System Requirements Specification

Index

For

Insurance Management Application

Version 1.0

TABLE OF CONTENTS

BACKEND-SPRING BOOT RESTFUL APPLICATION 3

1 Project Abstract 3

2 Assumptions, Dependencies, Risks / Constraints 4

2.1 InsurancePolicyConstraints: 4

3 Business Validations 4

4 Rest Endpoints 5

4.1 InsurancePolicyController 5

5 Template Code Structure 6

5.1 Package: com.insurancepolicy 6

5.2 Package: com.insurancepolicy.repository 6

5.3 Package: com.insurancepolicy.service 6

5.4 Package: com.insurancepolicy.service.impl 7

5.5 Package: com.insurancepolicy.controller 7

5.6 Package: com.insurancepolicy.dto 8

5.7 Package: com.insurancepolicy.entity 8

5.8 Package: com.insurancepolicy.exception 9

FRONTEND-ANGULAR SPA 12

1 Problem Statement 12

2 Proposed Insurance Policy Management Wireframe 12

2.1 Home Page 13

3 Business-Requirement: 13

4 Execution Steps to Follow for Backend 14

5 Execution Steps to Follow for Frontend 16

INSURANCE POLICY MANAGEMENT

System Requirements Specification

You need to consume APIs exposed by Backend application in Angular to make application work as FULLSTACK

BACKEND-SPRING BOOT RESTFUL APPLICATION

1 PROJECT ABSTRACT

The **Insurance Policy Management** is a FullStack Application with a backend implemented using Spring Boot with a MySQL database and a frontend developed using Angular. The application aims to provide a comprehensive platform for managing and organizing all insurance policies for a company.

Following is the requirement specifications:

	Insurance Policy Management
Modules	
1	Insurance Policy
Insurance Policy Module Functionalities	
1	Get all policies
2	Get policy by id
3	Create a new policy
4	Update a policy by id
5	Delete a policy by id

2 Assumptions, Dependencies, Risks / Constraints

2.1 POLICY constraints

- When fetching a policy by ID, if the policy ID does not exist, the service method should throw “Insurance Policy not found” message with NotFoundException class.
- When updating a policy, if the policy ID does not exist, the service method should throw “Insurance Policy not found” message with NotFoundException class.
- When removing a policy, if the policy ID does not exist, the service method should throw “Insurance Policy not found” message with NotFoundException class.

Common Constraints

- For all rest endpoints receiving @RequestBody, validation check must be done and must throw custom exception if data is invalid
- All the business validations must be implemented in dto classes only.
- All the database operations must be implemented on entity object only
- Do not change, add, remove any existing methods in service layer
- In Repository interfaces, custom methods can be added as per requirements.
- All RestEndpoint methods and Exception Handlers must return data wrapped in **ResponseEntity**

3 Business Validations

- PolicyNumber should not be null and size must be minimum 3 and maximum 10.
- PolicyType should not be null.
- PremiumAmount should not be null.
- StartDate should not be null.
- EndDate should not be null.

4 Database Operations

- InsurancePolicy should have a table created with the name “insurance_policies”.
- PolicyId should be treated as primary key and must be generated using IDENTITY technique.
- PolicyNumber should not be null.
- policyType should not be null.
- premiumAmount should not be null.
- startDate should not be null.
- endDate should not be null.
- isActive should not be null.
- customerId should not be null.

5 REST ENDPOINTS

Rest End-points to be exposed in the controller along with method details for the same to be created

5.1 INSURANACECONTROLLER

URL Exposed		Purpose
1. /api/policies		Fetches all the policies
Http Method	GET	
Parameter	-	
Return	List<InsurancePolicyDTO>	
2. /api/policies/{id}		Fetches a policy by id
Http Method	GET	
Parameter 1	Long (id)	
Return	InsurancePolicyDTO	
3. /api/policies/		Creates a new policy
Http Method	POST **The policy data to be created should be received in @RequestBody**	
Parameter	-	
Return	InsurancePolicyDTO	
4. /api/policies/{id}		Updates a policy by id
Http Method	PUT **The policy data to be updated should be received in @RequestBody**	
Parameter 1	Long (id)	
Return	InsurancePolicyDTO	
5. /api/policies/{id}		Deletes a policy by id
Http Method	DELETE	
Parameter 1	Long (id)	
Return	-	

6 Template Code Structure

6.1 Package: com.insurancepolicy

Resources

insurancePolicyManagementApplication (Class)	This is the Spring Boot starter class of the application.	Already Implemented

6.2 Package: com.insurancepolicy.repository

Resources

Class/Interface	Description	Status
InsurancePolicyRepository (interface)	• Repository interface exposing CRUD functionality for insurance policy Entity. • You can go ahead and add any custom methods as per requirements.	Partially implemented.

6.3 Package: com.insurancepolicy.service

Resources

Class/Interface	Description	Status
InsurancePolicyService (interface)	• Interface to expose method signatures for insurance policy related functionality. • Do not modify, add or delete any method.	Already implemented.

6.4 PACKAGE: COM.INSURANCEPOLICY.SERVICE.IMPL

Class/Interface	Description	Status
InsurancePolicyServiceImpl (class)	• Implements InsurancePolicyService. • Contains template method implementation. • Need to provide implementation for insurance policy related functionalities. • Do not modify, add or delete any method signature	To be implemented.

6.5 PACKAGE: COM.INSURANCEPOLICY.CONTROLLER

Resources

Class/Interface	Description	Status
insurancePolicyController (Class)	• Controller class to expose all rest-endpoints for insurance policy related activities. • May also contain local exception handler methods	To be implemented

6.6 PACKAGE: COM.INSURANCEPOLICY.DTO

Resources

Class/Interface	Description	Status
InsurancePolicyDTO (Class)	• Use appropriate annotations for validating attributes of this class.	Partially implemented.

6.7 PACKAGE: COM.INSURANCEPOLICY.ENTITY

Resources

Class/Interface	Description	Status
InsurancePolicy (Class)	● This class is partially implemented. ● Annotate this class with proper annotation to declare it as an entity class with **policyId** as primary key. ● Map this class with a **insurance policy** table. ● Generate the **policyId** using the IDENTITY strategy	Partially implemented.

6.8 PACKAGE: COM.INSURANCEPOLICY.EXCEPTION

Resources

Class/Interface	Description	Status
NotFoundException (Class)	● Custom Exception to be thrown when trying to fetch, update or delete the insurance policy info which does not exist. ● Need to create Exception Handler for same wherever needed (local or global)	Already implemented.

7 Method Descriptions

7.1 InsurancePolicyServiceImpl Class - Method Descriptions

- Declare a private variable named `insurancePolicyRepository` of type `InsurancePolicyRepository` and inject it using `@Autowired`.

Method	Task	Implementation Details
`getAllPolicies()`	Fetch all insurance policies	- Calls `insurancePolicyRepository.findAll()` - Converts `List<InsurancePolicy>` to `List<InsurancePolicyDTO>` using `convertToDTO()` - Return the list of InsurancePolicyDTO objects
`getInsurancePolicyById()`	Fetch a policy by its ID	- Calls `findById(id)` on the repository - If present, converts and returns the InsurancePolicyDTO - If not found, throws `NotFoundException` with message: "Insurance Policy not found"
`createInsurancePolicy()`	Create a new insurance policy	- Converts DTO to entity using `convertToEntity()` - Saves entity - Converts saved entity back to DTO - Return the newly created InsurancePolicyDTO
`updateInsurancePolicy()`	Update an existing insurance policy by ID	- Fetches policy by ID - If found, updates using DTO and converts result and returns the updated InsurancePolicyDTO - If not found, throws `NotFoundException` with message: "Insurance Policy not found"
`deleteInsurancePolicy()`	Delete an insurance policy by ID	- Calls `findById(id)` - If present, deletes using `deleteById()` and returns true - If not found, throws `NotFoundException` with message: "Insurance Policy not found"

7.2 InsurancePolicyController Class - Method Descriptions

- Declare a private variable named insurancePolicyService of type InsurancePolicyService and inject it using @Autowired.

Method	Task	Implementation Details
getAllPolicies()	To retrieve all insurance policies	- Request type: GET with URL `/api/policies` - Method name: `getAllPolicies` returns `List<InsurancePolicyDTO>>` - Calls `insurancePolicyService.getAllPolicies()` - Returns list of policies with `HttpStatus.OK`
getPolicyById()	To retrieve a specific policy by ID	- Request type: GET with URL `/api/policies/{id}` - Method name: `getPolicyById` returns `ResponseEntity<InsurancePolicyDTO>` - Extracts ID using `@PathVariable` - Calls `insurancePolicyService.getInsurancePolicyById(id)` - Returns the policy with `HttpStatus.OK`
createPolicy()	To create a new insurance policy	- Request type: POST with URL `/api/policies` - Method name: `createPolicy` returns `ResponseEntity<InsurancePolicyDTO>` - Uses `@Valid @RequestBody` to accept `InsurancePolicyDTO` - Calls `insurancePolicyService.createInsurancePolicy(dto)` - Returns created policy with `HttpStatus.CREATED`
updatePolicy()	To update an existing policy by ID	- Request type: PUT with URL `/api/policies/{id}`

		- Method name: `updatePolicy` returns `ResponseEntity<InsurancePolicyDTO>` - Uses `@Valid @RequestBody` to accept updated `InsurancePolicyDTO` - Extracts ID using `@PathVariable` - Calls `insurancePolicyService.updateInsurancePolicy(id, dto)` - Returns updated policy with `HttpStatus.OK`
`deletePolicy()`	To delete a policy by ID	- Request type: DELETE with URL `/api/policies/{id}` - Method name: `deletePolicy` returns `ResponseEntity<Void>` - Extracts ID using `@PathVariable` - Calls `insurancePolicyService.deleteInsurancePolicy(id)` - Returns empty response with `HttpStatus.NO_CONTENT`

FRONTEND-ANGULAR SPA

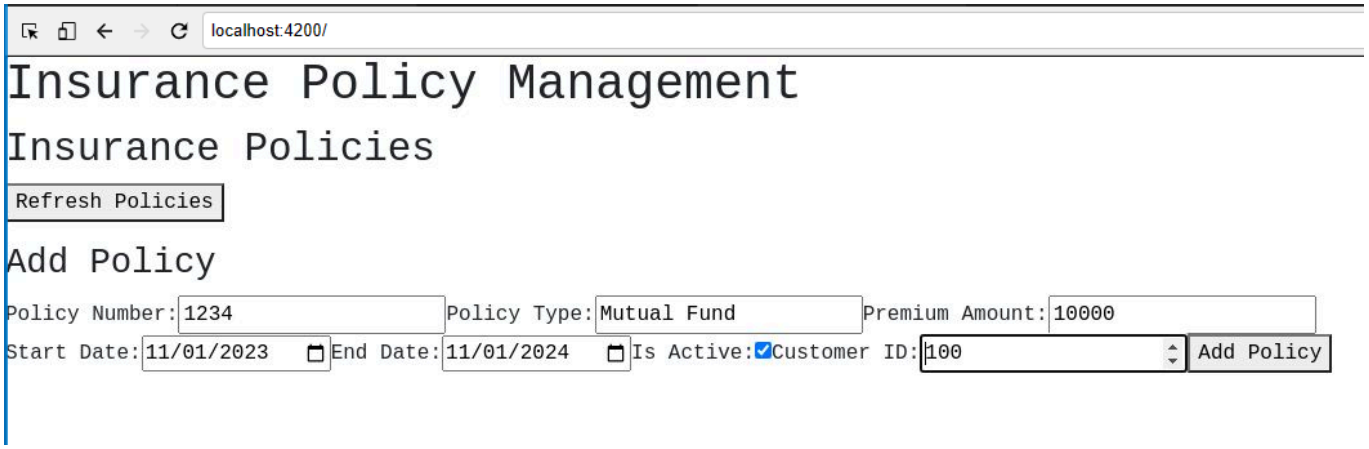
1 PROBLEM STATEMENT

Insurance Policy Application is SPA (Single Page Application), it allows you to add policy details, update policy details, delete policy and get all policies.

2 PROPOSED INSURANCE POLICY APPLICATION WIREFRAME

UI needs improvisation and modification as per given use case and to make test cases passed.

2.1 HOME PAGE



3 BUSINESS-REQUIREMENT:

As an application developer, develop the Insurance Policy Management (Single Page App) with below guidelines:

User Story #	User Story Name	User Story
US_01	Home Page	As a user I should be able to visit the Home page as the default page.
US_01	Home Page	As a user I should be able to see the homepage and perform all operations: **Acceptance criteria:** **App Component** **HTML Structure (`app.component.html`)** 1. **Main Heading** ○ Rendered inside an `<h1>` tag: ■ Text: **"Insurance Policy Management"** 2. **Embedded Component** ○ `<app-insurance-policy>` is used to render the policy UI. **Purpose**

- Acts as the root container.
- Only includes a title and delegates all UI functionality to the child component.

Insurance Policy Management Component

HTML Structure (`insurance-policy-management.component.html`)

1. **Heading Section**
   - `<h2>` displays: **"Insurance Policies"**

2. **Action Button**
   - A button labeled **"Refresh Policies"** triggers loading of data.

3. **List of Policies**
   - Uses `<ul>` to list each policy:
     - Displays:
       - `policyNumber`, `policyType`, `premiumAmount`
     - Includes buttons:
       - **Select** – to populate form
       - **Update** – to show update form
       - **Delete** – to remove the record
4. **Form Section**
   - `<h3>` dynamically displays: **"Add Policy"** or **"Update Policy"**
   - Form Fields:
     - **Policy Number** – text input
     - **Policy Type** – text input
     - **Premium Amount** – number input
     - **Start Date** – date input
     - **End Date** – date input
     - **Is Active** – checkbox
     - **Customer ID** – number input
   - Final button toggles between:
     - **"Add Policy"** or **"Update Policy"**

Functions & Responsibilities (`insurance-policy-management.component.ts`)

		1. **ngOnInit()** 1.1 Called on component load. 1.2 Triggers `loadPolicies()` to fetch initial data. 2. **loadPolicies()** 2.1 Retrieves all insurance policies from backend via `insurancePolicyService.getAllPolicies()` 2.2 Populates `policies` list. 2.3 On error → `console.error('Error loading policies:', error)` 3. **addPolicy()** 3.1 Calls `insurancePolicyService.createPolicy()` to save a new policy. 3.2 Refreshes list using `loadPolicies()` 3.3 Resets form via `createEmptyPolicy()` 3.4 On error → `console.error('Error adding policy:', error)` 4. **updatePolicyApi()** 4.1 Calls `insurancePolicyService.updatePolicy()` with updated data. 4.2 Refreshes list and resets form. 4.3 On error → `console.error('Error updating policy:', error)` 5. **deletePolicy(id)** 5.1 Calls `insurancePolicyService.deletePolicy()` 5.2 Removes policy from UI and clears selection. 5.3 On error → `console.error('Error deleting policy:', error)` 6. **selectPolicy(policy)** 6.1 Populates form with selected policy for editing. 7. **showUpdateForm(id)** 7.1 Retrieves policy from `policies` list by ID and loads it into form. 8. **createEmptyPolicy()** 8.1 Returns an empty template object to reset the form.

InsurancePolicyService

Purpose

- Acts as the **communication bridge** between the Insurance Policy component and the backend API.
- Uses Angular's `HttpClient` to perform **CRUD operations** for insurance policies.
- Communicates with API at: `http://127.0.0.1:8081/insurancepolicy/api/policies`

Service Functions & Responsibilities

1. **`getAllPolicies()`**
   - Sends a `GET` request to the backend.
   - Fetches **all insurance policies**.
   - Returns: `Observable<InsurancePolicy[]>`
2. **`getPolicyById(id: number)`**
   - Sends a `GET` request with a specific ID.
   - Fetches **one policy** by its id.
   - Returns: `Observable<InsurancePolicy>`
3. **`createPolicy(policy: InsurancePolicy)`**
   - Sends a `POST` request to add a **new policy**.
   - Sends the `policy` object in the request body.
   - Returns: `Observable<InsurancePolicy>`
4. **`updatePolicy(id: number, policy: InsurancePolicy)`**
   - Sends a `PUT` request to update an **existing policy**.
   - URL includes the policy `id`.
   - Sends updated `policy` object in the body.
   - Returns: `Observable<InsurancePolicy>`
5. **`deletePolicy(id: number)`**
   - Sends a `DELETE` request to remove a policy by its ID.
   - Returns: `Observable<void>`

Dynamic Behavior

		• On load: loadPolicies() fetches and displays all policies data.
		• Form resets after every successful **Add**, **Update**, or **Delete**
		• On **Add**:
		  ○ New entry is sent to the backend.
		  ○ UI refreshes with updated data.
		• On **Update**:
		  ○ Form is prefilled with selected policy.
		  ○ PUT request sent on submit.
		• On **Delete**:
		  ○ Deletes record and refreshes UI.
		• The form dynamically switches between "Add" and "Update" mode based on whether policyId is 0 or not.
		
		**** Kindly refer to the screenshots for any clarifications. ****

4 EXECUTION STEPS TO FOLLOW FOR BACKEND

1. **All actions like build, compile, running application, running test cases will be through Command Terminal.**
2. **To open the command terminal the test takers need to go to the Application menu (Three horizontal lines at left top) -> Terminal -> New Terminal.**
3. **cd into your backend project folder**
4. **To build your project use command:**

   mvn clean package -Dmaven.test.skip
5. **To launch your application, move into the target folder (cd target). Run the following command to run the application:**

   java -jar <your application jar file name>
6. **This editor Auto Saves the code.**
7. **These are time bound assessments the timer would stop if you logout and while logging in back using the same credentials the timer would resume from the same time it was stopped from the previous logout.**
8. **To test any Restful application, the last option on the left panel of IDE, you can find ThunderClient, which is the lightweight equivalent of POSTMAN.**
9. **To test any UI based application the second last option on the left panel of IDE, you can find Browser Preview, where you can launch the application.**
10. **Default credentials for MySQL:**
    a. **Username: root**
    b. **Password: pass@word1**

11. To login to mysql instance: Open new terminal and use following command:

a. sudo systemctl enable mysql
b. sudo systemctl start mysql

NOTE: After typing any of the above commands you might encounter any warnings.

>> Please note that this warning is expected and can be disregarded. Proceed to the next step.

c. mysql -u root -p
The last command will ask for password which is 'pass@word1'

12. Mandatory: Before final submission run the following command:

mvn test

5 Execution Steps to Follow for Frontend

1. **All actions like build, compile, running application, running test cases will be through Command Terminal.**
2. **To open the command terminal the test takers, need to go to**

   Application menu (Three horizontal lines at left top) -> Terminal ->New Terminal.

3. **This is a web-based application, to run the application on a browser, use the internal browser in the environment.**
4. **You can follow series of command to setup Angular environment once you are in your project-name folder:**
   a. **npm install -> Will install all dependencies -> takes 10 to 15 min**
   b. **npm run start -> To compile and deploy the project in browser. You can press <Ctrl> key while clicking on localhost:4200 to open project in browser -> takes 2 to 3 min**
   c. **npm run test -> to run all test cases. It is mandatory to run this command before submission of workspace -> takes 5 to 6 min**