Project Specification for Web-Based Onboarding Platform

Overview

The goal is to develop a web service platform for companies to manage the onboarding process of new employees, specifically for the stage after job acceptance but before the official start date. The platform will allow companies to sign up, create teams, define onboarding processes, assign those processes to teams, and track the status of each designated worker within the process stages. This project will be developed with HTML, CSS, and JavaScript (with no frameworks). It will involve server-side Java with JDBC for data persistence, following a RESTful approach to handle the communication between the backend and frontend.

Functional Requirements

1. User Authentication & Company Registration

- **Join Us Page**:
  - A welcoming page that allows companies to register and create a user.
  - Basic company details (name, contact email, etc.) are provided during registration.
  - The company can create a user with admin privileges (for managing teams, processes, and workers).

2. Company Dashboard

- **Overview**: A high-level dashboard that displays a summary of the company's teams, onboarding processes, and designated workers.
- **Features**:
  - View the list of teams, processes, and designated workers.
  - Quick overview of the onboarding stages for each team and its workers.
  - Link to create new teams or processes.

3. Team Dashboard

- **Overview**: A dashboard for each team showing detailed information about the team, its assigned onboarding process, and the workers.
- **Features**:
  - Team Information: Display team name, manager, and members.
  - Assigned Process: View the currently assigned onboarding process for the team.
  - Designated Workers: List of workers for the team, showing their status in the onboarding process (e.g., not started, in progress, completed).
  - Add New Worker: A link to a page where new workers can be added to the team.
  - Edit Team: Modify team details, including adding/removing workers and updating the onboarding process.

4. Onboarding Process Dashboard

- **Overview**: A page that displays information about the onboarding process, its assigned teams, and designated workers.
- **Features**:
  - View the onboarding process stages.
  - View teams assigned to the process and workers within those teams.
  - Add New Process: A link to a form where companies can define new onboarding processes.
  - Edit Process: Modify existing processes or detach them from teams.

5. Add/Edit Forms

- Forms for adding or editing the following entities:
  - **Company User**: A form to create or edit a company user in the platform.
  - **Team**: A form to create or modify a team (name, manager, etc.).
  - **Onboarding Process**: A form to create or edit an onboarding process (stages, process steps, etc.).
  - **Worker**: A form to add new workers to a team.

Data Entities & SQL Database Design

Entities (Java Classes)

- **User**: Represents a company's admin user.
  - **Fields**: `id`, `email`, `password`, `company_id`

- **Company**: Represents a company that uses the platform.
    - **Fields**: `id`, `name`, `email`, `address`
- **Team**: Represents a team in the company.
    - **Fields**: `id`, `team_name`, `manager_id (foreign key)`, `company_id (foreign key)`
- **Worker**: Represents an employee who is undergoing the onboarding process.
    - **Fields**: `id`, `name`, `status`, `team_id (foreign key)`
- **Onboarding Process**: Represents a predefined onboarding process for new employees.
    - **Fields**: `id`, `process_name`, `company_id (foreign key)`
- **Stage**: Represents the stages of the onboarding process.
    - **Fields**: `id`, `stage_name`, `process_id (foreign key)`

SQL Database (Recommended SQL Schema: PostgreSQL or MySQL)

- **Tables**:
    - `users (id, email, password, company_id)`
    - `companies (id, name, email, address)`
    - `teams (id, team_name, manager_id, company_id)`
    - `workers (id, name, status, team_id)`
    - `processes (id, process_name, company_id)`
    - `stages (id, stage_name, process_id)`

The relationships:

- **One-to-many**: A company can have multiple teams, and a team can have multiple workers.
- **Many-to-many**: A process can be assigned to multiple teams, and a team can have multiple processes assigned at different times.

Architecture & Components

Backend Architecture

- **Backend Language**: **Java**
- **Persistence Layer**: **JDBC** with SQL (PostgreSQL or MySQL)
- **Controller Layer**: Implement REST API controllers to interact with the frontend.
- **JDBC Servers**: Each backend feature will have its own separate server:
    1. **Users Server**: Handles authentication and user management.
    2. **Teams Server**: Manages team data (create, edit, view).
    3. **Processes Server**: Manages onboarding process data (create, edit, view).
    4. **Companies Server**: Manages company-specific data (view, edit).

Each controller will expose **REST API routes** to manage the data and interact with the frontend.

Frontend Architecture

- **Frontend Technology**: HTML, CSS, JavaScript (Vanilla JS, no frameworks)
- **Pages**:
    - **Join Us Page**: For company registration and user creation.
    - **Company Dashboard**: A summary of teams, processes, and workers.
    - **Team Dashboard**: Detailed info for teams, their assigned process, and workers.
    - **Onboarding Processes Dashboard**: Info about processes, teams assigned to them, and designated workers.
    - **Add/Edit Forms**: Forms for adding or editing company users, teams, processes, and workers.

The frontend will make **AJAX calls** to the RESTful APIs of the backend to fetch and send data. Each page will handle displaying the data and redirecting users to the appropriate form pages for creating or updating data.

Controller Layer & RESTful API Routes

Each backend server will expose specific REST routes to manage the data for the entities. Here are the main routes for each server:

1. **Users Server (Authentication & User Management)**
    - `POST /api/login`: Login for a company user.
    - `POST /api/signup`: Register a new company user.
2. **Teams Server**
    - `GET /api/teams`: Get all teams in the company.
    - `GET /api/team/{id}`: View team details.
    - `POST /api/team`: Create a new team.
    - `PUT /api/team/{id}`: Update team details.
    - `DELETE /api/team/{id}`: Delete a team.
3. **Processes Server**
    - `GET /api/processes`: Get all onboarding processes for a company.
    - `GET /api/process/{id}`: View details of a specific process.
    - `POST /api/process`: Create a new onboarding process.
    - `PUT /api/process/{id}`: Update an existing process.
    - `DELETE /api/process/{id}`: Delete a process.
4. **Companies Server**
    - `GET /api/companies`: View all companies using the platform.
    - `GET /api/company/{id}`: View a specific company's data.

Each controller will handle **GET**, **POST**, **PUT**, and **DELETE** requests and will communicate with the database through JDBC.

Roles & Team Structure

- **Team Leader (Senior Developer)**: Will oversee the entire development process, manage task delegation, and ensure deadlines are met.
- **Senior-Junior Pairs (3 Pairs)**: Each pair will be responsible for specific backend servers and frontend development tasks. The juniors will focus on frontend implementation, form development, and connecting the frontend with the API. Seniors will handle backend services, JDBC setup, REST API routes, and database management.

Frontend Dashboards Explanation & Wireframe Guidance

1. Company Dashboard

- **Overview**: The Company Dashboard serves as the entry point for administrators to manage their company's teams, processes, and workers. It will display a high-level summary and provide quick access to key areas for further management.
- **Dashboard Components**:
    - **Summary**: Displays a brief overview of the company's information (name, contact, etc.) and a high-level count of teams, processes, and workers.
    - **Teams**: A list of the teams within the company, with a link to the Team Dashboard.
    - **Processes**: A list of onboarding processes available to the company, with a link to the Process Dashboard.
    - **Designated Workers**: Quick summary of how many workers are assigned to each process.
- **Wireframe**:
    - **Header**: "Company Dashboard" with the company name, logo, and user profile.
    - **Sidebar**: Links to "Teams", "Processes", "Designated Workers", "Settings".
    - **Main Content Area**:
        - Summary section at the top with counts of teams, processes, and workers.
        - Below, lists of "Teams" and "Processes" with clickable links for further management.

2. Team Dashboard

- **Overview**: The Team Dashboard will display detailed information about each team, including their assigned onboarding process, workers, and their progress. This dashboard enables the team manager to track the team's onboarding progress and manage workers.
- **Dashboard Components**:

- **Team Information**: Basic info about the team such as team name, manager, and assigned onboarding process.
- **Onboarding Process**: A section showing the current onboarding process assigned to the team with clickable stages (linked to Process Dashboard).
- **Designated Workers**: A list of workers assigned to the team with their current onboarding status (e.g., "In Progress", "Completed").
- **Add New Worker**: A button to navigate to a form to add new workers to the team.

- **Wireframe**:

  - **Header**: "Team Dashboard" with the team name and manager.
  - **Sidebar**: Links to "Team Information", "Assigned Process", "Workers", "Add Worker".
  - **Main Content Area**:
    - Team info at the top.
    - Below, a list of stages in the assigned process.
    - Below that, a list of workers with their onboarding status.
    - An "Add New Worker" button to create a new worker.

3. Onboarding Processes Dashboard

- **Overview**: This dashboard provides an overview of all onboarding processes available within the company, as well as which teams are assigned to each process. It allows the creation of new processes and the editing of existing ones.

- **Dashboard Components**:

  - **Processes List**: A list of available processes with a summary of the stages involved.
  - **Assigned Teams**: Displays the teams assigned to each process.
  - **Add New Process**: A button to create a new process.
  - **Edit Process**: A button/link to edit existing processes.

- **Wireframe**:

  - **Header**: "Onboarding Processes" with the company name.
  - **Sidebar**: Links to "Processes", "Teams", "Designated Workers".
  - **Main Content Area**:
    - A list of processes, with links to see more details or edit.
    - A list of assigned teams under each process.
    - A button to create a new process.

4. Add/Edit Forms

- **Overview**: These pages provide forms for adding or editing data entities: teams, processes, workers, and companies.

- **Wireframe**:

  - **Header**: The title of the page (e.g., "Create New Team", "Edit Onboarding Process").
  - **Form Fields**:
    - For **Teams**: Team name, manager, assigned onboarding process.
    - For **Processes**: Process name, stages (multi-step), description.
    - For **Workers**: Worker name, email, team assignment.
  - **Form Buttons**: "Save", "Cancel".

Project Folder Structure (Model-View-Controller)

Here's a folder structure that will allow each pair to work on their respective tasks while ensuring separation of concerns. This structure follows the MVC (Model-View-Controller) design pattern: In a traditional Model-View-Controller (MVC) setup, controllers handle the application logic and interact with the models, while views are responsible for the user interface.

In your case, since you are working with a vanilla Java setup (no frameworks like Spring), servlets are essentially serving as the controllers. So instead of naming these classes UserController, TeamController, etc., we should call them UserServlet, TeamServlet, etc., since these servlets are the ones handling incoming requests (which is what the controllers would typically do in MVC).

```
/project-root
├── /backend
│   ├── /servlets
│   │   ├── UserServlet.java        // Handle user-related API requests (HTTP Requests → Services)
│   │   ├── TeamServlet.java        // Handle team-related API requests
│   │   ├── ProcessServlet.java     // Handle onboarding process API requests
│   │   └── CompanyServlet.java     // Handle company-related API requests
│   ├── /models
│   │   ├── User.java                  // User entity (company admin)
│   │   ├── Team.java                  // Team entity
│   │   ├── Process.java               // Onboarding process entity
│   │   ├── Worker.java                // Worker entity
│   │   └── Company.java               // Company entity
│   ├── /services
│   │   ├── UserService.java           // Logic for user operations
│   │   ├── TeamService.java           // Logic for team operations
│   │   ├── ProcessService.java        // Logic for process operations
│   │   └── CompanyService.java        // Logic for company operations
│   ├── /utils
│   │   ├── DBConnection.java         // Database connection setup
│   └── /config
│       ├── DBConfig.java             // DB configurations (JDBC setup)
│       └── AppConfig.java            // Application-level settings
└── /frontend
    ├── /assets
    │   ├── /css
    │   │   └── style.css            // General styles
    │   ├── /images
    │   └── /js
    │       └── app.js               // Frontend JS for AJAX requests
    ├── /views
    │   ├── companyDashboard.html
    │   ├── teamDashboard.html
    │   ├── processDashboard.html
    │   ├── addEditForm.html          // Template for adding/editing forms
    └── /controllers
        ├── companyController.js     // Frontend logic for company dashboard
        ├── teamController.js        // Frontend logic for team dashboard
        ├── processController.js     // Frontend logic for process dashboard
        ├── formController.js        // Logic for forms (create/edit)
        └── common.js                // Shared frontend logic (AJAX, validation)
```

- **Backend (Java)**:
  - **Controllers**: Will handle incoming API requests and return responses. Each pair can work on their respective controllers (one for users, teams, processes, or companies). Backend Controllers = Servlets: The servlets folder replaces the controllers. These servlets are now responsible for handling incoming HTTP requests, routing them to the appropriate service layer (where business logic is implemented), and sending back the response to the client (frontend).
  - **Models**: Represent the data entities, which can be shared across different backend components. Models are Java classes that represent the data entities of your application, like User.java, Team.java, Process.java, etc.
  - **Services**: Will contain the business logic, separate from the controllers. Each pair can work on implementing logic for their respective entities.
- **Frontend (HTML, CSS, JS)**:
  - **Views**: Will contain the HTML pages for rendering the dashboards and forms. Each pair can work on their respective dashboard and form view files.
  - **Controllers**: Will handle frontend logic, such as making AJAX calls to the backend and processing the responses. Each pair can handle the JS logic for their specific frontend views.

How It All Ties Together

Here's how a typical flow would work:

1. **Frontend**:
    - The user interacts with the frontend (e.g., a button click to create a user).
    - A request is made to the backend via **JavaScript** (AJAX call to a servlet, like `UserServlet`).
2. **Backend (Servlet)**:
    - The **UserServlet** receives the request, parses the incoming data (e.g., user info), and calls the corresponding **service** (e.g., `UserService.createUser()`).
3. **Service Layer**:
    - The **UserService** contains the business logic to process the request (e.g., create a new user).
    - It interacts with the **model** (e.g., `User.java`) and the **DAO** to save data to the database.
4. **Database**:
    - The **DAO layer** handles the direct interaction with the database (using JDBC or other methods).
5. **Response**:
    - After processing, the servlet sends back a response (e.g., success or failure message) to the frontend.
6. **Frontend Updates**:
    - The frontend processes the response and updates the UI accordingly (e.g., displaying a success or failure message to the user).

Conclusion

To clarify:

- **Servlets act as Controllers** in your case (handling HTTP requests).
- **Models** represent the data entities (like `User`, `Team`, `Process`).
- **Services** handle the business logic (interacting with models and performing actions like creating or updating records).
- **Frontend Controllers** (JavaScript files) handle client-side logic (AJAX requests, UI updates).

This setup follows the **MVC pattern** using servlets as controllers, and it will allow your team members to work on the specific components they are assigned while maintaining a clean architecture.

Example SQL Tables

Here's an example of the SQL schema that will be used for the database:

```sql
-- Users table (for company admins)
CREATE TABLE users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    email VARCHAR(100) NOT NULL,
    password VARCHAR(255) NOT NULL,
    company_id INT,
    FOREIGN KEY (company_id) REFERENCES companies(id)
);

-- Companies table
CREATE TABLE companies (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    email VARCHAR(100),
    address VARCHAR(255)
);

-- Teams table
CREATE TABLE teams (
    id INT AUTO_INCREMENT PRIMARY KEY,
    team_name VARCHAR(100) NOT NULL,
    manager_id INT,
    company_id INT,
    FOREIGN KEY (manager_id) REFERENCES users(id),
    FOREIGN KEY (company_id) REFERENCES companies(id)
);

-- Workers table (employees to be onboarded)
CREATE TABLE workers (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    status ENUM('Not Started', 'In Progress', 'Completed') DEFAULT 'Not Started',
    team_id INT,
    FOREIGN KEY (team_id) REFERENCES teams(id)
);

-- Onboarding Processes table
CREATE TABLE processes (
    id INT AUTO_INCREMENT PRIMARY KEY,
    process_name VARCHAR(100) NOT NULL,
    company_id INT,
    FOREIGN KEY (company_id) REFERENCES companies(id)
);

-- Stages table (to track process stages)
CREATE TABLE stages (
    id INT AUTO_INCREMENT PRIMARY KEY,
    stage_name VARCHAR(100) NOT NULL,
    process_id INT,
    FOREIGN KEY (process_id) REFERENCES processes(id)
);
```

Conclusion

This project structure enables parallel development by dividing responsibilities between frontend and backend and allowing each pair of developers to work on their own isolated tasks. The wireframe guidance helps ensure that the frontend will provide a consistent user experience, and the database schema supports the core functionality of the platform.

Timeline for 24 Hours

Hour 1-3: Initial Setup

- Set up the project repositories.
- Create the database schema and tables (SQL).
- Set up backend project structure (Java, JDBC).
- Set up the frontend project structure (HTML, CSS, JS).

Hour 4-8: Backend Development (Part 1)

- Implement the Users Server: Authentication routes (login/signup).
- Implement the Teams Server: CRUD routes for team management.
- Implement the Processes Server: CRUD routes for managing processes.

Hour 9-12: Backend Development (Part 2)

- Implement the Companies Server: Routes for managing company data.
- Connect JDBC for all servers.
- Begin implementing frontend pages for Company Dashboard, Team Dashboard, Process Dashboard.

Hour 13-16: Frontend Development (Part 1)

- Develop the frontend for the Company Dashboard, Team Dashboard, and Process Dashboard.
- Design the forms for adding/editing teams, processes, and workers.

Hour 17-20: Frontend Development (Part 2)

- Finalize all forms: Company user, team creation, onboarding process creation.
- Implement AJAX calls to the REST APIs from frontend pages.

Hour 21-24: Testing, Debugging & Final Touches

- Test all pages and ensure proper API connections.
- Debug any issues with data flow between frontend and backend.
- Add final CSS styling, ensure responsive design.

Additional Considerations

- **Security**: Ensure proper validation for login/signup and user roles.
- **Responsiveness**: Ensure frontend is responsive for desktop view.
- **Error Handling**: Implement proper error handling for API routes (e.g., invalid data input).

Conclusion

This project will provide companies with a robust platform for managing employee onboarding processes. By breaking the project into clearly defined tasks and using a team-based approach, we can efficiently deliver a working solution within the given 24-hour timeframe.

You're absolutely right! In the example I provided, I focused on using the servlet and JDBC to directly handle the request, but I didn't include the use of Java classes to represent the data or to abstract business logic. Let's fix that.

In a more structured, object-oriented approach, you would definitely use **Java classes** to represent entities like `Team`, `Process`, `Company`, etc. These classes would be part of the backend business logic layer, and they would help with managing the data and interaction with the database.

I'll modify the example to show where and how Java classes would fit into this flow. The main idea is that the **Java classes** should help:

- Represent the data entities (e.g., Team, Process).
- Encapsulate the business logic (e.g., assigning a process to a team).
- Serve as the interface between your servlet and database (e.g., handling data through `DAO` - Data Access Objects).

Full Front-To-Back Data Flow Example

Let's walk through the same route, but this time we will add the appropriate **Java classes** and **business logic** layers.

Step 1: Front-End (HTML and JavaScript)

This part will remain the same. The user selects a process and assigns it to a team.

```
<!-- /frontend/views/teamDashboard.html -->
<html>
<head>
    <title>Assign Onboarding Process</title>
</head>
<body>
    <h2>Assign Onboarding Process to Team</h2>

    <!-- Dropdown to select an onboarding process -->
    <select id="processSelect">
        <option value="1">Process 1</option>
        <option value="2">Process 2</option>
        <option value="3">Process 3</option>
    </select>

    <!-- Button to assign the selected process -->
    <button id="assignProcessBtn">Assign Process</button>

    <script src="/frontend/js/teamController.js"></script>
</body>
</html>
```

```
// /frontend/js/teamController.js
document.getElementById("assignProcessBtn").addEventListener("click", function() {
    const processId = document.getElementById("processSelect").value;
    const teamId = 1; // Example team ID

    // Sending POST request to the backend API to assign the process to the team
    fetch('/assignProcess', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify({
            teamId: teamId,
            processId: processId
        })
    })
    .then(response => response.json())
    .then(data => {
        // Show success message
        alert(data.message);
    })
    .catch(error => {
        // Show error message
        console.error('Error:', error);
    });
});
```

Step 2: Java Classes for Business Logic

Now, let's add some **Java classes** to represent the **Team** and **Process** entities, and a **TeamService** class to manage the logic.

1. `Team` Class:

This class represents a team in your system.

```
// /backend/models/Team.java
public class Team {
    private int id;
    private String name;
    private int processId; // The ID of the process assigned to this team

    // Constructor
    public Team(int id, String name, int processId) {
        this.id = id;
        this.name = name;
        this.processId = processId;
    }

    // Getters and Setters
    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getProcessId() {
        return processId;
    }

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

2. `Process` Class:

This class represents an onboarding process in your system.

```java
// /backend/models/Process.java
public class Process {
    private int id;
    private String name;

    // Constructor
    public Process(int id, String name) {
        this.id = id;
        this.name = name;
    }

    // Getters and Setters
    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

3. `TeamService` Class:

This class will contain the business logic for assigning a process to a team.

```
// /backend/services/TeamService.java
import java.sql.*;

public class TeamService {
    private static final String DB_URL = "jdbc:mysql://localhost:3306/company_db";
    private static final String DB_USER = "root";
    private static final String DB_PASS = "password";

    // Method to assign a process to a team
    public boolean assignProcessToTeam(int teamId, int processId) {
        // Create a team object (optional, depending on your business logic)
        Team team = new Team(teamId, "Team " + teamId, processId);

        // Execute the DB update
        try (Connection connection = DriverManager.getConnection(DB_URL, DB_USER, DB_PASS)) {
            String sql = "UPDATE teams SET process_id = ? WHERE id = ?";
            try (PreparedStatement stmt = connection.prepareStatement(sql)) {
                stmt.setInt(1, processId);
                stmt.setInt(2, teamId);
                int rowsAffected = stmt.executeUpdate();

                return rowsAffected > 0; // If rows affected > 0, the update was successful
            }
        } catch (SQLException e) {
            e.printStackTrace();
            return false; // If an error occurs, return false
        }
    }
}
```

- The **`TeamService`** class abstracts the logic of interacting with the database. It uses the `Team` class as an object to pass data and handle business logic.
- The **`assignProcessToTeam()`** method handles the process of updating the team's process in the database.

Step 3: Backend Servlet

The servlet will now use the `TeamService` class to handle the logic for assigning the process to the team.

```
// /backend/servlets/AssignProcessServlet.java
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import com.fasterxml.jackson.databind.ObjectMapper; // For parsing JSON

public class AssignProcessServlet extends HttpServlet {

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

    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        // Read the incoming JSON request body
        BufferedReader reader = new BufferedReader(new InputStreamReader(request.getReader()));
        StringBuilder sb = new StringBuilder();
        String line;
        while ((line = reader.readLine()) != null) {
            sb.append(line);
        }

        // Parse the JSON body using ObjectMapper (for deserialization)
        ObjectMapper objectMapper = new ObjectMapper();
        AssignProcessRequest assignRequest = objectMapper.readValue(sb.toString(), AssignProcessRequest.class);

        int teamId = assignRequest.getTeamId();
        int processId = assignRequest.getProcessId();

        // Call the service to assign the process
        boolean success = teamService.assignProcessToTeam(teamId, processId);

        // Prepare and send the response
        response.setContentType("application/json");
        PrintWriter out = response.getWriter();

        if (success) {
            out.println("{\"message\": \"Onboarding process assigned successfully\"}");
        } else {
            out.println("{\"message\": \"Failed to assign onboarding process\"}");
        }
    }
}
```

- **`AssignProcessRequest`**: This is a simple class to represent the incoming data. We use `ObjectMapper` to deserialize the JSON into this object.

```
// /backend/models/AssignProcessRequest.java
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;
    }
}
```

Step 4: Database (SQL)

The database is the same as before, with a `teams` table where the `process_id` is updated.

```
-- teams table in MySQL
CREATE TABLE teams (
    id INT PRIMARY KEY,
    name VARCHAR(255),
    process_id INT,
    FOREIGN KEY (process_id) REFERENCES processes(id)
);
```

Summary of File Structure and Flow

- **Frontend (JavaScript and HTML)**:
  - `/frontend/views/teamDashboard.html`: HTML page with dropdown and button for assigning a process.
  - `/frontend/js/teamController.js`: JavaScript that sends the POST request to the backend.
- **Backend (Java Classes and Servlets)**:
  - **Models**:
    - `/backend/models/Team.java`: Team class representing a team.
    - `/backend/models/Process.java`: Process class representing an onboarding process.
    - `/backend/models/AssignProcessRequest.java`: Class representing the incoming request data.
  - **Service**:
    - `/backend/services/TeamService.java`: Business logic for handling team-related operations (like assigning a process).
  - **Servlet**:
    - `/backend/servlets/AssignProcessServlet.java`: Servlet that handles the HTTP POST request and uses the `TeamService` to assign a process.
- **Database**:
  - MySQL `teams` table with a `process_id` foreign key.

Conclusion

In this updated example, I've integrated the Java classes (`Team`, `Process`, `TeamService`) into the flow, abstracting the logic from the servlet and using the service layer to handle the business logic of assigning a process to a team. These Java classes provide a clean structure for your application, helping organize your code into logical components while allowing you to interact with the database via JDBC.

Project Presentation: Developer Onboarding Management Platform

Timeframe: 7 minutes

Participants: 4 seniors and 3 juniors

Each member will cover a specific section of the project, with code snippets to demonstrate their part.

Opening (30 seconds)

- **Speaker**: Team Leader (Senior)
- **Overview**:
  - Introduce the project: A platform for companies to manage the onboarding process of new employees.
  - The project will provide a system for companies to register, create teams, assign onboarding processes to those teams, and track employee status.
  - Each team member will cover their part, explaining the backend, frontend, and overall architecture.

---

1. Backend Architecture Overview (1 minute)

- **Speaker**: Senior 1
- **Content**:
  - **MVC Architecture**: We follow the **Model-View-Controller** (MVC) design pattern.
  - **Backend Layers**:
    - **Controllers (Servlets)**: Handle HTTP requests and responses.
    - **Services**: Contain business logic.
    - **Models**: Represent the data entities (User, Team, Process).
  - **Database**: Use an SQL database for data persistence, connected via JDBC.

Code Snippet: UserServlet.java (Controller)

```
// /backend/servlets/UserServlet.java
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import com.fasterxml.jackson.databind.ObjectMapper;

public class UserServlet extends HttpServlet {
    private UserService userService = new UserService();

    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        // Read JSON data from the request body
        BufferedReader reader = new BufferedReader(new InputStreamReader(request.getReader()));
        StringBuilder sb = new StringBuilder();
        String line;
        while ((line = reader.readLine()) != null) {
            sb.append(line);
        }

        ObjectMapper objectMapper = new ObjectMapper();
        User user = objectMapper.readValue(sb.toString(), User.class);

        boolean success = userService.createUser(user);

        response.setContentType("application/json");
        PrintWriter out = response.getWriter();
        if (success) {
            out.println("{\"message\": \"User created successfully\"}");
        } else {
            out.println("{\"message\": \"Failed to create user\"}");
        }
    }
}
```

- **Explanation**: This servlet handles POST requests to create a new user. It reads the request body, parses it to a `User` object, calls the `UserService`, and sends a response back to the frontend.

---

2. Service Layer and Business Logic (1 minute)

- **Speaker**: Senior 2
- **Content**:
    - **Services**: Business logic is handled in the service layer, which interacts with the DAO (Data Access Object) to persist data.
    - **Example**: The `UserService` class that handles user operations like creating a user.

Code Snippet: UserService.java (Service Layer)

```
// /backend/services/UserService.java
public class UserService {
    public boolean createUser(User user) {
        // Logic to create a new user
        return UserDAO.create(user);  // This calls the DAO to interact with the database
    }
}
```

- **Explanation**: The `UserService` class contains business logic for user-related operations. Here, it uses the `UserDAO.create()` method to insert the user into the database.

---

3. Models and Database Interaction (1 minute)

- **Speaker**: Junior 1

- **Content**:
  - **Models**: Represent the entities (like `User`, `Team`, `Process`).
  - **Database Interaction**: We use JDBC to interact with the SQL database. Each model corresponds to a database table.

Code Snippet: User.java (Model)

```
// /backend/models/User.java
public class User {
    private String username;
    private String email;
    private String password;

    // Getters and setters
    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    // Other getters and setters...
}
```

- **Explanation**: The `User` class is a model representing the structure of the user data in the database. It has fields for `username`, `email`, and `password` (with corresponding getters and setters).

Database Schema Example (SQL Table)

```
CREATE TABLE users (
    id INT PRIMARY KEY AUTO_INCREMENT,
    username VARCHAR(100) NOT NULL,
    email VARCHAR(100) NOT NULL,
    password VARCHAR(100) NOT NULL
);
```

---

4. Frontend Dashboard and AJAX Requests (1.5 minutes)

- **Speaker**: Junior 2
- **Content**:
  - **Frontend**: The user interface consists of HTML, CSS, and JavaScript.
  - **AJAX Requests**: We use AJAX for making API calls from the frontend to the backend.
  - **Dynamic Dashboards**: Dashboards for the company, teams, and processes are rendered with the data fetched from the backend.

Code Snippet: companyController.js (Frontend)

```
// /frontend/controllers/companyController.js
document.getElementById("addUserBtn").addEventListener("click", function() {
    const userData = {
        username: document.getElementById("username").value,
        email: document.getElementById("email").value
    };

    fetch('/createUser', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify(userData)
    })
    .then(response => response.json())
    .then(data => {
        alert(data.message);  // Display success or error message
    })
    .catch(error => console.error('Error:', error));
});
```

- **Explanation**: The `companyController.js` file listens for button clicks (like `addUserBtn`), gathers user input, and sends an AJAX request to the `UserServlet` to create a new user. The response is then displayed in an alert box.

---

5. SQL Database Setup and DAO Layer (1 minute)

- **Speaker**: Junior 3
- **Content**:
    - **Database Setup**: We use an SQL database to store user, team, and process data.
    - **DAO Layer**: The DAO (Data Access Object) layer is responsible for database operations like `create`, `read`, `update`, and `delete`.

Code Snippet: UserDAO.java (DAO Layer)

```
// /backend/dao/UserDAO.java
import java.sql.*;

public class UserDAO {
    private static Connection getConnection() throws SQLException {
        // Database connection details
        return DriverManager.getConnection("jdbc:mysql://localhost:3306/onboarding", "username", "password");
    }

    public static boolean create(User user) {
        String sql = "INSERT INTO users (username, email, password) VALUES (?, ?, ?)";
        try (Connection conn = getConnection();
             PreparedStatement stmt = conn.prepareStatement(sql)) {
            stmt.setString(1, user.getUsername());
            stmt.setString(2, user.getEmail());
            stmt.setString(3, user.getPassword());
            int result = stmt.executeUpdate();
            return result > 0;
        } catch (SQLException e) {
            e.printStackTrace();
            return false;
        }
    }
}
```

- **Explanation**: The `UserDAO` class handles the interaction with the database. The `create()` method inserts a new user into the `users` table.

---

6. Conclusion and Team Collaboration (1 minute)

- **Speaker**: Team Leader (Senior)
- **Content**:
    - **Collaboration**: Each team member worked on a specific part of the project (backend logic, database, frontend, etc.).
    - **Next Steps**: The project is nearly ready for further testing and deployment. We'll continue refining features and ensuring seamless integration between all layers.
    - Thank the team and summarize the work done.

End of Presentation (7 minutes)

- **Team Leader** concludes the presentation, thanks the audience, and opens the floor for any questions.

Each team member provides clear and concise explanations, along with code snippets demonstrating their part in the project, ensuring a smooth and collaborative presentation.