|  |  |
| --- | --- |
|  | **Cognizant Academy**  **Policy Management System**  **FSE – Business Aligned Project**  **Case Study Specification**  **Version 1.0** |
| |  |  |  |  | | --- | --- | --- | --- | |  | **Prepared By / Last Updated By** | **Reviewed By** | **Approved By** | | **Name** | Khaleelullah Hussaini Syed |  |  | | **Role** | Trainer |  |  | | **Signature** | t-syed8 |  |  | | **Date** | 26 October 2022 |  |  | |
|  |

Table of Contents

[1.0 Important Instructions 3](#_Toc119075129)

[2.0 Introduction 4](#_Toc119075130)

[2.1 Purpose of this document 4](#_Toc119075131)

[2.2 Project Overview 4](#_Toc119075132)

[2.3 Scope 4](#_Toc119075133)

[3.0 Use Case Diagram 6](#_Toc119075134)

[4.0 System Architecture Diagram 7](#_Toc119075135)

[5.0 Development Phases 7](#_Toc119075136)

[6.0 System Requirements 8](#_Toc119075137)

[**6.1.1** **Module – Profile Management** 8](#_Toc119075138)

[**6.1.2** **Module – Policy Plans** 11](#_Toc119075139)

[**6.1.3** **Module – Premium Management** 15](#_Toc119075140)

[**6.1.4** **Module – Claims Management** 18](#_Toc119075141)

[7.0 Deployment requirements 22](#_Toc119075142)

[8.0 Design Considerations 22](#_Toc119075143)

[9.0 Reference learning 23](#_Toc119075144)

[10.0 Project Templates 25](#_Toc119075145)

[11.0 Change Log 25](#_Toc119075146)

# Important Instructions

1. Associate must adhere to the Design Considerations specific to each Technolgy Track.
2. Associate must not submit project with compile-time or build-time errors.
3. Being a Full-Stack Developer Project, you must focus on ALL layers of the application development.
4. Unit Testing is Mandatory, and we expect a code coverage of 100%. Use Unit testing and Mocking Frameworks wherever applicable.
5. All the Microservices, Client Application, DB Scripts, have to be packaged together in a single ZIP file. Associate must submit the solution file in ZIP format only.
6. If backend has to be set up manually, appropriate DB scripts have to be provided along with the solution ZIP file.
7. A READ ME has to be provided with steps to execute the submitted solution, the Launch URLs of the Microservices in cloud must be specified.

(Importantly, the READ ME should contain the steps to execute DB scripts, the LAUNCH URL of the application)

1. Follow coding best practices while implementing the solution. Use appropriate design patterns wherever applicable.
2. You are supposed to use an In-memory database or code level data as specified, for the Microservices that should be deployed in cloud. No Physical database is suggested for Microservice.

# Introduction

## Purpose of this document

The purpose of the software requirement document is to systematically capture requirements for the project and the system “Loan Management System” that has to be developed. Both functional and non-functional requirements are captured in this document. It also serves as the input for the project scoping.

The scope of this document is limited to addressing the requirements from a user, quality, and non-functional perspective.

High Level Design considerations are also specificed wherever applicable, however the detailed design considerations have to be strictly adhered to during implementation.

## Project Overview

Policy Market is a new startup in the business of various kinds of policies like life insurance, health insurance, etc., Your team has been assigned the account of policy market to develop their application. The application should have a responsive web based interface to perform various activies like – managing customer profiles, working with policies and subscription, tracking and paying the policy premiums and managing the claims associated with the policies.

## Scope

Below are the modules that needs to be developed part of the Project:

|  |  |  |
| --- | --- | --- |
| **Req. No.** | **Req. Name** | **Req. Description** |
| REQ\_01 | **Profile Management** module | * Customers will be able to create a new profile for themselves * Customers can register their nominees into the system * Customers can also remove a particular nominee added earlier |
| REQ\_02 | **Policy Plans** module | * Policy managers can create new policy plans in the system * Customers and policy managers can search the existing policies available * A customer will be able to subscribe to a policy of their choosing |
| REQ\_03 | **Premium Management** module | * The customers can pay the montly premium for their policies * Customers can also view the history of their premium payments * Policy managers can view the due premium for the current month |
| REQ\_04 | **Claims management** module | * The customers can raise a new claim request * Policy managers will be able to view all the new claim requests * The policy managers will be able to process a claim request |

Table 1 : Application Modules

# Use Case Diagram

The following use case diagram shows various users of the system and their responsibilities.

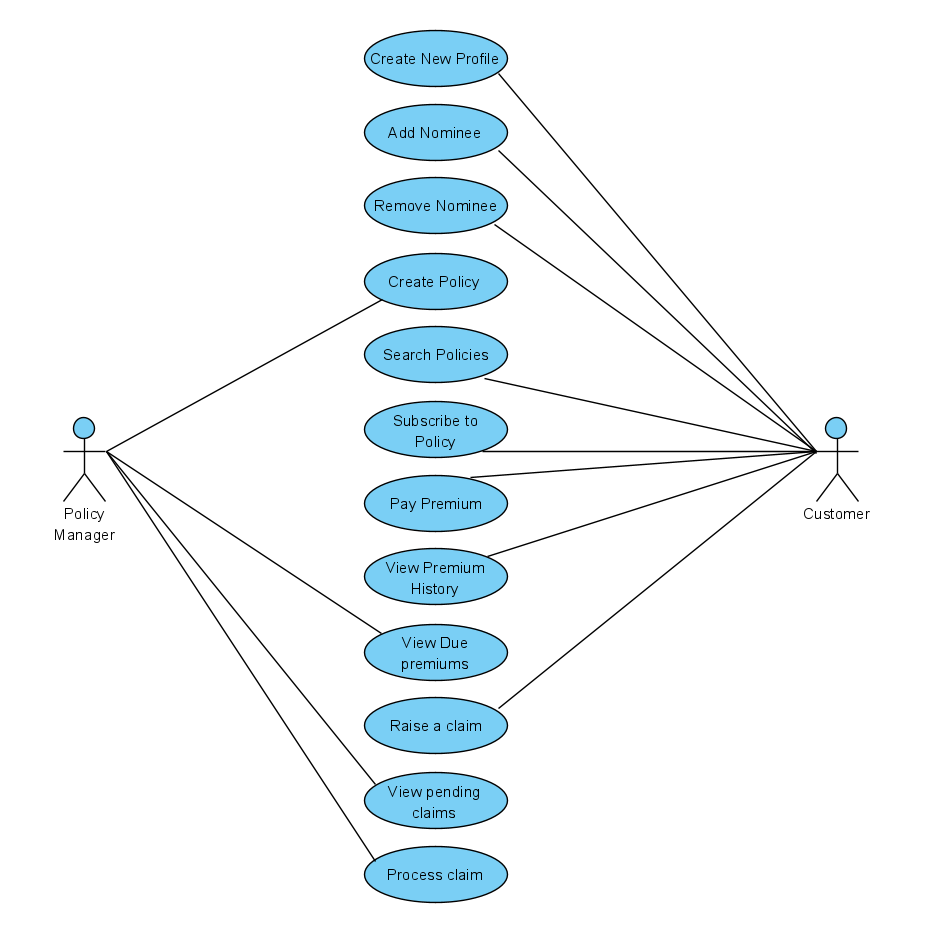


Figure 1 : Use case diagram

# System Architecture Diagram

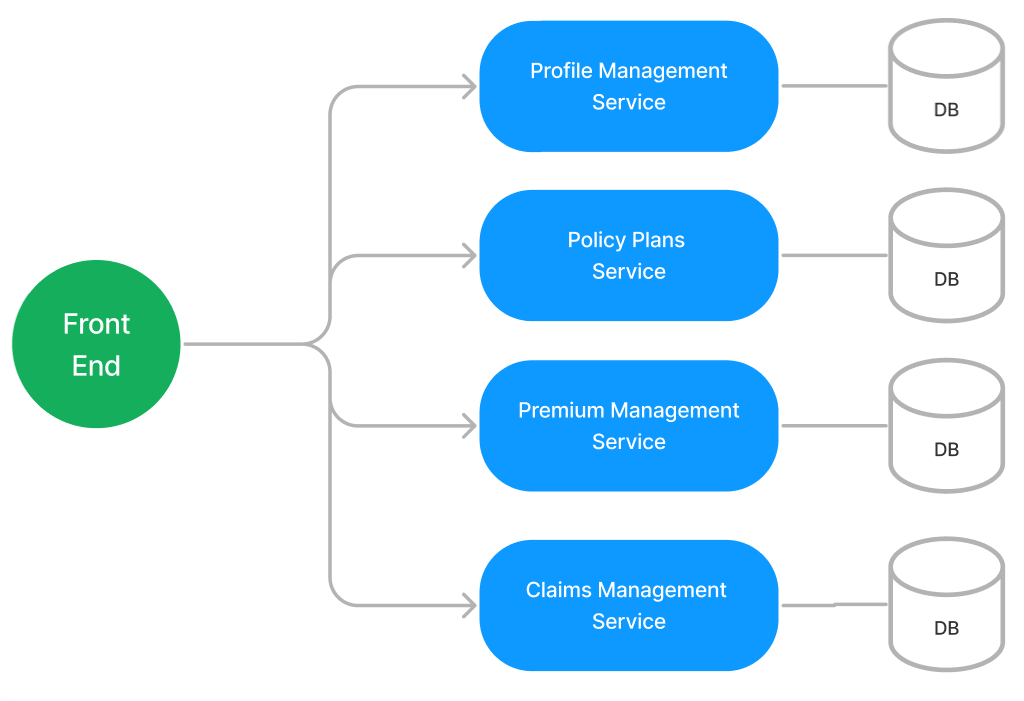


Figure 2 : Application Architecture Diagram

# Development Phases

* The application will be developed in 2 phase.
* Each phase will have 4 stages followed by a review at the end.
* The phase-1 output will be unit tested core business logic of the application.
* In phase-2 the output will be a functional application with micro-service and the Front end.
* Each stage of the development phase must be completed alongside the learning milestone

# System Requirements

### **Module – Profile Management**

This module is primarily used by the customers of the portal to manage their profiles. It allows the following operations.

1. Customers will be able to create a new profile for themselves
2. Customers can register their nominees into the system
3. Customers can also remove a particular nominee added earlier

**Stage: Database Implementation**

1. Design a data base as per the following ER diagram provided.

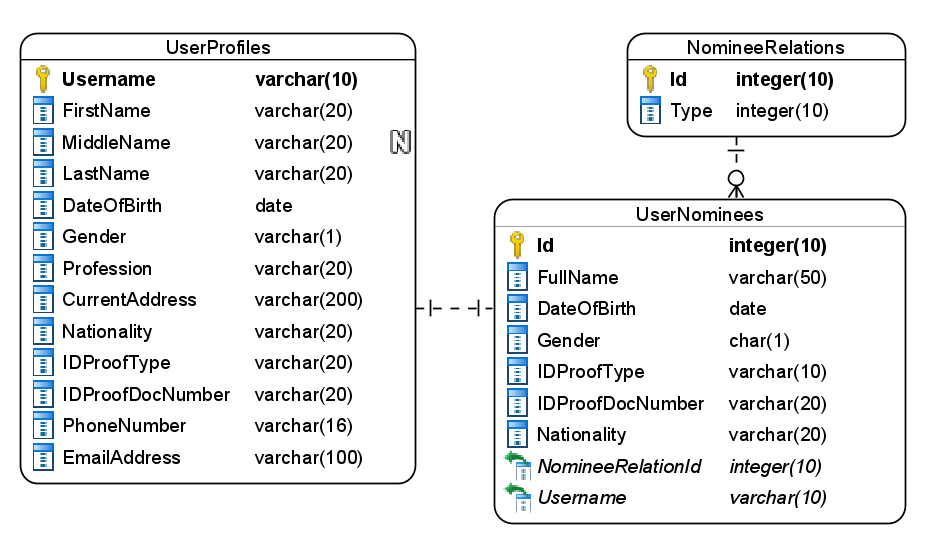


Figure 3 : ER Diagram – Profile Management

1. Enforce the following constraints on the database apart from primary key, foreign key and unique keys
   1. Gender should accept either M/F/O as the value
   2. Allowed values on IDProofType are – Passport/Aadhar/PAN/DrivingLicence
   3. Ensure that as per the date of birth user is 18 years old
   4. Username should always be 10 characters long

**Note**: Seed the data into the NomineeRelations table with values as – Parent, Child, Sibling, Spouse, Cousin, etc.,

**Stage: Data Access Layer Design**

1. Create a library project and add ORM support into it.
2. Use the ORM to map the entities to database as per the ER diagram provided.
3. Use repository per entity pattern and generate the repositories to perform the following operations
   1. Return list of nominee types
   2. Insert a new profile
   3. Insert a new nominee for customer
   4. Get a nominee for customer
   5. Remove a nominee for customer

**Stage: Business Logic Layer Development**

1. Develop a library which reference the Data Access Library project created earlier
2. This class library will contain various service classes which will encapsulate the business logic for the application.
3. Use dependency injection to in service classes to inject the required repositories.
4. Create the service classes following the single responsibility principle which perform the given operations as follows
   1. Fetch the list of nominee types
   2. Add a customer profile
   3. Add a nominee for customer
   4. Fetch the nominee for customer
   5. Remove a nominee for customer
5. Following business rules must be implemented as part of the business service class
   1. Each customer is allowed to add only 1 nominee. If a customer tries to add multiple raise a user-defined exception as “Only1NomineeAllowedException”
   2. If the nationality of customer is other than “Indian” then the only acceptable ID proof type is passport.
   3. Username should be auto-generated and return as in the following format – aabbbbXXXX – aa= first 2 letter of first name, bb= first 4 letters of last name, XXXX= 4 digit serial number.

**Stage: Unit Testing**

1. Create a new Unit test project to test the service classes created in business logic layers
2. Mock all the repositories using a mocking framework.

**Stage: Micro-service implementation**

1. Create a API project which references the business logic layer created earlier
2. Implement service documentation using swagger
3. All exceptions in the micro-service must be handled and logged using a logging library
4. Create the following end-points and test them using postman and export the requests into a json file.

Table 2 : Profile Management - Endpoint - 1

|  |  |
| --- | --- |
| **URL** | /api/nomineetypes |
| **Request Type** | GET |
| **User Role** | Customers |
| **Trigger** | Front end |
| **Description** | Customers can use this endpoint to choose the relationship with nominee |
| **Inputs** |  |
| **Outputs** | NomineeTypeDTOs |

Table 3 : Profile Management - Endpoint - 2

|  |  |
| --- | --- |
| **URL** | /api/profile |
| **Request Type** | POST |
| **User Role** | Customers |
| **Trigger** | Front end |
| **Description** | Using this endpoint a customer will be able to create a new profile and get a username |
| **Inputs** | ProfileDTO |
| **Outputs** | Status code and username |

Table 4 : Profile Management - Endpoint - 3

|  |  |
| --- | --- |
| **URL** | /api/profile/<username>/nominee |
| **Request Type** | POST |
| **User Role** | Customers |
| **Trigger** | Front end |
| **Description** | This endpoint will allow the customer to add a new nominee |
| **Inputs** | NomineeDTO, Username |
| **Outputs** | Status code |

Table 5 : Profile Management - Endpoint - 4

|  |  |
| --- | --- |
| **URL** | /api/profile/<username>/nominee |
| **Request Type** | GET |
| **User Role** | Customers |
| **Trigger** | Front end |
| **Description** | This endpoint will be responsible for returning the nominee for a customer |
| **Inputs** | Username |
| **Outputs** | NomineeDTO |

Table 6 : Profile Management - Endpoint - 5

|  |  |
| --- | --- |
| **URL** | /api/profile/<username>/nominee |
| **Request Type** | DELETE |
| **User Role** | Customers |
| **Trigger** | Front end |
| **Description** | This endpoint will allow the customers to delete a nominee from the system |
| **Inputs** | Username |
| **Outputs** | Status code |

**Stage: Font-end design**

Create the following components as per the specification provided below.

1. NewProfileFormComponent
2. Develop a component which is used by the customers to create new profile in the system
3. Provide a navigation to the component from main menu
4. Component should contain a form to accept the profile details
5. Use a set of radio buttons for gender
6. Provide a dropdown list for IDProof type and Profession
7. Once all details are validated the form should be allowed to be submitted and an acknowledgement must be displayed along with newly generated username
8. AddNomineeFormComponent
9. Develop a component which is used by the customers add a nominee in the system
10. Provide a navigation to the component from main menu
11. Component should contain a form to accept the nominee details
12. Provide a dropdown list for IDProof type, Gender, Nominee type
13. Once all details are validated the form should be allowed to be submitted and an acknowledgement must be displayed.
14. NomineeComponent
15. Design a component for customers to view their nominee details and provide a navigation to it via navbar
16. The component should fetch and display the details of existing nominee by accepting the username as the input view a search textbox.
17. The customer should also be allowed to delete the nominee by clicking the delete button.
18. Once the operation is successful, display an acknowledgement.

**Stage: Integration of Frontend and backend**

1. Create a data service in the font-end application which will communicate with the micro-services.
2. Use the data service in the components to make them interact with the API
3. Valid error messages should be shown based on various response status codes received form the API

### **Module – Policy Plans**

This module will be used for managing policies and the subscriptions to the policies by the customers and policy managers to perform the following tasks.

1. Policy managers can create new policy plans in the system
2. Customers and policy managers can search the existing policies available
3. A customer will be able to subscribe to a policy of their choosing

**Stage: Database Implementation**

1. Design a data base as per the following ER diagram provided.

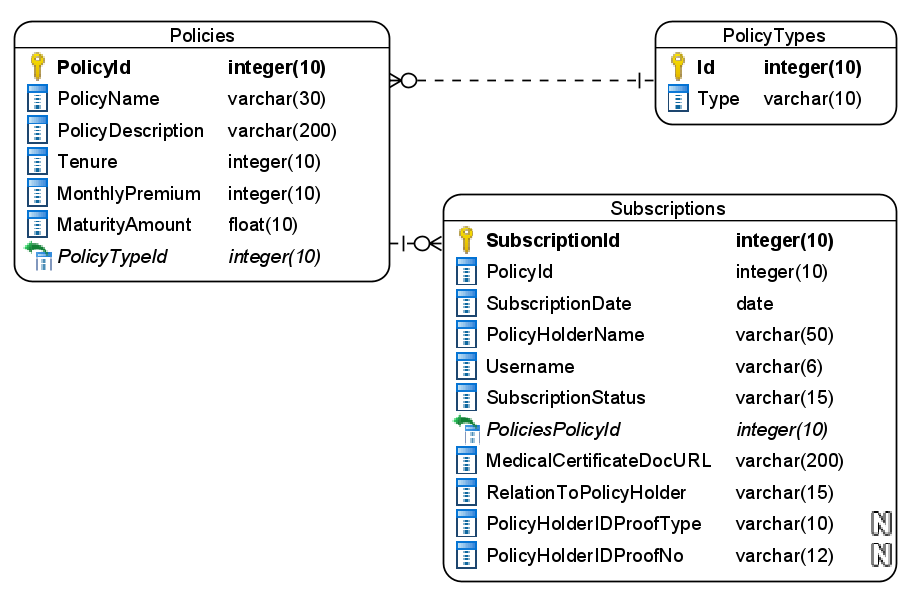


Figure 4 : ER Diagram – Policy Plans

1. Apply the following constraints apart from primary keys and foreign keys on the database
   1. PolicyHolderName should be minimum 5 characters long
   2. Subscription date should be automatically taken as today by default
   3. Allowed values for RelationToPolicyHolder are – Parent/Child/Sibling/Spouse/Cousin
   4. Subscription status should be – New/Matured/Defaulted/Terminated

Note: Seed the data for policy types as Life Insurance, Health Insurance, etc.,

**Stage: Data Access Layer Design**

1. Create a library project and add ORM support into it.
2. Use the ORM to map the entities to database as per the ER diagram provided.
3. Use repository per entity pattern and generate the repositories to perform the following operations
   1. Return policy types list
   2. Insert a new policy
   3. Return list of policies by tenure or premium amount or maturity amount
   4. Return a policy by id
   5. Insert new subscription

**Stage: Business Logic Layer Development**

1. Develop a library which reference the Data Access Library project created earlier
2. This class library will contain various service classes which will encapsulate the business logic for the application.
3. Use dependency injection to in service classes to inject the required repositories.
4. Create the service classes using the single responsibility principle which perform the given operations as follows
   1. Get all policy types list
   2. Create new policy
   3. Search policy by Tenure/Maturity Amount/Premium amount
   4. Get policy by id
   5. Add new subscription
5. Following business rules must be implemented as part of the business service class
   1. Each customer can have only 1 policy for himself and each member of ther family. Multiplie policies per customer or his family members is not allowed. For example – if Jojo has 2 children he has have a total of 3 policies 1 for himself and 1 for each of his children. If a user tries to create more generate a user defined exception as “PolicyLimitReachedException”.
   2. Maturity amount should be calculated as follows
      1. Policy type = Life Insurance = 10% interest on total premium
      2. Policy type = Health Insurance
         1. 10 years tenure –8% of total premium amount
         2. 10+ years tenure – 9% of total premium amount

**Stage: Unit Testing**

1. Create a new Unit test project to test the service classes created in business logic layers
2. Mock all the repositories using a mocking framework.

**Stage: Micro-service implementation**

1. Create a API project which references the business logic layer created earlier
2. Implement service documentation using swagger
3. All exceptions in the micro-service must be handled and logged using a logging library
4. Create the following end-points and test them using postman and export the requests into a json file.

Table 7 : Policy Plans - Endpoint - 1

|  |  |
| --- | --- |
| **URL** | /api/policy/types |
| **Request Type** | GET |
| **User Role** | Policy managers |
| **Trigger** | Front end |
| **Description** | Bank managers will use this endpoint to specify the type of policy they are adding in the system |
| **Inputs** |  |
| **Outputs** | PolicyTypeDTOs |

Table 8 : Policy Plans - Endpoint - 2

|  |  |
| --- | --- |
| **URL** | /api/policy |
| **Request Type** | POST |
| **User Role** | Policy managers |
| **Trigger** | Front end |
| **Description** | This endpoint is used to add a new policy in the system |
| **Inputs** | PolicyDTO |
| **Outputs** | Status code |

Table 9 : Policy Plans - Endpoint - 3

|  |  |
| --- | --- |
| **URL** | /api/policy/<searchcriteria>/<value> |
| **Request Type** | GET |
| **User Role** | Customers |
| **Trigger** | Front end |
| **Description** | This endpoint will help the customers to search for policies available in the system |
| **Inputs** | SearchCriteria, SearchValue |
| **Outputs** | PolicyDTOs |

Table 10 : Policy Plans - Endpoint - 4

|  |  |
| --- | --- |
| **URL** | /api/policy/<policyid> |
| **Request Type** | GET |
| **User Role** | Customers |
| **Trigger** | Front end |
| **Description** | This endpoint will allow the customer to view the details for a particular policy |
| **Inputs** | PolicyID |
| **Outputs** | PolicyDTO |

Table 11 : Policy Plans - Endpoint - 5

|  |  |
| --- | --- |
| **URL** | /api/policy/<policyid>/subscribe |
| **Request Type** | POST |
| **User Role** | Customers |
| **Trigger** | Front end |
| **Description** | Customers will use this endpoint to add a subscription to the policy |
| **Inputs** | PolicyID, SubscriptionDTO |
| **Outputs** | Status code |

**Stage: Font-end design**

Create the following components as per the specification provided below.

1. NewPolicyFormComponent
   1. Create a form component which can be navigated to from the navigation bar by the policy managers.
   2. The component should contain a form to accept the policy details.
   3. Use a dropdown to select the policy type.
   4. Once all details are validated and successfully submitted an acknowledgement must be displayed.
2. SearchPoliciesComponent
   1. Create a search policy component component which is accessible to policy managers and customers by navigation from application menu.
   2. Component should provide a dropdown list to choose the search criteria like tensure, maturity amount and premium amount.
   3. It should also provide a textbox to enter value for selected search criteria and a button to search
   4. Each policy should be displayed in bootstrap card style with a subscribe button.
   5. Upon clicking the button the users should be navigate to Subscribe policy component by passing policy id as route parameter
3. SubscribePolicyComponent
   1. Develop a component for customer which can be used to subscribe to the policy
   2. The component should display all details of the policy selected earlier using the policy id from route parameters
   3. It should also provide a form to accept the subscription details
   4. Upon successful, submission of form an acknowledgement should be displayed to the user.

**Stage: Integration of Frontend and backend**

1. Create a data service in the font-end application which will communicate with the micro services.
2. Use the data service in the components to make them interact with the API
3. Valid error messages should be shown based on various response status codes received form the API

### **Module – Premium Management**

The policy holder and policy managers can work with the premium amount of the policies subscribed by them. The module allows the following operations

1. The customers can pay the montly premium for their policies
2. Customers can also view the history of their premium payments
3. Policy managers can view the due premium for the current month

**Stage: Database Implementation**

1. Design a data base as per the following ER diagram provided.

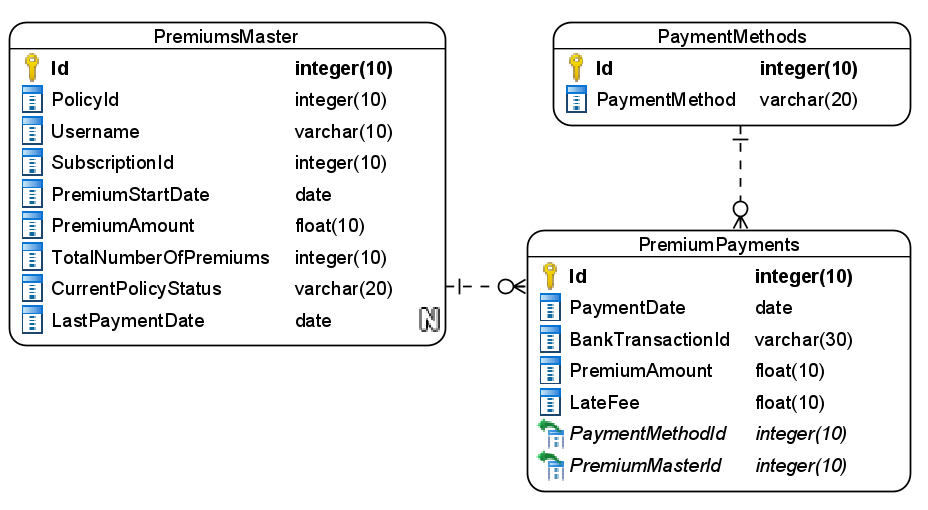


Figure 5 : ER Diagram - Premium Management

1. Apart from primary and foreing keys implement the following additional constraints
   1. Username should be exactly 10 characters long
   2. Allowed values for current policy status are – Ongoing, Matured, Defaulted, Cancelled
   3. Premium start date should not be a past date
   4. Payment date must be taken by default as today if not provided

Note: Seed data into the payment methods table as Card/NetBanking/UPI and PremiumMaster table with few records

**Stage: Data Access Layer Design**

1. Create a library project and add ORM support into it.
2. Use the ORM to map the entities to database as per the ER diagram provided.
3. Use repository per entity pattern and generate the repositories to perform the following operations
   1. Return list of payment methods
   2. Insert new premium payment
   3. Return list of premium payments for subscription
   4. Return list of due payments

**Stage: Business Logic Layer Development**

1. Develop a library which reference the Data Access Library project created earlier
2. This class library will contain various service classes which will encapsulate the business logic for the application.
3. Use dependency injection to in service classes to inject the required repositories.
4. Create the service classes following the single responsibility principle which perform the given operations as follows
   1. Get payment methods list
   2. Add new premium payment
   3. Get premium history for subscription
   4. Get due payments list
   5. Calculate late fee
5. Following business rules must be implemented as part of the business service class
   1. Total number of premium payments by the customer for a policy should not exceed the number of premiums mentiond in the Premium master. In-case of user paying more than the total number then raise a user-defined exception as “PolicyMaturedException”.
   2. In-case of late premium payments that is after 5th of the month, then a fine of 0.025% of premium amount should be levied per day.
   3. Once all the premiums are completed the current status of policy must be changed to Matured
   4. Any premium is considered due if it’s is not paid after 10th day of a month

**Stage: Unit Testing**

1. Create a new Unit test project to test the service classes created in business logic layers
2. Mock all the repositories using a mocking framework.

**Stage: Micro-service implementation**

1. Create a API project which references the business logic layer created earlier
2. Implement service documentation using swagger
3. All the exceptions must be handled and logged using a logging library.
4. Create the following end-points and test them using postman and export the requests into a json file.

Table 12 : Premium Management - End point - 1

|  |  |
| --- | --- |
| **URL** | /api/paymentmethods |
| **Request Type** | GET |
| **User Role** | Customers |
| **Trigger** | Front end |
| **Description** | Customers can choose one of the payments listed by this endpoint for paying premium |
| **Inputs** |  |
| **Outputs** | PaymentMethodDTOs |

Table 13 : Premium Management - End point - 2

|  |  |
| --- | --- |
| **URL** | /api/premium |
| **Request Type** | POST |
| **User Role** | Customers |
| **Trigger** | Front end |
| **Description** | The customers will be able to make a payment for the premium using this endpoint |
| **Inputs** | PremiumPaymentDTO |
| **Outputs** | Status code |

Table 14 : Premium Management - End point - 3

|  |  |
| --- | --- |
| **URL** | /api/premium/<username>/<subscriptionid>/history |
| **Request Type** | GET |
| **User Role** | Customers |
| **Trigger** | Front end |
| **Description** | Using this endpoint the customers can view the history of their payments towards a subscription |
| **Inputs** | SubscriptionId, Username |
| **Outputs** | PremiumPaymentDTOs |

Table 15 : Premium Management - End point - 4

|  |  |
| --- | --- |
| **URL** | /api/premium/dues |
| **Request Type** | GET |
| **User Role** | Policy managers |
| **Trigger** | Front end |
| **Description** | This endpoint will be responsible to display to the policy managers the information on premium dues for various policies |
| **Inputs** |  |
| **Outputs** | PolicyDTOs |

Table 16 : Premium Management - End point - 5

|  |  |
| --- | --- |
| **URL** | /api/premium/calculatelatefee |
| **Request Type** | GET |
| **User Role** | Customers |
| **Trigger** | Front end |
| **Description** | This endpoint will show the customers the late fee for a given premium due |
| **Inputs** | LateFeeCalcDTO |
| **Outputs** | Status code and LateFee amount |

**Stage: Font-end design**

Create the following components as per the specification provided below.

1. PremiumPaymentFormComponent
2. Develop a component to be used by customers which contains a form to make payment for a premium plan.
3. The payment method should be selected from a dropdown list
4. Once all the details are validated, user should be able to get an acknowledgement on submission of form.

1. PremiumHistoryComponent
2. Design a component which can be used by customers to view the details of their premium payments
3. The component should provide a textbox for accepting username and subscription id and search and display the premium payments data accordingly
4. Premium payment history for the policy should be displayed in a table
5. PremiumDueListComponent
6. Design a component which can be used by policy managers to view the details of due premium payments
7. Provide a navigation to the component from the application menu
8. Due history for the policies should be displayed in a table

**Stage: Integration of Frontend and backend**

1. Create a data service in the font-end application which will communicate with the micro-services.
2. Use the data service in the components to make them interact with the API
3. Valid error messages should be shown based on various response status codes received form the API

### **Module – Claims Management**

The policy owners will be able to work with the claims against their policies using this module. Following are the task this module supports

1. The customers can raise a new claim request
2. Policy managers will be able to view all the new claim requests
3. The policy managers will be able to process a claim request

**Stage: Database Implementation**

1. Design a data base as per the following ER diagram provided.

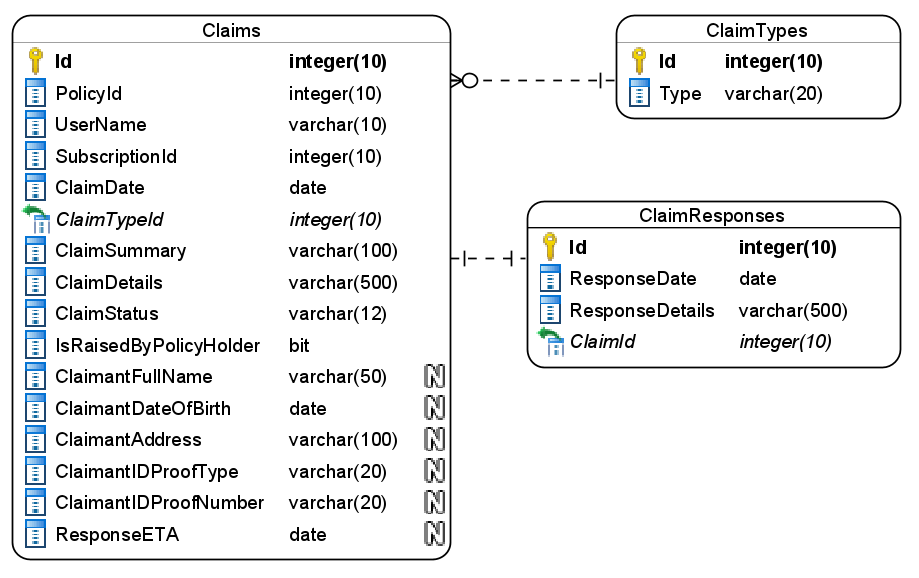


Figure 6 : ER Diagram – Claims Management

1. Enfore the following constraints along with primary and foreign keys
   1. ResponseETA date must be a future date
   2. ClaimDate must be taken as today by default
   3. ETAPaymentDate must be a future date
   4. Reponse should be atleast 50 characters long
   5. Allowed values for claim status are – New/Approved/Rejected
   6. ClaimantIDProofType should accept – Passport, Aadhar, PAN, DrivingLicence

Note: Pre-populate the Claim types table with types like Maturity Payout, Death, Premature Withdraw.

**Stage: Data Access Layer Design**

1. Create a library project and add ORM support into it.
2. Use the ORM to map the entities to database as per the ER diagram provided.
3. Use repository per entity pattern and generate the repositories to perform the following operations
   1. Return claim types list
   2. Insert a new claim request
   3. Return new claim requests list
   4. Return a claim request by id
   5. Insert a claim response

**Stage: Business Logic Layer Development**

1. Develop a library which reference the Data Access Library project created earlier
2. This class library will contain various service classes which will encapsulate the business logic for the application.
3. Use dependency injection to in service classes to inject the required repositories.
4. Create the service classes following the single responsibility principle which perform the given operations as follows
   1. Fetch all claim types list
   2. Add a new claim request
   3. Fetch new claim requests
   4. Fetch an claim request by ID
   5. Process claim request
5. Following business rules must be implemented as part of the business service class
   1. ResponseETA for claim must be generated based as follows
      1. Maturity Payout – 7 days from date of claim
      2. Death – 30 days from the claim date
      3. Premature Withdraw – 15 days from date of claim
   2. Once a claim request is accepted, user is not allowed to create more claim request for the given policy. If a user tries to create more than 1 claim request throw a user-defined exception as “PolicyAlreadyClaimedException”
   3. User can only have 1 new claim request per policy.
   4. Claims for Death cannot be raised by policy holder so the claimant details must be provided

**Stage: Unit Testing**

1. Create a new Unit test project to test the service classes created in business logic layers
2. Mock all the repositories using a mocking framework.

**Stage: Micro-service implementation**

1. Create a API project which references the business logic layer created earlier
2. Implement service documentation using swagger
3. Create the following end-points and test them using postman and export the requests into a json file.

Table 17 : Claims Management - End point - 1

|  |  |
| --- | --- |
| **URL** | /api/claims/types |
| **Request Type** | GET |
| **User Role** | Customers |
| **Trigger** | Front end |
| **Description** | The customers can choose the claim types from the list returned by the endpoint |
| **Inputs** |  |
| **Outputs** | ClaimTypeDTOs |

Table 18 : Claims Management - End point - 2

|  |  |
| --- | --- |
| **URL** | /api/claims |
| **Request Type** | POST |
| **User Role** | Customers |
| **Trigger** | Front end |
| **Description** | This endpoint will allow the customers to create a new claim request |
| **Inputs** | ClaimDTO |
| **Outputs** | Status code |

Table 19 : Claims Management - End point - 3

|  |  |
| --- | --- |
| **URL** | /api/claims |
| **Request Type** | GET |
| **User Role** | Front end |
| **Trigger** | Policy managers |
| **Description** | With this endpoint the policy managers can fetch details of newly created claim requests |
| **Inputs** |  |
| **Outputs** | ClaimDTOs |

Table 20 : Claims Management - End point - 4

|  |  |
| --- | --- |
| **URL** | /api/claims/<claimid> |
| **Request Type** | GET |
| **User Role** | Policy managers |
| **Trigger** | Front end |
| **Description** | The policy manager will use this endpoint to view details related to a claim request |
| **Inputs** | ClaimID |
| **Outputs** | ClaimDTO |

Table 21 : Claims Management - End point - 5

|  |  |
| --- | --- |
| **URL** | /api/claims/<claimid>/process |
| **Request Type** | PUT |
| **User Role** | Policy managers |
| **Trigger** | Front end |
| **Description** | The responsibility of this endpoint is to procces a claim request by providing a response |
| **Inputs** | ClaimID |
| **Outputs** | ResponseDTO |

**Stage: Font-end design**

Create the following components as per the specification provided below.

1. NewClaimRequestFormComponent
2. Create a component with a form and allow the navigation to it for customers
3. The component should allow the user to select the claim type from a dropdown list.
4. Validate all the data before it’s submitted
5. On successful submission of request display an acknowledgement.
6. ClaimListComponent
7. Create a component which is accessible to policy managers
8. The component should display the claim requests in a tabular formats using bootstrap tables
9. Each request should have “Process” button which should navigate to Process Claim Component component by passing the claim id through route parameters
10. ProcessClaimComponent
11. Design a new component for policy managers to process the claim requests.
12. The component must display the claim details along with accept/reject buttons.
13. The policy manager should also provide a proper response when processing the clainm request.
14. Once the details are updated an acknowledgement should be displayed.

**Stage: Integration of Frontend and backend**

1. Create a data service in the font-end application which will communicate with the micro-services.
2. Use the data service in the components to make them interact with the API
3. Valid error messages should be shown based on various response status codes received form the API

# Deployment requirements

1. All the Microservices must be deployed on a local web server like IIS or Apache Tomcat
2. All the Microservices must be independently deployable.
3. These services must be consumed from an Front end app running in a local environment.

# Design Considerations

Java and Dotnet specific design considerations are attached here. These design specifications, technology features have to be strictly adhered to.



Refer this link for the coding standards. <https://cognizantonline.sharepoint.com/:w:/r/sites/GTP-Solutions/Gencsharepath/Shared%20Documents/Internship2020/FSE/Coding%20standards/Effective%20coding%20standards.docx?d=w6430574d9db5478bbbe37c25b16e68e2&csf=1&web=1&e=84lTVf>

|  |  |
| --- | --- |
| **Category** | **Rule** |
| Database | Table names in database must be pascal cased and plural. All primary keys must be named as Pk\_<table>. All foreign keys must be named as FK\_<PrimaryKeyTable>\_<ForeignKeyTable> |
| Database | Column names must be pascal cased. Multi-word column must be split using \_ (underscore) |
| Coding | Follow pascal casing for naming classes, interfaces, methods, properties and other public members |
| Coding | Use camel casing for method parameter name, backing fields for properties and private variables. Consts must be capitalized |
| Coding | All exceptions must be handled and logged using a logging library |
| Coding | For communication between micro-services use the HttpClient class available in .Net and Java |
| Unit testing | Each method in services classes in business logic must be unit tested using nUnit/jUnit |
| Unit testing | Use a mocking library to mock the repositories while performing tests for business logic layer |
| Code Coverage | Should be minimum 90% |
| Front end(Angular/React ONLY) | Use pascal casing for the component names |
| Front end(Angular/React ONLY) | Create all components and data services in Angular/React project in dedicated folders |
| GitHub | Create ONLY Private Repositories.  No password should be stored.  DO NOT Mention in the Profile that You work for Cognizant |

# Reference learning

Please go through all of these k-point videos for

Microservices deployment into Azure Kubernetes Service.

|  |
| --- |
| [AzureWithCICD-1](https://cognizant.kpoint.com/app/video/gcc-19532393-d4e0-4fd9-8a0c-80ecbdb349d3) |
| [AzureWithCICD-2](https://cognizant.kpoint.com/app/video/gcc-6633a958-ab72-4c69-b926-fe832e4b56a1) |
| [AzureWithCICD-3](https://cognizant.kpoint.com/app/video/gcc-553eb186-c1cf-448e-96fc-a96fe37b2e6a) |
| [AzureWithCICD-4](https://cognizant.kpoint.com/app/video/gcc-fad7d4af-d651-4501-99c6-2785190670c2) |

**Other References:**

|  |  |
| --- | --- |
| Java 8 Parallel Programming | <https://dzone.com/articles/parallel-and-asynchronous-programming-in-java-8> |
| Feign client | [https://dzone.com/articles/Microservices-communication-feign-as-rest-client](https://dzone.com/articles/microservices-communication-feign-as-rest-client) |
| Swagger (Optional) | [https://dzone.com/articles/centralized-documentation-in-Microservice-spring-b](https://dzone.com/articles/centralized-documentation-in-microservice-spring-b) |
| ECL Emma Code Coverage | <https://www.eclipse.org/community/eclipse_newsletter/2015/august/article1.php> |
| Lombok Logging | <https://javabydeveloper.com/lombok-slf4j-examples/> |
| Spring Security | <https://dzone.com/articles/spring-boot-security-json-web-tokenjwt-hello-world> |
| H2 In-memory Database | <https://dzone.com/articles/spring-data-jpa-with-an-embedded-database-and-spring-boot>  <https://www.baeldung.com/spring-boot-h2-database> |
| AppInsights logging | <https://www.codeproject.com/Tips/1044948/Logging-with-ApplicationInsights> |
| Error response in WebApi | <https://stackoverflow.com/questions/10732644/best-practice-to-return-errors-in-asp-net-web-api> |
| Read content from CSV | <https://stackoverflow.com/questions/26790477/read-csv-to-list-of-objects> |
| Access app settings key from appSettings.json in .Net core application | <https://www.c-sharpcorner.com/article/reading-values-from-appsettings-json-in-asp-net-core/>  <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/?view=aspnetcore-3.1> |

# Project Templates









# Change Log

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Changes Made | | | |
| V1.0.0 | Initial baseline created on 11-November-2022 by Khaleelullah Hussaini Syed | | | |
|  |  | | | |
| **Section No.** | **Changed By** | **Effective Date** | **Changes Effected** |
|  |  |  |  |
|  |  |