1. **QUESTION**

[AMBITIOUS]

Multi-tenant architecture with single instance of application layer which will have separate database for each client. Expected the architecture and approach document as output.

**Source Code GIT Repository:**

**Web Browser Link:**

[**https://github.com/Ajay-Kumar-Aspiring-Minds-Round-2/MultiTenantApp**](https://github.com/Ajay-Kumar-Aspiring-Minds-Round-2/MultiTenantApp)

**GIT Clone Link:**

[**https://github.com/Ajay-Kumar-Aspiring-Minds-Round-2/MultiTenantApp.git**](https://github.com/Ajay-Kumar-Aspiring-Minds-Round-2/MultiTenantApp.git)

**Technology Stack:**

****

**What is Multi-tenancy?**

* Multi-tenancy is an architecture in which a single instance of a software application serves multiple customers. Each customer is called a tenant.
* We can implement multi-tenancy using any of the following approaches:

1. **Database per Tenant:** Each Tenant has its own database and is isolated from other tenants.
2. **Shared Database, Shared Schema:** All Tenants share a database and tables. Every table has a Column with the Tenant Identifier that shows the owner of the row.
3. **Shared Database, Separate Schema:** All Tenants share a database, but have their own database schemas and tables.

**Key Application Features & Configurations:**

* **Add Employee UI** for the Tenants (tenant\_1, tenant\_2 & tenant\_3)
* **Employee List UI** for Tenants (tenant\_1, tenant\_2 & tenant\_3)
* Tenant ID found from URL and operations are performed based on current tenant id.

(Ex: http://app.com/api/tenant\_id/, http://localhost:8080/tenant\_id)

* **Database per Tenant** approach is followed to implement the sample Multi-tenant App.
* We are going to use the **separate database approach**, with this approach Hibernate requires you to specify a **MultiTenantConnectionProvider**. This is an interface you need to implement and will allow Hibernate to obtain connections in a tenant specific manner.
* Hibernate also requires you to implement the interface **CurrentTenantIdentifierResolver**, as this is the contract for Hibernate to able to resolve what the application considers the current tenant identifier.
* In order to identify the current tenant id, we will intercept the incoming request by using a handler interceptor. Spring provides an abstract class (**HandlerInterceptorAdapter**) which contains a simplified implementation of the HandlerInterceptor interface for pre-only/post-only interceptors.
* **MultiTenancyInterceptor** class which extends **HandlerInterceptorAdapter** class is used to extract current tenant id from URL or incoming request which is then used by **CurrentTenantIdentifierResolverImpl**.
* **MultiTenancyJpaConfiguration** switches database connection, defined in **DataSourceConfig**. In **LocalContainerEntityManagerFactoryBean** we set the **MultiTenancyStrategy** as **DATABASE** and override the connection provider and Tenant Identifier Resolver.
* **MultitenancyProperties** class is used to map the datasource properties defined in the application.properties file. We have defined 3 datasources in our properties file.
* For providing the implementation of **MultiTenantConnectionProvider,** we extend the **AbstractDataSourceBasedMultiTenantConnectionProviderImpl** provided by Hibernate, we let Spring inject our datasources, load them in a map and resolve the datasource based on the tenantIdentifier.

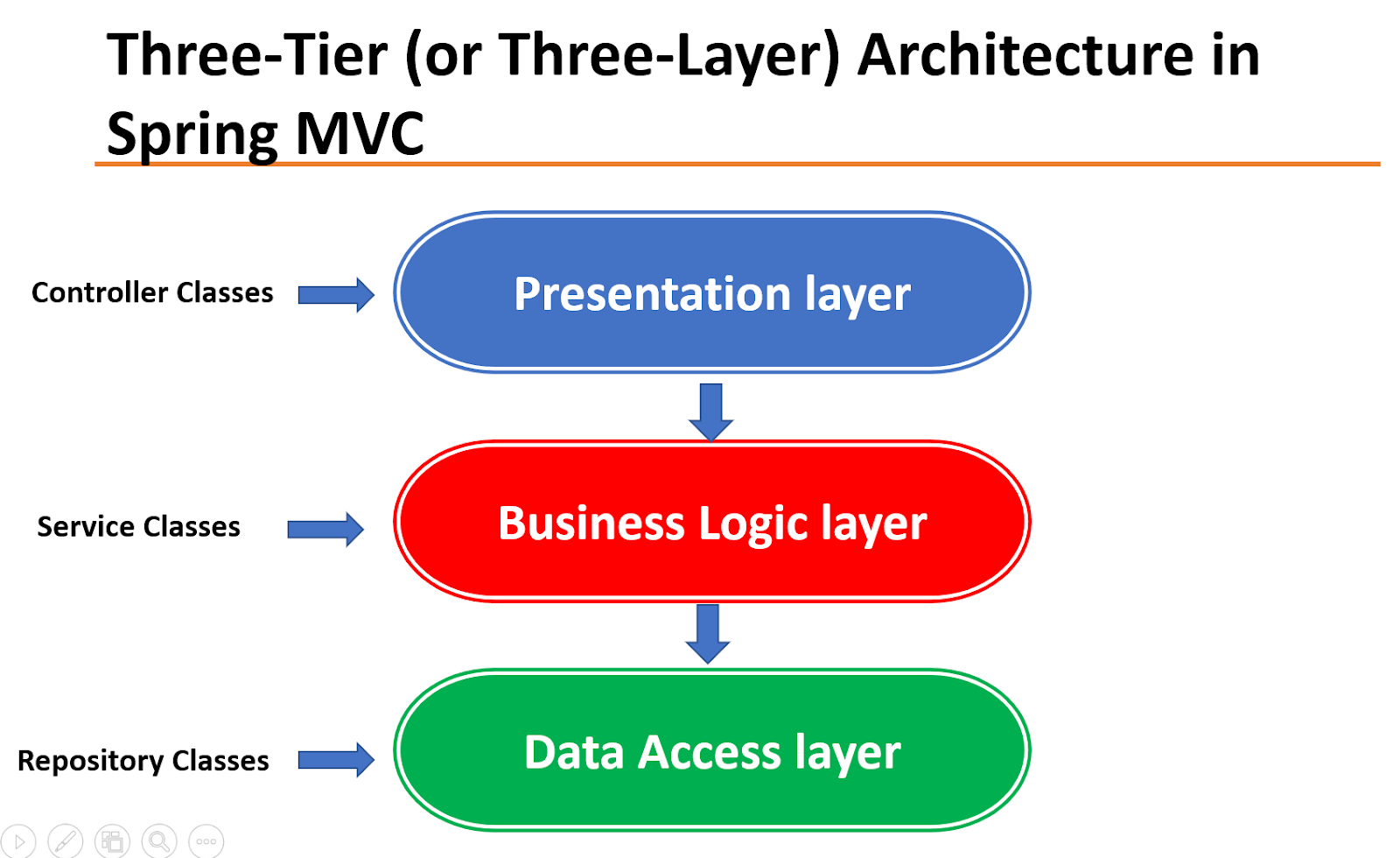
**MVC Framework:**

* Application is built using the **MVC** web application framework.
* The **Model-View-Controller** (**MVC**) is an architectural pattern that separates an application into three main logical components: **the model, the view, and the controller**.
  + The **Model** is responsible for managing the data of the application. It responds to the request from the view and it also responds to instructions from the controller to update itself.
  + The **View** is responsible for rendering the model data and in general it generates HTML output that the client's browser can interpret.
  + The **Controller** is responsible for processing user requests and building an appropriate model and passes it to the view for rendering.
* **Employee** is our Domain/POJO class.
* Creating a **Repository** for our **Employee** class is pretty simple using Spring. For our basic CRUD operations we only need to create an interface that extends Spring's own CrudRepository. Again nothing specific needs to be done here to support multi-tenancy.
* JPA with spring data repository model.

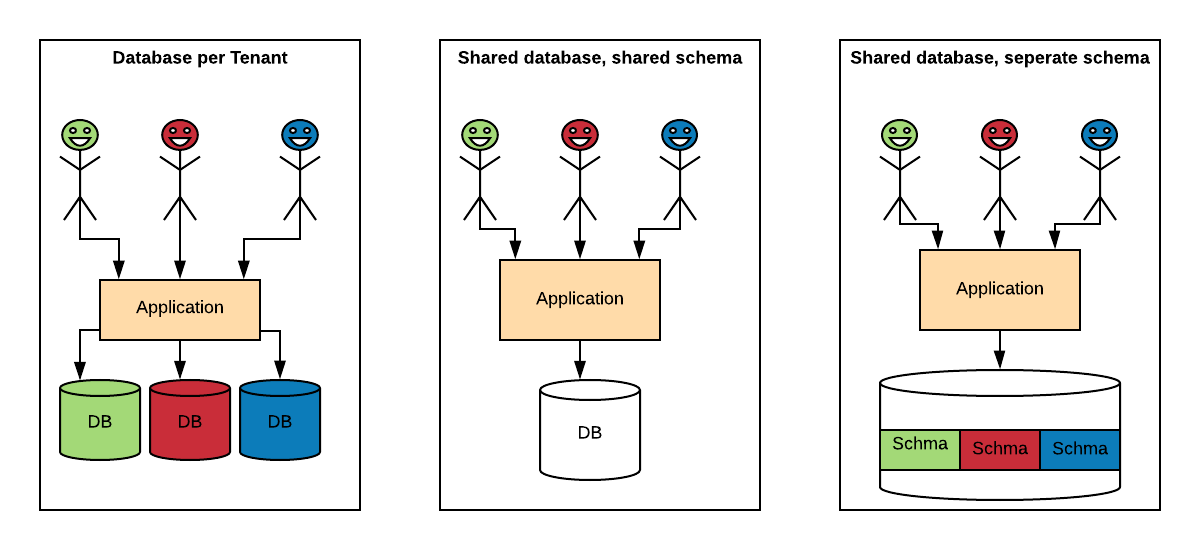
**MVC Architecture Diagram**



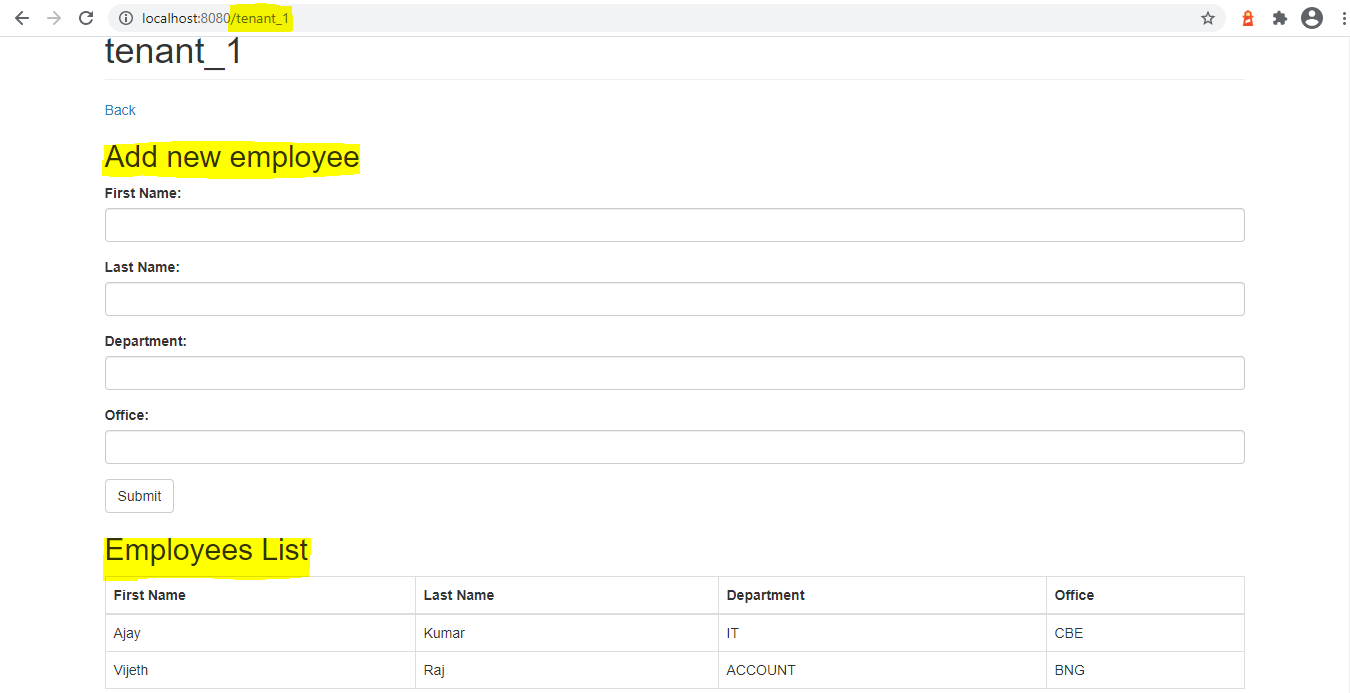
**Application Architecture Diagram**

****

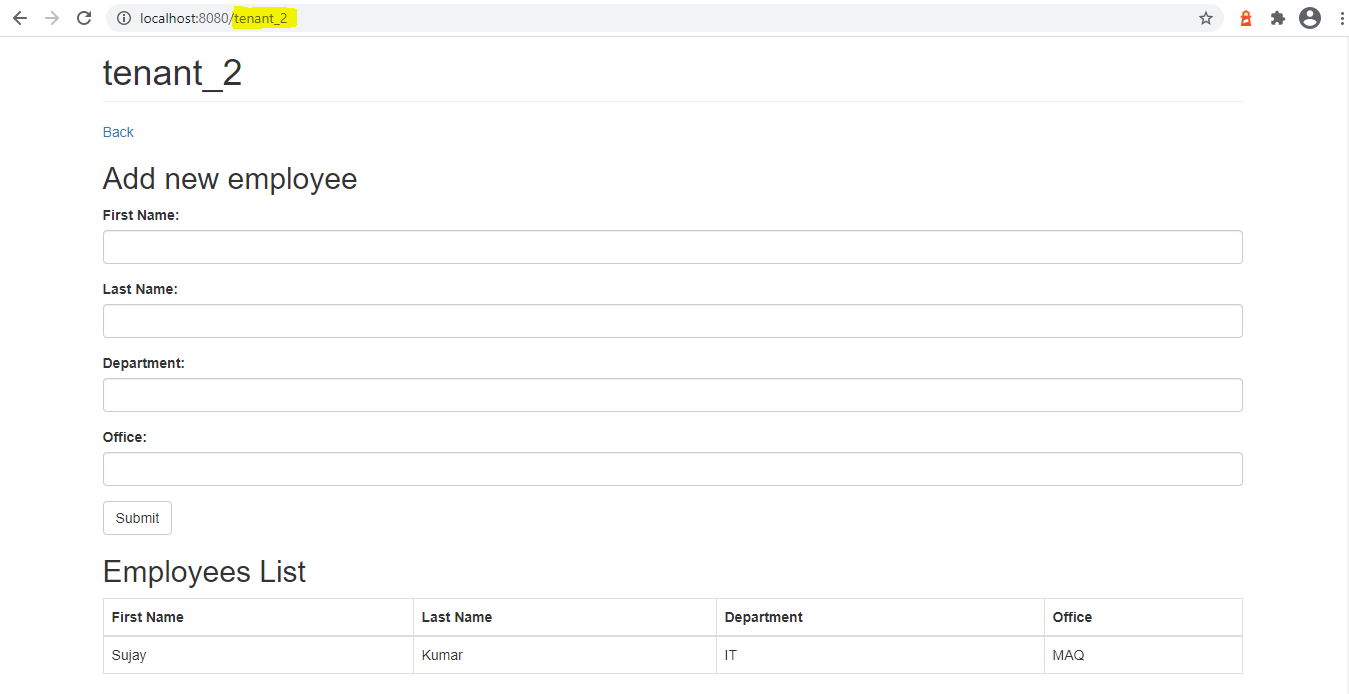
**Database per Tenant Architecture View**



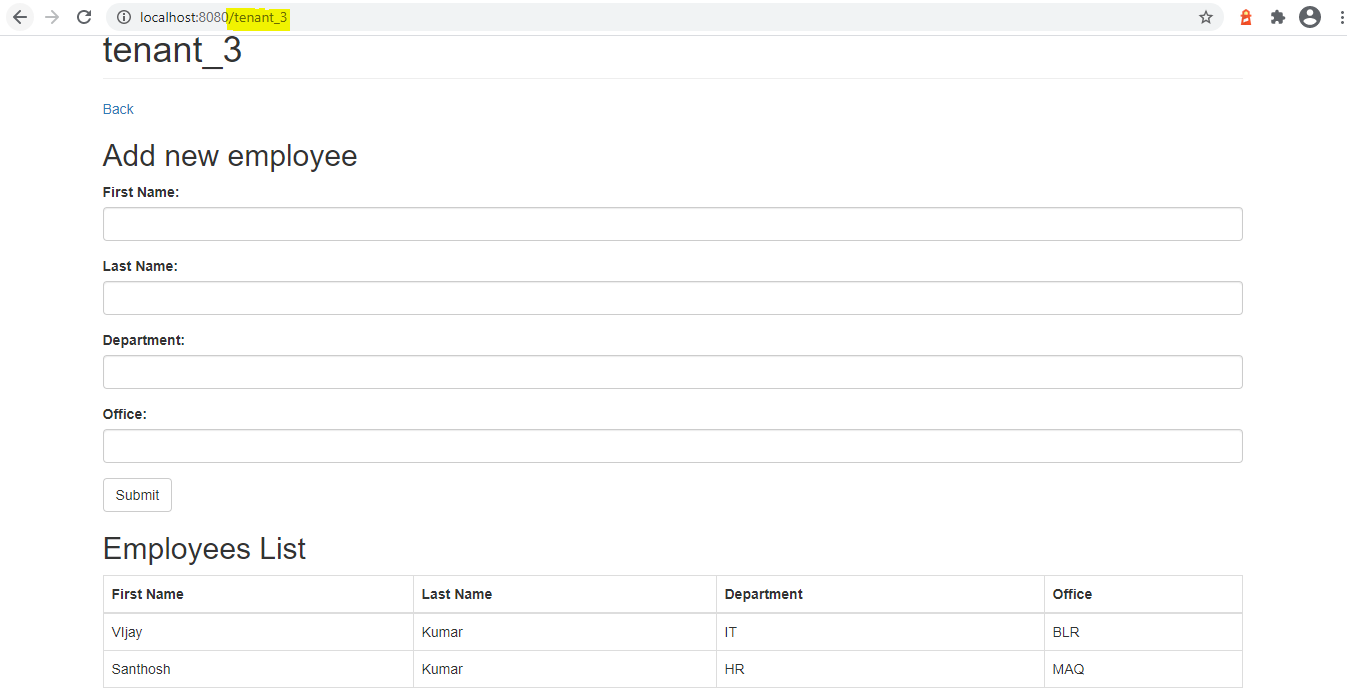
**tenant\_1 – Add New Employee & Employees List View**



**tenant\_2 – Add New Employee & Employees List View**



**tenant\_3 – Add New Employee & Employees List View**



**Multi-tenant Database View:**

