**Project Structure and MVC Flow in Spring Boot Web Application**

**Project Structure Overview**

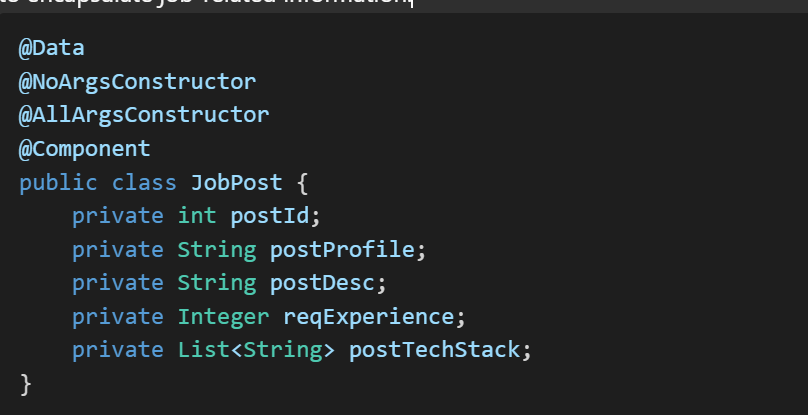
The application follows an MVC (Model-View-Controller) architecture, which divides the project into distinct layers to separate responsibilities. Here is a breakdown of the project structure:

* **Controller Layer**: Handles HTTP requests from the user.
  + **JobController**
* **Service Layer**: Implements business logic.
  + **JobService**
* **Repository Layer**: Manages data interactions.
  + **JobRepo**
* **Model Layer**: Defines data models.
  + **JobPost**
* **View Layer**: Represents JSP files for UI.
  + **addjob.jsp**, **home.jsp**, **success.jsp**, **viewalljob.jsp**
* **Configuration**: Configuration properties are defined in **application.properties**.

**Explanation of Each Class**

1. **JobPost Model Class**

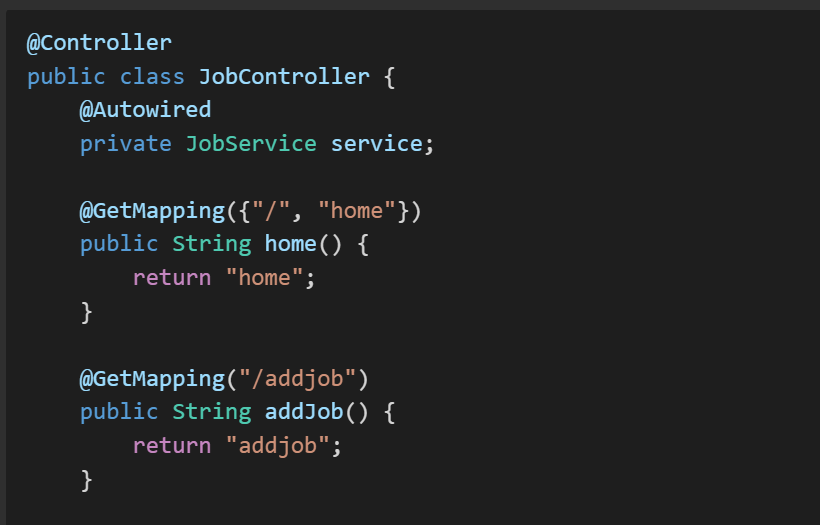
The JobPost class represents the data model for job postings in the application. It contains fields like postId, postProfile, postDesc, reqExperience, and postTechStack. This class is used across different layers to encapsulate job-related information.



1. **JobController (Controller Layer)**

The JobController class handles user requests and controls how data flows between the **view** and **model**.

* The controller contains methods such as handleAddJobForm() to handle form submissions and viewJobs() to display all jobs.
* It uses the **service layer** to interact with the business logic.





1. J**obService (Service Layer)**

The JobService class serves as an intermediary between the **controller** and the **repository**. It contains methods for interacting with job data, implementing business logic such as adding or retrieving job postings.



1. **JobRepo (Repository Layer)**

The JobRepo class is responsible for interacting with the data source, which, in this case, is a list that acts as an in-memory data store.

* The JobRepo constructor initializes the repository with sample job posts.
* It contains methods like getAllJobs() to fetch all jobs and addJob() to add a new job.

@Repository

public class JobRepo {

private List<JobPost> jobs = new ArrayList<>();

public JobRepo() {

jobs.add(new JobPost(1, "Java Developer", "Must have good experience in core Java", 3, List.of("Java", "Spring Boot")));

jobs.add(new JobPost(2, "Frontend Developer", "Experience in building responsive UIs", 3, List.of("HTML", "CSS", "JavaScript", "React")));

jobs.add(new JobPost(3, "Data Scientist", "Strong background in Machine Learning and Data Analysis", 4, List.of("Python", "Machine Learning", "Data Analysis")));

jobs.add(new JobPost(4, "Network Engineer", "Design and implement computer networks", 2, List.of("Networking", "Linux", "Cisco")));

jobs.add(new JobPost(5, "Mobile App Developer", "Experience in building mobile applications", 3, List.of("Android", "Kotlin", "iOS")));

}

public List<JobPost> getAllJobs() {

return jobs;

}

public void addJob(JobPost job) {

jobs.add(job);

System.out.println(jobs);

}

}

**Flow Explanation for Adding a Job Post**

**Method in Controller**: handleAddJobForm(JobPost jobPost)

1. **Client Request**: The user submits a form to add a job, triggering a POST request to /handleForm.
2. **Controller Interaction**: The handleAddJobForm() method in JobController receives the job data encapsulated in the JobPost object.
3. **Service Layer Call**: The controller calls service.addJobPost(jobPost) to pass the job post to the **service layer**.
4. **Service to Repository**: The JobService layer then calls the repo.addJob(jobPost) method in the **JobRepo** to add the job post to the in-memory list.
5. **View Rendering**: After successfully adding the job post, the controller returns "success", indicating the corresponding JSP page (success.jsp) should be rendered.

**Flow Explanation for Viewing All Jobs**

**Method in Controller**: viewJobs(Model m)

1. **Client Request**: The user sends a GET request to /viewalljobs by clicking a link or visiting the URL.
2. **Controller Interaction**: The viewJobs() method in the **JobController** is invoked to handle the request.
3. **Service Layer Call**: The controller calls service.returnAllJobPosts() to get all job postings.
4. **Repository Data Retrieval**: The JobService interacts with **JobRepo** by calling getAllJobs() to fetch all job posts from the list.
5. **Add Data to Model**: The **JobController** adds the list of job posts to the Model object using m.addAttribute("jobPosts", jobs), making it available to the view.
6. **View Rendering**: The controller returns "viewalljob" to indicate that the viewalljob.jsp page should be rendered, where job details are displayed to the user.

**Summary of Layer Interactions**

1. **User Request**: The user interacts with the web application via HTTP requests (form submission or link click).
2. **Controller (JobController)**: Handles HTTP requests, interacts with the service layer for data, and passes data to the view.
3. **Service (JobService)**: Acts as the business logic layer and interacts with the repository to retrieve or store data.
4. **Repository (JobRepo)**: Handles all data operations, such as adding or retrieving job posts.
5. **View (JSP Pages)**: Displays data to the user, facilitated by data provided via the Model object from the controller.

This MVC structure ensures clear separation of concerns, making the code more maintainable and scalable.