



Version 1.1

**YCompany eClaims**

High Level Design Document

Contents

[1 Introduction 3](#_Toc13497859)

[1.1 Purpose of Document 3](#_Toc13497860)

[1.2 Scope of Document 3](#_Toc13497861)

[2 System Overview 4](#_Toc13497862)

[2.1 Project Overview 4](#_Toc13497863)

[2.2 Functional Zones 5](#_Toc13497864)

[2.2.1 Frontend Applications 5](#_Toc13497865)

[2.2.2 Backend Services 6](#_Toc13497866)

[3 Design Considerations 12](#_Toc13497867)

[3.1 Assumptions and Dependencies 12](#_Toc13497868)

[3.2 General Constraints 12](#_Toc13497869)

[3.2.1 Availability 12](#_Toc13497870)

[3.2.2 Performance 12](#_Toc13497871)

[3.2.3 Thin Client 12](#_Toc13497872)

[3.2.4 Database considerations 12](#_Toc13497873)

[3.3 Goals and Guidelines 13](#_Toc13497874)

[3.3.1 Response Time 13](#_Toc13497875)

[3.3.2 Coding Guidelines 13](#_Toc13497876)

[4 NFR Considerations 15](#_Toc13497877)

[4.1 Usability 15](#_Toc13497878)

[4.2 Reliability 16](#_Toc13497879)

[4.3 Performance 17](#_Toc13497880)

[4.4 Security 18](#_Toc13497881)

[4.5 Supportability 18](#_Toc13497882)

[4.6 Infrastructure Requirements 19](#_Toc13497883)

[4.7 Implementation Constraints 20](#_Toc13497884)

[5 Solution Architecture 22](#_Toc13497885)

[5.1 System Design 22](#_Toc13497886)

[5.2 Information Flow 22](#_Toc13497887)

[5.3 Architecture Diagram 23](#_Toc13497888)

[5.3.1 Frontend Applications 23](#_Toc13497889)

[5.3.2 Business Services 24](#_Toc13497890)

[6 Framework/Infrastructural Components 28](#_Toc13497891)

[6.1 Logging 28](#_Toc13497892)

[6.2 Kafka Stream Flow 29](#_Toc13497893)

[6.3 Security 29](#_Toc13497894)

[6.4 Exception Handling 31](#_Toc13497895)

[6.5 Input Validation 32](#_Toc13497896)

[6.6 Caching 32](#_Toc13497897)

[6.7 Dependency Injection 33](#_Toc13497898)

[6.8 Data Access 33](#_Toc13497899)

[6.9 Data Transfer between Layers 33](#_Toc13497900)

[6.10 Test Cases 34](#_Toc13497901)

[6.11 Client-Side Frameworks 35](#_Toc13497902)

[7 Integration (External Systems) Architecture 36](#_Toc13497903)

[7.1 Document Store integration via Document Service 36](#_Toc13497904)

[7.2 External money transfer integration via Transaction Service 36](#_Toc13497905)

[7.3 SMS & Email notifications via Notification Service 36](#_Toc13497906)

[8 Deployment Architecture 37](#_Toc13497907)

[8.1 Deployment Components 37](#_Toc13497908)

[8.2 Deployment Diagram - Services 39](#_Toc13497909)

[8.3 Deployment Environments 39](#_Toc13497910)

[9 Glossary 41](#_Toc13497911)

[9.1 Requirements Traceability Matrix 41](#_Toc13497912)

[9.2 Technology POCs 41](#_Toc13497913)

[9.2.1 eClaim Incident Reporting 41](#_Toc13497914)

[9.2.2 Kubernetes based service orchestration 41](#_Toc13497915)

# Introduction

“YCompany” is one of the renowned name in Insurance providers in industry providing auto insurance to customers in US. The company's product offerings include term and life insurance. Company servers more than 200 million customers across various geographies has been facing challenges with the manual process currently in place

The company is planning to modernize claims processing part of the application to all its users Customers, Partner and Internal company users. The system needs to be fast, highly interactive and user friendly to achieve following goals:

1. Highly interactive with user friendliness portal so that customer can report an accident to start the claim process, pay their dues online by themselves to avoid manual interactions.
2. Online portal to its partner will enable a hassle free medium to report status of claims easily and efficiently which also gain the trust of customers and partner both.
3. At the same time access to internal portal will allow internal user like surveyor, adjustor and claim manager to manage the whole process smoothly and avoiding manual intervention.
4. A very flexible & easily maintainable system is required to meet ever increasing pace of requirement, regulatory, statutory compliances.

## Purpose of Document

This document provides the high-level design of the envisioned Insurance Claim Solutions with eClaims Facility for YCompany. Throughout this documents details will be listed about various design considerations, application architecture, non-functional requirements, various components of the system. This document is meant to be the basis of development of the proposed system.

## Scope of Document

This document gives the high-level design for the envisioned eClaims System.

The HLD specification will include the following:

* System Overview
* High level design considerations
* Non-functional requirements-based design considerations
* High level solution architecture
* Component description
* Deployment architecture and considerations
* Integration architecture
* References & research details

The design discussed in this document can be considered final in most aspects unless otherwise mentioned. However, considering the evolving nature of development, design and implementation may change to small extent.

# System Overview

## Project Overview

The proposed eClaims System will be designed to fulfil following requirements in three parts:

* eClaim online portal available on customers
  + Registration
    - Signup via third party
    - Signup on portal
  + Claims Initiation
    - Incident detail
    - Supporting documents
    - Status tracking a query posting
  + Payments
    - Online payment
  + Downloads
    - Invoices
    - Policy
    - Statements
  + Appointment
  + Feedback
    - Feedback via portal
    - Twitter
    - Facebook
    - WhatsApp
* eClaim internal portal for company users
  + Surveying
    - Inspection
    - Data submission
    - Image uploads
  + Claim processing
    - View and update details
    - Query posting
    - Adjustment
    - Upload document
  + Approval and Rejection
    - Rejection
    - Adjustments
    - Query posting
    - Signing
    - Reports
* eClaim internal portal for partners
  + Work order
    - Initiate
    - Update
  + Incident
    - Detail view
    - Inclusion and exclusion
    - Query to insurer
  + Billing
    - Invoice
    - Payment tracking
  + Downloads
    - Statements
    - Work orders

## Functional Zones

Based on the requirement, the overall functioning of the system can be divided in two zones. The frontend applications, backend services.

### Frontend Applications

There will be several frontend applications for system interaction, management and administration purposes. The applications with their brief summary are listed below:

Frontend Applications

**Customer Portal:** Will be a single page web application accessible over internet where insurance customers can login and initiate claim process etc. The portal will be available 24x7 and would provide nice and easy user interface to perform various tasks.

**Partner Portal:** Will be a single page web application accessible from various partner of the insurance company where partner employees can login and perform various activities like creating a work order, updating the status, payment tracking etc. The portal will be available 24x7 and would provide nice and easy user interface to perform various tasks.

**Internal Portal:** Will be a single page web application accessible from internal user of the insurance company where its employees can login and perform various activities like filling inspection detail, uploading supporting images, adjusting amount, approving or rejecting amount etc.

**Kubernetes Web UI:** Will be used by DevOps admins to monitor application server clusters that are running the app services in containerized environment (pods). This is also a readymade dashboard provided by Kubernetes that require minimal configuration for setting up access. It also helps in monitoring health & state of clusters in real time, as well can provide visual insight into scaling levels of various services.

### Backend Services

In order to implement a more robust, efficient & reliable system, the business logic behind the whole eClaim systems are envisioned in the form of API services that would follow microservice architecture. Moreover, considering the requirement for huge expansion in future, it is necessary to break down the backend business processing in individual highly scalable & partitionable chunks to cater to a huge user base. With these considerations, the system can be broken down into following subsystems:

* 1. Customer Services
  2. Insurance Claims Services
  3. Partner Services
  4. Document Services
  5. Insurance Product Services

This distribution of servies into groups is described below.

#### Customer Services

Customer services will have set of APIs that would handle most operations being performed by customers such as Incident reporting, claim start, uploading supportive documents and payment etc. The APIs in Customer services space are described below:

Customer Services

* + **Customer Data API:** Provide methods to the customers to get customer information registered with company.
  + **Incident API:** This API will allow customer to report details of incident happened with them. Here some details such as location, dates and time customer policy number etc can be sent via this api.
  + **Claim API:** This API will allow customer to start with claim process at first place. Here with the exposed end point customer can send all details about claim data such as policy number , incident details, nominee, canceld cheque etc.
  + **Payment API:** This API will allow customer to pay their dues to company partner and also in case customer is willing to a policy online.
  + **Policy API:** Policy API provides methods to get policy detail for a provided customer. Customer can view his policy detail any time via customer portal.
  + **Document API:** Provides concrete implementation for implementing a document upload workflow. Endpoints exposed will allow to accept documents uploaded by customers reporting an incident and while uploading supprting documents with respect to their claim.
  + **Tracking API:**This api will provide tracking a cliam feature to customer.

#### Insurance Claim Services

Branch Services are the set of services that enables insurance branch operations. E.g. policy opening, approval, Inspection, Billing etc. The APIs are discussed below:

Branch Services

* + **Auth API:** It will be responsible for authenicating and authorizing internal users. This api has been chooses as different from customer and partner auth process because it will more connect with internal user role access policy etc.
  + **Profile API :** will expose endpoints for internal user info, validating their credentials, source of truth for their roles & access levels, will also store ruleset for endpoints invocation mappled to various roles & access levels. It will store all the information at rest around who can perform what & the same can be quried with the exposed endpoints from this service.
  + **PartnerAPI:** This API will provide methods to register a partner which will be a different process as compared to customer registration hence it is choosen a different api al together.
  + **Products API :** This api will provide all avaiable insurance products. This api will be more detailed one as compared to the common api as it gives all detaile data of product which any internal user may want while dealing with claims agains a policy availed by customer.
  + **Policy API :** this api will provide methods to create , update and read plocies for a customer.
  + **Claim API:** This is major claim api as compared to customer claim api as it will provide more interface to internal user to deal with customer claims. Actions such as adjusting amount, interncommunication with other apis via common Bridge API will happen under the same.
  + **CustomerData API:** It will consist customer detail data to help advisor to check the saame at the time he/she evaluates claim requested by a customer. Detail data may conatisn personal information, previous claim occurred and dues pending from last claim etc.
  + **Survey API:** This api will dedicately be designed for surveor to send the survey report while accessing customer about damage and claim.
  + **Payment API:** This API will allow user to pay requested amount y partner against a
  + **Transactions API:** This API will allow user to check their payments statements agains each transaction happened in past and can also download Invoices.
  + **Document API:** This API will have methods to upload documents by user agains a claim or plocy generation documents along with other customer document which may be provided in branch by them.
  + **Report API:** This API will consist methods to get data from different data sources to generate reports which would in the interest of Adjuster, Claim Manager or higer authority.
  + **Message API:** This API provide methods to give the response agains a query made by parner or customer which is very likely to happpen when a follow is going on unde claim processing.

#### Partner Services

Branch Services are the set of services that enables insurance partner operations. E.g. work order, Payment, Billing etc. The APIs are discussed below.

Partner Services

* + **Work Order API** :This API will alow partner to create a work order against a claim made by customer. This api will communicate common Bridge API to read customer claim/ incident relavant data to proceed with work order
  + **Document API** :This API will allow partner to upload all relvant documents such as invoices, images of progress etc.
  + **Billing API** :This API will provide methods to generate a bill against work order with respect to a claim made by customer.
  + **Report API** :This will majorly allow partner to view and download reports of their interset such as invoices, number of claims per quartert/half year etc..
  + **Query API** :This API will be designed for partner to ask any query to the comapny against a claim.
  + **Incident API** :This API will be designed for partner to get the detail of incident.

## Mobile Application

There would also be a mobile application built for IOS/Android which would allow customers and surveyors to see their policy and claims and report the incident , also the inspection reports to back office.

The mobile application would be built using Angular with Ionic framework.

# Design Considerations

There are many design considerations that defines the overall usability, efficiency, availability, performance of the system.

## Assumptions and Dependencies

* Data Location – Data can live outside the country boundary but will be encrypted at rest and in transit.
* No payment gateways already exist in the company and cost for payment gateway services will be considered as part the system.
* Document store – archiving of documents to a low cost or infrequent access store is not in consideration.
* It is assumed that internal user management system already exists and not part of eClaim system.
* Active directory server is available on premises, employee list integration needs to be done with this existing active directory.
* Insurance product creation is not part of system and it is assumed that customer have a valid policy against which he/she willing to claim though policy generation API has been considered
* It is assumed that agreement has already been done for paying cloud services for customer and partner portal deployment. And internal user portal will be deployed on premise.
* It is assumed that no data migration will be performed by the system and it is out of scope

## General Constraints

### Availability

The system must be available 24x7 accept scheduled maintenance time if any.

### Performance

The system is required to be reasonably performing under peak load. Our target would be to deliver a system which performs better than the performance criteria defined in NFRs.

### Thin Client

The system would work based on thin client design. So, eCalim portal would be developed as a thin client web application, thereby not requiring special software /hardware to run the system. Client machines should just be installed with browser.

### Database considerations

Following general guideline are applicable when designing a database clusters & database:

* Primary keys will be of GUID type.
* Credentials will be stored encrypted.
* Database will be designed to be in 3NF except for
  + Special optimization scenarios
  + Decoupling is required while following microservices architecture
* Customer Claims and partners data will be stored in multiple micro application to gain the performance.
* Read replicas will be created for the databases instances and will be dedicatedly used for certain use cases.

## Goals and Guidelines

### Response Time

Expected response time and peak load conditions are defined in NFR. To achieve the desired level of response time, the hardware must itself be scaled appropriately. The exact specification and configuration of the hardware will be communicated after initial load testing and optimization.

### Coding Guidelines

* Coding guidelines will be followed as per Nagarro coding guideline standards.
* For server-side coding, quality of code will be tested by visual studio code analysis and sonar code analysis which is widely accepted code analytics tool. The target is to achieve zero code analysis errors for these rules. Refer below image for sonar setup.

A screenshot of a cell phone

Description automatically generated

* For client-side coding in Typescript, Angular frameworks provided linting mechanism will be used to identify issues with the code. A ruleset will be created to which the development team will adhere. The target is to achieve zero lint errors from this ruleset.
* Rest service architecture will be followed for the services. There will be no real session, but to boost performance some key information will be stored mapped with token to avoid unnecessary lookup.
* Unit test cases will be written to cover functional aspects not just to meet the code coverage percentage value, so a code coverage value of 85% is prescribed.

# NFR Considerations

NFR considerations for various fronts are mentioned below. Wherever there is a variation in NFR consideration for various components the name of the component will be mentioned.

## Usability

* Speed of Use
  + Is it particularly important that users be protected from making errors?
    - Yes
* What sort of input/output devices for the human interface are available and what are their characteristics?
  + - Keyboard, Mouse and Mobile
* Required User Ability
  + What type of user will be using the system?
    - Public and internal users can use the system.
    - Internal users can be staff members or admin.
  + Will more than one type of user be using the system?
    - Yes
* Learnability
  + Is it particularly important that the system be easy to learn?
    - Yes.
* Training Material
  + What sort of training will be required for each type of user?
    - eClaim Partner Portal: Classroom sessions for partner company staff.
    - eClaim Internal Portal: Classroom sessions for company staff.
    - eClaim Customer Portal: No training, the application should be easily understandable without any prior knowledge. Help pages and user guidance links will be provided to make internet user learn from there itself.
* Documentation
  + What kind of documentation is required?
    - eClaim Partner/Internal Portal: User manual for admin & staff members.
    - eClaim Customer Portal: Help Section on the screens.
  + What audience is to be addressed by each document?
    - eClaim Internal Portal: Internal users.
    - eClaim Partner Portal: Partner company internal users.
  + On-line Help
    - eClaim customer portal: in form of help sections
* On-line Help
  + What format of online help is required?
    - eClaim– Help sections
  + How extensive help needs to be – page wise or functionality wise?
    - eClaim– Functionality wise
  + Any customer support required in terms of call centre or chat?
    - Customer support is required but call centre and chat are not in scope
    - Online query page will be available to post in case of any help or information required.
* Consistency
  + What type of user-interface consistency in the system is required?
    - Application theme should be consistent.
    - Only the common controls should be used to implement user interface to ensure consistency across the application.
    - Help sections, terms & conditions should be placed consistently across the forms.
    - Action links/button should be disabled while an operation is being performed.
    - Event for saving state should be consistent e.g. while navigating to another screen it will be ensured that is saved prior to navigate
    - Validation presentation on client side will remain consistent throughout the system
  + Tabular interface.
    - Field labels adjacent to form fields.
    - Pagination should be consistent throughout the user interface.
* Language and Culture
  + What language is supported?
    - English (en-US)
* Accessibility Features
  + Will the system support differently abled users?
    - No

## Reliability

* Maximum Failure Rate
  + What is the acceptable % range of failure in delivering a request?
    - Less than 1%
* Maximum Down Time
  + What is the acceptable system downtime per 24-hour period?
    - Zero downtime other than scheduled maintenance.
* Ease of Recovery
  + How should the system respond to errors?
    - Log inform user and continue if possible.
  + Is there a maximum acceptable time for restarting the system after a failure?
    - No
  + How system should behave if the backend connection is unavailable?
    - Discard the operation
  + How code base should be recoverable?
    - Git as a source control strategy
    - Branch creation on each release
* Continuous integration
  + Is it important the system code should be automatically buildable and shippable?
    - Yes
  + How system code will be built and shipped?
    - Jenkins continuous integration and delivery
* Maximum known bugs
  + What is the acceptable number of known high severity bugs?
    - Zero High Severity Bugs for acceptance at sprint end
  + What is the acceptable number of known medium severity bugs?
    - <=5 Medium Severity bugs for acceptance at sprint end
  + What are the criteria of low severity?
    - <=10 low severity bugs for acceptance at sprint end.

## Performance

* Throughput
  + How many concurrent users may be using the system at any given peak-use time?
    - Customer Portal: 20,000 concurrent users
  + How many additional resource requirements (server, load balancers, etc.) are acceptable by the system to maintain acceptable latency and throughput with increasing load?
    - System should be able to scale up to tolerate increased load.
* Response Time
  + What is the acceptable response time range (in seconds) to a user action on the system?
    - Acceptable Response time depend on type of operation/size of response. Benchmarking is also done to while measuring response time in terms of number of iterations and the network bandwidth.
    - The request needs to be categorized among one of the following categories to map with acceptable response time, benchmarked at 5Mbps bandwidth & 5 iterations:
      * Landing Page: Less than 3 seconds.
      * Small to medium resource intensive operations: less than 3 seconds.
      * Highly resource intensive operations: less than 5 seconds.
      * Reports: less than 5 seconds, it may also be decided at the time when system is brought to data load testing.
* Resource Usage
  + Are there size or capacity constraints on the data to be processed by the system?
    - No size / capacity constraints
  + Are there any CPU usage constraints while the application is executing?
    - During normal load hours less than 50%
    - During peak load hours less than 75%
* Degradation Under Overload Conditions
  + How should the system respond to extreme conditions?
    - System should be able to scale to increased load conditions.

## Security

* Internal Security
  + Must access to any data or the system itself be controlled?
    - Yes
  + Are there any requirements to ensure the integrity of the system from accidental or malicious damage?
    - Yes
* External Security
  + Is physical security an issue?
    - Yes
  + Must all external communications between the system’s data server and clients be encrypted?
    - SSL encryption will be used for all communications.

## Supportability

* Ease of Installation
  + Is there any need for an automatic installation package? Or manual installation will suffice?
    - App services installation - Using containerized method via Kubernates.
    - Front end portal installation: Using containerized method via Kubernates.
    - Database server management: update scripts: will be manual
  + Who is responsible for system installation?
    - Developing Organization
  + How many installations of the system will be required?
    - Installation will be repeatable
    - Deployment should gracefully switch from an old version to new version ensuring availability
* Planned Maintenance
  + What would be the frequency of system maintenance?
    - To be defined by client.
  + Who will be responsible for system maintenance?
    - Developing organization.
* Backup
  + How often will the system be backed up?
    - To be defined by client.
  + Who will be responsible for the back up?
    - Automated cloud services
    - On premise cloud department
* Ease of Configuration
  + What type of configuration requirements of the system?
    - Database connection string
    - Shard based configuration
    - SMTP sender email profile credentials
    - SMS service credential configuration
    - Document repository path configuration
    - Initial admin configuration in the databases.
  + Would configuration items specified in application configuration file be acceptable?
    - Yes
* Ease of Testing
  + Is part of system needs to be testable via unit test cases?
    - Yes
  + Is system needs to be testable via automated test cases??
    - Yes
  + What level of user acceptance testing is required?
    - UAT
  + Are there any requirements to load test the system?
    - Yes
  + What level of feature isolation/factoring is required from testability perspective?
    - Feature isolation is required on use case level.

## Infrastructure Requirements

* Clients
  + What are the client system hardware, memory and browser requirements?
    - Internet Explorer 9+/Firefox/Chrome/Safari browser/ Edge browser.
* Servers
  + What hardware is the proposed server system to be used on?
    - Azure virtual machines
    - On premise windows server installed with docker and Kubernetes services
  + What are the characteristics of the target hardware, including memory size and storage space?
    - Specifications will be provided after initial load testing and optimization.
* Networks
  + What is the connection bandwidth required connecting Web Server and Application Server?
    - Specifications will be provided after initial load testing and optimization.
* Web Services
  + What are the external web services on which system will depend on?
    - Payment gateway services decided by client
    - File storage services
    - Third party payment hubs
    - SMS notification services
    - Email notification services
* Environment
  + Where will the target equipment operate?
    - Internet and Intranet
  + Will the target equipment be in one or several locations?
    - Single region, multiple availability zones, but for Disaster Recovery Multiple region
  + Will the environmental conditions in any way be out of the ordinary (for example, unusual temperatures, vibration, and magnetic fields)?
    - N/A

## Implementation Constraints

* Development Platform
  + Is there any specific requirement that the system should be implemented on a platform such as .NET 3.5?
    - .Net Core, MY SQL, Visual studio 2019 IDE, Visual Studio Code, Mongo Db, Kafka, Docker, Kubernetes, Azure
* Languages
  + Is there any specific requirement that system should be implemented in a programming language?
    - C#, Typescript, SQL
* Operating Systems
  + Is it important that the system be portable (able to move to different hardware or operating system environments)?
    - Yes
  + What Operating systems application should be able to execute on?
    - Platform independent
* Standards
  + Are there any specific implementation standards that need to be followed during development?
    - Microsoft Standard Coding Guidelines
    - Visual Studio and Sonar Code Analysis
    - Typescript coding guidelines through TS Lint
* System Interfaces
  + How many different types of interfaces (presentation layers) required for the system?
    - Web interface and Mobile interface
  + Is input coming from systems outside the proposed system?
    - Yes, Third party payment hubs
  + Is output going to systems outside the proposed system?
    - Yes, Third party payment hubs
    - SMS Service
    - Mail Services
  + Are there restrictions on the format or medium that must be used for input or output?
    - No
* Legacy Systems
  + Is there any type of interaction required with any existing legacy system?
    - No.
* Databases
  + What database implementation application needs to support?
    - MY SQL
    - Mongo Db
  + Is support for more than one database required?
    - Yes
  + Will there be any database mirroring required?
    - Mirroring - required.

# Solution Architecture

The envisioned solution is composed of client-side applications, API gateway, services relying on transactions databases, document databases, Radis cache, Kafka log streams etc. for implementing various business requirement.

Various aspects of solution architecture are explained in following subsections.

## System Design

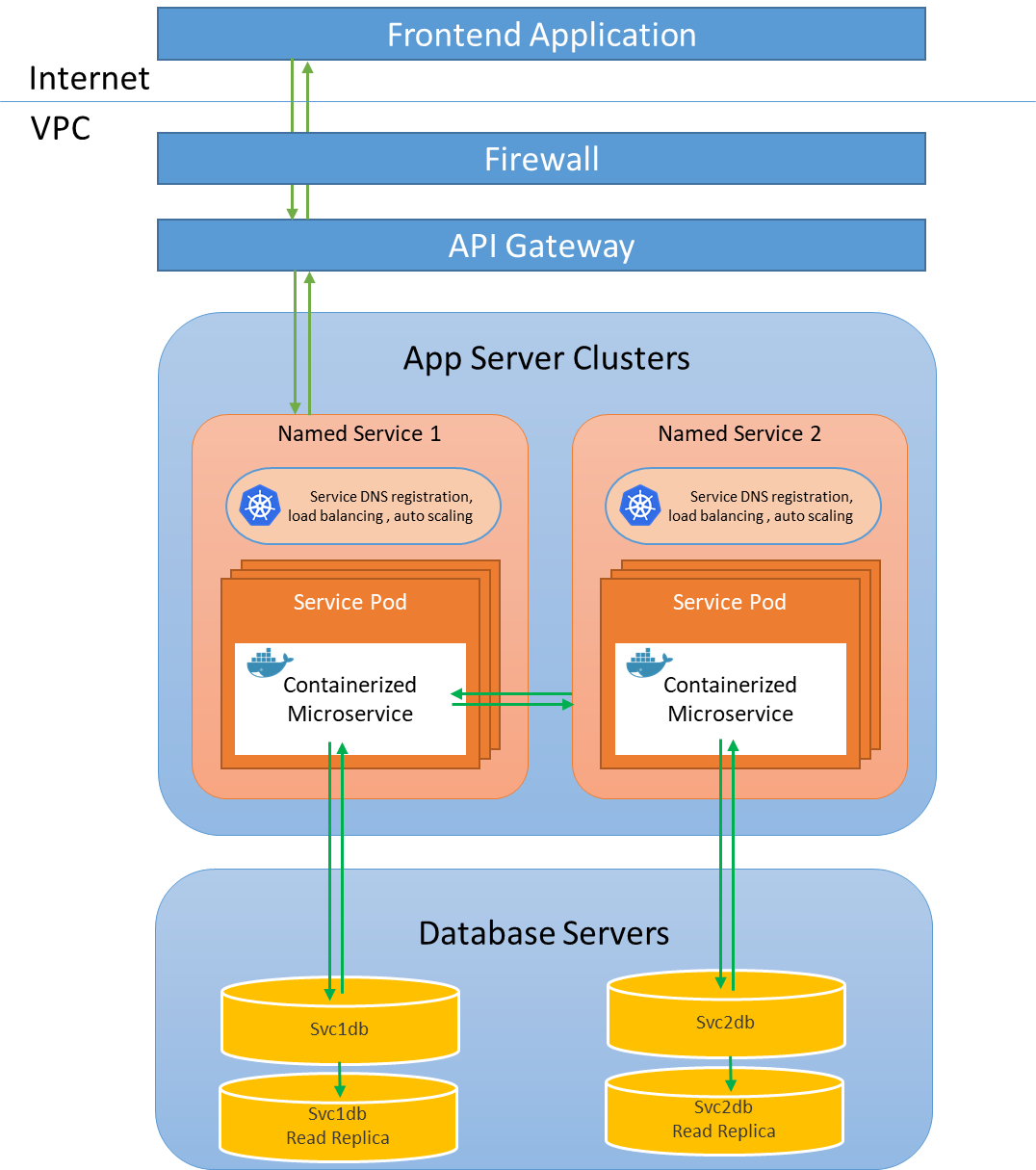
The overall system design is shown in the system design diagram below:Graphical user interface, application

Description automatically generated

System Design Diagram

## Information Flow

The diagram below depicts flow of information across core physical components of the system. Though it doesn’t show communication of all the components but depicts how the system scale & how service resolution is being done. The frontend portals are static html, CSS, JavaScript based applications, accessible over internet. Backend applications are platform agnostic microservices that run on docker containers. Many aspects such as service naming (DNS based service access), autoscaling (to meet increased load on system), load balancing (distributing traffic across service instances) is handled by Kubernetes master. All the services can talk to each other on internal network.



Information flow in service – Simplified view

## Architecture Diagram

### Frontend Applications

The presentation layer contains the client-side applications – Customer Portal, Partner Portal and Internal Portal. These applications will be a Single Page Application, built in Angular Framework. The architecture for the client-side application is shown in the diagram below:

A screenshot of a cell phone

Description automatically generated

Client-side application architecture

Typescript will be used as programming language to write client-side logic that would run on browsers. TypeScript is a [free and open source](https://en.wikipedia.org/wiki/Free_and_open_source) programming language developed and maintained by [Microsoft](https://en.wikipedia.org/wiki/Microsoft).

Angular framework will be used on client side to handle view rendering, two-way data binding, event handling, creating custom UI controls, layout. Angular is a structural framework for dynamic web apps and hybrid apps. It will allow us to use HTML as template language and lets us extend HTML's syntax to express the application's components clearly and succinctly. Angular data binding and dependency injection helps in writing independent components that can be used throughout the application.

### Business Services

The diagram below shows the core design of the services. In the diagram show how a service communicate to the other layer or systems.

A screenshot of a cell phone

Description automatically generated

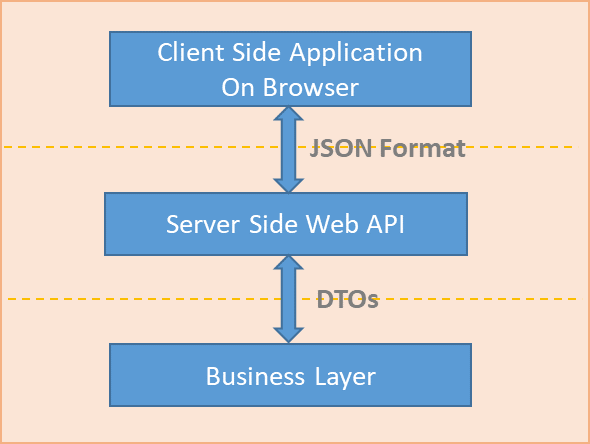
Service architecture (shows example of Internal Portal)

#### Service Components Description

##### API Layer

The API layer will contain a controller classes defining rest-full endpoints exposed from the service. All client-server data communication will be done through these restful http interfaces exposed by the Web API on the server. This application will be entertaining requests from the client in http verbs GET, POST, PUT, DELETE etc. The data format for communication between the client-side application and the server-side REST API will be JSON.

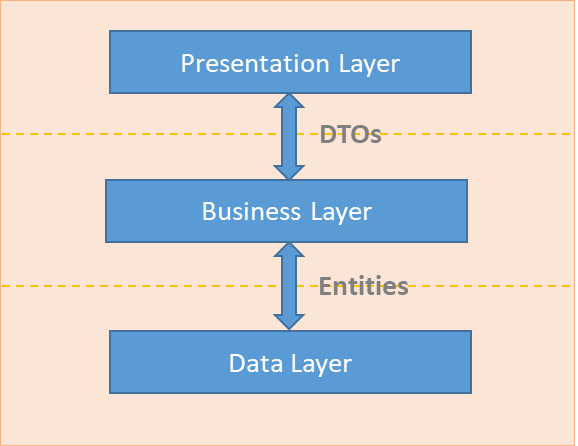
Web APIs will be implemented using Microsoft’s ASP.NET Core Web Application project template. .NET Core is a platform agnostic framework that makes it easy to build HTTP services that reach a broad range of clients, including browsers and mobile devices; and is an ideal platform for building RESTful applications.



Data communication to and from Web API layer

##### Business Layer

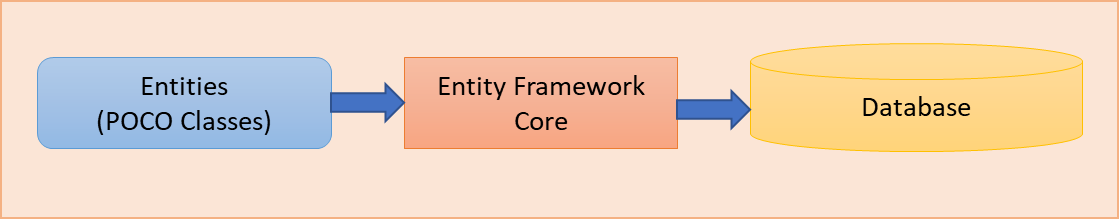
All business layer projects will class libraries projects in .NET Core framework, will expose their API through interfaces, and any inbound/outbound data communication with the upper layers will be done using data transfer objects (DTOs).



Data communication to and from business layer

##### Data Layer

The data layers will be implemented by .NET Core framework class library project templates. It would be responsible for implementing Entity Framework Core as ORM for handling database communication. For quicker and efficient database design Code-First approach will be used. With the Code-First approach, focus will be on domain driven design and models will be created as per domain requirement. Code-First implementation is used to create/update the database on the fly based on entity classes and configuration, resulting into much faster development.



Code first database generation

The Entities will be available in the domain project inside the cross-cutting concerns layer. Entities will be placed there so that they can be consumed by the business layer class libraries.

##### Cross Cutting Concerns

All the cross-cutting concerns will be part of this layers. This will include data structures, logging, exception handling, model mapping, interfaces. This layer will be containing two projects in general per micro application other than that global assemblies shared by all the micro applications. These two projects will contain artifacts required just by that microservice. More information of each of these projects is given below:

###### Domain Project

The domain project will contain all the data structures that will be consumed by various layers in the services. This would include Data Transfer Objects (DTOs), Domain Entities (representing database), configurations that specifies how a DTO map with an Entity & vice-versa.

###### Shared Layer

The Shared project contains Logging framework, Custom Exceptions, Interfaces that will be exposed by various layers, constants and other utilities.

# Technology /Framework/Infrastructural Components

## Technology

To develop the complete solution based on design consideration, below technology stack will be chosen.

Frontend- Angular

Backend - .NetCore

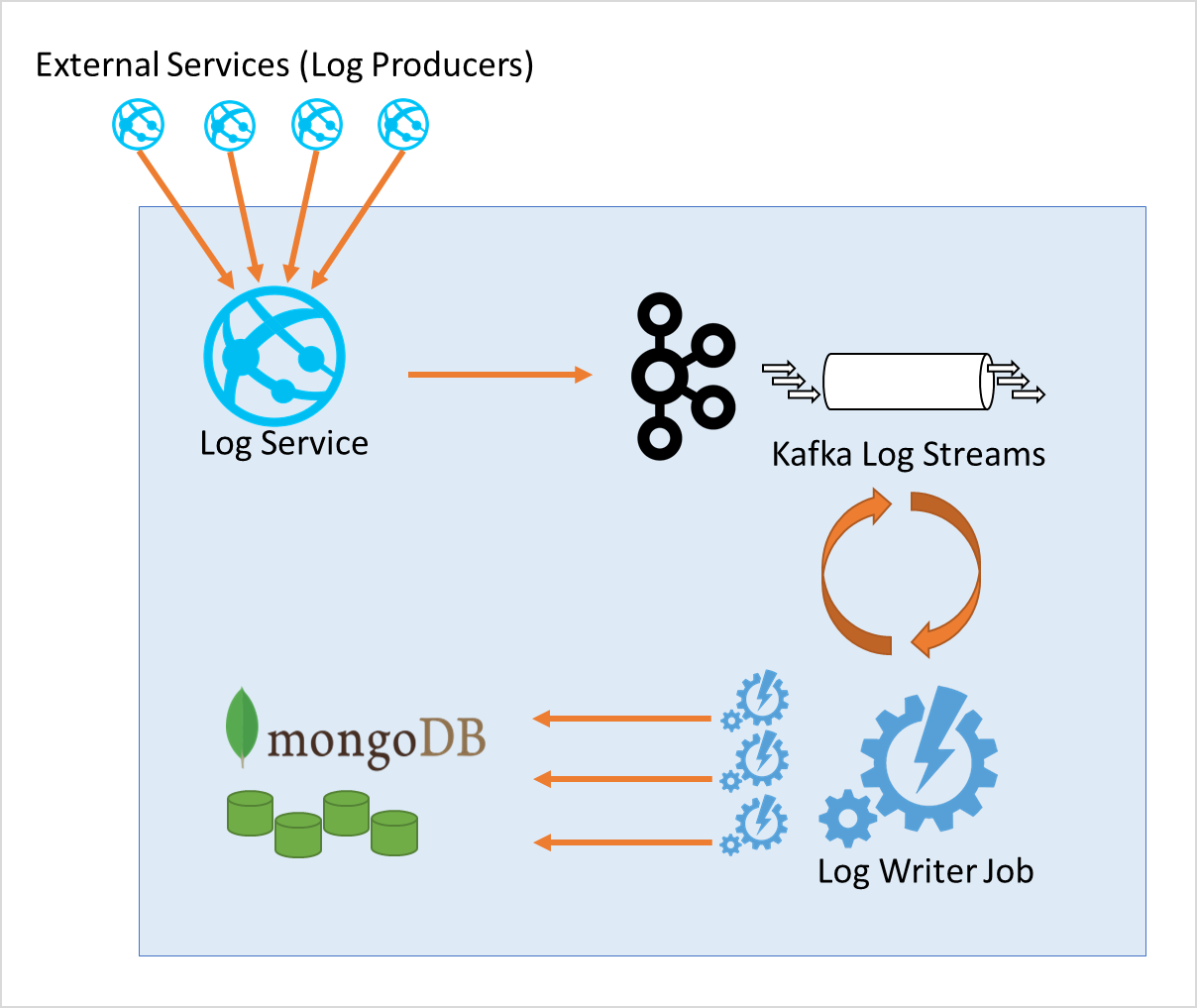
Database - MySQL

Log Aggregator and Message Queuing – Kafaka

By the time when system becomes ready, it can decided to upgrade the technology based on scheduled timeline and decision to be taken by client. But in a continuous process technologies upgradation will happen

## Logging

Logs will be generated on server side using log service. .Net CoreFramework’s ILogger instance writes logs to the Kafka log streams which would be part of log provider, which are then pushed to Mongo Db via a log writer job. The diagram below shows various component involved in implementing a generic logging framework.



Logging framework architecture

Logging will be done in two categories:

1. Client-Side Logging
2. Server-Side Logging

Client-side logs will be tagged with application, along with other details related to severity, location, event, trace etc. Similarly, Server-side logs will contain information about which service generated the logs, trace, levels, event etc. details.

Kafka log streams will provide streaming for various kind of logging messages. In future it can also serve as a platform for real time log analytics in combination with other tools.

Log writer job will be used to write logs to permanent storage using MongoDB. The service will work on a distributed architecture with master & worker nodes. Worker nodes will be scalable if there are too many logs coming in needed to be written to persistent storage.

## Security

Security will be a prime feature of the system as the system process financial & confidential data with huge customer base. There are many areas in which we will deal with implementing proper security aspects as mentioned below:

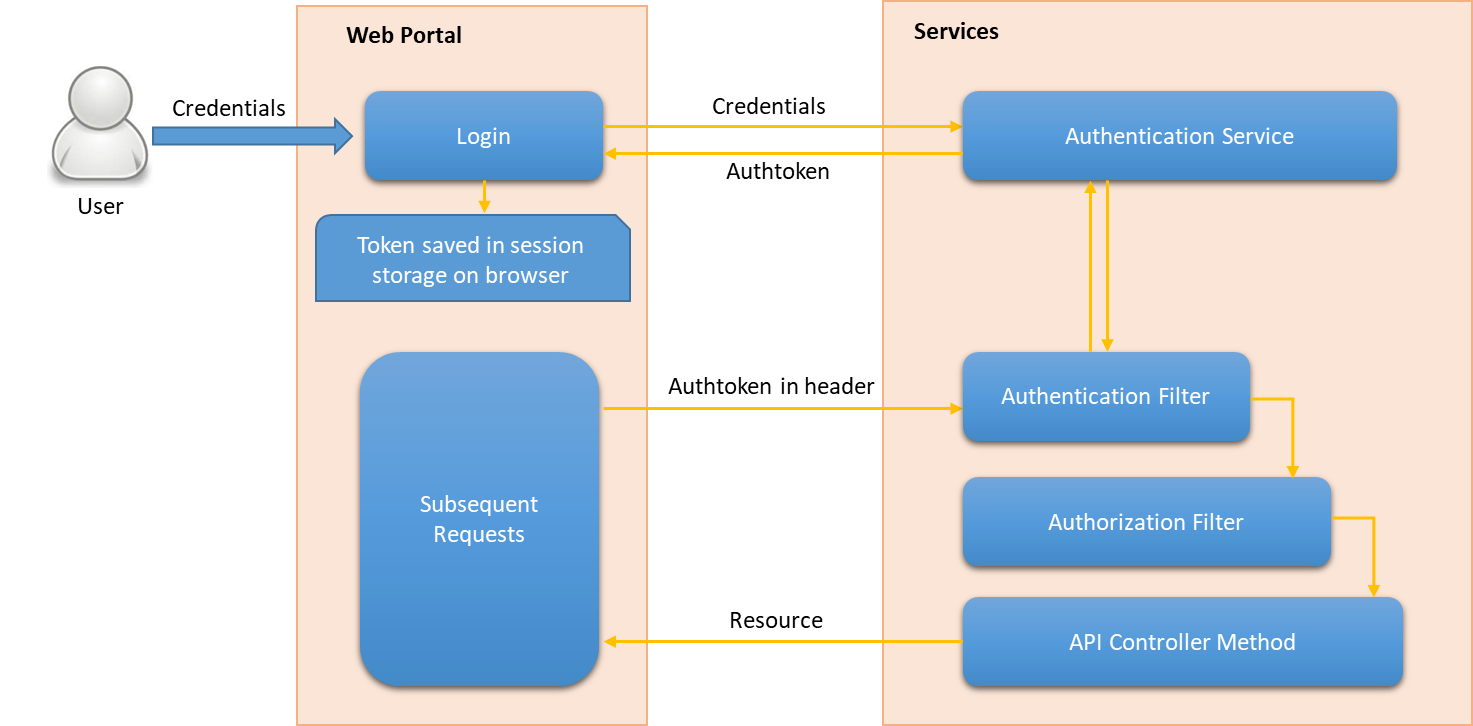
#### OWASP compliance

eClaims web application design will ensure that primary security requirements related to user authentication, authorization, data security, data integrity and web security are based on recommendations by Open Web Application Security Project (OWASP) and WASC (Web Application Security Consortium). eClaims web application will follow the guidelines of OWASP Top 10 to



#### Authentication & Authorization

All the endpoints exposed for information passing are embedded in the service layer web api project. Authentication & authorization will be taken care by a dedicated service. Token based authentication will be used as explained in the diagram below.



Token based authentication

When user lands on the system through the browser, there won’t be any authentication token available. During login, user credentials are verified, and server returns auth. This token will be saved in browser session and will be enclosed in the header of each subsequent Web API call. For all subsequent request on the server (if the request belong to a controller marked authorized) the authentication & authorization filters comes into play and use the access token to identify the user and make sure that the user has access to the asked operation. In order to logout, the authentication token needs to be cleared from the session storage on browser.

#### Session Expiration

Session on server side essentially means that the token generated for the user is valid and can be used for invoking operations. A dictionary will be maintained to store tokens generated by user with some additional properties to lookup & identify active sessions. These tokens will be considered expired and hence will be removed if their expiration time has elapsed. On the next token check call to the auth service, if the service identifies that the token expiration time has passed, it returns unauthenticated user response. In same way, a logout operation will result in finding and deleting the token in the auth service.

Sliding expiration of the token will be used to keep the token validity to minimum. For the customer portal the validity of this token will be kept 5 minutes. Reliable use cases will be implemented on frontend to logout inactive session & to keep active sessions alive if there is user activity.

#### Auditing

Auditing of the user actions would be done. Any change in the system data would bear a stamp of the user who made the change. Non-repudiation would effectively be achieved through logging of actions and stamping of modified data in the database

#### Sensitive data handling

Any communication above the business layer is done using data transfer objects. Having only the required properties in the DTOs allow us to make sure that we are not sending any undesired filed of the requested object to the client such as password or card details.

All the communication will be using SSL encryption and the same will be used across various components on the server side as well. E.g. communication between Frontend & API Gateway, API Gateway & Services, Services & Databases, Services & Cache, between two services, Services & third-party integration services will be SSL encrypted.

#### Encryption at rest

Encryption at rest will be implemented using Azure inbuilt mechanism that ensures all the block store volumes are encrypted and any data transfer to and from the block store volumes is also encrypted. SSL communication over network between components along with encryption at rest ensures that entire system ensures data security from end to end.

#### Function level access control

The API core framework will ensure all endpoints are secured to allow only authenticated & authorized requests. Other than that, many utilities will be implemented at the framework level on client side as well as server side that makes it fast and easy to check user access to operation at a granular level. Operation level access configuration will be stored in database and will be implemented in the core API & business framework.

#### Input validation

Input validation is an important security aspect. Model level validations along with business level validations will be used on the Frontend as well as API Services to ensure valid inputs.

#### Protection against SQL Injections

Entity Framework Core will be used as ORM for the application. It is a very stable object relational mapping framework and ensures that all database communications are secured from known SQL injections attacks. Further code first approach will be used and unless required stored procedures will be avoided as such calls can allow bypassing SQL injection security provided by entity framework. In cases where a stored procedure is required proper checks would be implemented on the data inputs.

#### Protection against XSS attacks

Angular Framework will be used as client-side rendering framework. It takes care of sanitizing the data while rendering on the page and properly encode special characters.

#### Protection against CSRF attacks

Token based authentication will be used instead of cookie-based authentication to ensure CSRF proof system.

## Exception Handling

Exception handling will be done in all business methods wherever a business constraint is breaking. A custom exception class BusinessException will be implemented in the global shared library. This would allow us to specify a string message and an error code for the business rule that is break. Whenever a business rule is breaking, an instance of BusinessException will be thrown from the code. All public business methods would catch, log and re-throw the exceptions.

All other exceptions will automatically bubble up to the upper layers. Finally, all exceptions will be cached in the Web API project with the help of a custom exception filter on Web API controllers. Inside this exception filter a decision is made whether the bubbled-up exception is a BusinessException or any other type of exception. Based on the type of exception and environment (debug/published), HTTP response will be formulated with HTTP Error codes (500 range). This HTTP response will be sent over to the client side in JSON format. In case of request validation failure, the response status codes will be of range 400 and the response body may include list of messages / validation error useful for the frontend.

On the client side, there will be an interceptor that would check all requests for server errors and will do a generic handling in form of a notification to the user about what went wrong.

## Input Validation

Input validation is performed to minimize malformed data from entering the system. In the system, input validation would be done in three different flavors as described.

1. **Client-side input validation:** On client side before submission of any input all the form fields would be checked for valid input. This would lead to faster data validation and save server round trip in case it was possible to identify issue with the data on client side. These checks would be implemented in typescript.
2. **Server-side model validation:** On server side, model validation will be done using data annotations on the model properties. This would ensure that no malformed data is accepted by the server. In service-oriented architecture data can come from several places, and model validation on server side becomes single point of truth to ensure that data is in correct shape.
3. **Business validation:** On server side, wherever possible business validation will be applied on the input data to ensure that the input data in not violating any business constraint. As a standard practice, all public business methods must first check for the input parameter and will raise business exceptions if any business / data constraint is violated.

## Caching

Caching will be readily used in the entire system to boost the overall performance. Various kinds of caching scenarios are listed here:

* Caching of generated tokens in Auth API for quickly checking of active session. This will be implemented with Radis Cache.
* Caching of reference/static data in Radis cache because this data is changed infrequently.
* Caching of security rules to verify access to endpoints & operation to avoid database lookup on each API call.
* Caching of front-end applications on Azure to serve the apps from nearest edge location for fastest application load time.

A close up of a device

Description automatically generated

Redis Cache diagram

For service layer caching, Radis cache cluster will be used with master slave replication for high availability and optimum performance.

## Dependency Injection

Dependency injection pattern will be used to allow layers to communicate effectively, implement loose coupling, and to implement clean code. .Net Core’s own dependency engine will be used to implement dependency injection.

Interfaces exposed by various layers will be available in the shared project, and thus can be implemented/consumed by any other project in the service. In brief, the API layer projects will be consuming interfaces exposed by the business layer projects, the business layer projects will consume interfaces exposed by the data and shared layer projects.

## Data Access

Data will be saved in the system majorly in relations.

My SQL Server will be used to store the data in relational schema. All the access to the data will be governed through data layer using Entity Framework Core ORM. The data layer will expose a generic repository that can be queried for any type derived from the base type. Code-First approach will be used for the database design, and entities will be modelled following a domain driven design.

All entities will be derived from a base entity that would contain common properties such as unique id, auditing field (created on, modified on, deleted on, created by, modified by, deleted by).

## Data Transfer between Layers

Data transfer between layers will be done using various formats and data structures. Below diagram show what format of data flows from layer to layer, between services and from frontend to services.

A screenshot of a cell phone

Description automatically generated

Data transfer between layers

* The presentation layer, containing the web project will totally run on the browser and send request to the Web API on server in JSON format through ajax request. Web API respond in JSON format too. Other than that, it will download static content – Html, CSS, JavaScript & graphics from the server deploying the static frontend portals.
* Ajax requests originating from frontend goes to API Gateway where there are rerouted to configured services.
* JSON data arriving in Web API is de-serialized to DTOs and is validated. These DTOs are feed to the business method invocation in business layers.
* Business layer looks for required information in the DTO and perform required desired operation on the data entities either by adding new entities or by modifying existing entities.
* Integration layer services take JSON data as input and perform desired operations. The result is updated to the data entities and the integration layer again respond to the business layer with JSON response.
* Some use cases require files to be uploaded / downloaded from the file servers, this communication is done by document service and the persistent store is Azure File. Data transit and storage in Azure File is encrypted.

## Test Cases

Test cases will be written for the presentation layer and the business layer of the services.

Presentation Layer: presentation layer is coded in typescript using MVVM pattern. Test cases will be written using Jasmine and Karma framework. These test case would cover the services and controllers. These test cases can be run on visual studio task runner or on karma browser framework using any suitable browser.

Business Layer: Business layer test cases will be written on server side using Microsoft unit testing projects.

## Client-Side Frameworks

There frontend will be completely based on latest version of Angular Framework. Angular framework provides a combination of tools & package all in one place to develop, unit test, end to end test, environment handling, bundling, minification & build generation. As compared to legacy approaches where a combination of various framework was required to complete tasks mentioned above, Angular Framework now provide an all in one approach and promote a good architecture for developing front end applications with cross browser support.

# Integration (External Systems) Architecture

As per the system design wherever there is external system communication involved it will be dealt with a new service and other services will just make use of the former to decouple the responsibility & standardize communication to external systems. As per requirement there are three categories of systems external to the proposed system with which communication needs to happen. These are - Document Service, Transactions Service and Notification Service respectively, making use of following external systems:

1. Third party payment hubs
2. SMS & Email notifications gateway

Communication to these externals systems will be implemented in Document Service, Transactions Service and Notification Service respectively. More details around integration characteristics of each service is mentioned below.

## Document Store integration via Document Service

The system needs to cater to upload and download of various kinds of documents such as – customer documents, incident, claim documents, work order reports and invoices etc. This requirement will be dealt by document service. The document service will be built generically and can be deployed with varying configuration to support different kinds of documents in different areas of the system. E.g. separate instances of document service will be deployed for global services space, customer, internal user & partner. This is to ensure horizontal scalability, high availability & load distribution.

As per our proposition of cloud vendor, Azure File will be used as backing store for the documents. Azure File provides 99.999999999% durability & 99.99% availability. It will also take care of encryption of data at rest & in transit. The document service instances will maintain document metadata in their local databases and will store the Azure File URL of the document for accessing the document.

## External money transfer integration via Transaction Service

The proposed system assigns the responsibility of communicating with payment gateways to the Transaction Service. There are two flavors of this service, one for the customers and other for the internal purpose where company user can pay to the partner.

## SMS & Email notifications via Notification Service

The feature to enable this communication will be implemented in the Notifications API. All other services that need to generate such notifications will be sending their requests to the notification service which will further responsible to deal with SMS and email sending.

# Deployment Architecture

Deployment process and deployment architecture will be the backbone of the system and will keep the company far ahead in the competition with other vendors still relying on legacy technologies and incurring huge bills to use third party services for their existing claim system & infrastructural components.

The proposed system will ensure best use of the dedicated infrastructure and will have ability to scale up and down within few minutes to cater to peak load time as well as temporary spikes by adding/removing required infrastructure in cloud environment.

The overall process of deployment will be divided into two phases. Phase1 will be one-time configuration of the above services on Azure cloud. Phase2 will be ongoing deployment process whenever next version/upgrade of the application is available.

## Deployment Components

For deployment some Azure managed & some self-managed services will be used. Overall the physical components used during deployment are shown in the diagram below.

Graphical user interface, application, Word

Description automatically generated

General AKS Deployment Diagram

1. **Azure AKS** - Azure Kubernetes Service (AKS) is a managed container orchestration service, based on the open source Kubernetes system, which is available on the Microsoft Azure public cloud. An organization can use AKS to deploy, scale and manage Docker containers and container-based applications across a cluster of container hosts.

The fully managed Azure Kubernetes Service (AKS) makes deploying and managing containerized applications easy. It offers serverless Kubernetes, an integrated continuous integration and continuous delivery (CI/CD) experience and enterprise-grade security and governance. Unite your development and operations teams on a single platform to rapidly build, deliver and scale applications with confidence.

1. **Kubernetes cluster-** AKS is responsible for deploying the Kubernetes cluster and for managing the Kubernetes masters. You only manage the agent nodes.
2. **Ingress**­ an ingress exposes HTTP(S) routes to services inside the cluster. It acts as API gateway.

The Kubernetes Ingress resource type abstracts the configuration settings for a proxy server. It works in conjunction with an ingress controller, which provides the underlying implementation of the Ingress.

1. **Azure Active Directory -** AKS uses an Azure Active Directory (Azure AD) identity to create and manage other Azure resources such as Azure load balancers. Azure AD is also recommended for user authentication in client applications.
2. **Azure Container Registry -** Container Registry to store private Docker images, which are deployed to the cluster. AKS can authenticate with Container Registry using its Azure AD identity. Note that AKS does not require Azure Container Registry. You can use other container registries, such as Docker Hub.
3. **Azure Pipelines -** Pipelines is part of Azure DevOps Services and runs automated builds, tests, and deployments. You can also use third-party CI/CD solutions such as Jenkins.
4. **Helm -** Helm is as a package manager for Kubernetes — a way to bundle Kubernetes objects into a single unit that you can publish, deploy, version, and update.
5. **Azure Monitor -** Azure Monitor collects and stores metrics and logs, including platform metrics for the Azure services in the solution and application telemetry. Use this data to monitor the application, set up alerts and dashboards, and perform root cause analysis of failures. Azure Monitor integrates with AKS to collect metrics from controllers, nodes, and containers, as well as container logs and master node logs.
6. **Kafka Streams** - Kafka is used for building real-time data pipelines and streaming apps. It is horizontally scalable, fault-tolerant, amazingly fast. We will be using Kafka for aggregating log stream for real time analysis and dumping to document DB for prolonged storage.
7. **Azure SQL Database-** Microsoft Azure SQL Database is a managed cloud database (SaaS).

It can scale quickly and reach global distribution without worrying about costly downtime. It also provides some cool features like full management and monitoring of multitenant apps with isolation benefits of one-customer-per-database, leverage open source tools like VS Code and Microsoft tools like Visual Studio and SQL Server Management Studio, Azure Management Portal, PowerShell, and REST APIs, Data protection with encryption, authentication, limiting user access to the appropriate subset of the data, continuous monitoring and auditing to help detect potential threats and provide a record of critical events in case of a breach.

Our implementation will make use of several Azure MySQL database. It is a relational database service based on the open-source MySQL Server engine. It's a fully managed database as a service offering that can handle mission-critical workloads with predictable performance and dynamic scalability.

1. **Mongo DB Cluster -** MongoDB is a free and open-source cross-platform document-oriented database program. Classified as a NoSQL database program, MongoDB uses JSON-like documents with schemata. MongoDB provides high availability with replica sets.

We will be using Mongo Db for various purposes such as storing logs

## Deployment Diagram - Services

While most of the components in the deployment architecture are simple and follow well know deployment standards for enterprise applications, the business services are deployed using cutting edge technologies. These services are containerized & deployed on Kubernetes clusters inside the Kubernetes pods. The deployment is explained below.

A screenshot of a cell phone

Description automatically generated

The diagram shows how the app containers would run inside service pods hosted over AKS cluster. Scaling of the pods is controlled by AKS cluster to meet required load as per inputs available from Metrics Server; Whereas whenever required, scaling of pods instances will be handled by AKS Cluster Auto Scaler.

## Deployment Environments

There would be four deployment environments:

1. Test Environment – for the testing team
2. UAT Environment – for client testing
3. Staging Environment – for production stability testing
4. Prod Environment – for production

All four environments will be available on Azure cloud infrastructure. The capability of these three environments would vary. The type of virtual machines will be selected based on the performance output of the application when tested progressively on various environment with load test scenarios.

# Glossary

## Requirements Traceability Matrix

Unavailable at this point of time, since requirements, test cases, functional design are not available

## Technology POCs

### eClaim Incident Reporting

The POC illustrates single page application specifically a user interface to report the incident.

A screenshot of a cell phone

Description automatically generated

### Kubernetes based service orchestration

The POC illustrates deployment of services with orchestration being handles by Kubernetes configuration.

A screenshot of a cell phone

Description automatically generated

Screenshot from POC



**COPYRIGHT NOTICE**

Copyright © 2013-15 Nagarro Software Pvt. Ltd. **All rights reserved.**

This document is confidential and proprietary to Nagarro Software Pvt. Ltd. and no part of it should be reproduced, published, transmitted or distributed in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, or stored in any information storage or retrieval system of any nature nor should it be disclosed to third parties without the prior express written authorization of Nagarro Software Pvt. Ltd.

|  |  |  |  |
| --- | --- | --- | --- |
| **Revision History** | | |  |
| Version | Date | Author/Contributor | Comments |
| 0.1 | 20-06-2019 | Sanjeet Chauhan | Initial Version |
| 1.0 | 20-01-2021 | Sanjeet Chauhan |  |