

---

# **CASE STUDY ON**

## **Insurance Claim And**

## **Management System**

---

---

# INTRODUCTION

- The **Insurance Management System** is a web-based application designed to digitally manage insurance operations, including policy administration, claim processing, and system oversight through dedicated portals.
- The system is structured into **three modules** - Admin, Policy, and Claim portals - ensuring clear role separation, secure access, and efficient workflow management.
- Built using modern web and backend technologies, the project emphasizes scalability, responsiveness, and maintainability, making it suitable for real-world insurance management scenarios.

---

# ABSTRACT

- The project focuses on **centralized data management**, allowing insurance records to be stored, retrieved, and updated efficiently within a single integrated system.
- It enhances **operational transparency** by enabling systematic tracking of policies and claims throughout their lifecycle.
- The application supports informed **decision-making** by providing structured data flow and consistent record maintenance for insurance operations.

---

## WHAT WE ARE GOING TO BUILD: CLIENT REQUIREMENT?

**Client wants an Insurance Management System** to manage insurance policies, claims, and administrative records efficiently.

### **System Requirements:**

- Admin manages policy and claim records (create, update, delete, view).
- Policy module handles policy details and premium information.
- Claim module manages claim requests and claim status tracking.
- Policies and claims are categorized for organized and efficient processing.

---

## WHAT WE ARE GOING TO BUILD: SOME TECHNICAL TERMS

- Develop REST APIs using Spring Boot with proper Login and Registration modules.
- Implement JWT-based authentication and role-based authorization using Spring Security.
- Apply input validation using Hibernate Validator and handle errors with global exception handling.
- Enable pagination and sorting for APIs using Spring Data JPA.
- Document APIs using OpenAPI for easy consumer understanding.
- Deploy the Spring Boot backend on a cloud platform for scalable access.

---

# WHAT ARE THE TECHNOLOGIES AND TOOLS WE USE ?

- Framework:** Spring Boot Java Framework
- Programming Language:** Java 8+
- Build Tool:** Maven
- IDE:** Spring Tool Suite (STS)
- Server:** Apache Tomcat
- Spring Modules:** Spring Core, Spring Security (JWT), Spring Data JPA (Hibernate), etc.
- Database:** MySQL
- API Testing:** Postman REST Client
- API Documentation:** Swagger

---

## SYSTEM REQUIREMENTS

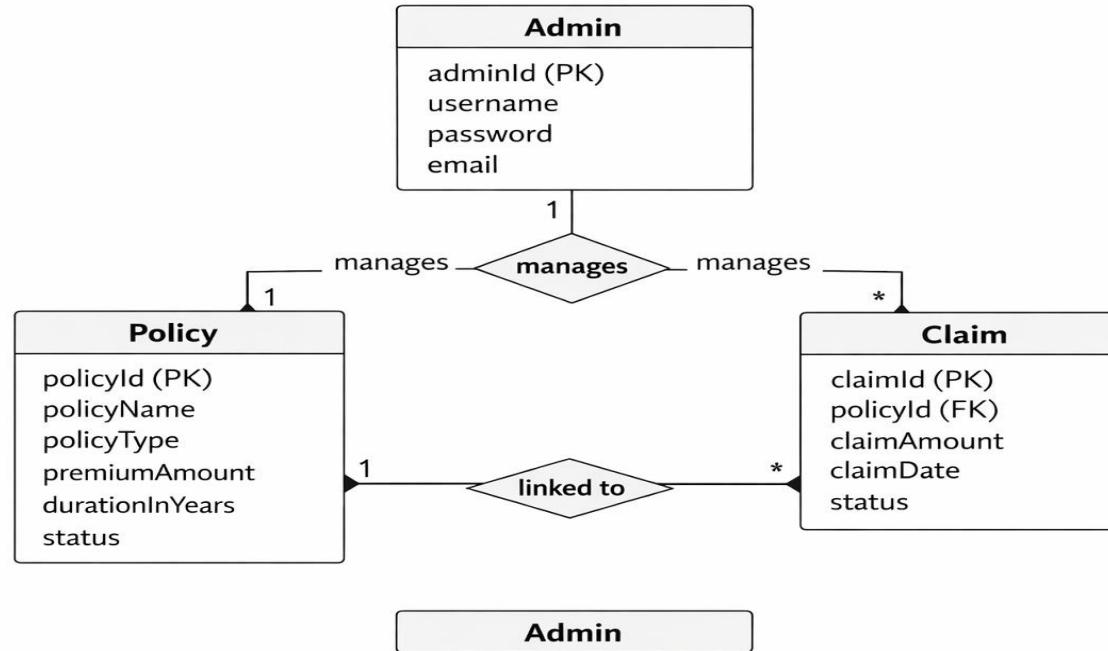
- Back End:** Spring Boot, JPA Hibernate, and Postman Tool
- DB Server:** MySQL Server 8.0 CE
- Web Server:** Apache Tomcat
- IDE:** Spring Tool Suite (STS)
- Browser:** Google Chrome
- Test:** JUnit

---

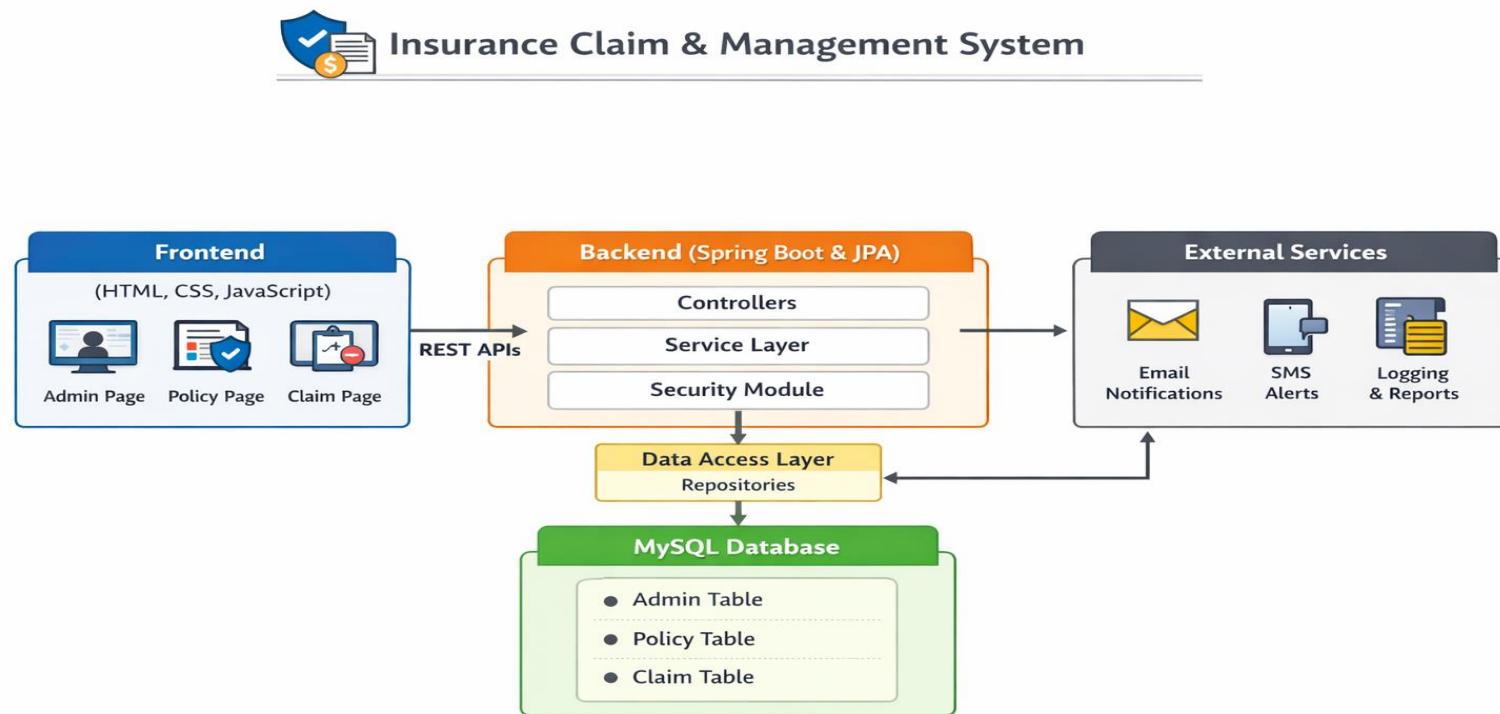
## PROJECT MODULES

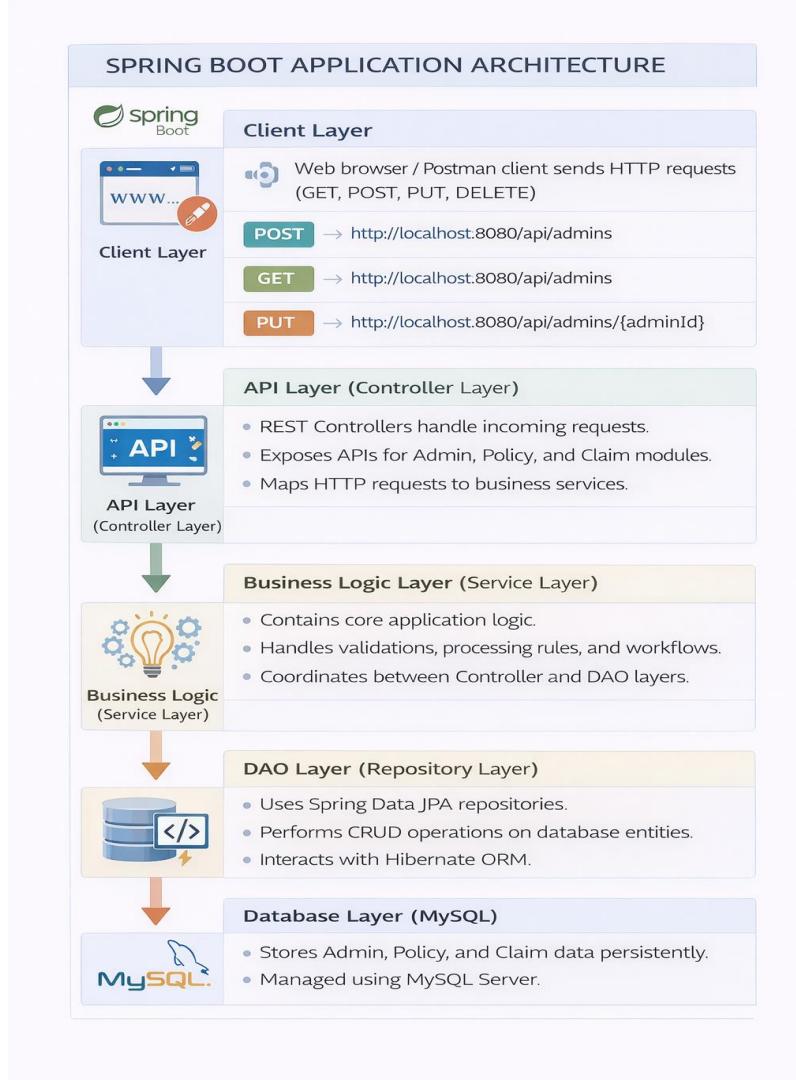
- Admin Module
- Policy Module
- Claim Module

# ER DIAGRAM



# CLIENT – SERVER ARCHITECTURE





# SPRING BOOT APP

---

## ADMIN MODULE

- Manages insurance policies and claim records.
- Performs create, update, delete, and view operations.
- Controls system access and oversees overall insurance operations.

---

## POLICY MODULE

- Manages creation and maintenance of insurance policy details.
- Stores premium amount, policy type, duration, and status information.
- Allows updating and viewing of active and inactive policies.
- Ensures organized and centralized policy data management.

---

## CLAIM MODULE

- Handles insurance claim requests linked to specific policies.
- Records claim amount, claim date, and current claim status.
- Supports updating and tracking claim approval or rejection.
- Maintains accurate and organized claim records for processing.

# HTTP REQUEST METHODS

METHOD	URL	
ADMINS API	<b>POST</b>	→ http://localhost:8080/api/admins
	<b>GET</b>	→ http://localhost:8080/api/admins
	<b>PUT</b>	→ http://localhost:8080/api/admins/{adminId}
	<b>DELETE</b>	→ http://localhost:8080/api/admins/{adminId}
POLICIES API	<b>POST</b>	→ http://localhost:8080/api/policies
	<b>GET</b>	→ http://localhost:8080/api/policies
	<b>PUT</b>	→ http://localhost:8080/api/policies/{policyId}
	<b>DELETE</b>	→ http://localhost:8080/api/policies/{policyId}
CLAIMS API	<b>POST</b>	→ http://localhost:8080/api/claims
	<b>GET</b>	→ http://localhost:8080/api/claims
	<b>PUT</b>	→ http://localhost:8080/api/claims/{claimId}
	<b>DELETE</b>	→ http://localhost:8080/api/claims/{claimId}

---

# DATA DICTIONARY

# Tables of Database

The screenshot shows the MySQL Workbench interface. In the Query Editor (Query 1), the following SQL commands are run:

```
1 • CREATE DATABASE insurance_db;
2 • USE insurance_db;
3 • SHOW TABLES;
```

The Result Grid displays the output of the SHOW TABLES command:

Tables_in_insurance_db
admin
claims
policies

In the Output pane, the Action Output table shows the results of the executed queries:

#	Time	Action	Message	Duration / Fetch
2	18:19:57	USE insurance_db	0 row(s) affected	0.016 sec
3	20:39:45	SHOW TABLES	3 row(s) returned	0.015 sec / 0.000 sec

# Admin Database

The screenshot shows the MySQL Workbench interface. The title bar reads "MySQL Workbench" and "Mysql@localhost:3306". The menu bar includes File, Edit, View, Query, Database, Server, Tools, Scripting, and Help. The toolbar has various icons for database management tasks.

The Navigator pane on the left lists categories like MANAGEMENT, INSTANCE, and PERFORMANCE. Under MANAGEMENT, it shows Server Status, Client Connections, Users and Privileges, Status and System Variables, Data Export, and Data Import/Restore. Under INSTANCE, it shows Startup / Shutdown, Server Logs, and Options File. Under PERFORMANCE, it shows Dashboard.

The central area contains two tabs: "Query 1" and "SQL File 1\*". The "Query 1" tab displays the following SQL code:

```
1 •  SELECT * FROM insurance_db.admin;
2 •  show tables;
```

The "Result Grid" pane below shows the results of the first query:

admin_id	email	password	username
2	admin1@example.com	admin123	admin1
*	NULL	NULL	NULL

The "Output" pane at the bottom shows the execution history:

#	Time	Action	Message	Duration / Fetch
9	20:47:12	SELECT * FROM claims LIMIT 0, 1000	1 row(s) returned	0.015 sec / 0.000 sec
10	20:49:01	SELECT * FROM insurance_db.admin LIMIT 0, 1000	1 row(s) returned	0.000 sec / 0.000 sec

A message in the SQLAdditions pane states: "Automatic context help is disabled. Use the toolbar to manually get help for the current caret position or to toggle automatic help."

# Policy Database

The screenshot shows the MySQL Workbench interface. The top navigation bar includes File, Edit, View, Query, Database, Server, Tools, Scripting, and Help. The left sidebar has sections for MANAGEMENT (Server Status, Client Connections, Users and Privileges, Status and System Variables, Data Export, Data Import/Restore), INSTANCE (Startup / Shutdown, Server Logs, Options File), and PERFORMANCE (Dashboard, Administration, Schemas). The main area displays a query results grid and an output pane.

**Query Editor:**

```
1 •  SELECT * FROM insurance_db.policies;
2 •  show tables;
```

**Result Grid:**

policy_id	duration_in_years	policy_name	policy_type	premium_amount
2	2	Life Insurance	Family	7500
3	1	Health Insurance	Individual	5000
4	1	Health Insurance	Individual	5000
5	1	Health Insurance	Individual	5000
*	HULL	HULL	HULL	HULL

**Output:**

#	Time	Action	Message	Duration / Fetch
10	20:49:01	SELECT * FROM insurance_db.admin LIMIT 0, 1000	1 row(s) returned	0.000 sec / 0.000 sec
11	20:49:43	SELECT * FROM insurance_db.policies LIMIT 0, 1000	4 row(s) returned	0.000 sec / 0.000 sec

# Claim Database

The screenshot shows the MySQL Workbench interface with the following details:

- Navigator:** MANAGEMENT (Server Status, Client Connections, Users and Privileges, Status and System Variables, Data Export, Data Import/Restore), INSTANCE (Startup / Shutdown, Server Logs, Options File), PERFORMANCE (Dashboard), Administration (selected), Schemas.
- Query Editor:** Query 1 tab, SQL File 1\* tab. The query is:

```
1 •  SELECT * FROM insurance_db.claims;
2 •  show tables;
```
- Result Grid:** Shows the results of the first query. The table has columns: claim\_id, claim\_amount, claim\_date, claim\_status, claim\_type. One row is displayed:

claim_id	claim_amount	claim_date	claim_status	claim_type
2	10000	2025-12-13	Pending	Accident
- Action Output:** Shows two recent actions:

#	Time	Action	Message	Duration / Fetch
11	20:49:43	SELECT * FROM insurance_db.policies LIMIT 0, 1000	4 row(s) returned	0.000 sec / 0.000 sec
12	20:50:14	SELECT * FROM insurance_db.claims LIMIT 0, 1000	1 row(s) returned	0.000 sec / 0.000 sec

# POST METHOD - ADMIN

The screenshot shows the Postman application interface. On the left, the sidebar displays 'Tissina Gold's Workspace' with sections for Collections, Environments, History, Flows, and Files (BETA). The main workspace shows a 'POST New Request' for the 'Insurance-management / New Request' collection. The request method is set to 'POST' and the URL is 'http://localhost:8080/api/admins'. The 'Body' tab is selected, showing raw JSON data:

```
1 {
2   "username": "admin",
3   "password": "admin123",
4   "email": "admin@example.com"
5 }
```

The response section shows a '200 OK' status with a response time of 728 ms and a response size of 246 B. The response body is identical to the request body:

```
1 {
2   "adminId": 1,
3   "username": "admin",
4   "password": "admin123",
5   "email": "admin@example.com"
6 }
```

At the bottom, there are navigation links for Cloud View, Find and replace, Console, Terminal, Runner, Start Proxy, Cookies, Vault, Trash, and Help.

# GET METHOD - ADMIN

The screenshot shows the Postman application interface. On the left, the sidebar displays 'Tissina Gold's Workspace' with sections for Collections, Environments, History, Flows, and Files (BETA). The main workspace shows a 'New Request' dialog for an 'Insurance-management / New Request'. The method is set to 'GET' and the URL is 'http://localhost:8080/api/admins'. The 'Headers' tab shows 7 headers. The 'Body' tab is selected and set to 'none'. Below the request area, the response section shows a status of '200 OK' with a response time of 19 ms and a size of 248 B. The response body is displayed as JSON:

```
1 [  
2 {  
3   "adminId": 1,  
4   "username": "admin",  
5   "password": "admin123",  
6   "email": "admin@example.com"  
7 }]  
8 ]
```

# DELETE METHOD - ADMIN

The screenshot shows the Postman application interface. On the left, the sidebar displays 'Tissina Gold's Workspace' with sections for Collections, Environments, History, Flows, and Files (BETA). The main workspace shows a 'New Request' dialog for the 'Insurance-management' collection. The method is set to 'DELETE' and the URL is 'http://localhost:8080/api/admins/1'. The Headers tab shows 7 items. The Body tab is selected and set to 'none'. Below the request, the response section shows a status of '200 OK' with a response body containing '1 Admin deleted successfully'.

# POST METHOD - POLICY

The screenshot shows the Postman application interface. On the left, the sidebar displays 'Tissina Gold's Workspace' with sections for Collections, Environments, History, Flows, and Files (BETA). The main workspace shows an environment named 'Insurance-management' with two collections: 'APICRUD' and 'Insurance-management'. Under 'Insurance-management', there are two GET requests: 'ADMIN API' and 'POLICY API'. The 'POLICY API' is selected, and a POST request is being configured. The request URL is 'http://localhost:8080/api/policies'. The 'Body' tab is selected, showing a JSON payload:

```
1 {  
2   "policyName": "Health Insurance",  
3   "policyType": "Health",  
4   "premiumAmount": 5000,  
5   "durationInYears": 2  
6 }  
7
```

The response section shows a successful '200 OK' status with a response time of 566 ms and a response size of 275 B. The response body is identical to the request body, indicating a successful creation of a new policy.

# GET METHOD - POLICY

The screenshot shows the Postman application interface. On the left, the sidebar displays 'Tissina Gold's Workspace' with collections like APICRUD and Insurance-management, and environments like My Collection. The main workspace shows a 'Policy API' collection under 'Insurance-management'. A 'GET' request is selected with the URL `http://localhost:8080/api/policies`. The 'Body' tab is active, showing the response body is empty: 'This request does not have a body'. Below the request details, the response section shows a '200 OK' status with a response time of 165 ms and a size of 277 B. The response body is displayed as JSON:

```
[{"policyId": 1, "policyName": "Health Insurance", "policyType": "Health", "premiumAmount": 5000.0, "durationInYears": 2}]
```

# DELETE METHOD - POLICY

The screenshot shows the Postman application interface. On the left, the sidebar displays 'Tissina Gold's Workspace' with sections for Collections, Environments, History, Flows, and Files (BETA). The main workspace shows two environments: 'DEL ADMIN API' and 'DEL POLICY API'. The 'DEL POLICY API' environment is selected. A collection named 'Insurance-management' is expanded, showing two GET requests: 'ADMIN API' and 'POLICY API'. A DELETE request is being configured with the URL `http://localhost:8080/api/policies/1`. The Headers tab shows 7 items. The Body tab is selected and set to 'none'. The response section shows a 200 OK status with the message 'Policy deleted successfully'.

# POST METHOD - CLAIM

The screenshot shows the Postman application interface. The left sidebar displays 'Tissina Gold's Workspace' with sections for Collections, Environments, History, Flows, and Files (BETA). The main workspace shows a collection named 'Insurance-management' under 'APICRUD'. Within this collection, there are three API endpoints: 'GET ADMIN API', 'GET POLICY API', and 'GET CLAIM API'. The 'GET CLAIM API' endpoint is currently selected. A POST request is being prepared for this endpoint, with the URL set to 'http://localhost:8080/api/claims'. The request body is defined as follows:

```
1 {  
2   "claimType": "Accident",  
3   "claimAmount": 10000,  
4   "claimDate": "2025-12-13",  
5   "claimStatus": "Pending"  
6 }  
7
```

The response section shows a successful '200 OK' status with a response time of 546 ms and a response size of 271 B. The response body is identical to the request body.

# GET METHOD - CLAIM

The screenshot shows the Postman application interface. On the left, the sidebar displays 'Tissina Gold's Workspace' with sections for Collections, Environments, History, Flows, and Files (BETA). The main workspace shows a collection named 'Insurance-management' containing three APIs: 'ADMIN API', 'POLICY API', and 'CLAIM API'. The 'CLAIM API' is selected. A request card for a 'GET' method at 'http://localhost:8080/api/claims' is displayed, with the status bar showing '200 OK', '181 ms', and '273 B'. The response body is shown as JSON:

```
[{"claimId": 1, "claimType": "Accident", "claimAmount": 10000.0, "claimDate": "2025-12-13", "claimStatus": "Pending"}]
```

# DELETE METHOD - CLAIM

The screenshot shows the Postman application interface. On the left, the sidebar displays 'Tissina Gold's Workspace' with sections for Collections, Environments, History, Flows, and Files (BETA). The main workspace shows a collection named 'Insurance-management / CLAIM API' under 'APICRUD'. A specific request is selected: a 'DELETE' method against the URL `http://localhost:8080/api/claims/1`. The Headers tab shows 7 items. The Body tab is selected, showing the message 'This request does not have a body'. Below the request details, the response section shows a green '200 OK' status with a timestamp of '37 ms' and a size of '190 B'. The response body contains the message '1 Claim deleted successfully'. At the bottom of the screen, there are various navigation and utility icons.

# JUNIT TEST CASE - ADMIN

The screenshot shows the Spring Tools for Eclipse interface with the following details:

- Title Bar:** workspace-spring-tools-for-eclipse-4.32.2.RELEASE - insurance-management/src/test/java/com/example/insurance/repository/AdminRepositoryTest.java - Spring Tools for Eclipse
- Toolbar:** Includes standard Eclipse icons for File, Edit, Source, Refactor, Navigate, Search, Project, Run, Window, Help.
- Project Explorer:** Shows files like Admin.java, Policy.java, AdminReposit..., PolicyRepos..., ClaimReposit..., application....
- JUnit View:** Shows "Finished after 5.771 seconds" with 2 runs, 0 errors, and 0 failures.
- Code Editor:** Displays AdminRepositoryTest.java code:

```
1 package com.example.insurance.repository;
2
3 import static org.junit.jupiter.api.Assertions.*;
4
5 import java.util.Optional;
6
7 import org.junit.jupiter.api.Test;
8 import org.springframework.beans.factory.annotation.Autowired;
9 import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
10
11 import com.example.insurance.entity.Admin;
12
13 @DataJpaTest
14 class AdminRepositoryTest {
```
- Console View:** Shows Java runtime logs including Hibernate schema creation and destruction, and a warning about sharing support.

```
<terminated> AdminRepositoryTest [JUnit] C:\Program Files\Java\jdk-17.0.5\bin\javaw.exe (14-Dec-2025, 7:24:44 pm – 7:24:51 pm elapsed: 0:00:06.936) [pid: 5316]
Hibernate: drop table if exists admin cascade
Hibernate: drop table if exists claims cascade
Hibernate: drop table if exists policies cascade
Hibernate: create table admin (admin_id bigint generated by default as identity, email varchar(255), password varchar(255), user
Hibernate: create table claims (claim_amount float(53) not null, claim_date date, claim_id bigint generated by default as identit
Hibernate: create table policies (duration_in_years integer not null, premium_amount float(53) not null, policy_id bigint generat
2025-12-14T19:24:49.766+05:30 INFO 5316 --- [           main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
2025-12-14T19:24:50.264+05:30 INFO 5316 --- [           main] c.e.i.repository.AdminRepositoryTest : Started AdminRepository
Java HotSpot(TM) 64-Bit Server VM warning: Sharing is only supported for boot loader classes because bootstrap classpath has been
Hibernate: insert into admin (email,password,username,admin_id) values (?,?=?,default)
Hibernate: insert into admin (email,password,username,admin_id) values (?,?=?,default)
2025-12-14T19:24:51.110+05:30 INFO 5316 --- [ionShutdownHook] j.LocalContainerEntityManagerFactoryBean : Closing JPA EntityManagerFactory for persistence unit 'default'
Hibernate: drop table if exists admin cascade
Hibernate: drop table if exists claims cascade
Hibernate: drop table if exists policies cascade
```
- Failure Trace:** An empty panel.

# JUNIT TEST CASE - POLICY

The screenshot shows the Spring Tools for Eclipse interface with the following details:

- Title Bar:** workspace-spring-tools-for-eclipse-4.32.2.RELEASE - insurance-management/src/test/java/com/example/insurance/repository/AdminRepositoryTest.java - Spring Tools for Eclipse
- Toolbar:** Standard Eclipse-style toolbar with icons for file operations, search, and run.
- Project Explorer:** Shows the project structure with files like Admin.java, Policy.java, AdminRepositoryTest.java, PolicyRepositoryTest.java, ClaimRepositoryTest.java, and application.properties.
- Run View:** Displays the results of the test run:
  - Finished after 7.432 seconds
  - Runs: 2/2 Errors: 0 Failures: 0
  - PolicyRepositoryTest [Runner: JUnit 5] (0.781 s)
    - testFindPolicyById() (0.773 s)
    - testSavePolicy() (0.006 s)
- Console View:** Shows the command-line output of the test run, including Hibernate SQL logs and Java HotSpot VM warnings.
- Failure Trace:** A small panel showing the failure trace for any failed tests.

# JUNIT TEST CASE - CLAIM

The screenshot shows the Spring Tools for Eclipse interface with the following details:

- Project Explorer:** Shows files like Admin.java, Policy.java, AdminRepositoryTest.java, PolicyRepositoryTest.java, and ClaimRepositoryTest.java.
- JUnit View:** Shows a green bar indicating the test completed successfully with "Runs: 2/2 Errors: 0 Failures: 0".
- Code Editor:** Displays the code for AdminRepositoryTest.java, which includes imports for JUnit Jupiter, Java.util.Optional, and Spring Boot Data JPA, along with annotations for @DataJpaTest and a class definition.
- Console View:** Shows the command-line output of the JUnit run, including Hibernate schema creation and drop statements, and a warning about Java HotSpot VM sharing support.
- Failure Trace:** A panel at the bottom left showing the failure trace for the test.

---

## **DEMO VIDEO LINK**

[https://drive.google.com/drive/folders/1-Bc2Rnny0cODpOJUbY\\_KwzibzfT3Cn4\\_?usp=drive\\_link](https://drive.google.com/drive/folders/1-Bc2Rnny0cODpOJUbY_KwzibzfT3Cn4_?usp=drive_link)

---

**SUBMITTED BY**

**TISSINA GOLD TS**

**BE CSE – DATA SCIENCE (III YR)**

**REGISTER NO : 43733078**