What is a Servlet?

A **Servlet** is a Java class that runs on a **Java-enabled web server** or **servlet container** (like **Apache Tomcat**, **Jetty**, etc.). It is used to extend the capabilities of a web server by processing client requests (usually HTTP requests) and generating responses (typically HTML, JSON, XML, etc.). Servlets are part of the **Java EE** (Enterprise Edition) specification but are now widely used in **Jakarta EE**.

In simple terms, servlets allow Java applications to handle web requests and generate dynamic web content.

Key Features of Servlets:

1. **Handles HTTP Requests**: Servlets receive HTTP requests from a client (such as a web browser), process the request (e.g., query a database, perform some business logic), and return an HTTP response.
2. **Server-side**: Servlets run on the server-side, not in the user's browser (unlike JavaScript). They provide a way for a web application to dynamically generate content based on client requests.
3. **Persistent**: Servlets are designed to be reusable and persistent within a servlet container. The servlet container manages the lifecycle of servlets, such as creating, initializing, and destroying them when required.

How Does a Servlet Work?

1. **Client Sends HTTP Request**:
   - The client (usually a browser) sends an HTTP request to a server.
2. **Request is Routed to Servlet**:
   - The web server routes the request to the appropriate servlet based on URL patterns defined in the server's configuration (usually in the `web.xml` file or servlet annotations).
3. **Servlet Processes the Request**:
   - The servlet processes the request, which may involve tasks like:
     - Querying a database.
     - Processing form data.
     - Authenticating users.
     - Generating dynamic content (HTML, JSON, etc.).
4. **Servlet Sends Response**:
   - Once the servlet has processed the request, it generates an HTTP response, typically in the form of HTML or JSON, and sends it back to the client.
5. **Client Receives Response**:
   - The client (browser or API client) receives the response and processes or renders it as needed (e.g., displaying an HTML page).

Servlet Lifecycle

The lifecycle of a servlet is managed by the **Servlet Container**. It involves the following stages:

1. **Loading and Instantiation**:
   - When a request is made to a servlet, the servlet container loads the servlet class and creates an instance of it.
2. **Initialization (`init()` method)**:
   - The servlet is initialized using the `init()` method, which is called once when the servlet is first loaded. This is where setup or configuration tasks can be performed.
3. **Request Handling (`service()` method)**:
   - The servlet container calls the `service()` method to handle each request. This method processes the incoming request and generates a response.
4. **Destruction (`destroy()` method)**:
   - The `destroy()` method is called when the servlet is removed from memory, such as when the web application is shut down or when the servlet is no longer in use. This allows cleanup resources (like database connections) to be released.

Example of a Simple Servlet

Here's an example of a very basic **HTTP servlet** that handles a `GET` request and responds with a simple message:

```
package com.example;

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

public class HelloWorldServlet extends HttpServlet {

    // Called to handle GET requests
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws Se
        // Set response content type
        response.setContentType("text/html");

        // Writing a simple response to the output
        PrintWriter out = response.getWriter();
        out.println("<h1>Hello, World!</h1>");
    }
}
```

Key Components in a Servlet Example:

1. **Extends `HttpServlet`**:
   - A servlet class typically extends the `HttpServlet` class (or other specialized servlet classes), which provides methods like `doGet()`, `doPost()`, `doPut()`, etc., for handling different HTTP methods.
2. **`doGet()` or `doPost()` Methods**:
   - The `doGet()` method is called when the server receives an HTTP GET request. Similarly, `doPost()` is called for POST requests. These methods handle the actual logic for processing requests.
3. **`PrintWriter` for Response**:
   - The `PrintWriter` object is used to write the HTTP response back to the client. In the example, we are simply writing some HTML content.

Deploying a Servlet:

To deploy a servlet in a Java web application:

1. **Create the Servlet Class**:
   - Write the Java class that extends `HttpServlet` and implements `doGet()`, `doPost()`, etc.
2. **Configure the Servlet**:
   - In the `web.xml` file, define the servlet and URL mapping, like so:

```
<servlet>
    <servlet-name>HelloWorldServlet</servlet-name>
    <servlet-class>com.example.HelloWorldServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>HelloWorldServlet</servlet-name>
    <url-pattern>/hello</url-pattern>
</servlet-mapping>
```

   This maps the servlet to the `/hello` URL.
3. **Build the Web Application**:
   - You can package your Java web application as a **WAR** file (Web Application Archive) and deploy it to a servlet container like **Apache Tomcat**.
4. **Start the Web Server**:
   - Once the web server (e.g., Tomcat) is started, you can access the servlet by visiting the mapped URL (`http://localhost:8080/your-web-app/hello`).

Servlet Mapping with Annotations (Optional):

Instead of using `web.xml`, you can map the servlet using **annotations** in modern Java web applications (Servlet 3.0+):

```
@WebServlet("/hello")
public class HelloWorldServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws Se
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<h1>Hello, World!</h1>");
    }
}
```

Conclusion:

- A **Servlet** is essentially a Java class used to handle HTTP requests and generate responses on the server side.
- It forms the basis of web applications in Java, allowing dynamic content generation and interaction with data (e.g., databases).
- Servlets are managed by a **servlet container** like **Tomcat**, which handles their lifecycle and request routing.

Servlets are a fundamental part of Java web development, and they play a crucial role in building scalable, dynamic web applications.

Servlet Example in our OnBoarding Project:

The `AssignProcessServlet` is a **Java Servlet** that acts as a backend endpoint for handling HTTP requests related to assigning onboarding processes to teams in your application. It receives HTTP requests (typically `POST` requests), processes the incoming data (in this case, the team and process details), and responds with appropriate messages.

In the context of your project, it is used for assigning a specific onboarding process to a particular team.

`AssignProcessServlet.java`

```
package com.company.onboarding.backend.servlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.annotation.*;
```

```java
import com.company.onboarding.backend.models.AssignProcessRequest;
import com.company.onboarding.backend.services.TeamService;
import com.fasterxml.jackson.databind.ObjectMapper; // For parsing JSON

@WebServlet("/assignProcess")  // Servlet mapped to /assignProcess URL
public class AssignProcessServlet extends HttpServlet {

    // Create a TeamService instance to use the business logic
    private TeamService teamService = new TeamService();
    private ObjectMapper objectMapper = new ObjectMapper(); // Reusable ObjectMapper instan

    // Override the doPost method to handle the POST request
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws S
        // Set the response content type to JSON
        response.setContentType("application/json");

        // Read the incoming JSON request body using request.getReader()
        StringBuilder sb = new StringBuilder();
        String line;

        try (BufferedReader reader = request.getReader()) {
            while ((line = reader.readLine()) != null) {
                sb.append(line);
            }
        } catch (IOException e) {
            // Log and return an error response if reading the request fails
            handleError(response, HttpServletResponse.SC_BAD_REQUEST, "Invalid JSON data");
            return;
        }

        // Deserialize the JSON request to AssignProcessRequest object
        AssignProcessRequest assignRequest;
        try {
            assignRequest = objectMapper.readValue(sb.toString(), AssignProcessRequest.class
        } catch (Exception e) {
            // Handle JSON parsing errors
            handleError(response, HttpServletResponse.SC_BAD_REQUEST, "Failed to parse reque
            return;
        }

        // Validate the input data (e.g., teamId and processId must not be null or negative)
        if (assignRequest.getTeamId() <= 0 || assignRequest.getProcessId() <= 0) {
            handleError(response, HttpServletResponse.SC_BAD_REQUEST, "Invalid team ID or pr
            return;
        }
```

```
        // Call the service to assign the process to the team
        boolean success = teamService.assignProcessToTeam(assignRequest.getTeamId(), assignR

        // Prepare the response based on the result of the assignment operation
        if (success) {
            response.setStatus(HttpServletResponse.SC_OK);
            sendResponse(response, "Onboarding process assigned successfully");
        } else {
            response.setStatus(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
            sendResponse(response, "Failed to assign onboarding process");
        }
    }

    /**
     * Helper method to send JSON response with a message
     * @param response The HTTP response object
     * @param message The message to send in the response body
     * @throws IOException
     */
    private void sendResponse(HttpServletResponse response, String message) throws IOExcepti
        PrintWriter out = response.getWriter();
        out.println("{\"message\": \"" + message + "\"}");
    }

    /**
     * Helper method to handle error responses
     * @param response The HTTP response object
     * @param statusCode The HTTP status code (e.g., 400, 500)
     * @param errorMessage The error message to include in the response
     * @throws IOException
     */
    private void handleError(HttpServletResponse response, int statusCode, String errorMessa
        response.setStatus(statusCode);
        sendResponse(response, errorMessage);
    }
}
```

Key Concepts:

1. **Servlet**:
   - A **Servlet** is a Java class that is designed to handle HTTP requests and responses.
   - It runs on a **web server** (like Tomcat, Jetty, or any Java EE server) and interacts with clients (browsers or other HTTP clients).
   - Servlets are usually used in web applications to provide dynamic

content and business logic processing.

2. **AssignProcessServlet**:
   - This servlet is responsible for processing the business logic when a request to assign an onboarding process to a team is made.
   - It listens to HTTP requests (specifically POST requests), extracts data from the request body, and sends a response to the client indicating whether the operation was successful.

Breakdown of `AssignProcessServlet`:

1. Servlet Declaration:

- The class extends `HttpServlet`, which is the base class for all servlets that handle HTTP requests in Java.
- The servlet is mapped to a URL pattern (e.g., `/assignProcess`) in the `web.xml` file or via annotations.

2. doPost Method:

- `doPost()` is an HTTP method that handles **POST requests**. This method processes the incoming request, performs necessary business logic (like calling a service), and sends a response.
- In your case, the `doPost()` method receives an HTTP request, extracts the body (in JSON format), and passes the data to the `TeamService` to assign a process to a team.

3. Reading and Parsing JSON:

- The request body is expected to be in **JSON format**, so the servlet reads the input stream, converts it into a `String`, and then uses an `ObjectMapper` from the Jackson library to **deserialize** the JSON string into a Java object (`AssignProcessRequest`).
- This allows the servlet to easily handle structured data coming from the frontend.

4. Business Logic Call:

- The `TeamService` class is invoked to process the assignment of the onboarding process to the team. If the assignment is successful, the servlet prepares a success response. If not, it sends an error message.

5. Response Handling:

- After processing the request, the servlet sends a JSON response back to the frontend with a message (either success or failure).
- The response is sent with the `Content-Type` set to `application/json`, and the message is sent in the response body.

Important Components:

1. **AssignProcessRequest**:
   - This is the class that represents the data sent from the frontend (JSON data). It holds the `teamId` and `processId` that the frontend sends to the backend to request the process assignment.
   - For example, `AssignProcessRequest` might look like this:

```
public class AssignProcessRequest {
    private int teamId;
    private int processId;

    // Getters and Setters
    public int getTeamId() {
        return teamId;
    }

    public void setTeamId(int teamId) {
        this.teamId = teamId;
    }

    public int getProcessId() {
        return processId;
    }

    public void setProcessId(int processId) {
        this.processId = processId;
    }
}
```

2. **TeamService**:
   - The `TeamService` class handles the business logic. In this case, it contains the method `assignProcessToTeam(int teamId, int processId)`, which performs the actual process assignment logic.
   - Example:

```
public class TeamService {
    public boolean assignProcessToTeam(int teamId, int processId) {
        // Logic to assign the process to the team (e.g., update a database, etc.)
        return true; // Return true if successful
    }
}
```

3. **ObjectMapper**:
   - The `ObjectMapper` from the Jackson library is used to **deserialize** the JSON request body into a Java object (`AssignProcessRequest`).
   - Example usage:

```
ObjectMapper objectMapper = new ObjectMapper();
AssignProcessRequest assignRequest = objectMapper.readValue(sb.toString(), AssignPr
```

4. **Response**:

- The servlet sends back a **JSON response** containing a success or error message. This message is displayed to the frontend after the request is processed.
- Example response:

```
{
  "message": "Onboarding process assigned successfully"
}
```

How it works in the application:

1. **Frontend makes a request**:
   - A frontend (e.g., JavaScript in the browser) makes an HTTP `POST` request to the `/assignProcess` endpoint with JSON data, containing the `teamId` and `processId`.
2. **Servlet processes the request**:
   - The servlet reads the JSON data, converts it into a Java object, calls the `TeamService` to perform the logic (assigning the process), and sends back a response.
3. **Frontend handles the response**:
   - After the frontend receives the response, it may display a success or failure message to the user, depending on the JSON response received.

Improvements with Annotations (Servlet 3.0+):

Instead of using `web.xml` for servlet mapping, you can use **Servlet annotations** (available in Servlet 3.0 and above), which simplify the configuration.

For example, you can use the `@WebServlet` annotation to map the servlet directly to a URL pattern:

```
@WebServlet("/assignProcess")
public class AssignProcessServlet extends HttpServlet {
    // Servlet implementation
}
```

With the `@WebServlet` annotation, the servlet is automatically mapped to the `/assignProcess` URL pattern, and you no longer need to configure it in the `web.xml`.

Conclusion:

The `AssignProcessServlet` is a backend component of your web application that listens for requests to assign an onboarding process to a team. It reads incoming data, processes it with business logic, and sends a response back to the client. This servlet can be improved and simplified using annotations for servlet mapping.

Key Properties:

1. **Annotations**:
   - The `@WebServlet("/assignProcess")` annotation is used to map the servlet to the `/assignProcess` URL pattern. This eliminates the need for `web.xml` configuration.
2. **Error Handling**:
   - The method `handleError()` is created to handle different error cases (e.g., invalid JSON, invalid IDs). This reduces code duplication and centralizes error handling.
3. **JSON Parsing**:
   - The `ObjectMapper` is used to deserialize the incoming JSON data into the `AssignProcessRequest` model. This is done inside a `try-catch` block to handle potential parsing errors gracefully.
4. **Request Validation**:
   - The servlet validates the `teamId` and `processId` to ensure they are positive values. If they are not valid, it responds with an HTTP 400 status code (`BAD_REQUEST`) and an appropriate error message.
5. **Response Handling**:
   - The response is sent in JSON format using the helper method `sendResponse()`. Depending on the success of the process assignment, the servlet responds with either a success message or an error message.
   - The HTTP status codes are set based on the outcome:
     - **200 OK** for successful assignments.
     - **400 BAD REQUEST** for invalid inputs (e.g., invalid JSON or IDs).
     - **500 INTERNAL SERVER ERROR** for failures in the assignment process.
6. **Cleaner Code**:
   - The use of `try-with-resources` for reading the request body ensures that the `BufferedReader` is properly closed after reading, even if an exception occurs.

Improvements in Action:

1. If the request contains invalid JSON or missing parameters, the servlet will send a clear error message along with a `400` HTTP status code.
2. The servlet ensures that only valid team IDs and process IDs are processed, helping to prevent errors at later stages.
3. The client (frontend) will receive JSON responses that clearly indicate the success or failure of the operation.

Example Request:

The frontend (like `teamController.js`) might send a POST request like this:

```
fetch('/assignProcess', {
    method: 'POST',
    headers: {
        'Content-Type': 'application/json'
    },
    body: JSON.stringify({
        teamId: 1,  // valid team ID
        processId: 101  // valid process ID
    })
})
.then(response => response.json())
.then(data => {
    alert(data.message);
})
.catch(error => {
    console.error('Error:', error);
});
```

1. Using a Relative URL (`/assignProcess`)

When you use a relative URL like `/assignProcess`, it refers to the same domain and port that the client (frontend) is currently running on. This is generally the best practice when your frontend and backend are served from the same server or if you're using the same domain for both.

Why it's a good option:

- **Simpler**: It's shorter and more maintainable.
- **Environment flexibility**: When you're working in different environments (e.g., development, staging, or production), the relative URL will automatically adapt to the server your frontend is running on. It avoids hardcoding the server address.
- **No duplication of URLs**: If you're deploying the project on a server where both frontend and backend share the same domain (e.g., `http://example.com`), you don't need to repeat the URL or worry about port numbers.

For example:

```
fetch('/assignProcess', {
    method: 'POST',
    headers: {
        'Content-Type': 'application/json'
    },
    body: JSON.stringify({
        teamId: 1,
        processId: 101
```

```
    })
})
.then(response => response.json())
.then(data => {
    alert(data.message);
})
.catch(error => {
    console.error('Error:', error);
});
```

In this case, `/assignProcess` is automatically resolved based on the base URL of the current page. So, if your frontend is served at `http://localhost:8080/onboarding-platform/`, the request will go to `http://localhost:8080/onboarding-platform/assignProcess`.

Use case:

- If both your **frontend** (HTML, CSS, JS) and **backend** (Servlets) are deployed on the same server, and you don't need to specify a domain or port.

2. Using an Absolute URL (`http://localhost:8080/onboarding-platform/assignProcess`)

If you use an absolute URL, like `http://localhost:8080/onboarding-platform/assignProcess`, you're specifying the full path to your backend, including the domain and port. This method is useful when your frontend and backend are running on **different servers** or **ports**, and you need to make sure the frontend knows exactly where to send requests.

Why it might be necessary:

- **Different origins**: If your frontend is served from one domain (e.g., `http://localhost:3000`) and your backend is served from another (e.g., `http://localhost:8080`), you'll need to specify the full URL.
- **Development or Debugging**: During development, your frontend might be served on a different port (e.g., `3000` for a React app, or a local file server), while your backend is running on a different port (e.g., `8080` for your Java servlet-based backend). In such cases, an absolute URL ensures the request is sent to the correct server.

For example:

```
fetch('http://localhost:8080/onboarding-platform/assignProcess', {
    method: 'POST',
    headers: {
        'Content-Type': 'application/json'
    },
    body: JSON.stringify({
        teamId: 1,
```

```
        processId: 101
    })
})
.then(response => response.json())
.then(data => {
    alert(data.message);
})
.catch(error => {
    console.error('Error:', error);
});
```

In this case, you are directly specifying the **server and port** where your backend is running. This is especially useful in situations where your frontend is not served from the same server as your backend (e.g., when using frameworks like React, Vue.js, or Angular that are served from `localhost:3000`).

Use case:

- When your **frontend** and **backend** are running on different ports during development (e.g., React on `3000`, Java servlet on `8080`).
- When you want to make sure the request goes to a **specific server** regardless of the current page.

Conclusion: Which one should you use?

- **During local development**, if both your frontend and backend are on the same server but different ports (e.g., frontend on `localhost:3000` and backend on `localhost:8080`), use the **absolute URL** (`http://localhost:8080/onboarding-platform/assignProcess`).
- **For production** or **when both frontend and backend are served from the same server and domain** (e.g., `http://example.com`), **use the relative URL** (`/assignProcess`).

In general: - **Relative URLs** are more flexible, maintainable, and the preferred choice. - **Absolute URLs** are more rigid but may be necessary in development if your frontend and backend are served from different origins (domains/ports).

Practical Example for Local Development:

If you're working locally with a backend on `localhost:8080` and a frontend on `localhost:3000`, you'd use:

```
fetch('http://localhost:8080/onboarding-platform/assignProcess', {
    method: 'POST',
    headers: {
        'Content-Type': 'application/json'
    },
    body: JSON.stringify({
```

```
        teamId: 1,
        processId: 101
    })
})
.then(response => response.json())
.then(data => {
    alert(data.message);
})
.catch(error => {
    console.error('Error:', error);
});
```

But when both frontend and backend are on the same server, **relative URL** like `/assignProcess` would work better:

```
fetch('/assignProcess', {
    method: 'POST',
    headers: {
        'Content-Type': 'application/json'
    },
    body: JSON.stringify({
        teamId: 1,
        processId: 101
    })
})
.then(response => response.json())
.then(data => {
    alert(data.message);
})
.catch(error => {
    console.error('Error:', error);
});
```

Best Practice:

For production environments where frontend and backend are deployed together, **always use relative URLs** unless you have a very specific reason to hardcode absolute URLs (like when the frontend and backend are on different domains).