**Project Overview:**

**Domain:** Insurance

**Project Name:** LINQ Core Application (Sub-project of Marsh) from Nov 2022 – April 2023.

**Technical Details:** Java, Spring Boot, MYSQL, Angular.

**Summary:**

The LINQ Core Application is a business insurance platform designed to provide businesses with an integrated system for managing their insurance policies, processing claims, assessing risks, and handling customer data. The platform is designed to simplify the complex processes involved in business insurance, offering a user-friendly interface and efficient backend services.

* Key stakeholders included the client (Archipology), business owners, product owners, development teams, and end users like insurance agents and policyholders.

The architecture of LINQ application:

* Policy Management: Handles policy creation, updates, policy issuances, renewals, and validations.
* Claims Management: Manages claims filing, claims tracking.
* Customer Management: Maintains customer profiles and customers self-service portals.
* Billing & Payments: Invoice Generation, payment remainders are managed.

**My Contribution:**

As the **backend developer**, my role used to involve building, maintaining, and improving the **server-side logic** that powers the Business Insurance Platform. I used to collaborate with front-end developers, QA teams, and project managers to deliver a seamless, highly functional insurance platform.

**Roles and Responsibilities:**

* I am responsible for designing and implementing the database schema using **MySQL**. This includes creating tables for customer information, insurance policies, payments, claims, invoices, and reminders.

Example: Created relational tables for customers, policies, invoices, payments, claims, etc.

* Develop RESTful APIs using **Spring Boot** to expose services for interacting with the frontend (built using **Angular**). These APIs should be designed to handle requests related to user registration, policy management, payment processing, and more.
* Writing unit and integration tests to validate API endpoints and ensure error-free application.

**User Stories:**

1. Implement Error Messages for Policy-Related Operations in Backend

* 1. Define Error Response Structure for Policy Operations:
* Create a standardized error response structure that includes:
* Error Code: Unique code for each error type (e.g., POLICY-404 for policy not found).
* Message: User-friendly description of the error (e.g., “The specified policy could not be found.”).
* Details: Technical information for logs (e.g., policy ID, requested operation) to aid in troubleshooting.
* Implement Error Handling for Policy Service:
* In the backend policy service, add handling for common errors, such as:
* Policy Not Found: Return a 404 Not Found response if the policy ID does not exist.
* Invalid Policy ID: Return a 400 Bad Request response if the policy ID format is incorrect.
* Data Validation Errors: Return a 422 Un processable Entity response if policy data fails validation (e.g., missing required fields).
* For each error, return a structured error message to the frontend and log the full error details.

2. Writing Junit test cases for policy creation.

Policy Creation Tests

* When a valid policy is provided, it should be successfully saved to the database.
* All required fields in a policy should be validated.
* If any required data is missing or invalid, the creation process should fail with appropriate error messages.
* Edge cases, such as duplicate policies or invalid data formats, should be tested.

3. View Records for 5-Year or Custom Date Range in Program, Premium, and Policy Tables

* I want to view records of customers from the program, premium, and policy tables for either the last 5 years or a custom date range of my choice.

4. Create an API so that users can add multiple companies to their portfolio.

* To achieve this, I have created a method addCompanyToUserPortfolio(userId, companyId). So, it adds a selected company to a user's portfolio.
* Created an endpoint to add a company to user’s portfolio.

5. Created a Trail register form using Responsive forms in angular.

**Challenges:**

1. This project involves complex calculations for premiums, policy coverage, and regulatory compliance. Implementing this logic internally led to performance issues or incorrect outputs if not well-optimized.

* Use Spring Boot's transaction management features to ensure that business-critical transactions, such as policy issuance or claims settlement, are executed atomically. Write unit tests and integration tests to validate the correctness of all business logic. SonarQube can be used to enforce good coding practices and prevent technical debt.

2. Maintaining code quality is essential in a long-term project, especially in this project, because it is a single application where business logic is concentrated.

* Integrate SonarQube into my IntelliJ Idea to continuously review the codebase. SonarQube helps catch issues like code smells, duplicated code, low test coverage, and potential security vulnerabilities.

**Outcomes:**

This project is expected to deliver a highly secure, efficient, and scalable system that significantly improves the way insurance policies and claims are managed. The automation of manual processes, high code quality, and CI/CD capabilities contribute to long-term sustainability, allowing the business to grow and evolve with confidence.

**Learning Experience:**

Working with both Spring Boot on the backend and Angular on the frontend provides hands-on experience with full-stack development. I learnt how to build RESTful APIs in Spring Boot and integrate them with a responsive, user-friendly Angular front end.