

Cognizant Academy

Daily Insurance Portal

Case Study Specification

Daily Insurance Portal

Contents

[1 Problem statement 4](#_Toc113530290)

[2 Use case Diagram for the Problem Statement 4](#_Toc113530291)

[3 Use case details 5](#_Toc113530292)

[4 Functional/Non-Functional Requirement of the Problem Statement 5](#_Toc113530293)

[4.1 Client Registration 5](#_Toc113530294)

[4.1.1 Acceptance Criteria 6](#_Toc113530295)

[4.1.2 UI-Screen View 6](#_Toc113530296)

[4.1.3 REST End Points 7](#_Toc113530297)

[4.1.4 Entity 7](#_Toc113530298)

[4.1.5 Non-Functional Requirement 7](#_Toc113530299)

[4.1.6 Business Validations 7](#_Toc113530300)

[4.2 Client Login 8](#_Toc113530301)

[4.2.1 Acceptance criteria 8](#_Toc113530302)

[4.2.2 UI-Screen View Login 8](#_Toc113530303)

[4.2.3 REST End Points 9](#_Toc113530304)

[4.2.4 Entity 9](#_Toc113530305)

[4.2.5 Detailed Requirement 9](#_Toc113530306)

[4.2.6 Non- Functional Requirements 9](#_Toc113530307)

[4.2.7 Business Validations 9](#_Toc113530308)

[4.3 Add Wallet Balance 9](#_Toc113530309)

[4.3.1 Acceptance criteria 10](#_Toc113530310)

[4.3.2 UI-Screen View 10](#_Toc113530311)

[4.3.3 REST End Points 10](#_Toc113530312)

[4.3.4 Entity 11](#_Toc113530313)

[4.3.5 Detailed Requirement 11](#_Toc113530314)

[4.3.6 Non-Functional Requirements 11](#_Toc113530315)

[4.3.7 Business Validations 11](#_Toc113530316)

[4.4 Purchase policy 11](#_Toc113530317)

[4.4.1 Acceptance criteria 12](#_Toc113530318)

[4.4.2 UI-Screen View 12](#_Toc113530319)

[4.4.3 REST End Points 12](#_Toc113530320)

[4.4.4 Entity 13](#_Toc113530321)

[4.4.5 Detailed Requirement 13](#_Toc113530322)

[4.4.6 Non-Functional Requirements 13](#_Toc113530323)

[4.4.7 Business Validations 14](#_Toc113530324)

[4.5 Process Claim 14](#_Toc113530325)

[4.5.1 Acceptance criteria 15](#_Toc113530326)

[4.5.2 UI-Screen View 15](#_Toc113530327)

[4.5.3 REST End Points 15](#_Toc113530328)

[4.5.4 Entity 15](#_Toc113530329)

[4.5.5 Detailed Requirement 16](#_Toc113530330)

[4.5.6 Non-Functional Requirements 16](#_Toc113530331)

[4.5.7 Business Validations 16](#_Toc113530332)

[5 Expected Deliverables 16](#_Toc113530333)

[6 Milestone 17](#_Toc113530334)

[7 Skills to develop the project 17](#_Toc113530335)

[8 Implementation Notes 18](#_Toc113530336)

[9 Evaluation rubrics 18](#_Toc113530337)

# Problem statement

The purpose of the requirements document is to systematically capture requirements for the project and the system “**Daily Insurance Portal**” to be developed. The application should be Cloud Native Architecture with Microservices. Both functional and non-functional requirements are captured in this document. It also serves as the input for the project scoping.

**About the System**

The client would like to develop a Daily Insurance Portal application to store the client information, wallet and let the client purchase a daily insurance plan and process the claims through the portal to Downstream Systems.

**Scope of the System**

The scope of the system is explained through its modules as follows

1. Client Induction – used by client to register the details of self into the system and select a predefined wallet with money into it. The system stores the details of the Client and his/her wallet in the system and able to edit it.
2. Process Claim – The user should be able to request a claim to Downstream system. (Just Update the claim amount back in the wallet)

# Use case Diagram for the Problem Statement

**Client Registration**

Save to DB

First Time User

Client Registration

**Client Login**

Query from DB

First Time User

Client Login

**Add wallet**

Save to DB

Add Wallet: Mode + Money

Landing Page

Logged In client

**Purchase policy**

Save to DB

Select Policy + Pay

Landing Page

Logged In client

**Process Claim**

Save to DB

Request Claim and Submit Proof

Landing Page

Logged In client

# Use case details

|  |  |  |
| --- | --- | --- |
| **User Story #** | **User Story Name** | **User Story** |
| US\_01 | Client Registration | User to sign up in the system with user name, password, first name, last name and email address |
| US\_02 | Client Login | Client should be able to log in using registered username and password |
| US\_03 | Add Wallet Balance | Client should opt for pre-defined wallet with details like Mode (Credit card, Debit card, UPI) and amount of money |
| US\_04 | Purchase policy | Client should be able to select a suitable daily insurance policy and pay from wallet. |
| US\_05 | Process Claim | Client should be able to raise a claim request with details and proof and get the amount added in the wallet. |

# Functional/Non-Functional Requirement of the Problem Statement

## Client Registration

|  |  |
| --- | --- |
| US-01 | Client Registration |
| Acceptance Criteria User needs to provide Username, Password, First Name, Last Name and Email address | |
| UI-Screen View | |
| REST End Points POST - /register  **Input**   * First name * Last Name * Username * Email ID * Password in encrypted format * Confirm Password in encrypted format   **Output**   * The service should response 200 ok along with success registration message. * Error code 400 for any business validation error. * Error code 500 for internal prog. error | |
| Entity **User**   * **ID**   + Variable name as <id> * **First Name**   + Variable name as <first-name> * **Last Name**   + Variable name as <last-name> * **Username**   + Variable name as <user-name> * **Email ID**   + Variable name as <email-id> * **Password**   + Variable name as <password> | |
| Non-Functional Requirement  * Proper error message & logging should be in-place. * Swagger end points should be available | |
| Business Validations  * All fields are mandatory. * Email id should be in valid email pattern, containing a single @. * Password contains at least 8 characters and at most 20 characters.   + It contains at least one digit.   + It contains at least one upper case alphabet.   + It contains at least one lower case alphabet.   + It contains at least one special character which includes! @#$%&\*()-+=^.   + It doesn’t contain any white space. | |

## Client Login

|  |  |
| --- | --- |
| US\_02 | Client Login |
| Acceptance criteria Client should be able to log in using registered username and password | |
| UI-Screen View Login | |
| REST End Points POST - /login  **Input**   * Username * Password in encrypted format   **Output**   * The service should response 200 ok along with success message. * Error code 400 for any business validation error. * Error code 500 for internal prog. error | |
| Entity **User entity used earlier** | |
| Detailed Requirement  * Needs to call API and this method should query the db for valid username and password. * When the details are verified successfully, the service should response 200 ok along with success message and allow user to proceed. * The next screen should be the insurance purchase and claims history view page, where all the previous insurance purchase claims made by user and balance wallet value will be showing in table structure. * User will have option to go to **Add Wallet Balance**, **Purchase Insurance** and **Request Claim** pages. * If there are any exceptions while connecting/querying to DB, the service should throw corresponding error with error status as 500. | |
| Non- Functional Requirements  * Swagger end points should be available * Proper error message & logging should be in-place. | |
| Business Validations  * All fields are mandatory. * For a new customer the claims history table is empty and wallet balance is zero. * User will have option to go to **Add Wallet Balance** first. There will be options to **Purchase Insurance** and **Request Claim** options as well, but the former should be disabled till balance value is added. * Balance value can’t be minus. No Insurance can be purchased when the fund value is zero. | |

## Add Wallet Balance

|  |  |
| --- | --- |
| US\_03 | Add Wallet Balance |
| Acceptance criteria Client should be able to opt for pre-defined wallet with details like Mode (Credit card, Debit card, UPI) and amount of money | |
| UI-Screen View | |
| REST End Points POST - /add-wallet-balance  **Input**   * Wallet from dropdown * Amount in numeric format   **Output**   * The service should response 200 ok along with success message. * Error code 400 for any business validation error. * Error code 500 for internal prog. error | |
| Entity **Wallet**   * **Mode**   + Variable name as <mode> * **Amount**   + Variable name as < amount > | |
| Detailed Requirement  * Wallet should be drop-down with values: **Debit Card**, **Credit Card** and **UPI** * Once the user enters the details, they should be sent to the POST method and saved in the db. * Mandatory fields should be validated as mentioned in the rules above and 400 exception response should be sent with the missing field details. * When the details are saved successfully, the service should response 200 ok along with success message. * If there are, any exceptions while connecting/saving to DB. The service should throw corresponding error with error status as 500. | |
| Non-Functional Requirements  * Swagger end points should be available. * Proper error message & logging should be in-place. | |
| Business Validations  * Amount can hold only numeric values and used should not be able to give value more than 10,000 and less than 0. | |

## Purchase policy

|  |  |
| --- | --- |
| US\_04 | Purchase policy |
| Acceptance criteria Client should be able to select a suitable daily insurance policy and pay from wallet. | |
| UI-Screen View | |
| REST End Points POST - /purchase-policy  **Input**   * Policy from drop-down * Auto-populated one-time insurance premium calculated based on selected insurance   **Output**   * The service should response 200 ok along with success message. * Error code 400 for any business validation error. * Error code 500 for internal prog. error | |
| Entity **Policy (Look up/predefined Table)**   * **ID**   + Variable name as <policy-id> * **Policy name**   + Variable name as <policy-name> * **Policy premium**   + Variable name as <policy-premium> * **Policy Maximum Coverage**   + Variable name as <policy-coverage>   **UserPolicyClaim (Transactional Table)**   * **User ID**   + Variable name as <user-id> foreign key from user table * **Policy id**   + Variable name as <policy-id> foreign key from policy table * **DateTime of Purchase**   + Variable name as <purchase-dttm> in yyyy-MM-ddTHH:mm:ss format * **Claim amount**   + Variable name as <claim-amount> * **Claim status**   + Variable name as <claim-status> * **DateTime of Claim**   + Variable name as <claim-dttm> in yyyy-MM-ddTHH:mm:ss format | |
| Detailed Requirement  * Policy table will be a table with pre-defined values for policy id, name and the premium details. One sample:  |  |  |  | | --- | --- | --- | | policy id | policy name | policy premium | | POL\_1 | Daily Accidental Coverage policy upto 10,000 | 100 |  * Once the user enters the details, they should be sent to the POST method and saved in the db. * When the details are saved successfully, the service should response 200 ok along with success message. * If there are any exceptions while connecting/saving to DB, the service should throw corresponding error with error status as 500. | |
| Non-Functional Requirements  * Swagger end points should be available. * Proper error message & logging should be in-place. | |
| Business Validations  * Both front-end and backend validation should be present to accept correct values. * Once the premium value is deducted, the Wallet should show the updated amount. * On a single day, a given user won’t be able to purchase more than one insurance. * At the time of policy purchase, claim details should be inserted as null. | |

## Process Claim

|  |  |
| --- | --- |
| US\_04 | Process Claim |
| Acceptance criteria Client should be able to raise a claim request with details and proof and get the amount added in the wallet. | |
| UI-Screen View | |
| REST End Points PUT - /process-claim  **Input**   * Policy Name from drop-down * Date of purchase * Claim value   **Output**   * The service should response 200 ok along with success message. * Error code 400 for any business validation error. * Error code 500 for internal prog. error | |
| Entity **UserPolicyClaim entity used earlier** | |
| Detailed Requirement  * Once the user enters the details, they should be sent to the PUT method and saved in the db. * When the details are saved successfully, the service should response 200 ok along with success message. * If there are any exceptions while connecting/saving to DB, the service should throw corresponding error with error status as 500. | |
| Non-Functional Requirements  * Swagger end points should be available. * Proper error message & logging should be in-place. | |
| Business Validations  * Both front-end and backend validation should be present to accept correct values. * Claim value should be only numeric and can’t be more than the max coverage value associated with the insurance. * No claim can be submitted for the same day of purchase the insurance. * If the Insurance plan type and date purchased is not matching with the plan and date selected for coverage, the same will be discarded. * Claim status should be updated based on successful claim processing and the wallet amount should also reflect the updated value always. | |

# Expected Deliverables

The following deliverables are expected as outcomes:

* Application Code base
* Readme document on the complete application
  + Setup of the application
  + How to run the application
  + Any inference
  + Screenshot of UI results
* Reports:
  + Unit/Functional Test Report
  + SonarQube report with no Blocker/Critical bugs, more than 70% coverage and less than 3% code duplication
  + Architecture/design diagram
  + Swagger API documentation (screenshot of Swagger page should be fine)

# Milestone

The milestone for the project use is given below

|  |  |  |
| --- | --- | --- |
| **Milestone** | **Duration (in weeks)** | **Topic** |
| Milestone -1 | 1.5 | * Develop the APIs for the application with Java, Spring boot along with Database design, set-up, table creation with test data. |
| Milestone – 2 | 1 | * Design and develop the UI for the application |
| Milestone -3 | 1 | * Integrate service layer with UI component, Containerize the application and deploy in any cloud |

# Skills to develop the project

List the Technology based on your respective technology stack, that will be used to development the project.

|  |  |
| --- | --- |
| **Layer** | **Tech Stack** |
| Front end | * React * JavaScript * Karma/ Cypress/ Jest |
| Back End | * JDK11 * Spring Boot > 2.5 * JPA/Hibernate * Maven * SonarQube * Junit 5 |
| Database | * MongoDB / MySQL / PostgreSQL |
| Deployment Infra | * AWS/Azure/GCP <<TBD>> * Amazon ECR /DockerHub |
| Code repository | * GitHub/GitLab |
| IDE | * IntelliJ Community Ed/Eclipse/STS |
| Container | * Docker |

# Implementation Notes

As per the project requirement modification can be done in the below table.

|  |  |
| --- | --- |
| Milestone -1 | **Spring Boot**  Create Spring Boot REST Microservices to perform SAVE Operation using POST method.   * Use Microservices Architecture * Follow coding standards * Follow Standard project structure * Message input/output format should be in JSON (Read the values from the property/input files, wherever applicable). Input/output format can be designed as per the discretion of the participant * Database connections and web service URLs should be configurable. * Use browser / POST Man to invoke APIs * Swagger implementation. * Run SonarQube for code quality. * Implement Junit for unit testing. |
| Milestone -2 | * Implement user-stories using any one of the UI frameworks [React] * Implement Forms, databinding, validations * Use appropriate unit test framework. |
| Milestone -3 | * Integrate service layer with UI component. * Containerize the application * Push your Docker images to an Amazon ECR repository |

# Evaluation rubrics

|  |  |
| --- | --- |
| Angular | 1. Associate must have used Angular Components, Modules, Databinding, data validation, CLI commands. 2. Associate must have used Forms and Forms validation 3. Associate must have used Directives 4. Associate must have developed Reusable Components 5. Associate must have followed coding standards |
| REACT | 1. Associate must have used Component, Databinding, data validation, CLI commands. 2. Associate must have used Forms and Forms validation 3. Associate must have defined React state 4. Associate must have followed coding standards |
| Microservices, Java, JPA/Hibernate, Database | 1. REST controller 2. Follow controller ->service->Dao model 3. Entity and Model classes 4. Appropriate logging statements 5. Exception handling 6. Usage of Java 8 features such as streams, lambda, Async |
| Docker | 1. Containerize the application 2. Build docker containers 3. Push your Docker images to an Amazon ECR repository with the docker push command |
| AWS | 1. Code is committed in AWS 2. Push your container images to Amazon Elastic Container Registry |