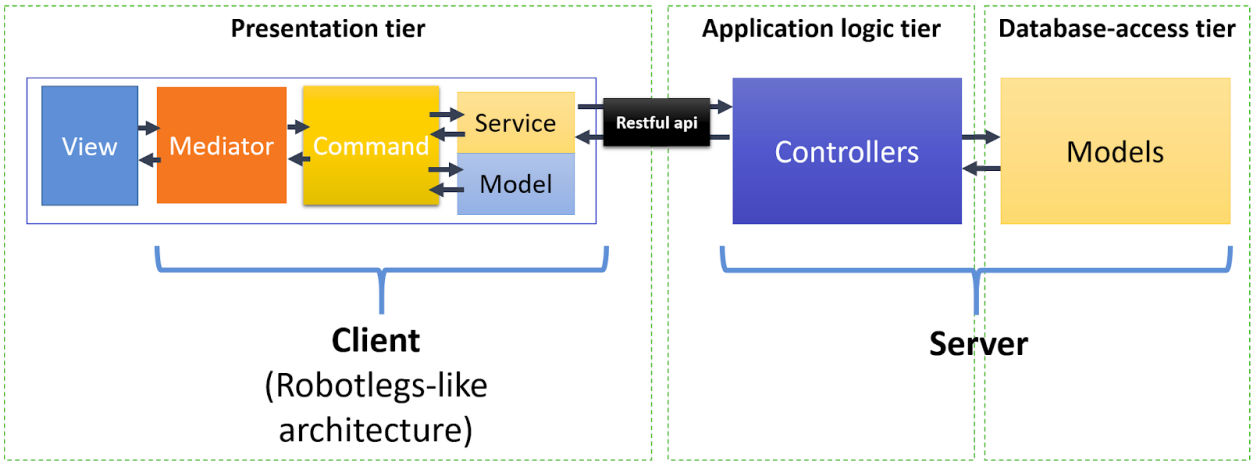
OVERVIEW DESIGN ARCHITECTURE OF ESTORE MANAGER - GROUP 01

1. **Choose software architectural patterns for front end (if any) and back end of your project**
    a. **Front-end:**
        i. Framework: JavaFX
        ii. Programming language: Java
    b. **Back-end:**
        i. Framework: SpringBoot
        ii. Programming language: Java
2. **Describe the software architecture of your project (implementation)**
    a. **Original Design Architecture:**
    - We use 3 tier architecture for our whole application and Robotlegs-like architecture for client-side desktop application.
    b. **The implementation of the design:**
    - The design of the whole system is described by following figure:

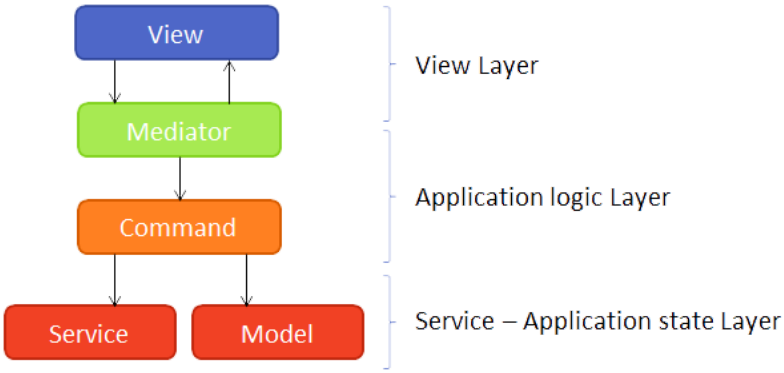


The whole system design

- **The whole system** follows 3-tier architecture, including: **Presentation tier** (client side application or webpages), **Application logic tier** and **Database-access tier**. 3-tier architectures provide many benefits for production and development environments by modularizing the user interface, business logic, and data storage layers.
    + **Presentation Tier:** The presentation tier is the front end layer in the 3-tier system and consists of the user interface.
    + **Application Tier:** The application tier contains the functional business logic which drives an application's core capabilities.
    + **Database-access tier:** The data tier comprises of the database/data storage system and data access layer. Data is accessed by the application layer via API calls.

Client application design

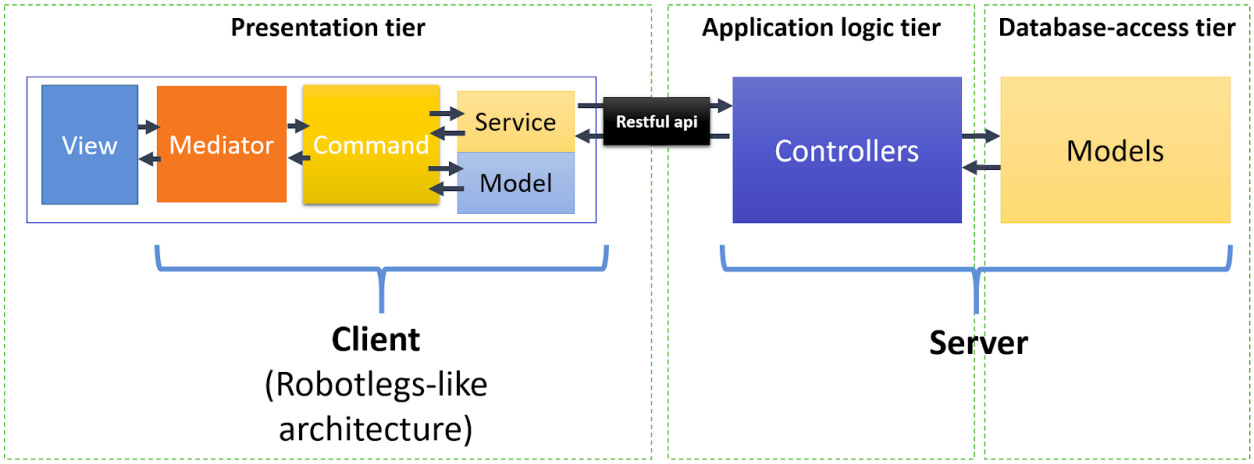
- The structure of client-side application is based on **Robotlegs**, an Apache Flex micro-architecture library.
- The diagram below shows the different components and layers in the client-side application architecture. The arrows indicate the dependencies.



-

- **View:** This is where the JavaFx components are used to build the UI.
- **Mediator:** The mediator facilitates the communication between the view and the application logic layer. This is why it's both included in the view layer as the application logic layer. This layer should not contain application logic, it's just a postman who receives calls from the view and delegates them to the application logic layer and vise versa.
- **Command:** This is where your application logic will reside. A Command is a class which executes a single unit of work and will be disposed when it's finished.
- **Service:** A service is used to access external resources. This can be a restful api (mostly), a file on the filesystem or a device attached to the user's pc (printer, barcode scanner, etc…).
- **Model:** A model stores the state of your application. This can be the selected record which should be accessible in multiple views.

Main flow of the application



- **We will take an example to describe the main flow of the application: *Delete a customer from customer list.* The flow of application will be:**

1. The view asks the mediator to delete the item by calling the deleteCustomer(customer) method on the ICustomerListMediator interface.

2. The mediator creates a new DeleteCustomerCommand and registers a success handler on it. Then it calls the start() method on the command.
3. The command calls the customer service to delete the contact from the remote server
4. Customer service connect to the server by a restful api and give request to delete customer to CustomerController.
5. CustomerController delete the customer from database by calling deleteCustomer(customer) method of CustomerModel and give the result back to CustomerService at client side.
6. The command calls a method on the model to remove the customer from the application state. It receives the updated list of customers from the model
7. The command finishes and triggers the registered success listeners. It provides the updated list of customers to the success handlers.
8. The success handler in the mediator asks the view to update the list of customers based on the updated customer list it got from the command.