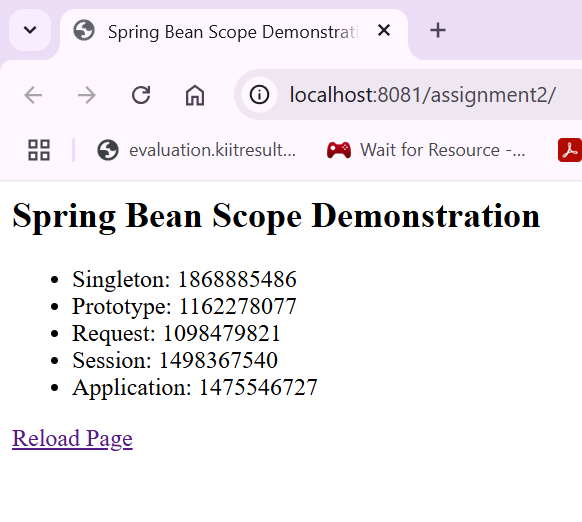
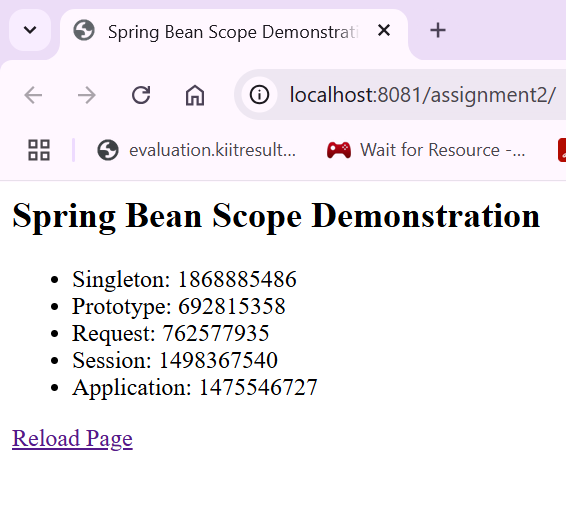
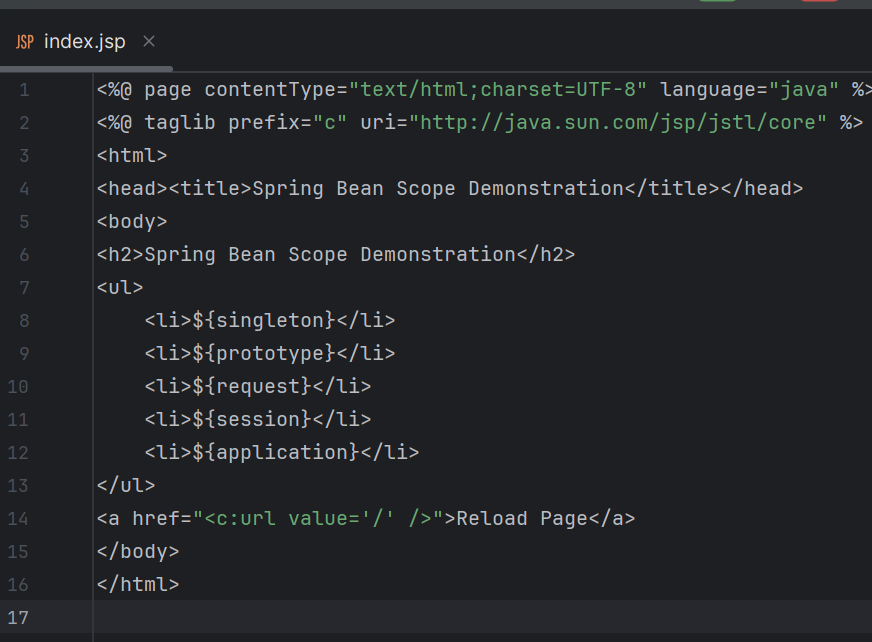
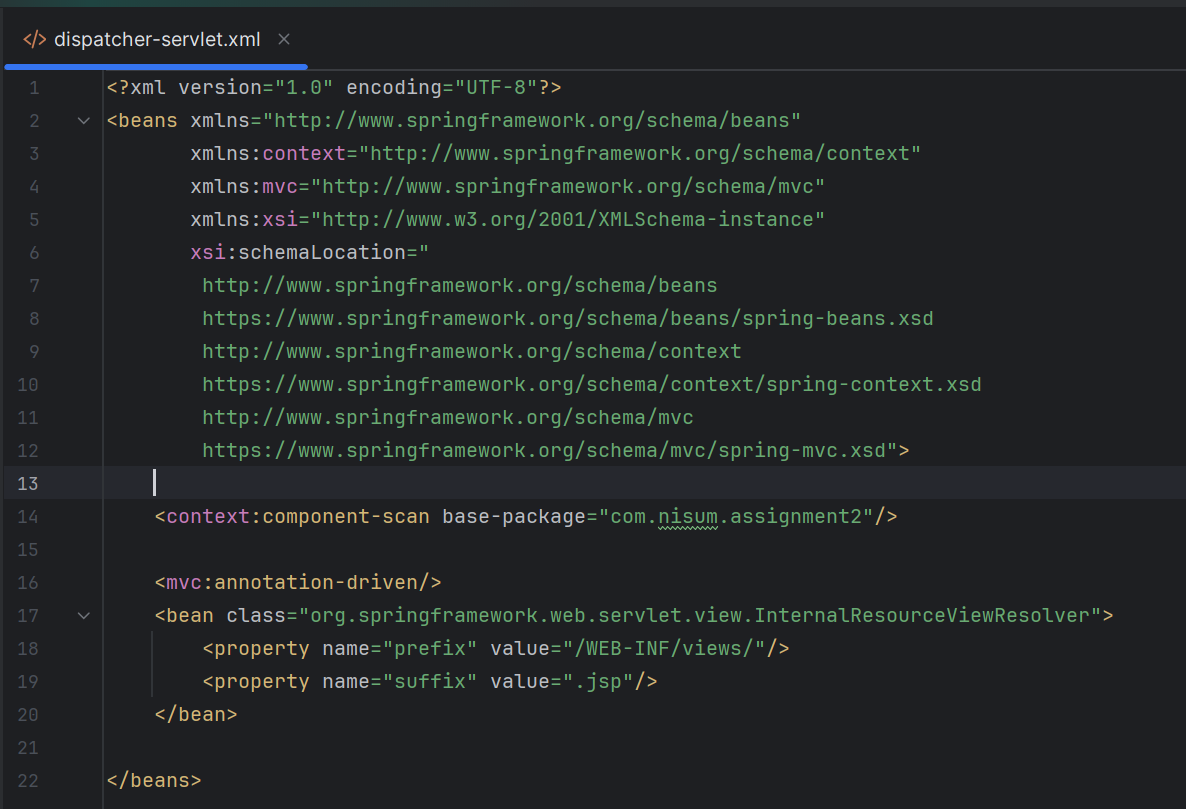
*Bean Lifecycle (Spring Beans and Bean Scopes) : Assignment:*

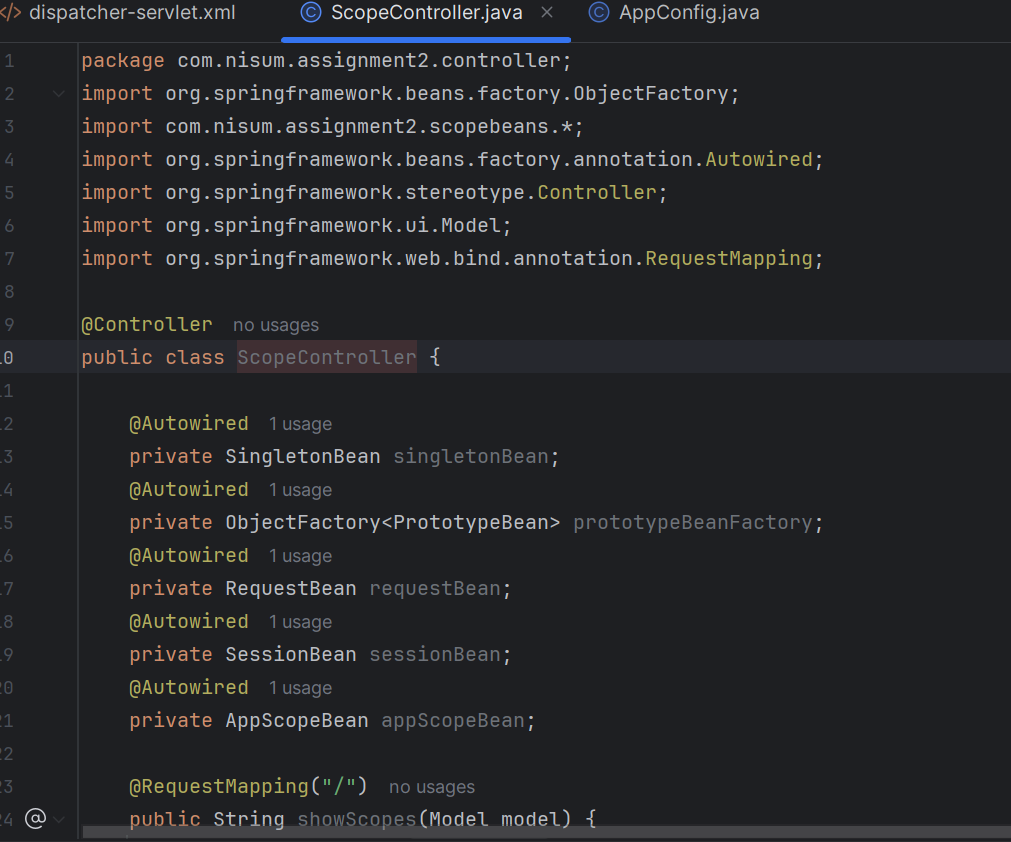
*1 I learned that s*pring provides different bean scopes to define the lifecycle and visibility of a bean in the application context. The commonly used scopes are singleton, prototype, request, session, and application. Singleton scope creates a single shared instance of the bean per Spring container. Prototype scope creates a new instance every time the bean is requested. Request scope creates a new instance for every HTTP request, session scope creates a new instance per HTTP session, and application scope creates a single instance shared across the entire servlet context. Understanding these scopes helps in managing object creation and memory usage efficiently depending on the context of use in web applications.

2 I created a Spring MVC web application using Maven and deployed it via Smart Tomcat in IntelliJ Community Edition. The application demonstrates all five Spring bean scopes: singleton, prototype, request, session, and application. I defined each bean using @Component along with the appropriate @Scope annotations. For the request, session, and application scopes, I used proxyMode = ScopedProxyMode.TARGET\_CLASS to ensure correct injection into the singleton controller. To correctly demonstrate prototype behavior, I used ObjectFactory to fetch a fresh instance of the prototype bean on each request. I designed a JSP page to display the hashCode of each bean, which visually confirms the behavior of each scope. Reloading the page shows which beans are persistent and which are recreated, effectively validating the lifecycle and visibility of Spring beans in a web environment.

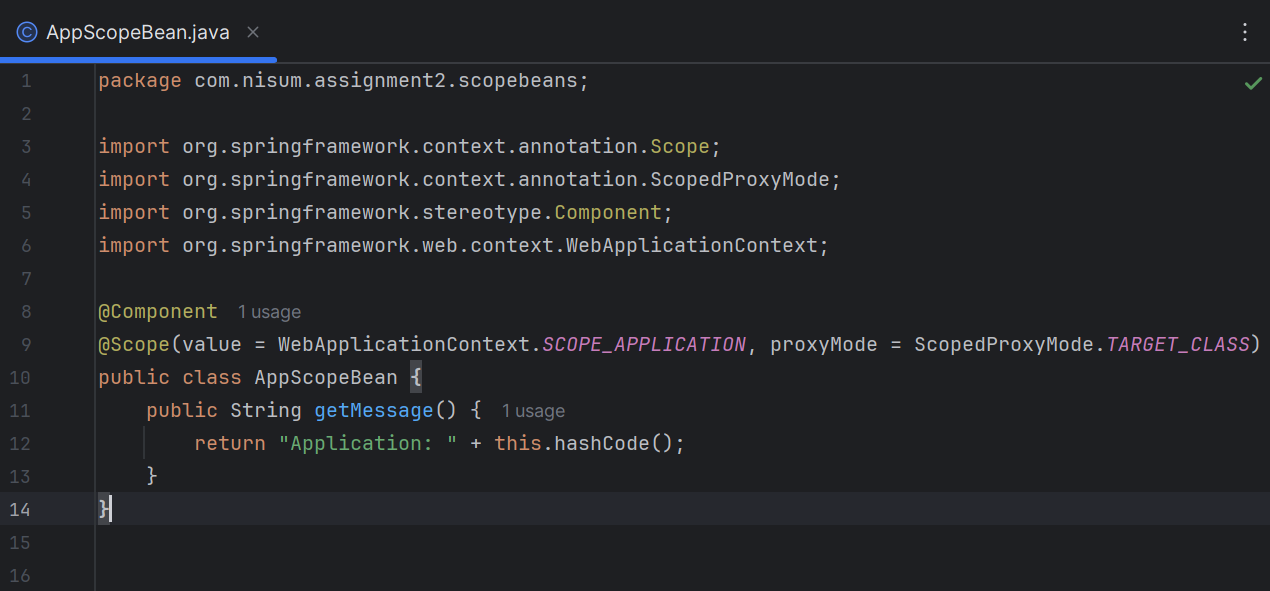


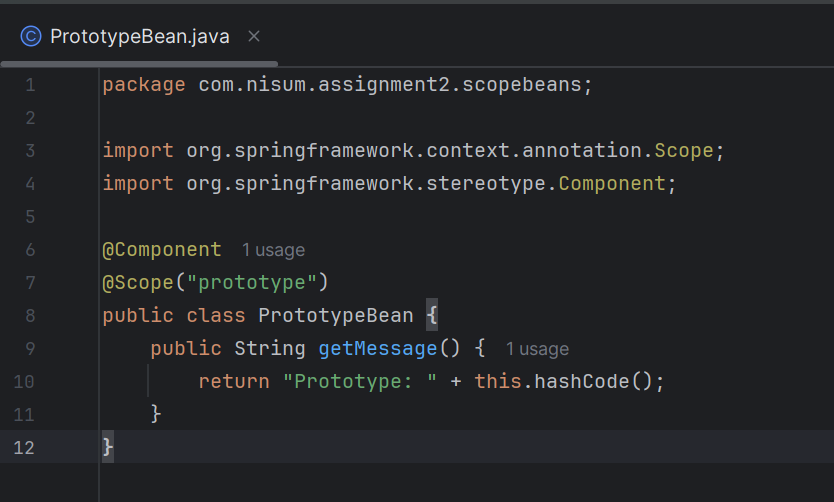


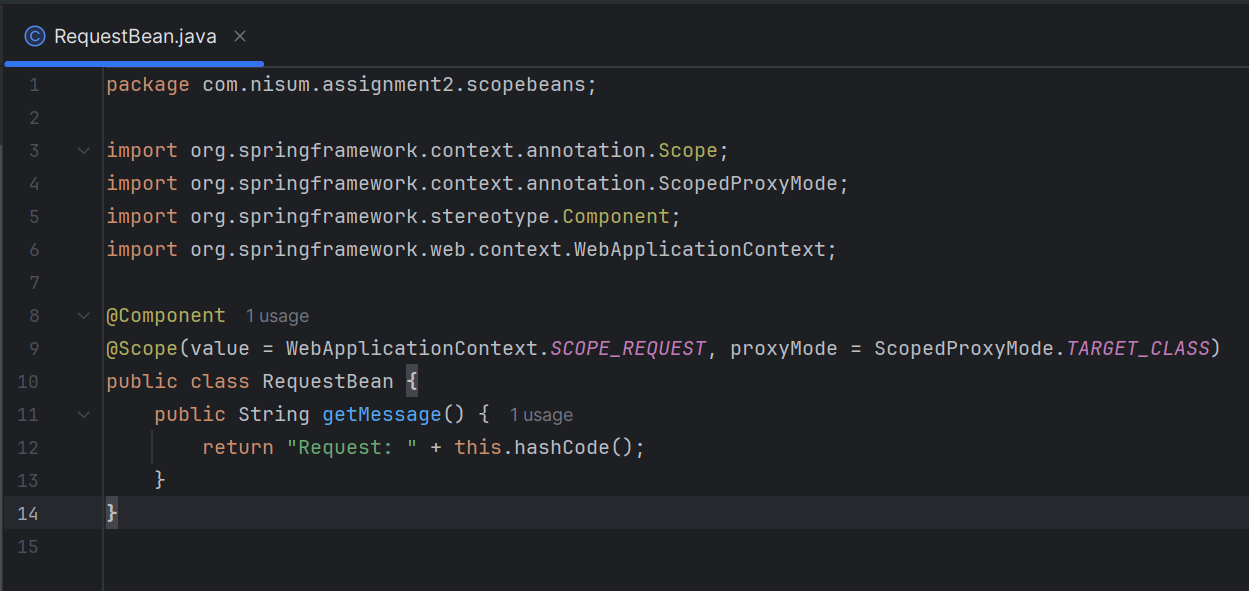


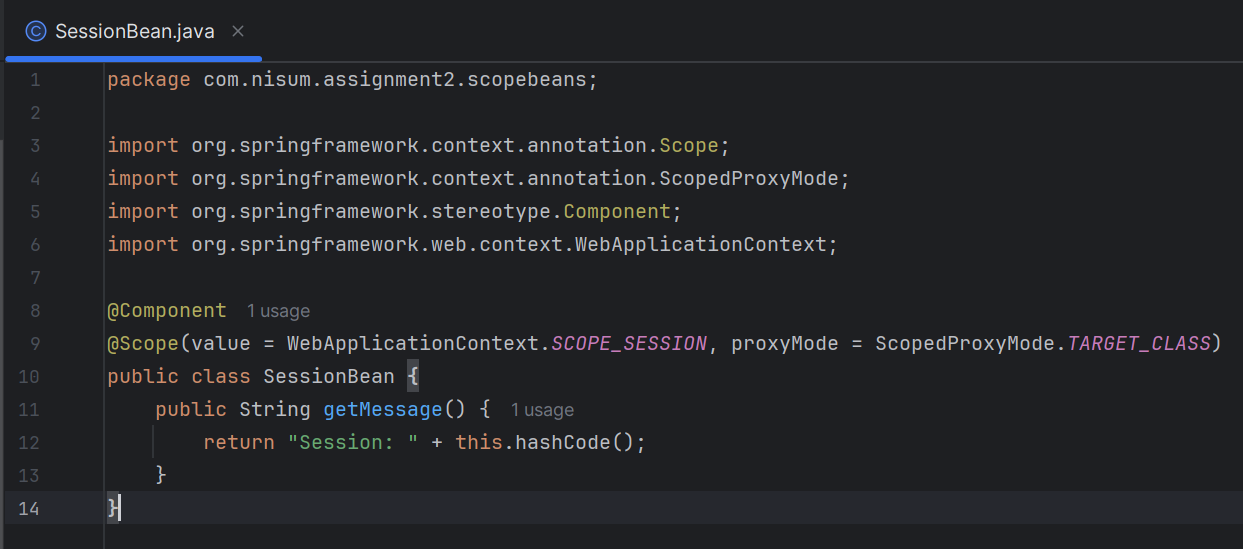


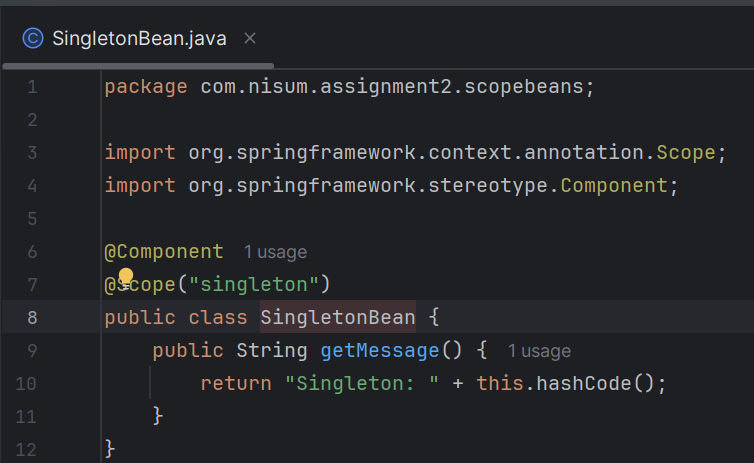












3

A .)In Spring, I learned that there are five main bean scopes. The default is **singleton**, which creates one shared instance per container — I use this for stateless, reusable components like services. **Prototype** creates a new bean every time it's requested, which I find useful for stateful or short-lived objects. **Request** scope gives a new bean for every HTTP request, so it's helpful for things like handling form data. **Session** creates one bean per user session, useful when I need to keep user-specific data across multiple requests. **Application** scope creates a single instance for the entire web application, which I would use for application-wide shared objects.

B .)Lazy initialization in Spring means that a bean is created only when it is first needed, rather than during application startup. This can be enabled using the @Lazy annotation or lazy-init="true" in XML. It’s useful for reducing startup time and avoiding unnecessary memory usage, especially for beans that may not always be used.

In addition to these configurations, I learned that **method injection** is another way to achieve lazy behavior. By calling a method to request the bean (instead of autowiring it directly), Spring can create a fresh instance at runtime. For example, using ObjectFactory.getObject() or ApplicationContext.getBean() inside a method allows me to delay bean creation until the method is actually executed. This is particularly helpful when injecting prototype or request-scoped beans into a singleton.

C .)I believe Spring Beans and their scopes help make applications more robust by giving precise control over how objects are managed. For example, singleton beans reduce memory use and improve performance for shared logic, while request and session scopes help manage user-specific data without conflicts. By choosing the right scope, I can avoid bugs related to shared state and ensure that the application behaves correctly across different users and requests. It also makes the code more modular and easier to test.

*Spring MVC*

1While learning Spring MVC, I focused on understanding the **MVC architecture**, which separates the application into three layers: Model, View, and Controller. I explored how **Spring MVC controllers handle requests and responses**, and how annotations like @Controller, @RequestMapping, and @ResponseBody are used to simplify request handling. I also learned how to integrate with other **MVC modules** such as view resolvers and validation. Additionally, I went through examples of implementing **CRUD operations** in a Spring MVC setup, which helped me understand how to connect a controller with a database-backed model using service and repository layers.

2 Practical Exercise Summary

I developed a Spring MVC web application using Maven and deployed it through Smart Tomcat in IntelliJ IDEA Community Edition. The application demonstrates both JSP-based traditional Spring MVC and RESTful APIs, integrated with a MySQL database. I implemented full CRUD (Create, Read, Update, Delete) operations, allowing data to be managed via both UI and API.

Technologies I Used

Spring MVC (DispatcherServlet, Controllers, ViewResolvers)

JSP for frontend view rendering

Spring JDBC Template for database connectivity

MySQL as the backend database

RESTful APIs using @RestController

IntelliJ Community Edition with Smart Tomcat plugin

Modules I Implemented

Spring MVC (JSP-based CRUD)

I created controllers to handle web requests, views using JSPs to render the UI, and a DAO layer for MySQL interaction. I added a service layer to abstract business logic from controllers. The student data is displayed in a list view, with options to add, edit, and delete records.

JSP URL Flow:

GET /students/ → displays list of all students

GET /students/add → opens a form to add a new student

POST /students/save → saves the student to database

GET /students/edit/{id} → loads edit form with student data

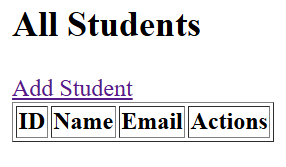
POST /students/update → updates the student record

GET /students/delete/{id} → deletes the student

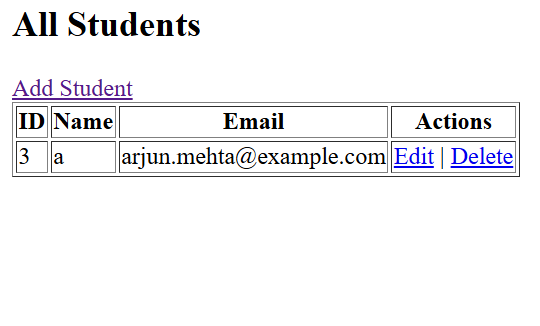
**Database Configuration (MySQL)**

I created a MySQL database named springmvcdb and used it to store student records for both JSP-based and RESTful CRUD operations. The student table contains three fields: an auto-incremented primary key id, and the name and email fields.

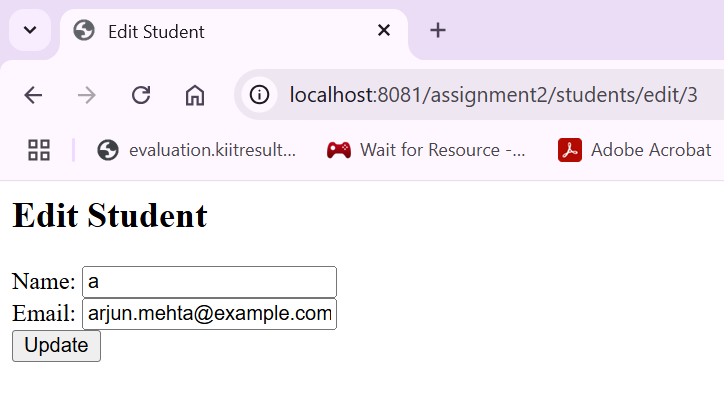
<http://localhost:8081/assignment2/students/>

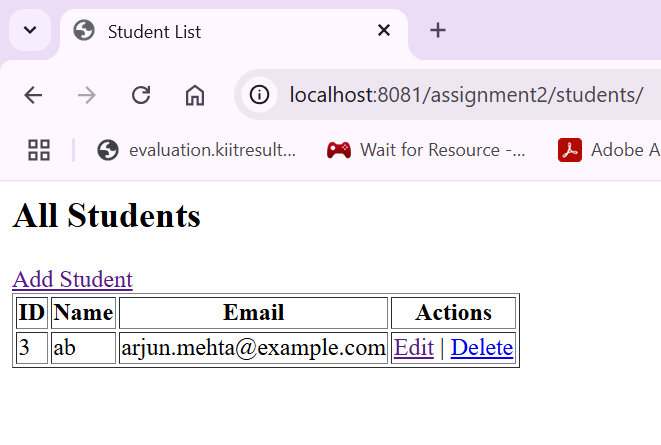


Add:

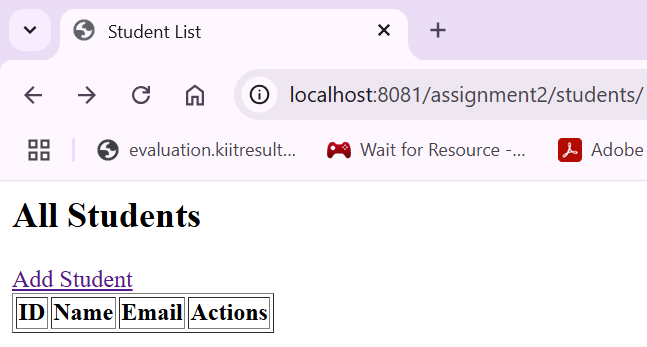


Update:





Delete:



3

A )Spring MVC solves the problem of separating concerns in web applications by dividing the application into three interconnected components: Model, View, and Controller. It helps isolate the UI logic from business logic, making the code more maintainable, testable, and scalable.

B )When a request is received, the DispatcherServlet acts as the front controller. It forwards the request to a suitable HandlerMapping, which determines the controller. The controller processes the request and returns a ModelAndView object. The ViewResolver resolves the logical view name to a physical view, and finally, the view (like a JSP page) is rendered back to the client with the model data.

C )Spring Interceptors are used to pre-process and post-process requests in a Spring MVC application. They work similarly to servlet filters and allow execution of code before the controller method, after the method execution, and after the view is rendered. They're typically used for logging, authentication, and request modification.

D )Spring MVC provides exception handling using @ExceptionHandler methods inside controllers, @ControllerAdvice for global handling, and XML-based SimpleMappingExceptionResolver. These mechanisms help in gracefully handling exceptions and returning appropriate error responses or views.

E )Common patterns include Front Controller (via DispatcherServlet), MVC (Model-View-Controller), Singleton (for beans), Dependency Injection, Factory (for bean creation), Template (in JdbcTemplate, RestTemplate), and Proxy (used in AOP features).

MCQ:

d ) Singleton

d )@Value

b ) Dispatcher Servlet

b )Void

b )BindingResult

a )@CrossOrigin

d) @RequestMapping(produces="application/json")

a) It sends the return value of a method back to the web response body.

d) @PostMapping