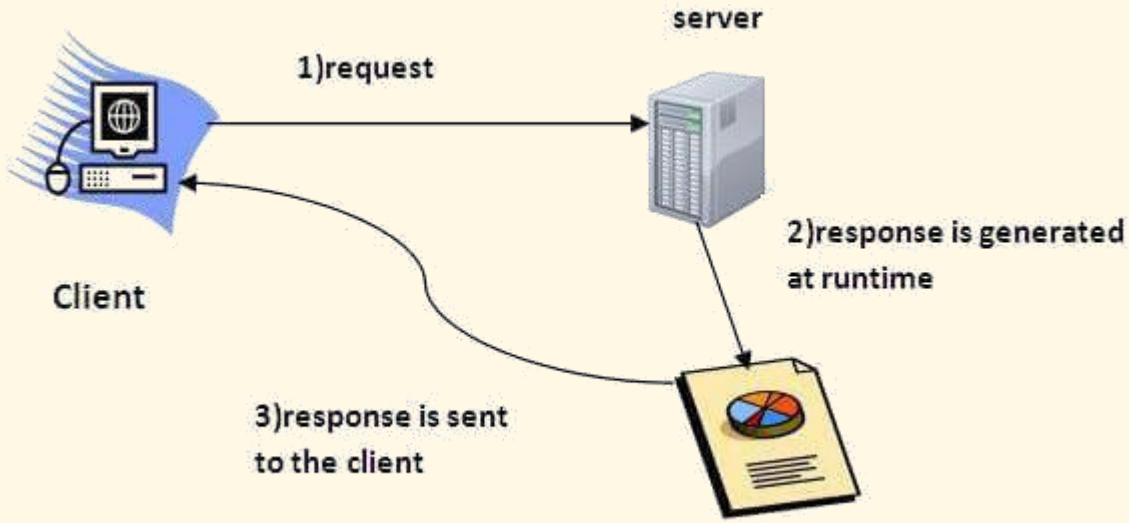
Web

- Servlet
- JSP
- JSP with Servlet
- Spring Boot

Servlet

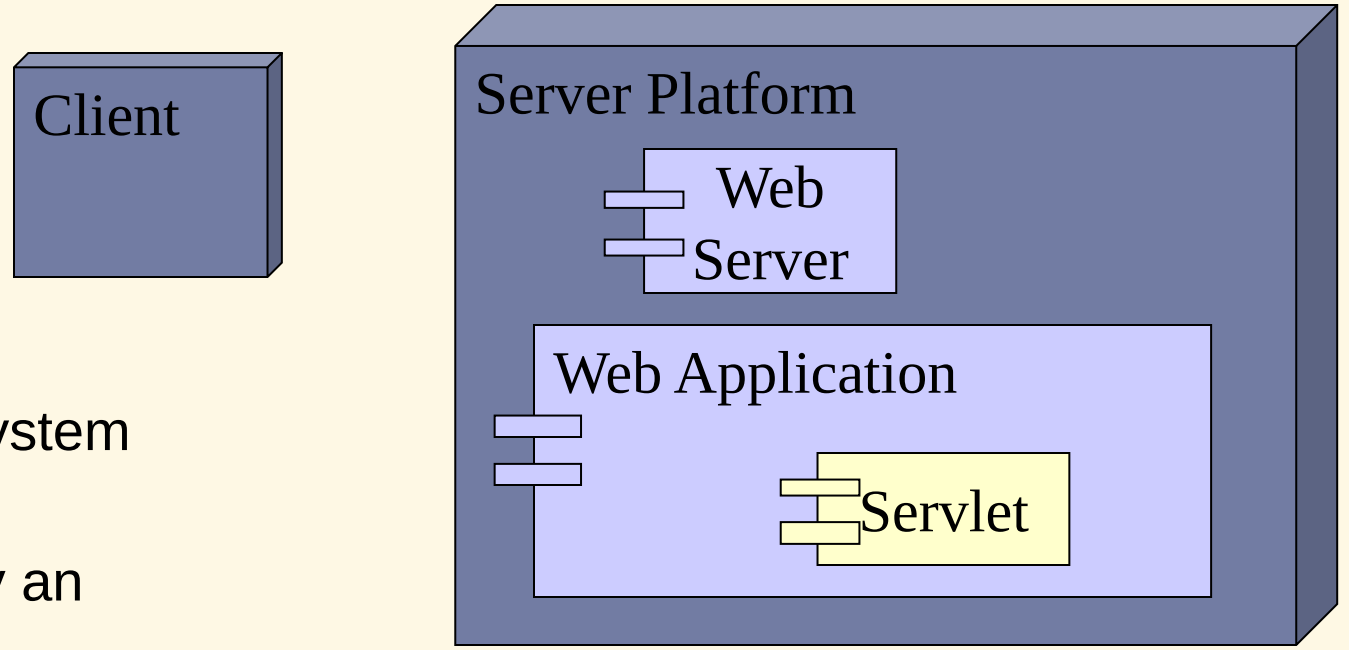
Introduction

- A servlet is a class that is used to extend the capabilities of servers that host applications accessed by means of a request-response programming model
- Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by web servers



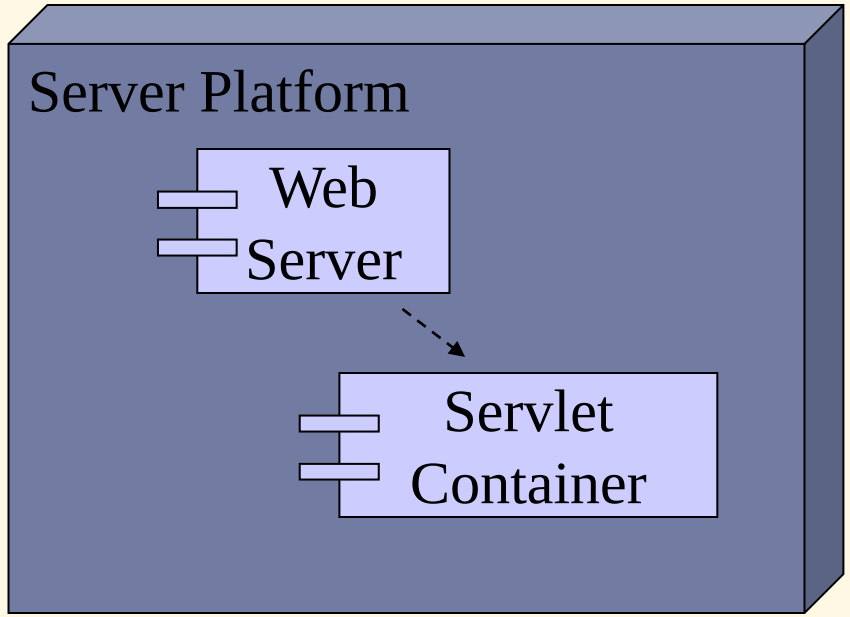
Servlets

- A servlet is a Java program that is invoked by a web server in response to a request
- Together with web pages and other components, servlets constitute part of a web application
- Servlets can
  - create dynamic (HTML) content in response to a request
  - handle user input, such as from HTML forms
  - access databases, files, and other system resources
  - perform any computation required by an application



Servlets

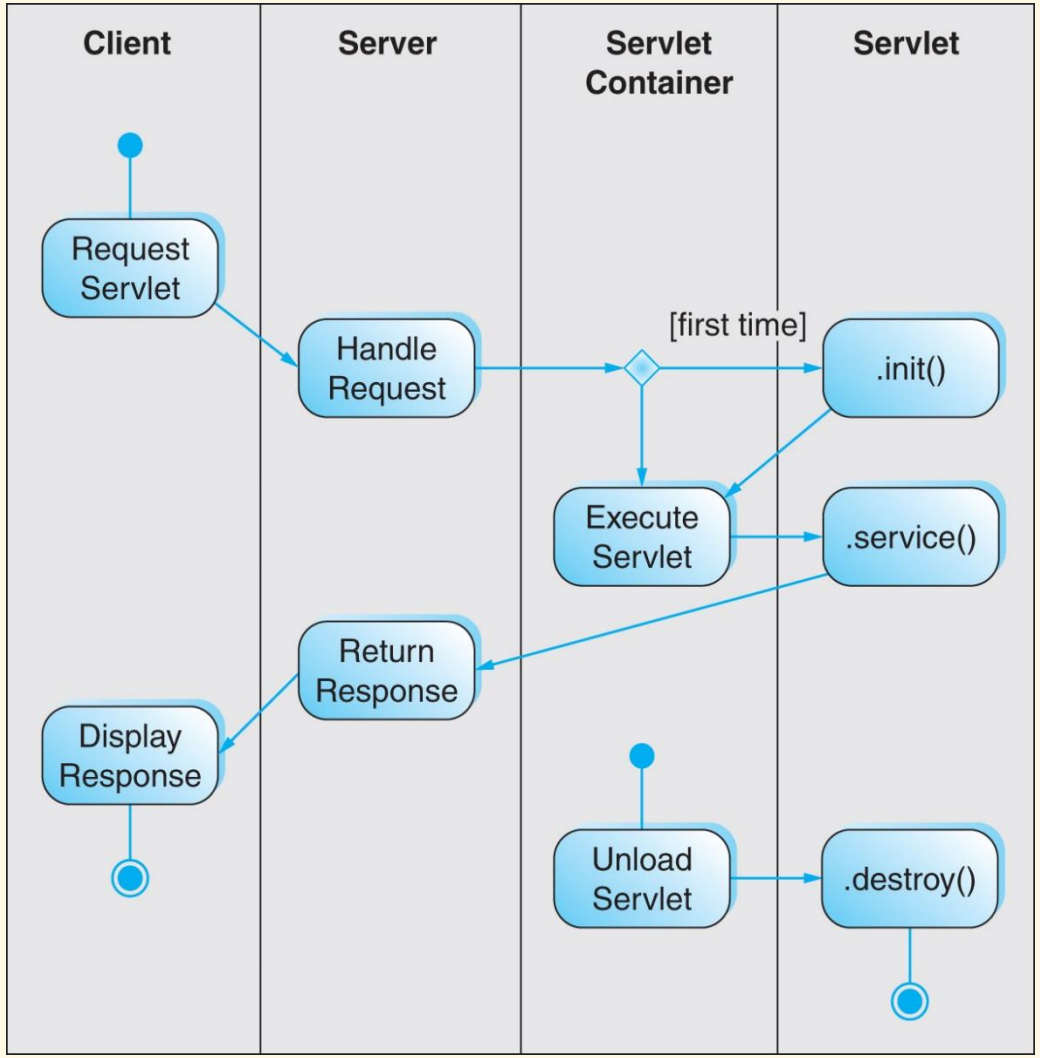
- Servlets are hosted by a *servlet container*, such as Apache Tomcat
  - Apache Tomcat can be both a web server and a servlet container



The web server handles the HTTP transaction details

The servlet container provides a Java Virtual Machine for servlet execution

Servlet Operation

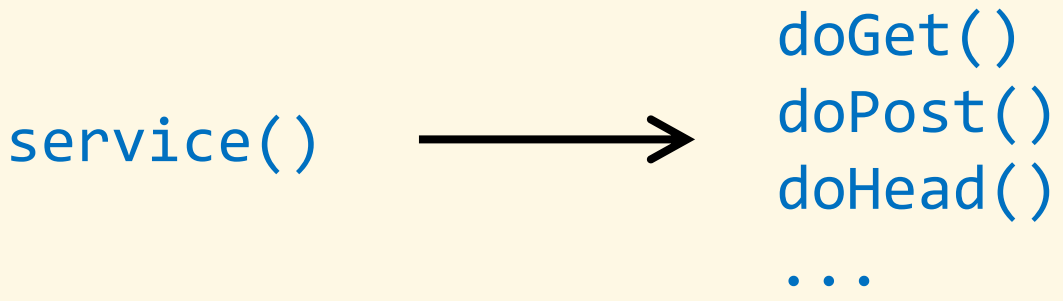


Servlet Methods

- Servlets have three principal methods
  - `init()`
    - Invoked once, when the servlet is loaded by the servlet container (upon the first client request)
  - `service(HttpServletRequest req, HttpServletResponse res)`
    - Invoked for each HTTP request parameters encapsulate the HTTP request and response
  - `destroy()`
    - Invoked when the servlet is unloaded (when the servlet container is shut down)

Servlet Methods

- The default `service()` method simply invokes method-specific methods depending on the HTTP request method
  - Prerequisites: HTTP protocol and HTTP methods



HTTPServlet

- Methods of `HttpServlet` and HTTP requests
  - All methods take two arguments: an `HttpServletRequest` object and an `HttpServletResponse` object
  - Return a BAD_REQUEST (400) error by default

Methods	HTTP Requests	Comments
`doGet()`	GET, HEAD	Usually overridden
`doPost()`	POST	Usually overridden
`doPut()`	PUT	Usually not overridden
`doOptions()`	OPTIONS	Almost never overridden
`doTrace()`	TRACE	Almost never overridden

First Servlet

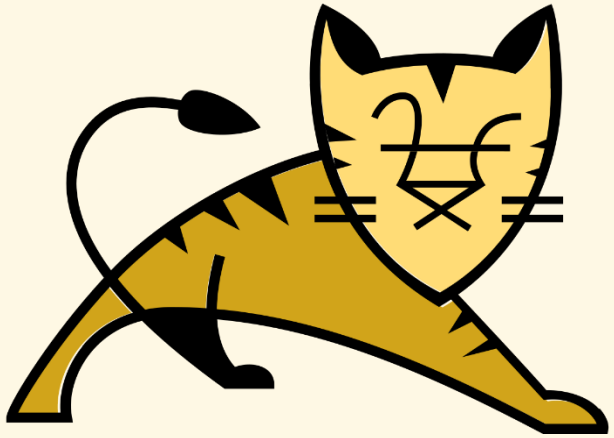
- This servlet will say "Hello!" (in HTML)
  - ```
    package com.example;

    import java.io.*;
    import jakarta.servlet.http.*;

    public class HelloServlet extends HttpServlet {
      @Override
      public void doGet(HttpServletRequest req,
        HttpServletResponse res) throws IOException {
        res.setContentType("text/html");

        try (PrintWriter out = res.getWriter()) {
          out.println("<html><head><title>Servlet Example</title></head>");
          out.println("<body> Hello! </body></html>");
        }
      }
    }
    ```


    ```

Servlet Configuration

- The web application configuration file `web.xml` identifies servlets and defines a mapping from requests to servlets

```
<web-app>
  <servlet>
    <servlet-name>HelloServlet</servlet-name>
    <servlet-class>com.example.HelloServlet</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>HelloServlet</servlet-name>
    <url-pattern>/hello</url-pattern>
  </servlet-mapping>
<web-app>
```

An identifying name for the servlet

The servlet's package and class names

The pathname used to invoke the servlet (relative to the web application URL)

- Alternatively, just put this annotation to the servlet class:

```
@WebServlet("/hello")
public class WelcomeServlet extends HttpServlet { ... }
```

Step by Step

1) Create a Maven webapp project (`maven-archetype-webapp`)
2) Change the JDK version in `pom.xml` to make it compatible with Tomcat:
   - ```
     <properties>
       <maven.compiler.source>11</maven.compiler.source>
       <maven.compiler.target>11</maven.compiler.target>
     </properties>
     ```
3) Add dependency to support servlets: `jakarta.jakartaee-web-api`
4) Write `HelloServlet.java` in `src/main/java/com/example/...` folder
5) Add an endpoint to `web.xml`
6) Package and deploy the output WAR file on Tomcat
7) Open https://localhost:8080/hello to check the result
     ```

Apache Tomcat

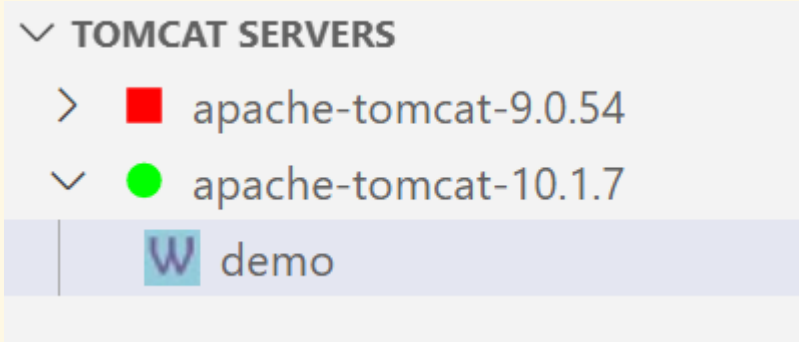
- A free servlet and JSP engine (Servlet/JSP container)



- Installation:
  - Download and unzip from:
    - http://jakarta.apache.org/tomcat/
  - Make sure the Tomcat version is compatible the JDK you use:
    - Tomcat 10 with JDK 11
    - Tomcat 11 (not yet released as of 2023) with JDK 17

Deployment from VS Code (for Development Purpose)

- In the **Tomcat Server**s view, add one
- Right click on the added server, choose **Start**
- To deploy a WAR file, right click on it and choose **Run/Debug on Tomcat Server**
  - If the app is launched without error, now you should see the app attached under the Tomcat server



Manual Deployment (for Production Purpose)

- Install Tomcat as a service:
  - Open the file *tomcat-folder*\conf\tomcat-users.xml and add the following line:
    - `<user username="admin" password="your-password" roles="manager-gui"/>`
  - Run tomcat-folder\bin\TomcatXXw.exe then:
    - On **Startup** tab, change **Mode** to **Java**
    - On **Shutdown** tab, change **Mode** to **Java**
  - Open a terminal to *tomcat-folder*\bin and run:
    - `service install`
- Start the service:
  - Open Windows' **Services** settings, find **Apache Tomcat…** then click **Start**
- To deploy and manage apps:
  - Open https://localhost:8080/manager/html from a browser

Environment Entries

- Servlets can obtain configuration information at run-time from the configuration file (`web.xml`): a file name, a security code, a database password,…
- Example:

```
<env-entry>
  <description>Security code for encryption</description>
  <env-entry-name>security-code</env-entry-name>
  <env-entry-value>hD%lo5*1</env-entry-value>
  <env-entry-type>java.lang.String</env-entry-type>
</env-entry>
```

Environment Entries

- Getting value from the servlet code:
  - ```
    @Override
    public void init() throws ServletException {
        super.init();

        try {
            Context envCtx = (Context)(new InitialContext())
                .lookup("java:comp/env");
            securityCode = (String)envCtx.lookup("security-code");
        } catch (NamingException e) {
            e.printStackTrace();
        }
    }
    ```


    ```

Handling Form Submission: GET Method

```
res.setContentType("text/html");

String name = req.getParameter("name");
// in production code, input validation and
// sanitization is necessary here for security!

try (PrintWriter  out = res.getWriter()) {
    out.println("<html><head><title>Servlet Example</title></head><body>");

    if (name == null || name.isEmpty()) {
        out.println("<form method='GET'>Enter your name:");
        out.println("<input type='text' name='name' /></form>");
    } else out.println("<p>Hello " + name + "!</p>");

    out.println("</body></html>");
}
```

Serving Static Contents

- Using a mapping to the `default` servlet (provided by Tomcat)
  - ```
    <servlet-mapping>
      <servlet-name>default</servlet-name>
      <url-pattern>*.jpg</url-pattern>
    </servlet-mapping>
    ```
- Put static contents in `src\main\webapp` folder, e.g.:
  - http://localhost:8080/demo/image/car.jpg
    → `src\main\webapp\image\car.jpg`
    ```

Exercises

1. Create a webpage showing a personal profile, which includes a photo
2. Create a web form allowing the user to enter two values, and show their product when the user submits them

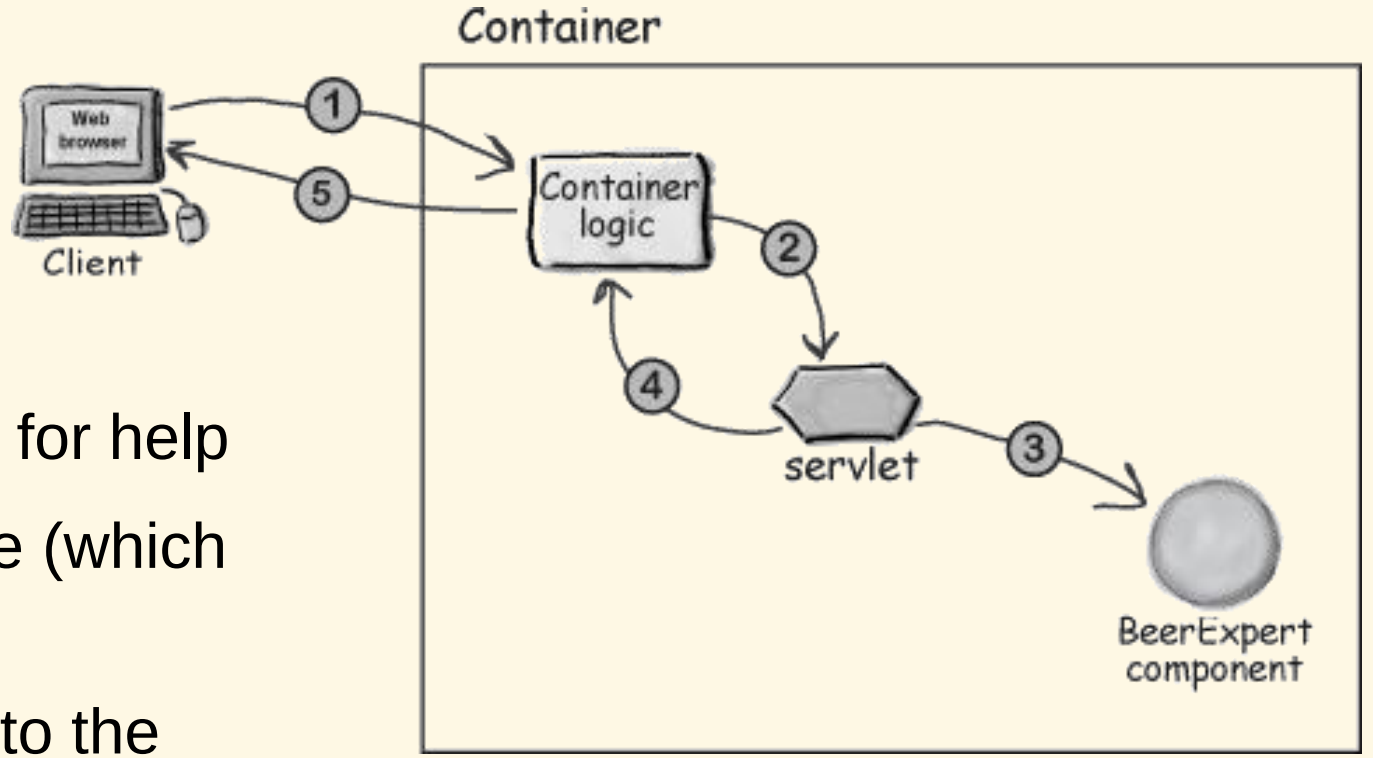
JSP (Java Server Page)

Introduction

- Server-side Java:
  - Scheme 1:
    - HTML files placed at location for web pages
    - Servlets placed at special location for servlets
    - Call servlets from HTML files
  - Scheme 2:
    - JSP: HTML + servlet codes + JSP tags
    - Placed at location for web pages
    - Converted to normal servlets when first accessed

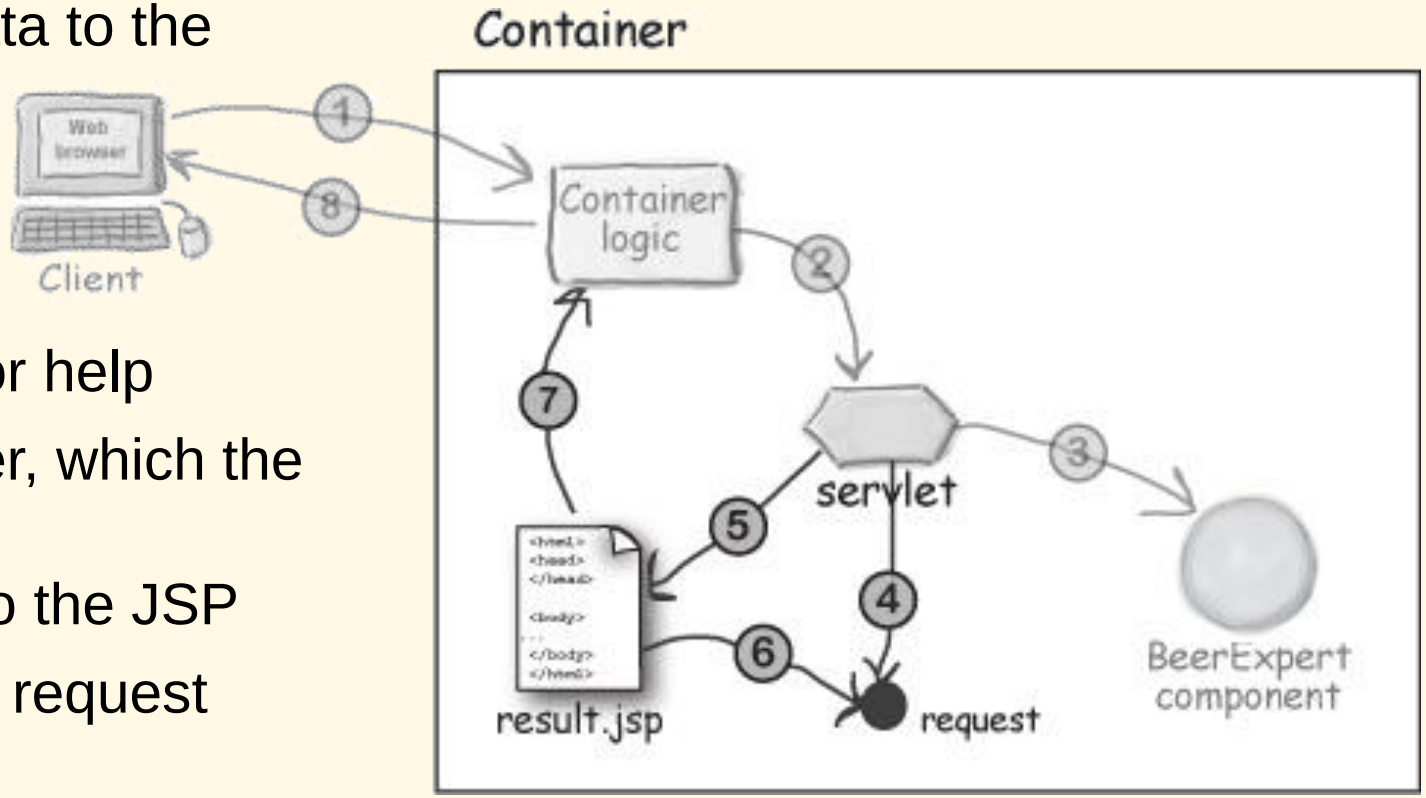
Scheme 1: What's Working So Far

1) The browser sends the request data to the Container
2) The Container finds the correct servlet based on the URL, and passes the request to the servlet
3) The servlet calls the BeerExpert for help
4) The servlet outputs the response (which prints the advice)
5) The Container returns the page to the happy user



Scheme 2: What We Want

1) The browser sends the request data to the Container
2) The Container finds the correct servlet based on the URL, and passes the request to the servlet
3) The servlet calls the BeerExpert for help
4) The expert class returns an answer, which the servlet adds to the request object
5) The servlet forwards the request to the JSP
6) The JSP gets the answer from the request object
7) The JSP generates a page for the Container
8) The Container returns the page to the happy user



JSP Example

- Create `time.jsp` file and put it in `webapp` folder:

```
<html>
    <head>
        <title>Example JSP</title>
    </head>
    <body>
        <h2>Current time: <%= new java.util.Date() %></h2>
    </body>
</html>
```

- JSP are converted to servlet at the first access

JSP Ingredients

- Regular HTML
  - Simply “passed through” to the client by the servlet created to handle the page
- JSP constructs
  - *Scripting elements*: let you specify Java code that will become part of the resultant servlet
    - Expressions: Evaluated and inserted into the output
    - Scriptlets: Inserted into the servlet’s service method
    - Declarations: Inserted into the body
  - *Directives*: let you control the overall structure of the servlet
  - *Actions*: let you specify existing components that should be used, and control the behavior of the JSP engine
    - JavaBeans: a type of components frequently used in JSP

Scripting Elements: Expressions

- Processing:
  - Evaluated, converted to a string, and inserted in the page
  - At run-time (when the page is requested)
- Several variables predefined to simply JSP expressions:
  - `request`, the `HttpServletRequest`
  - `response`, the `HttpServletResponse`
  - `session`, the `HttpSession` associated with the request (if any)
  - `out`, the `PrintWriter` (a buffered version of type `JspWriter`) used to send output to the client
- Example:

```
<h2>Current time: <%= new java.util.Date() %></h2>
<p>Your hostname: <%= request.getRemoteHost() %></p>
```

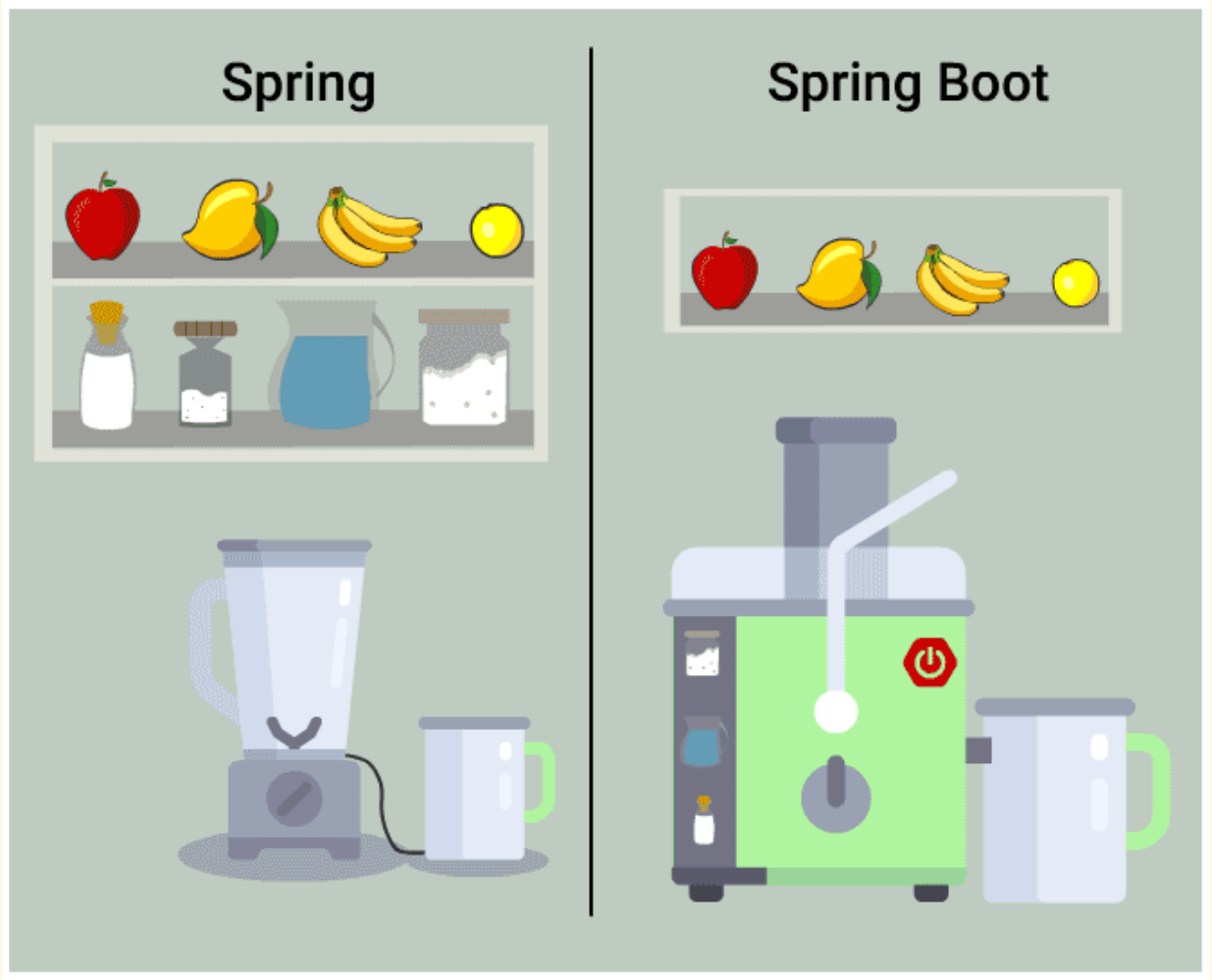
Scripting Elements: Scriptlets

- Processing:
  - Inserted into the servlet's service method *exactly* as written
  - Can access the same predefined variables as JSP expressions

- Example:
  - ```
    <% String queryData = request.getQueryString();
       out.println("Attached GET data: " + queryData); %>
    ```
    ```

Scripting Elements: Declarations

- Processing:
  - Inserted into the main body of the servlet class (outside of the service method processing the request)
  - Normally used in conjunction with JSP expressions or scriptlets

- Example:
  -
```
<%! private int accessCount = 0; %>

<p>Accesses to page since server reboot:
    <%= ++accessCount %></p>
```

JSP Directives

- Affect the overall structure of the servlet class
- Two commonly used directive types:
  - Page directives:
    - `<%@ page import="java.util.*" %>`
    - `<%@ page language="java" import="java.util.*" %>`
    - `<%@ page contentType="text/plain" %>`
      - Same as : `<% response.setContentType("text/plain"); %>`
    - `<%@ page session="true" %>`
  - Include directives: to include files at the time the JSP page is translated into a servlet (static including)
    - `<%@ include file="/navbar.html" %>`

JSP Actions

- JSP actions control the behavior of the servlet engine. Let one
  - Dynamically insert a file
  - Forward the user to another page
  - Reuse JavaBeans components
  - ...
- Include action: Inserts the file at the time the page is requested
  - Differs from the include directive, which inserts file at the time the JSP page is translated into a servlet
  - `<jsp:include page="show-product.jsp" flush="true" />`
- Forward action: Redirect to the page specified
  - `<jsp:forward page="payment.jsp?orderId=<%= getOrderId() %>" />`

JSP vs Servlet

Servlet	JSP
HTML code in Java	Java-like code in HTML
Not easy to author	Very easy to author
When making a change, one need to recompile and redeploy	Compiled into a servlet

Benefits of Using JSP

- Contents and display logic (or presentation logic) are separated
- Web application development can be simplified because business logic is captured in the form of JavaBeans or custom tags while presentation logic is captured in the form of HTML template
- Because the business logic is captured in component forms, they can be reused in other web applications
- And again, for web page authors, dealing with JSP page is a lot easier than writing Java code
- And just like servlet technology, JSP technology runs over many different platforms

JSP with Servlet

Benefits of Using JSP with Servlet

- In practice, both servlet and JSP are very useful in MVC model
  - Servlet plays the role of Controller
  - JSP plays the role of View
- Exploit both two technologies
  - The power of servlet is “controlling and dispatching”
    - Servlet passes data to JSP:
      - `req.setAttribute("attribute-name", value);`
    - Servlet dispatches the request to JSP:
      - `req.getRequestDispatcher("page.jsp").forward(req, res);`
  - The power of JSP is “displaying”:
    - JSP gets data passed by the servlet:
      - `request.getAttribute("attribute-name");`

Example:

- The servlet (see `WeatherServlet.java` and `WeatherInfo.java`):
  - ```
    String city = req.getParameter("city");
    WeatherInfo weather = WeatherInfo.getWeatherInfo(city);
    req.setAttribute("city", city);
    req.setAttribute("weather", weather);
    req.getRequestDispatcher("weather.jsp").forward(req, res);
    ```
- The JSP (see `weather.jsp`):
  - ```
    <%@ page import = "com.example.WeatherInfo" %>

    <h2>City: <%= request.getAttribute("city") %></h2>

    <% WeatherInfo weather = (WeatherInfo)request.getAttribute("weather"); %>
    <p>Status: <%= weather.getStatus() %></p>
    <p>Temperature: <%= weather.getTemperature() %></p>
    <p>Humidity: <%= weather.getHumidity() %></p>
    ```

Exercise

- Rewrite the calculation web app using servlet and JSP

Spring Boot

Spring and Spring Boot

- Spring is a framework to provides comprehensive infrastructure support for developing Java applications: JDBC, MVC, ORB,…
- Spring Boot is an extension that makes it easy to create Spring based applications
  - It's mostly used for the web development



Creating a Spring Boot Web App

- From the **Command Palette** in VS Code, choose **Spring Initializr: Create a Maven Project…** then follow the instructions, with attention to:
  - Choose WAR as packaging type
  - Add Spring Web as a dependency
- Create `Hello.java` source file with the following code:

```
@RestController
public class Hello {
    @GetMapping("/welcome")
    public String welcome() {
        return "<html><body>Welcome</body></html>";
    }

    @GetMapping("/hello")
    public String hello(@RequestParam String name) {
        return "<html><body>Hello " + name + "!</body></html>";
    }
}
```

Remarks

- No need to set up Tomcat, as an embedded Tomcat instance is added automatically to the project
- Spring Boot uses a system of annotations to minimize the code writing

@Controller, @ResponseBody

- `@Controller` allows to auto-detect implementation classes through the classpath scanning
  - Use in combination with a `@RequestMapping` for request handling methods
- `@ResponseBody` tells a controller that the object returned is automatically serialized into JSON and passed back into the `HttpResponse` object
- Example:
  - ```
    @Controller
    public class Hello {
        @GetMapping("/")
        public String home() {
            return "home";
        }

        @GetMapping("/welcome")
        @ResponseBody
        public String welcome() {
            return "<html><body>Welcome</body></html>";
        }
    }
    ```
    ```

@RestController

- Is the combination of @Controller and @ResponstBody for more convenience
- Example
  - ```
    @RestController
    public class Hello {
        @GetMapping("/welcome")
        public String welcome() {
            return "<html><body>Welcome</body></html>";
        }
    }
    ```
    ```

@RequestMapping, @GetMapping, @PostMapping

- Used to map web requests to Spring Controller methods
- Examples:
  - Map any method:
    - `@RequestMapping("/path")`
  - Map GET method:
    - `@GetMapping("/path")`
    - `@RequestMapping(value = "/path", method = RequestMethod.GET)`
  - Map POST method:
    - `@PostMapping("/path")`
    - `@RequestMapping(value = "/path", method = RequestMethod.POST)`
  - Map multiple paths:
    - `@PostMapping(value = {"/path1", "/path2"})`

@PathVariable

- Binds parts of the mapping URL to variables
- Example:

```
@GetMapping("/hello/{name}/{country}")
@ResponseBody
public String hello(
    @PathVariable String name,
    @PathVariable("country") String countryName
) {
    return "<html><body>Hello " +
        name + " from " + countryName +
        "!</body></html>";
}
```

@RequestParam

- Allows to get the value of a GET or POST request parameter
- Example:

```
@GetMapping("/hello")
@ResponseBody
public String hello(
        @RequestParam String name,
        @RequestParam(name = "country", required = false)
            String countryName
) {
        return "<html><body>Hello " + name +
            (countryName == null ? "" : " from " + countryName) +
            "!</body></html>";
}
```

Spring Boot with JSP (1)

- Spring Boot plays as controller, JSP as view
  - Add dependencies: `javax.servlet` / `jstl`, `org.apache.tomcat.embed` / `tomcat-embed-jasper`
  - Settings (`src\main\resources\application.properties`):
    - ```
      spring.mvc.view.prefix=/WEB-INF/jsp/
      spring.mvc.view.suffix=.jsp
      ```
- Example:
  - Controller:
    - ```
      @Controller
      @GetMapping("/hello/{name}")
      public String hello(Model model, @PathVariable("name") String name) {
          model.addAttribute("name", name);
          return "hello";
      }
      ```
  - View (`src\main\webapp\WEB-INF\jsp\hello.jsp`):
    - ```
      <h2>Hello ${name}!</h2>
      ```
      ```

Spring Boot with JSP (2)

- Weather example:
  - Controller:
    - Reuse the `WeatherInfo` class in the JSP section
    - ```
      @Controller
      @GetMapping("/weather/{city}")
      public String weather(Model model, @PathVariable("city") String city) {
          model.addAttribute("city", city);
          model.addAttribute("weather", weatherInfo.getWeatherInfo(city));
          return "weather";
      }
      ```
  - View (`src\main\webapp\WEB-INF\jsp\weather.jsp`):
    - ```
      <h2>City: ${city}</h2>
      <p>Status: ${weather.getStatus()}</p>
      <p>Temperature: ${weather.getTemperature()}</p>
      <p>Humidity: ${weather.getHumidity()}</p>
      ```

Serving Static Contents

- With the default Maven project setup, put static contents into:
  - `src\main\resources\static`
- To change the default static content folders, add this setting to `application.properties`:
  - `spring.web.resources.static-locations=classpath:/static-folder1,classpath:/static-folder2`

Exercise

- Rewrite the calculation web app using Spring Boot and JSP

More on Web

- HTML
- CSS for presentation
- Client-side scripting with JavaScript
- Cookie handling
- Session handling
- …