**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 simplify the management of business insurance needs for companies. It provides a digital platform that streamlines processes like obtaining insurance quotes, managing policies, submitting claims, and monitoring compliance with insurance requirements. By offering a central solution for managing risk portfolios, the application enhances decision-making, simplifies data access, and improves interaction with insurance offerings for employees, customers, and contractors.

* Key stakeholders included the client (Archipology), business owners, product owners, development teams, and end users like insurance agents and policyholders.
* In the LINQ Core project, R2D2 is a powerful tool that serves several key roles in managing application configurations and naming conventions, as well as supporting multilingual functionality.

The architecture of LINQ application:

* Policy Management: Handles policy creation, updates, renewals, and validations.
* Claims Management: Manages claims filing, adjudication, and processing.
* Customer Management: Maintains customer information and account details.
* Premium Calculation: Calculates premiums based on policy details, coverage type, and other factors.

**Roles and Responsibilities:**

* As a developer my roles and responsibilities are to develop restful APIs in LINQ Core Application for insurance functionalities such as, user registration, policy applications, renewals and claims.
* Work closely with the business team to understand requirements and implement logic that aligns with insurance workflows, like eligibility checks and policy validation.
* Writing unit and integration tests to validate API endpoints and ensure error-free deployment.

**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 from the program, premium, and policy tables for either the last 5 years or a custom date range of my choice,
* so that I can analyze historical data or specific periods based on my requirements.
* Assume that the program, premium, and policy tables each contain a created date or date field that stores the date of each record, which will be used for filtering.
* Each record must have an indexed date field for efficient querying, especially if the tables contain a large volume of data.

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, that 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 form using Responsive forms in angular.

**Challenges:**

1. This project involves complex calculations for premiums, policy coverage, claims adjudication 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 CI/CD pipeline 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.