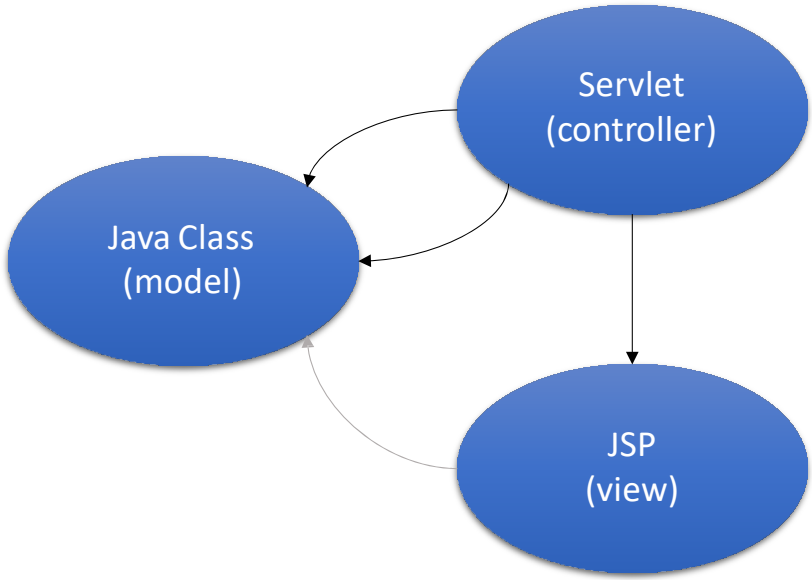
- Presentation logic
  - How the information encapsulated in the model should be displayed to the user
- Not concerned with how the information was obtained or calculated
  - Or where it came from
- Contains controls (forms, buttons, links, etc.) that notify controllers of user interaction
  - i.e. User pressed the “Submit Booking” button with the following information
  - i.e. User is requesting to book seat “A7”
- No business/application logic

MVC – Controllers

- Is the connection between the **View** and the **Model**
- Deals with requests (user interaction) and
- Makes decisions about which **view** should be displayed
  - Success vs Error
  - i.e. Booking confirmation vs Error (you have already booked 3 seats)
  - i.e. Display booking form vs display information about who booked the seat
- Uses the models to make decision
- Has knowledge of both the **View** and the **Model**

MVC – Java

- Model – Java class (maybe a JavaBean)
- View – JSP
- Controller – Servlet (or a JSP)

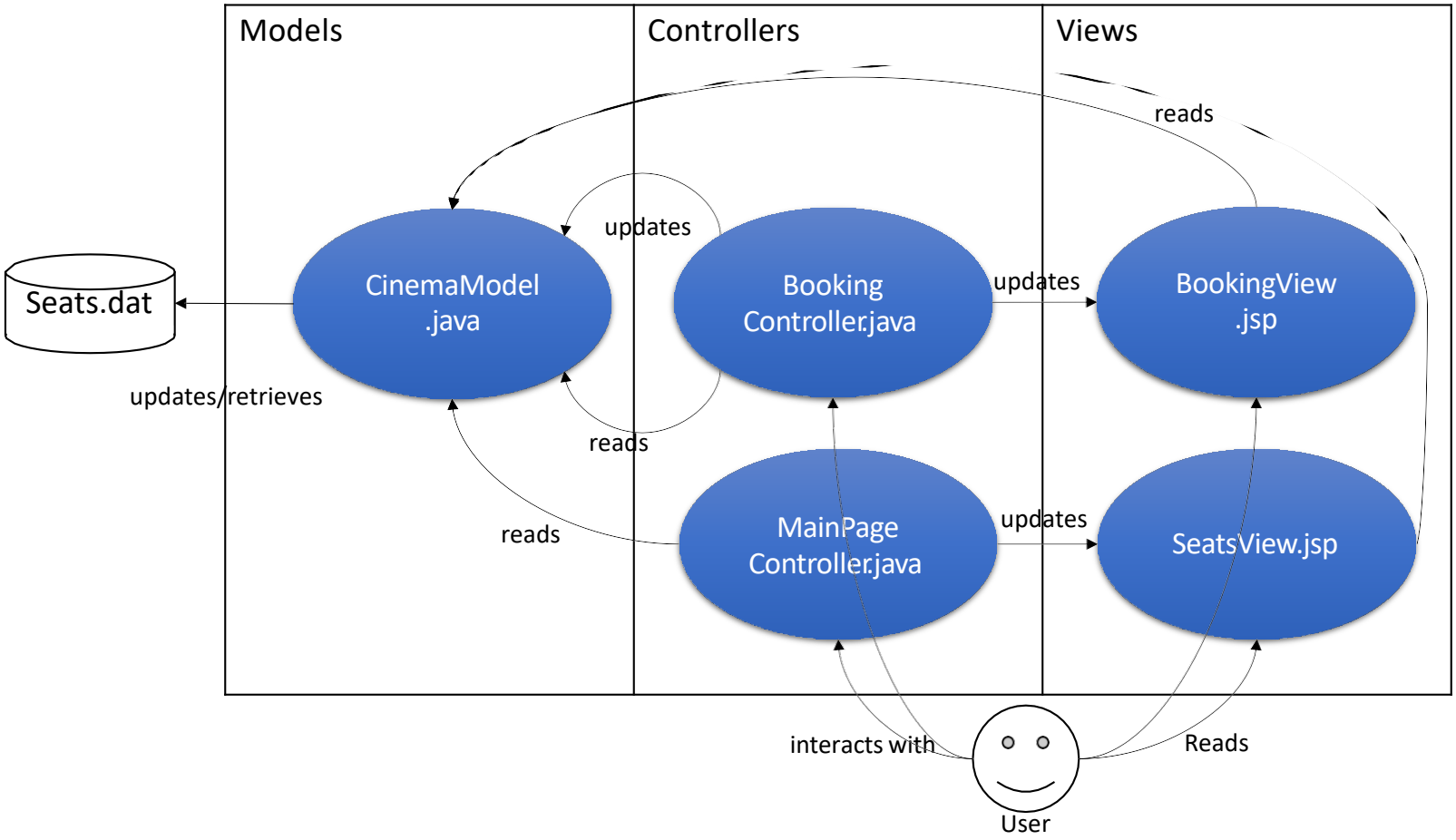


MVC – J2EE Mappings

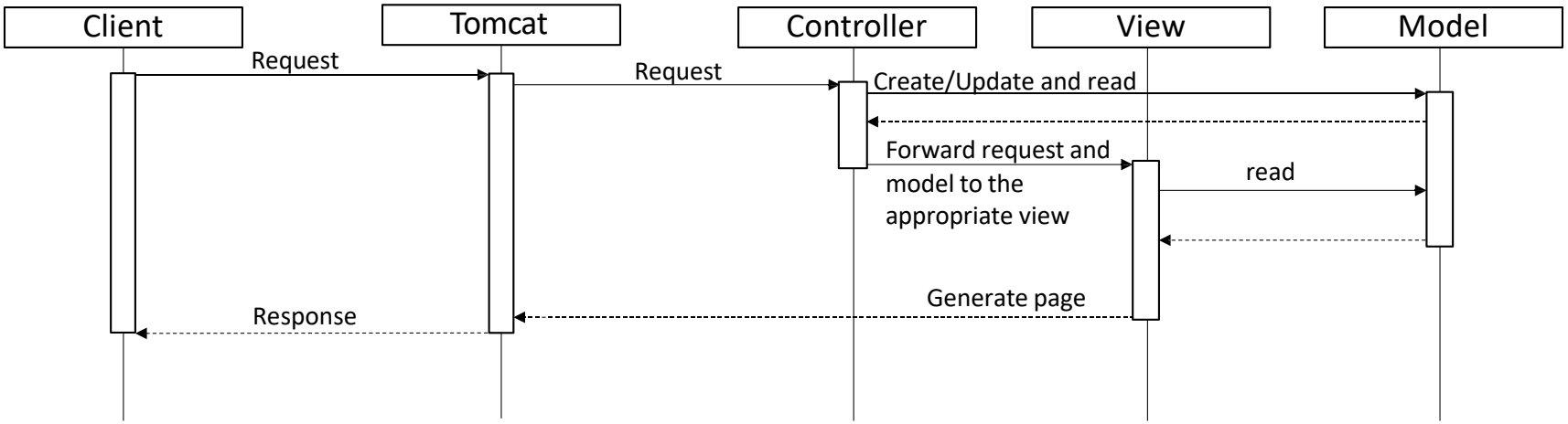
1. **Controller** handles user interaction
2. Populate the **Model**
   - **Controller** invokes business logic through public methods on the **Model**
3. Pass the **Model** to the **View**
4. The **View** generates the user interface (from the model data) to display to the user
5. The user will interact with the **View** which will notify the **Controller** (repeat steps 1-5)

1. A **servlet** handles requests
   - **Servlet** reads request parameters, checks for missing or malformed data, etc.
2. Populate the **JavaBeans**
   - **Servlet** invokes business logic through public bean methods
3. Store the processed data in the request, session, or servlet context
   - Using *setAttribute(String, Object)* methods
   - This processed data can be the bean itself, or a different bean
4. Forward the request to a **JSP** to generate HTML to send to the client web browser
   - **JSP** uses <jsp:usebean …> and <jsp:geProperty …/> tags along with JSTL and EL (more on these later)
   - Ensure you aren't manipulating these beans in your JSPs though. i.e. maybe create an *interface* for your bean that only contains the *getXXX* methods.
5. The user will interact with the generated web page which will generate a new request for a **controller** (repeat steps 1-5)

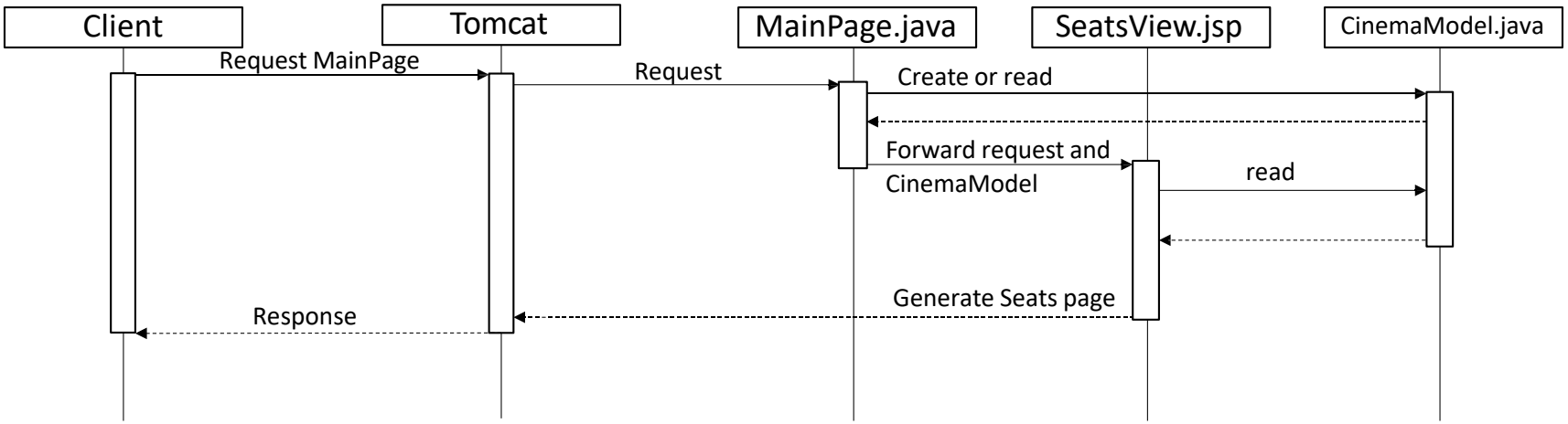
Example 1 – Seat Booking System



Example 2 – MVC – Sequence Diagram



Example 3 – Seat Booking – Sequence Diagram



B - MVC in Practice

Keep in mind …

- We have many frameworks that claim to be MVC
- A framework itself cannot be MVC
  - Only provides the scaffolding for developing a MVC application
- We, as engineers, need to use the frameworks to design MVC applications
- Servlets, JSPs and Beans are not designed as a MVC framework
  - They are simply components for building an application
  - In this course we like to use them in a MVC-compliant manner

Different MVC Frameworks

- We never use Servlets + JSPs + Beans as a ‘framework’
- There exist countless ‘MVC’ frameworks
- We will look at Struts2 later in this course (optional for Assignment 3)

Java	Scala	C#	Ruby / Python	JavaScript
spring NINJA web framework Struts 2	play	Microsoft .NET MVC 5	RAILS python django	ANGULARJS by Google ember JavaScript MVC

Servlets + JSPs + Beans

- Model – The Java Beans
  - Implement *ALL* the business logic
  - Not necessarily 'JSP-compliant' beans
  - A bean can be used by other Beans + Servlets

- View – The JSPs
  - Simple display of data
  - No actual Java code (will learn more later – limit yourself to conditional/loop statements)

- Controller – The Servlet
  - Read request, invoke bean logic, handle *forwarding* to JSP for view
  - Handle all setup of beans + injection for JSP
  - Manage application state

MVC – In Practice

- Avoid accessing JSPs directly (through URLs)
  - This is because your JSP probably requires a model that a controller has prepared for it
  - You may of course have simple JSPs accessed directly:
    - If they don't rely on business logic
    - Maybe your index.jsp
- Instead servlets should be the only URLs the user is accessing
- Your servlet can populate the session, request, or application context objects
  getSession().setAttribute("key", value); // session-scope variables request.setAttribute("key", value); // request-scope variables getServletContext().setAttribute("key", value); // application-scope variables
- When a servlet has completed its job it should forward the request and response objects to the JSP using:

```
RequestDispatcher dispatcher = getServletContext().getRequestDispatcher("/Page.jsp");
dispatcher.forward(request, response);
```

  - *RequestDispatcher* is included in the *javax.servlet* package
  - */Page.jsp* is the relative address (in respect to your web application) of your **View** page
    - If *Page.jsp* is in a *views* folder it would be:

```
RequestDispatcher dispatcher = getServletContext().getRequestDispatcher("/views/Page.jsp");
```

MVC – In Practice

- Problem:
  - Because our JSPs rely on a servlet to process its request it may be invalid if a user navigates directly to the JSP
- Solution:
  - Protect them
  - Store them in a directory that the user can't access by default
    - The *WEB-INF* directory
  - This way the only way a JSP is displayed is if our web application *forwards* to it using:

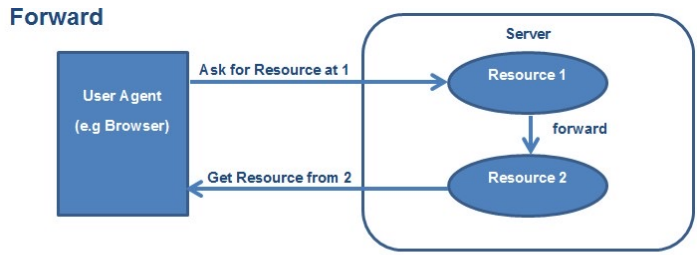
```
RequestDispatcher dispatcher = getServletContext().getRequestDispatcher("/WEB-INF/Page.jsp");
dispatcher.forward(request, response);
```

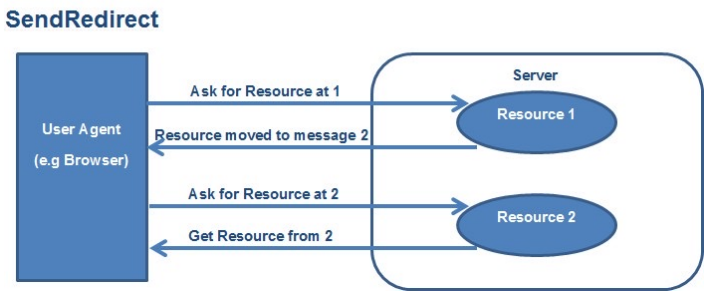
Redirects and Forwards

- For proper MVC, the servlet needs to 'forward' a request to a JSP

```
RequestDispatcher dispatcher = getServletContext().getRequestDispatcher("/Page.jsp");
dispatcher.forward(request, response);
```

- This is not the same as a HTTP Redirect!
  - A redirect sends a HTTP response to the client saying 'please go to this URL'
  - A forward is internal to the server
  - It passes the current request to a new page on the server
  - All without returning back to the client





MVC – In Practice

- Remember:
  - A servlet can *forward* to one of many different pages (or even other servlets):
    - i.e. when making a booking there are 2 possible actions:
      1. The user is allowed and the booking is performed. The servlet forwards to a confirmation page; or
      2. The user has previously booked 3 seats. The servlet forwards to a page explaining why their booking was not successful.
  - The **Model** should not know about how it is being presented
    - It simply holds/manipulates/returns data related to the business
    - Putting your **Models** in a different package will make it obvious if it has a dependency on a **Controller** or a **View**
      - i.e. avoid *import controllers.MyController;* from a **Model** class
  - There are flavours of MVC with active **Models**
    - This means the **Model** knows about the **Controller** and will call methods on the **Controller**
    - These **Models** are useful in event-driven software such as those found in desktop, Android, iOS, and even JavaScript applications

MVC – In practice

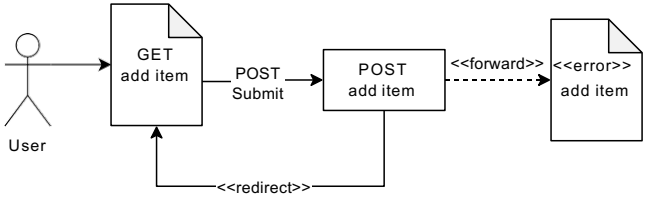
- The processing of a request is boiled down into 5 broad steps
  - Note: Not all need to be executed!

1. The Servlet accepts a HTTP requests
2. The Servlet processes inputs
3. The Servlet calls appropriate bean methods
4. The Servlet passes view data to the JSP
5. The Servlet forwards the request to a JSP for display
   - Or redirects to another URI for display

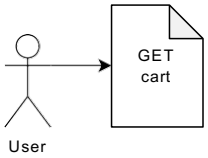
Example 1: Servlets, JSPs and Beans

- Consider a simple online web store

- User can add items to a cart
  - Item must have a valid name
  - Item must have a qty > 0



- User can view their cart



Example 1: Servlets, JSPs and Beans

```java
public class ProductService {
    private static String[] itemNames = {
        "bananas",
        "apples",
        "museli bars",
        "salad",
        "juice"
    };

    public static String[] getItemNames() {
        return itemNames;
    }
}
```

Model

```java
@WebServlet(urlPatterns = {"/additem"})
public class AddItemServlet extends HttpServlet {

    public void doGet(
        HttpServletRequest request,
        HttpServletResponse response
    ) throws IOException, ServletException {
        // Setup for view - get valid items
        String[] itemNames = ProductService.getItemNames();
        request.setAttribute("itemNames", itemNames);

        // Pass items to view
        RequestDispatcher dispatcher = this.getServletContext()
            .getRequestDispatcher("/WEB-INF/additem.jsp");
        dispatcher.forward(request, response);
    }
```

Controller

```jsp
<%
    String[] itemNames = (String[]) request.getAttribute("itemNames");
%>


<html>
<head>
    <title>Add Item</title>
</head>
<body>
    <h1>Add Item</h1>

    <% if (java.util.Objects.equals(
      request.getParameter("success"), "true")) { %>
        <p>
            Item added successfuly.
        </p>
    <% } %>

    <form action="/week4examples/additem" method="POST">
        <p>
            Item:
            <select name="name" required>
                <% for (String name : itemNames) { %>
                    <option value="<%= name %>"><%= name %></option>
                <% } %>
            </select>
        </p>

        <p>
            Qty: <input type="number" name="qty" required />
        </p>

        <p>
            <input type="submit" />
        </p>
    </form>
</body>
</html>
```

Add Item

Item: bananas

Qty: 0

Submit

View

Viewing the 'additem' page

Example 1: Servlets, JSPs and Beans

Adding an item to the cart - success

```java
public class ProductService {
    private static String[] itemNames = {
        "bananas",
        "apples",
        "museli bars",
        "salad",
        "juice"
    };

    public static String[] getItemNames() {
        return itemNames;
    }

    public static boolean isValidName(String name) {
        return Arrays.binarySearch(itemNames, name) >= 0;
    }
}
```

```java
public class CartBean implements Serializable {

    private List<CartItemBean> items = new LinkedList<>();

    public CartBean() {
        items.add(new CartItemBean("bananas", 1));
        items.add(new CartItemBean("apples", 2));
        items.add(new CartItemBean("museli bars", 1));
        items.add(new CartItemBean("salad", 3));
    }

    public List<CartItemBean> getItems() {
        return items;
    }
}
```

Models

```java
@WebServlet(urlPatterns = {"/additem"})
public class AddItemServlet extends HttpServlet {
    public void doPost(
        HttpServletRequest request,
        HttpServletResponse response
    ) throws IOException, ServletException {
        List<String> errors = new ArrayList<>();

        // Get parameters
        String name = request.getParameter("name");
        String qtyStr = request.getParameter("qty");

        // Validate name
        if (name == null) {
            errors.add("No name passed");
        } else if (!ProductService.isValidName(name)) {
            errors.add("Item name is not a valid product name");
        }

        // Validate qty
        int qty = 0;
        try {
            qty = Integer.parseInt(qtyStr);

            if (qty <= 0) {
                errors.add("Qty is less than zero");
            }
        } catch (NumberFormatException ex) {
            errors.add("Qty is not a number");
        }

        // Show an error page if an error was gound
        if (!errors.isEmpty()) {
            // Pass errors to the view
            request.setAttribute("errors", errors);

            // Forward on to the error page
            RequestDispatcher dispatcher = this.getServletContext()
                .getRequestDispatcher("/WEB-INF/additem-error.jsp");
            dispatcher.forward(request, response);
            return;
        }

        // Get the current users cart
        HttpSession session = request.getSession();
        CartBean cart = (CartBean) session.getAttribute("cart");
        if (cart == null) {
            cart = new CartBean();
            session.getAttribute("cart");
        }

        // Add the item, and redirect to success page
        cart.getItems().add(new CartItemBean(name, qty));
        response.sendRedirect("/week4example/additems?success=true");
    }
}
```

Controller

```jsp
<%
    String[] itemNames = (String[]) request.getAttribute("itemNames");
%>

<html>
<head>
    <title>Add Item</title>
</head>
<body>
    <h1>Add Item</h1>

    <% if (java.util.Objects.equals(
      request.getParameter("success"), "true")) { %>
        <p>
            Item added successfuly.
        </p>
    <% } %>
```

Add Item

Item added successfuly.

Item: bananas

Qty:

Submit

View

Example 1: Servlets, JSPs and Beans

Adding an item to the cart - error

```java
public class ProductService {
    private static String[] itemNames = {
        "bananas",
        "apples",
        "museli bars",
        "salad",
        "juice"
    };

    public static String[] getItemNames() {
        return itemNames;
    }
}
```

Model

```java
@WebServlet(urlPatterns = {"/additem"})
public class AddItemServlet extends HttpServlet {
    public void doPost(
        HttpServletRequest request,
        HttpServletResponse response
    ) throws IOException, ServletException {
        List<String> errors = new ArrayList<>();

        // Get parameters
        String name = request.getParameter("name");
        String qtyStr = request.getParameter("qty");

        // Validate name
        if (name == null) {
            errors.add("No name passed");
        } else if (!ProductService.isValidName(name)) {
            errors.add("Item name is not a valid product name");
        }

        // Validate qty
        int qty = 0;
        try {
            qty = Integer.parseInt(qtyStr);

            if (qty <= 0) {
                errors.add("Qty is less than zero");
            }
        } catch (NumberFormatException ex) {
            errors.add("Qty is not a number");
        }

        // Show an error page if an error was gound
        if (!errors.isEmpty()) {
            // Pass errors to the view
            request.setAttribute("errors", errors);

            // Forward on to the error page
            RequestDispatcher dispatcher = this.getServletContext()
                .getRequestDispatcher("/WEB-INF/additem-error.jsp");
            dispatcher.forward(request, response);
            return;
        }

        // Get the current users cart
        HttpSession session = request.getSession();
        CartBean cart = (CartBean) session.getAttribute("cart");
        if (cart == null) {
            cart = new CartBean();
            session.getAttribute("cart");
        }

        // Add the item, and redirect to success page
        cart.getItems().add(new CartItemBean(name, qty));
        response.sendRedirect("/week4example/additems?success=true");
    }
}
```

Controller

```jsp
<%
    List<String> errors = (List<String>) request.getAttribute("errors");
%>

<html>
<head>
    <title>Error</title>
</head>
<body>
    <h1>An error occured</h1>

    <ul>
        <% for (String error : errors) { %>
            <li><%= error %></li>
        <% } %>
    </ul>

</body>
</html>
```

An error occured

- No name passed
- Qty is not a number

View

But, do we always need controllers?

- Having a servlet-jsp-bean tuple is tedious
  - Think for every ‘action’
- What if the Servlet does nothing?
  - Sometimes the Servlet simply directs a JSP to be displayed
  - i.e. we need to display a simple page
  - No prior setup
  - No processing of the request
- Sometimes we have ‘controller-less’ MVC
  - We have a model and view, but no dedicated controller!

Controller-less MVC - JSPs + Beans

- Model – The Java Beans
  - Implement *ALL* the business logic
  - Must be 'JSP-compliant' beans
  - As well as other beans to be used by the JSP beans

- View – The JSPs
  - Simple display of data
  - Invoke the logic in the Beans + process result

- Controller - … ?
  - Largely the JSP framework itself
  - Can be JSP actions, custom tags, *useBean*, *setProperty*, *getProperty*

Example 2: JSPs & Beans

Controller?

```java
public class CartBean implements Serializable {

    private List<CartItemBean> items = new LinkedList<>();

    public CartBean() {
        items.add(new CartItemBean("bananas", 1));
        items.add(new CartItemBean("apples", 2));
        items.add(new CartItemBean("museli bars", 1));
        items.add(new CartItemBean("salad", 3));
    }

    public List<CartItemBean> getItems() {
        return items;
    }
}
```

Model

```jsp
<%@page import="pkg.CartItemBean" %>
<%@page import="java.util.List" %>

<jsp:useBean id="cart" class="pkg.CartBean" scope="session" />

<%
    List<CartItemBean> items = cart.getItems();
%>

<html>
<head>
    <title>My Cart</title>
</head>
<body>
    <h1>My Cart</h1>
    <% for(CartItemBean item : items) { %>
        <li><%= item.getName() %> - <%= item.getQty() %></li>
    <% } %>
</body>
</html>
```

View

Do we always need views?

- Not all HTTP requests have pages generated as a response

- Simple example is processing a POST request
  - If a POST request succeeds: best to ‘redirect’ to a new page
    - To avoid some funny browser behavior
    - The redirect will be handled by a new controller
  - If a POST request fails: re-display a form page (or error page)
    - Used to ask user to fix inputs

- In the first case, we don’t have a view component!

Overall …

- MVC is a design pattern
- There are many ways of using design patterns
- For a traditional user interaction
  - We need the full MVC
- But on the web things aren’t so simple
  - Sometimes we don’t need controllers (presenting simple pages)
  - Sometimes we don’t need views (processing POST requests)
- Always up to the developer to decide!

What to use?

- Depends …
  - Pros and cons to each

- If simple interaction – JSPs + Beans
  - No need to control state
  - Very simple manipulation of data
    - Just that stored in beans

- If complex interaction – Servlets + JSPs + Beans
  - Need to control state
  - Need access to: request, cookies, response, session or application context.
  - Validate inputs
  - Need resources (e.g. database connection, file IO, etc)

- If processing POST data – Servlet + Beans
  - As above, but don’t directly present a page

C - Implicit Objects & Directives

Servlet Objects

- Servlets are provided with:
  - HttpServletRequest
  - HttpServletResponse

- From these we can retrieve
  - PrintWriter (for the HttpServletResponse)
  - HttpSession (stores data for the current user)
  - ServletContext (an object representing our application)
  - ServletConfix (any configuration values applied to the application)

- All useful in the design of our applications

JSP Implicit Objects

- Where are these objects in JSP?
  - Provided `implicitly' to the JSP Page
  - Hence, we call them "Implicit Objects"

- JSPs have 9 implicit objects
  - `config, application`
  - `request, response, out`
  - `pageContext, page`
  - `session`
  - `exception`

- **Important!**
  - **You don't declare them, you just use them!**

JSP Implicit Objects

- **`request`**
  - represents the request this JSP is serving
  - contains request parameters
- **`response`**
  - represents the response the JSP is generating for the client
  - contains the body (HTML), cookies, etc
- **`out`**
  - a PrintWriter used to generate output for the client
  - only needed in scriptlets
- **`session`**
  - represents the session associated with the request
  - stores data about the current user

JSP Implicit Objects

- **`application`**
  - the ServletContext object
  - shared by all the servlets in the current application
- **`config`**
  - the ServletConfig object
  - stores configuration for the current application
- **`pageContext`**
  - the PageContext object
  - used for sharing Java Beans between servlets
- **`page`** = this
- **`exception`**
  - The last thrown exception

Four Scopes

- Our implicit objects provide four scopes:
  - application
  - request
  - session
  - page

- All available in Servlets (page == ’this’)

- Used to store values
  - e.g. configuration, shared objects `setAttribute("name", value)` and `getAttribute("name")`

- Primarily used for storing beans
  - This is where the beans ‘live’, but can store any data.

JSP Implicit Objects - `request`

- Same as servlet HTTPServletRequest parameter!
  - **getProtocol()** – HTTP/1.1, FTP, SMTP, ...
  - **getServerName()** – the name of the computer running the server
  - **getPort()** – the port the server is listening to
  - **getRemoteAddr()** – the IP number the request came from
  - **getRemoteName()** – the DNS name the request came from
  - **getParameter("*name*")** – the value of a passed request parameter

JSP Implicit Objects - `request`

- Same as Servlet HTTPServletResponse parameter
  - **getHeader(“*name*”)** – the value of any header passed in the request
  - **getMethod() –** Usually GET or POST
    - can be PUT, PATCH, DELETE, INFO, HEAD,
  - **getPathInfo() –** the path portion of the requesting URI
  - **getQueryString() –** the query portion of the requesting URI
  - **getRemoteUser() –** the name of the user who sent the request
    - (if it can be determined)
  - **getRequestURI() –** full URI of the request

JSP Implicit Objects - `out`

- print(*string*), println(*string*)
  - the standard PrintWriter methods
  - Inside the servlet code out is a java.io.PrintWriter
  - you get one by calling ServletResponse.getWriter()
- In JSP, out is a java.servlet.jsp.JspWriter
  - you get it implicitly through the pageContext
- For all practical purposes they are interchangeable
- Very few reasons to use this …

```
Path Info: <%= request.getPathInfo() %>
<% out.print("Path Info: "); out.println(request.getPathInfo() ); %>
```

Equivalent!

JSP Implicit Objects Examples

```
<% int num1=10;int num2=20;
out.println("num1 is " +num1);
out.println("num2 is "+num2);
%>
```

out

```
<%
String bg = request.getParameter("bg");
boolean hasBg;
if (bg != null) {
  hasBg = true;
} else {
  hasBg = false;
  bg = "white";
}
%>

<body style="background-color: <%= bg %>">
```

request

JSP Implicit Objects Examples

```
<form action="guru.jsp">
<input type="text" name="username">
<input type="submit" value="submit">
</form>
```

Submit parameter

```
<% String username = request.getParameter("username");
out.println("Welcome " +username);
%>
```

Retrieve through request

JSP Implicit Objects Examples

```
<% session.setAttribute("user","GuruJSP"); %>
<a href="implicit_jsp8.jsp">Click here to get user name</a>
```

set session value

```
<% String name = (String)session.getAttribute("user");
out.println("User Name is " +name);
%>
```

print session value

JSP Actions

- Have the form **<jsp:*action* ... />**
- Special tags that implement some dynamic functionality
  - Coordinate JSP with Java Beans
  - Import classes
  - Include external files
  - 'JSP Tags' (next week)
- Forward requests to another JSP or servlet
- Already used:
  - jsp:useBean
  - jsp:setProperty
  - jsp:getProperty
- Others:
  - jsp:include
  - jsp:forward
- Can define own:
  - JSP Tags
- Or use
  - JSTL – JSP Standard Tag Library

Bean Actions

- Declaration:

```
<jsp:useBean id="beanName"
  scope="page|request|session|application"
  class="my.bean.classname" />
```

- Get Property:

```
<jsp:getProperty name="beanName"
  property="propertyName" />
```

- Set Property:

```
<jsp:setProperty name="beanName"
  property="propertyName"
  value="newValue" />
```

- Note: Many optional attributes
  - Check the lecture slides and docs!

JSP Include

```
<jsp:include page="url" flush="true" />
```

- Includes the output from the specified page
  - But limited in what in can control – cannot modify request & response

- Why?
  - Useful for common site elements
  - E.g. headers, navigation, footers, etc

- Can pass parameters
  - Used as configuration

```
<jsp:include page="url" flush="true">
  <jsp:param name="name" value="value" />
</jsp:include>
```

JSP Forward

```
<jsp:forward page="url" />
```

- Forwards the request to the passed url
- Similar to jsp:include
  - Has full control of request & response
- Very similar to `response.sendRedirect(url)`
  - jsp:include handled internally
  - response.sendRedirect(url) handled externally

JSP Directives

- Control global properties of the page
  - Class imports
    ```
    <%@page import="java.util.List" %>
    ```
  - Import tag libraries
    ```
    <%@taglib uri="uri" prefix="prefix" %>
    ```
  - Customising the servlet class
    ```
    <%@page extends="my.servlet.ClassName" %>
    ```
  - Session handling
    ```
    <%@page session="true|false" %>
    ```
  - errorPage
    ```
    <%@page errorPage="url.jsp" %>
    ```

JSP Include Directive

- **<%@ include file="*url*" %>**
  - Pastes the source text of the relative URL into the JSP at this point
  - like a C **#include “filename”**

- This happens only when the JSP is converted into a Java servlet

- If the included file changes, you have to **force** the servlet to recompile
  - most JSP servers will not do this automatically

- Useful for including headers and footers

Notes on include

- Literally copies the referenced file into the current
  - Break large pages into smaller
- Which form to use?
  - Core Servlets: Use directive if you need JSP constructs
  - Hans Bergsten: Use directive if the included page rarely changes
- Include action is safer
  - Does not share variables
  - Cannot set headers or cookies

Notes on include

- `include` directive is more efficient
- `include` directive is less secure
  - Included page can contain malicious JSP
- `include` action cannot share page variables (use Beans or sessions)
- `include` action cannot set headers or cookies

Resources

- Tomcat HttpSession API
  - https://tomcat.apache.org/tomcat-8.0-doc/servletapi/javax/servlet/http/HttpSession.html

- Tomcat Cookies API
  - https://tomcat.apache.org/tomcat-8.0-doc/servletapi/javax/servlet/http/Cookie.html