



Version 1.1

**YCompany eClaims**

High Level Design Document

Contents

[1 Introduction 3](#_Toc529963059)

[1.1 Purpose of Document 3](#_Toc529963060)

[1.2 Scope of Document 3](#_Toc529963061)

[2 System Overview 4](#_Toc529963062)

[2.1 Project Overview 4](#_Toc529963063)

[2.2 System Data categorization 5](#_Toc529963064)

[2.3 Functional Zones 6](#_Toc529963065)

[2.3.1 Frontend Applications 6](#_Toc529963066)

[2.3.2 Backend Services 7](#_Toc529963067)

[2.3.3 Background Jobs 11](#_Toc529963068)

[3 Design Considerations 13](#_Toc529963069)

[3.1 Assumptions and Dependencies 13](#_Toc529963070)

[3.2 General Constraints 13](#_Toc529963071)

[3.2.1 Availability 13](#_Toc529963072)

[3.2.2 Performance 13](#_Toc529963073)

[3.2.3 Thin Client 13](#_Toc529963074)

[3.2.4 Database considerations 13](#_Toc529963075)

[3.3 Goals and Guidelines 14](#_Toc529963076)

[3.3.1 Response Time 14](#_Toc529963077)

[3.3.2 Coding Guidelines 14](#_Toc529963078)

[4 NFR Considerations 15](#_Toc529963079)

[4.1 Usability 15](#_Toc529963080)

[4.2 Reliability 16](#_Toc529963081)

[4.3 Performance 17](#_Toc529963082)

[4.4 Security 17](#_Toc529963083)

[4.5 Supportability 18](#_Toc529963084)

[4.6 Infrastructure Requirements 19](#_Toc529963085)

[4.7 Implementation Constraints 19](#_Toc529963086)

[5 Solution Architecture 21](#_Toc529963087)

[5.1 System Design 21](#_Toc529963088)

[5.2 Information Flow 22](#_Toc529963089)

[5.3 Architecture Diagram 22](#_Toc529963090)

[5.3.1 Frontend Applications 22](#_Toc529963091)

[5.3.2 Services 23](#_Toc529963092)

[6 Framework/Infrastructural Components 27](#_Toc529963093)

[6.1 Logging 27](#_Toc529963094)

[6.2 Security 28](#_Toc529963095)

[6.3 Exception Handling 30](#_Toc529963096)

[6.4 Input Validation 30](#_Toc529963097)

[6.5 Caching 30](#_Toc529963098)

[6.6 Dependency Injection 31](#_Toc529963099)

[6.7 Data Access 31](#_Toc529963100)

[6.8 Data Transfer between Layers 31](#_Toc529963101)

[6.9 Test Cases 32](#_Toc529963102)

[6.10 Client-Side Frameworks 33](#_Toc529963103)

[7 Integration (External Systems) Architecture 34](#_Toc529963104)

[7.1 Document Store integration via Document Service 34](#_Toc529963105)

[7.2 External money transfer integration via Transaction Service 34](#_Toc529963106)

[7.3 SMS & Email notifications via Notification Service 35](#_Toc529963107)

[8 Deployment Architecture 36](#_Toc529963108)

[8.1 Deployment Components 36](#_Toc529963109)

[8.2 Deployment Diagram - Services 37](#_Toc529963110)

[8.3 Deployment Environments 38](#_Toc529963111)

[9 Glossary 39](#_Toc529963112)

[9.1 Requirements Traceability Matrix 39](#_Toc529963113)

[9.2 Technology POCs 39](#_Toc529963114)

[9.2.1 Kubernetes based service orchestration 39](#_Toc529963115)

# 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 an 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.

**Kibana Dashboard:** Will be used by DevOps admins to monitor logs, errors, application performance & instrumentation details. This is a readymade portal that will be configured for various kinds of analytical insights required for the overall system monitoring.

**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. Global Services
  2. Customer Services
  3. Branch Services
  4. Partner Services

This distribution of servies into groups is described below.

#### Global Services

Global services are the one that are common for all users and there is no partitioning of data in multiple database. Various APIs in the global services subsytem are described below:

Global Services

* + **Auth API :** Auth api will be majorly responsible for storing generated tokens. Any service before performing any sensitive operation should be checking the validity of token supplied in the request with the auth api & rely on it to provide roles & access levels of the requesting user to allow/disallow the operation. This commpn API will be consumed on customer and partner level as Internal suer will have more complex access role devision in profilinf system.
  + **Profile API :** will expose endpoints for customer info, partner 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.
  + **Products API :** This api will provide avaiable insurance products to any user which may contain Life insurent, motor insurent etc. It will be made as common api as can be requested by anyone in system
  + **Promotion API :** Promotion api will majorly provide company’s promotions to end users and partners. It will provide methods to get offeres and discounts , events etc.
  + **Documents API :** This will be a base level document api which may contain some specifc operations for all inner level document api like document comprassion. It is important to have comparission in place for all type of docuemnts uploaded by any user in the sysem as doc storage later will be a concern.
  + **Notification API:** Notification API will be responsible for producing notifications to users be it SMS / Email or app notification in an active session. On one hand this API will expose endponts that can be used by all other services to post notification to a user, on other hand the service would implement various communication channels such as – third party SMS service intergration, integration with mail services for emails, web sockets connection with active sessions to post realtime notifications on browser.
  + **Reports API:** Will expose endpoint for searching & downloading various reports from the each portal.
  + **Logs API:** Will be implementing a highly scalable & efficient log stream using Kafka that can be used by all the services to write logs & instrumentation info.
  + **Path Resolution API:** Will be used to resolve service location when the services / data is partitioned. E.g. What would be the endpoint for branch api for a given branch. As the components will be partitioned / shareded, the role of path resolution become really important. This api will also be using the Profile API to resolve user specific service endpoints.
  + **Reference Data API :** This api will store all the reference data such as dropdown values, countries, currencies. Other than that it will store all the global reference entities such as account types, report types, policy, terms & conditions etc.

#### 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 API:** Provide methods to the customers to get registered wuth the company portal.
  + **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.
  + **Transactions API:** This API will allow customer to check their payments statements agains each transaction happened in past and can also download Invoices.
  + **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.
  + **CarRental API:** This API provides method to get available cars fro, partner whichc ustomer may want to get.
  + **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.
  + **Contact API:** Provide customer to make an appointment with partner workshop and to connect with chat messagenger os the company.
  + **Tracking API:**This api will dedicatly be designed to monitor customer actions on portal to get the sense of interest customer shows on custmer portal. Data collected with this api will help company to make better decision against their business products and other operations etc.

#### Branch 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.
  + **CustomerAccount API:** This will consist methods to open an account for a customer and managing their policies under the account.
  + **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.
  + **Signing API:** It will allow claim manager to approve, reject or partial approve a cliam.
  + **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.

# 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.
* It is assumed that no legacy system exists.
* Exact number of regulatory feeds is not known, and a realistic value is assumed ~10.
* 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.

## 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 and Claims data will be shared 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.
* There won’t be dedicated database servers for each shard. Instead, multiple shards will share same database server. So, on database server level, there will be logical sharding of databases. This will help in better resource utilizing for scenarios where many Db servers could be idle whereas other Db servers would be highly loaded if servers are dedicated to the shards. Approximation is to keep ~5 shards per Db server.

## 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.
* 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.
* The APIs will not be simple CRUD operations but will follow more of operation-based semantics. E.g. to approve a flow, the client won’t post the entire workflow data, instead will just post the workflow id & desired modification. The service will always respond with the latest full object after modification so that frontend always display a true picture after each backend operation.
* 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 and Mouse
* 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.
* Documentation
  + What kind of documentation is required?
    - eClaim Partner/Internal Portal: User manual for admin & staff members.
    - E-Banking 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
* 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 other screen it will be ensured that is saved prior to navigate
  + 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.
    - Scheduled maintenance time must not be over 30 mins.
* 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
* 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 High severity and Medium Severity bugs?
    - <=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.
* 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.
    - Within a 5 minutes window system should be able to scale up or scale down to cater to 50% increase or decrease in load.

## 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 - should be automatic.
    - Front end portal installation: should be automatic
    - Background jobs installation: should be automatic
    - Database server management, database sharding, 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
    - Internal service account configuration for cross service communication
    - Document repository path configuration
    - Initial admin configuration in the databases.
  + Is any type of administration panel is required to configure the system?
    - eClaim Internal System: capable of all configurations
  + Would configuration items specified in application configuration file be acceptable?
    - Yes
* Ease of Testing
  + Is part of system needs to be testable 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 auxiliary 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
  + Will the target equipment be in one or several locations?
    - Single region, multiple availability zones
  + 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, MS SQL Server, Visual studio 2017 IDE with typescript installed, Visual Studio Code, Mongo Db, Kafka, Docker, Kubernetes
* 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
  + 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
    - File Service
  + 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?
    - MS SQL server
    - Mongo Db
  + Is support for more than one database required?
    - Yes
  + Will there be any database partitioning or mirroring required?
    - Partitioning – not required
    - Mirroring - required

# Solution Architecture

The envisioned solution is composed of client-side applications, API gateway, autoscaling & load balancing components, service discovery components, containerized service deployment, services relying on transactions databases, document databases, Radis cache, Kafka log streams, messaging queues, payment switches, scheduled jobs etc. for implementing various business requirement.

Various aspects of solution architecture are explained in following subsections.

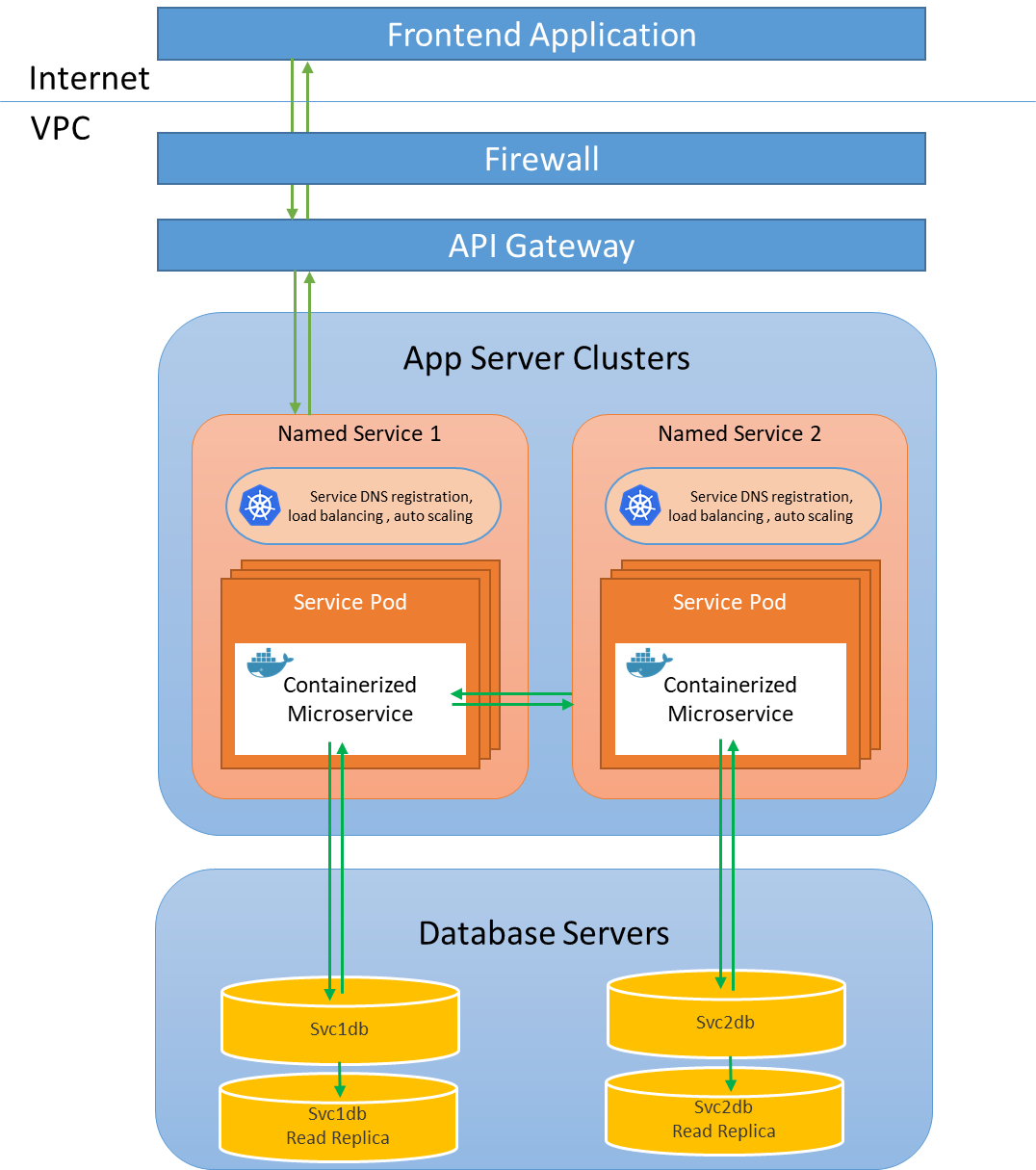
## System Design

The overall system design is shown in the system design diagram below:

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.

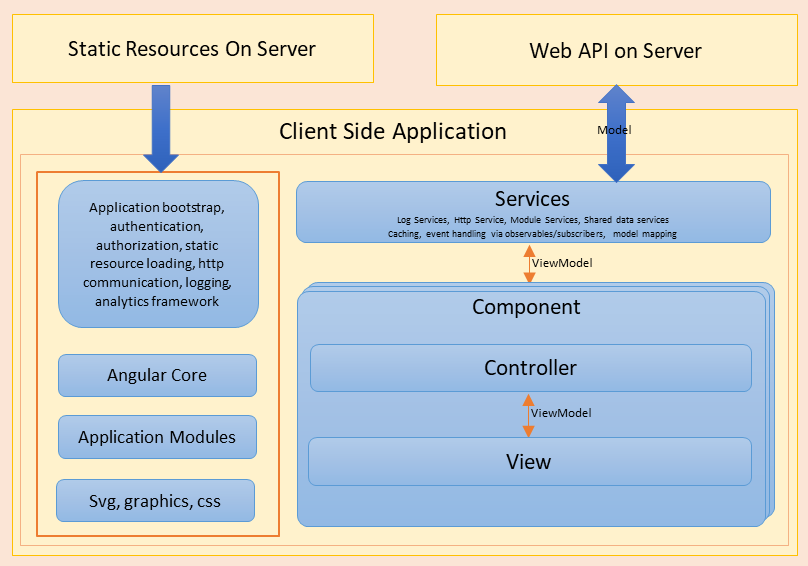


Information flow in service – Simplified view

## 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 MVVM pattern with Angular Framework. The architecture for the client-side application is shown in the diagram below:



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). It is a strict superset of JavaScript and adds optional static typing and class-based object-oriented programming to the language. For a large JavaScript project, adopting Typescript results in more robust software, while still being deployable where a regular JavaScript application would run.

Angular framework will be used on client side to handle view rendering, data binding, event handling, creating custom UI controls, layout. Angular is a structural framework for dynamic web 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. The region enclosed by the red boundary will essential be contained inside a service implementation, remaining components in the diagram show how a service communicate to the other services or systems.



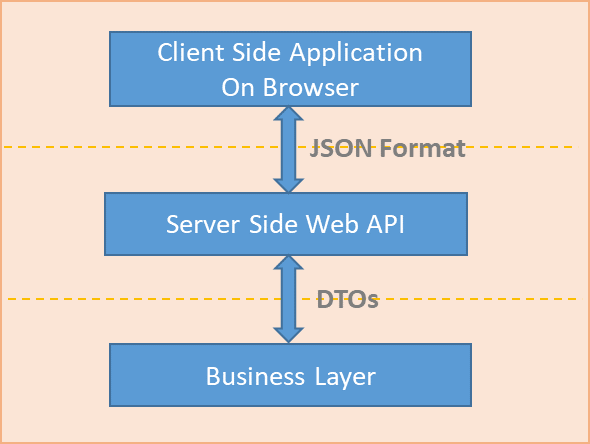
Service architecture (shows example of Loan Accounts Service)

#### 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.

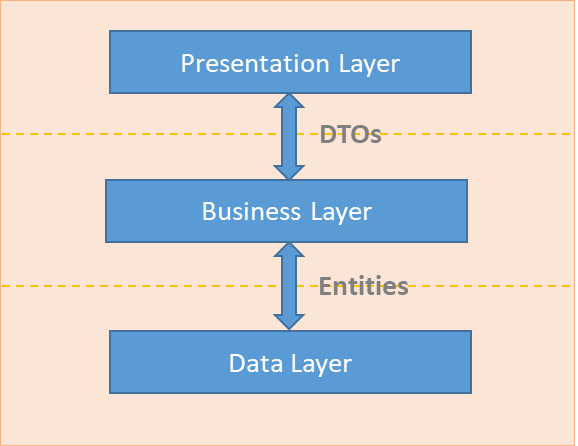
Web APIs will be implemented using Microsoft’s ASP.NET Core Web Application project template. .NET Core is a platform agnostic framework that makes it easy to build HTTP services that reach a broad range of clients, including browsers and mobile devices; and is an ideal platform for building RESTful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



Data communication to and from Web API layer

##### Business Layer

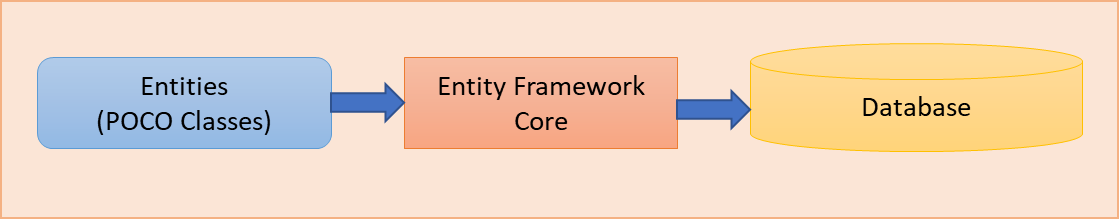
All business layer projects will class libraries projects in .NET Core framework, will expose their API through interfaces, and any inbound/outbound data communication with the upper layers will be done using data transfer objects (DTOs).



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 microservice other than that global assemblies shared by all the microservices. 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.

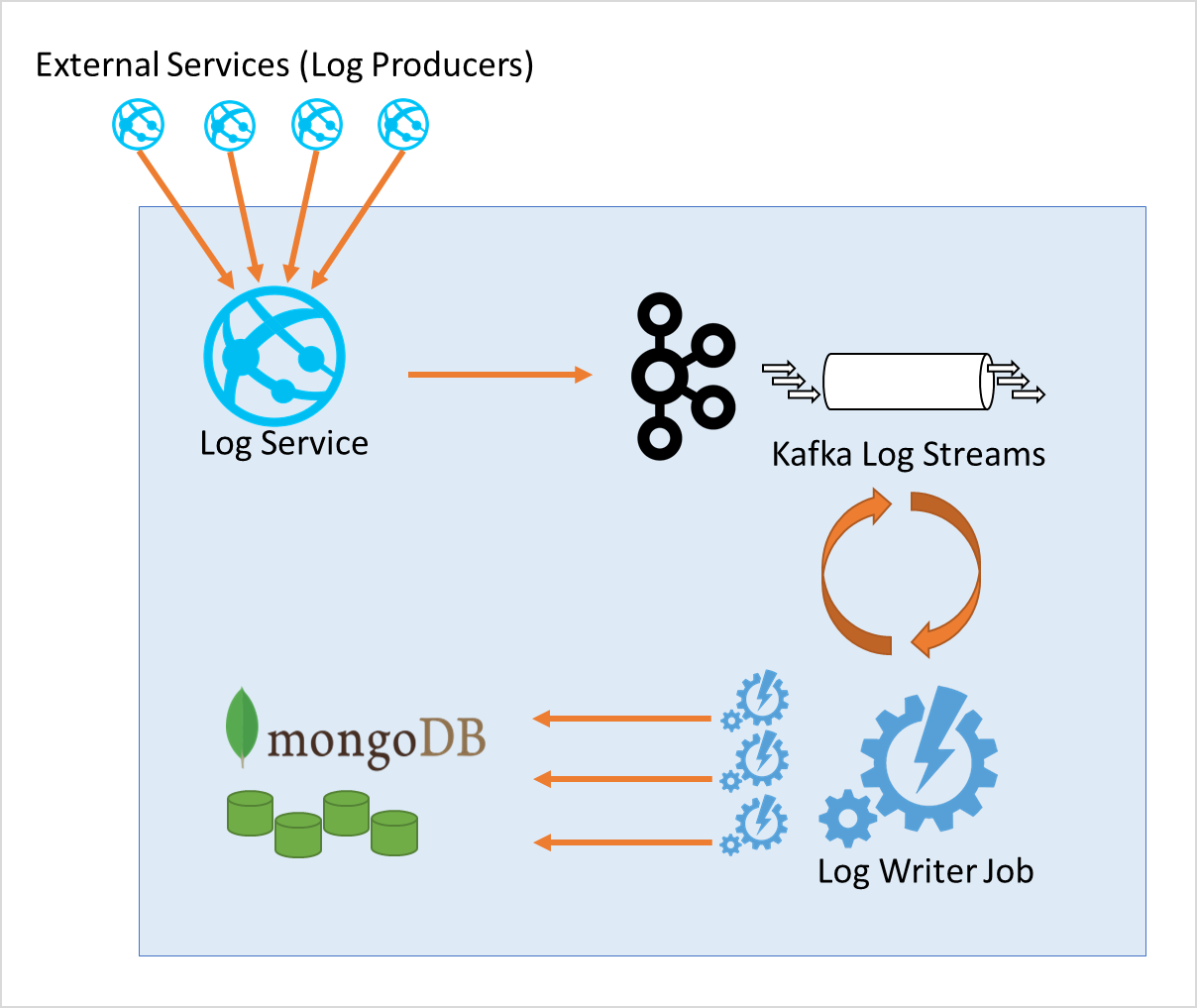
###### Shared Layer

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

# Framework/Infrastructural Components

## Logging

Logs will be generated on server side using log service. The log service writes logs to the Kafka log streams, 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.

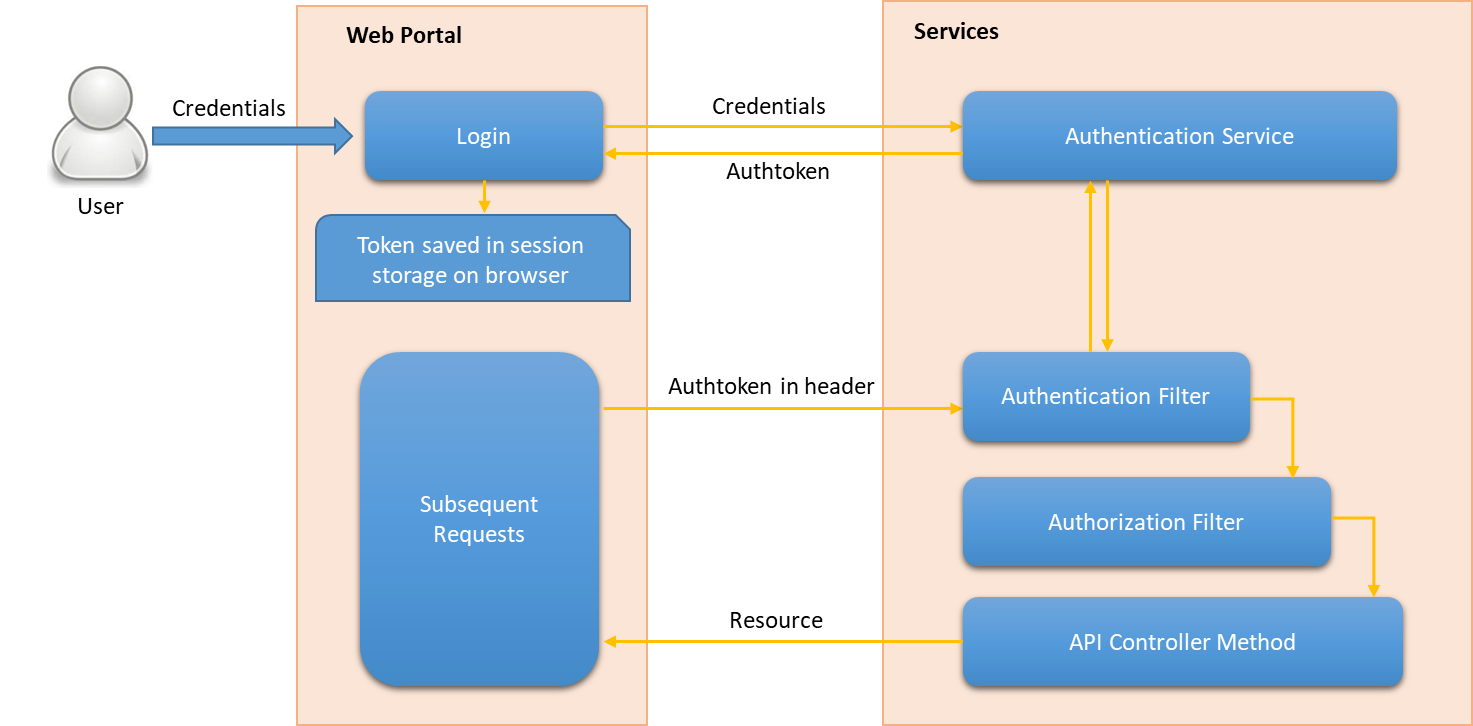
Log writer job will be used to write logs to permanent storage using MongoDB. The service will work on a distributed architecture with master & worker nodes. Worker nodes will be scalable if there are too many logs coming in needed to be written to persistent storage.

## 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:

#### 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 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.

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. Unity framework 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

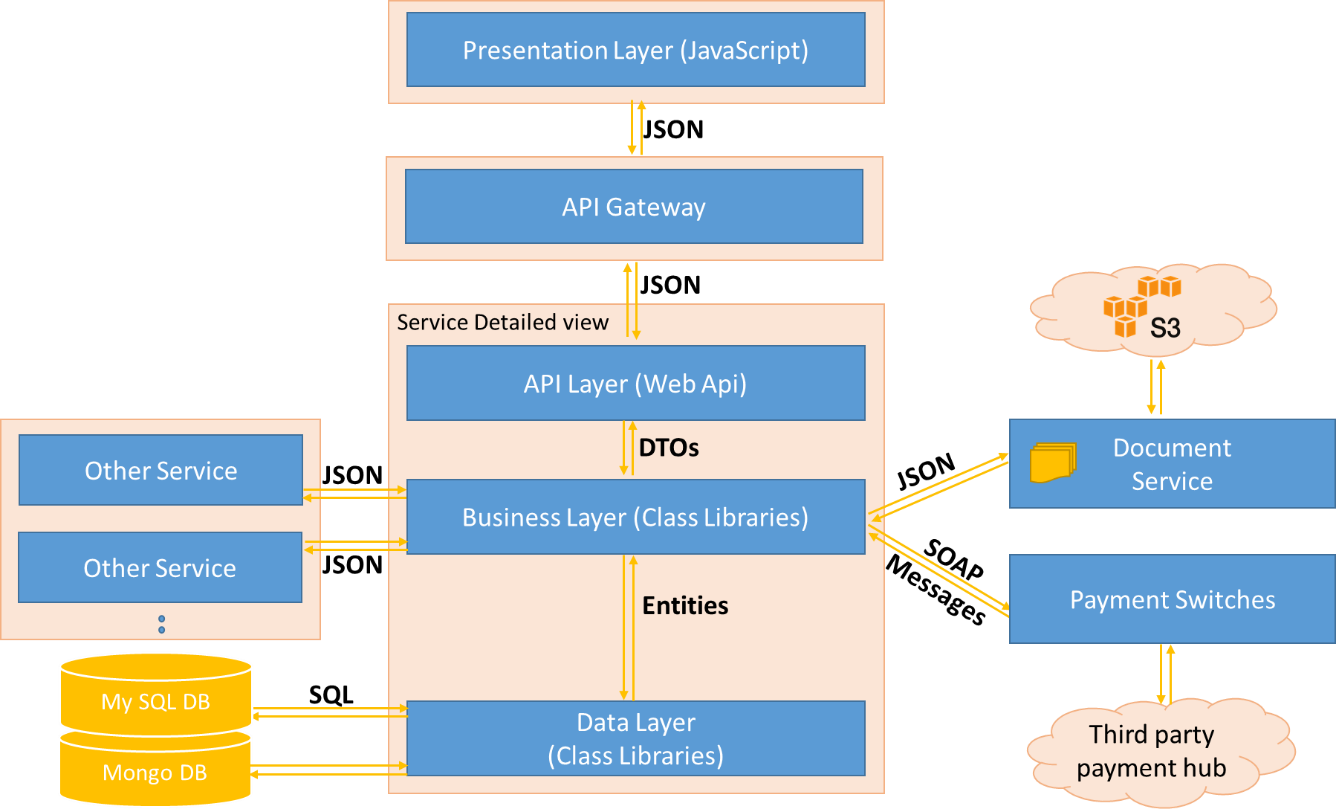
Data will be saved in the system majorly in relations & document format. For No SQL data, Mongo DB will be implemented. It would be used in following use cases:

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.



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. Document store
2. Third party payment hubs
3. 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 and claim documents work order reports 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 shards, internal user shard & partner shards. Wherever sharding is involved, one instance of document service will be required per shard. 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 system will rely on external / bank’s notification system to generate SMS & Email notifications. 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.

# Deployment Architecture

Deployment process and deployment architecture will be the backbone of the system and will keep the client (bank) far ahead in the competition with other vendors still relying on legacy technologies and incurring huge bills to use third party services for their core banking & 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 AWS cloud. Phase2 will be ongoing deployment process whenever next version/upgrade of the application is available.

## Deployment Components

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



1. **Amazon API Gateway** – Amazon API Gateway is a fully managed service that makes it easy for developers to create, publish, maintain, monitor, and secure APIs at any scale. With a few clicks in the AWS Management Console, we can create an API that acts as a “front door” for applications to access data, business logic, or functionality from back-end services.

In our case, Amazon API gateway will be used for guarding, throttling, managing & exposing the business services. Since this is an AWS managed & auto scaled service, we don’t need to know or manage the internal infrastructure.

1. **Amazon CloudFront -** Amazon CloudFront is a fast content delivery network (CDN) service that securely delivers data, videos, applications, and APIs to customers globally with low latency, high transfer speeds, all within a developer-friendly environment. CloudFront is integrated with AWS – both physical locations that are directly connected to the AWS global infrastructure, as well as other AWS services.

In our case CloudFront will be used to host static web portal to minimize the download time on browsers across the globe.

1. **Amazon EKS** - Amazon Elastic Container Service for Kubernetes (Amazon EKS) makes it easy to deploy, manage, and scale containerized applications using Kubernetes on AWS. Amazon EKS runs the Kubernetes management infrastructure for you across multiple AWS availability zones to eliminate a single point of failure. Amazon EKS is certified Kubernetes conformant, so you can use existing tooling and plugins from partners and the Kubernetes community. Applications running on any standard Kubernetes environment are fully compatible and can be easily migrated to Amazon EKS. All our services will reside in the EKS space, with application containers hosted inside Kubernetes pods. The scaling of pods, DNS registry, hardware scaling etc. will be managed by Amazon EKS service.
2. **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 implementing log stream for real time analysis and dumping to document DB for prolonged storage. Kafka implementation will be a self-managed deployment clustered using placement group of VMs in different availability zones for high availability.
3. **Amazon RDS -** Amazon Relational Database Service (or Amazon RDS) is a distributed relational database service by AWS, running in the cloud designed to simplify the setup, operation, and scaling of a relational database for use in applications. Administration processes like patching the database software, backing up databases and enabling point-in-time recovery are managed automatically.

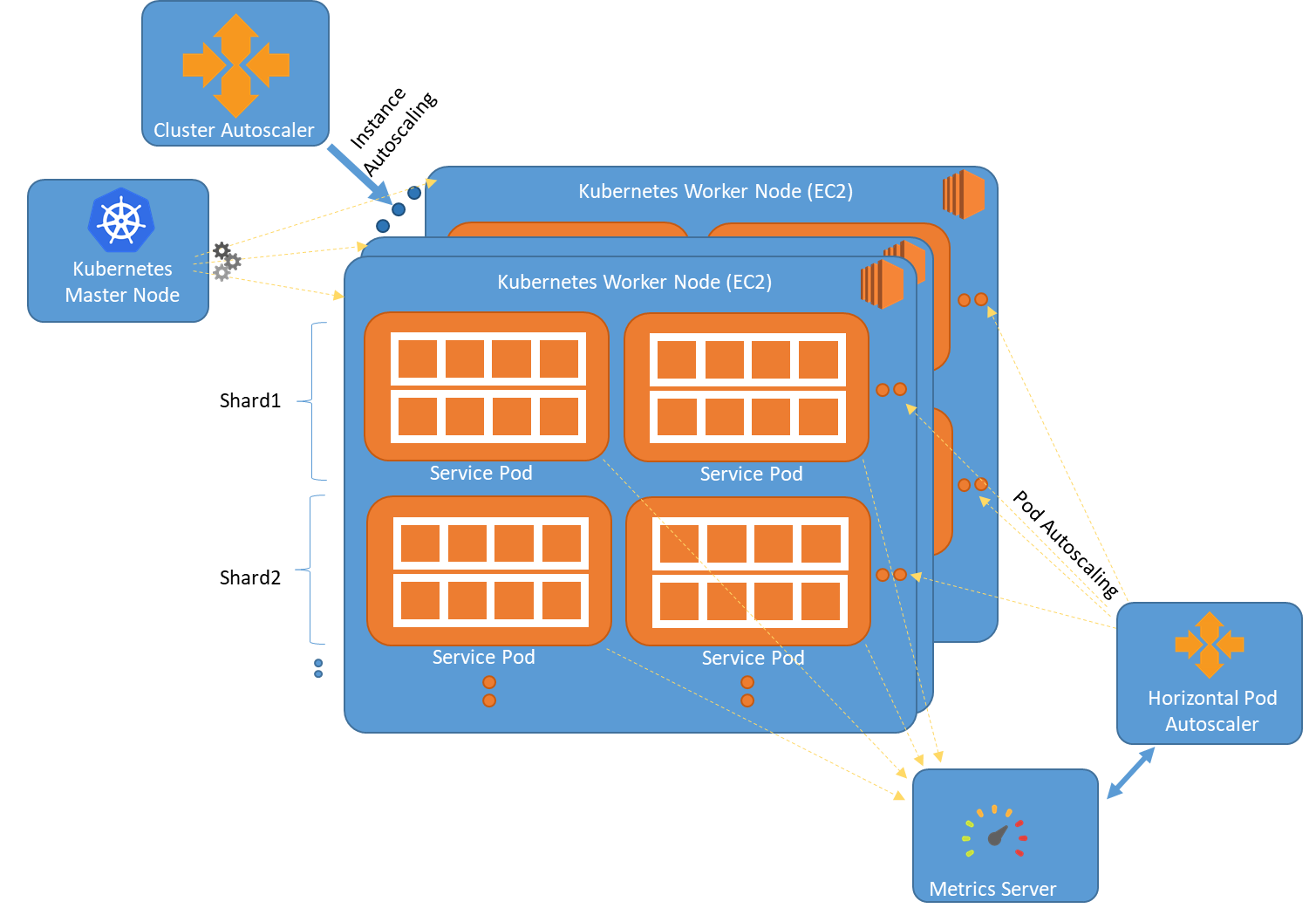
Our implementation will make use of several Multi AZ MySQL server cluster with replica sets for high availability & failover to most updated read replicas. There will be logical grouping of databases from various app service shards on these database servers to ensure better resource consumption because autoscaling is not an option.

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, storing analytic data generated post ETL processes.

## 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.



The diagram shows how the app containers would run inside service pods hosted over EC2 virtual machines. Scaling of the pods is controlled by Kubernetes master node to meet required load as per inputs available from Metrics Server; Whereas whenever required, scaling of EC2 instances will be handled by AWS’s 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 AWS cloud infrastructure. The capability of these three environment 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

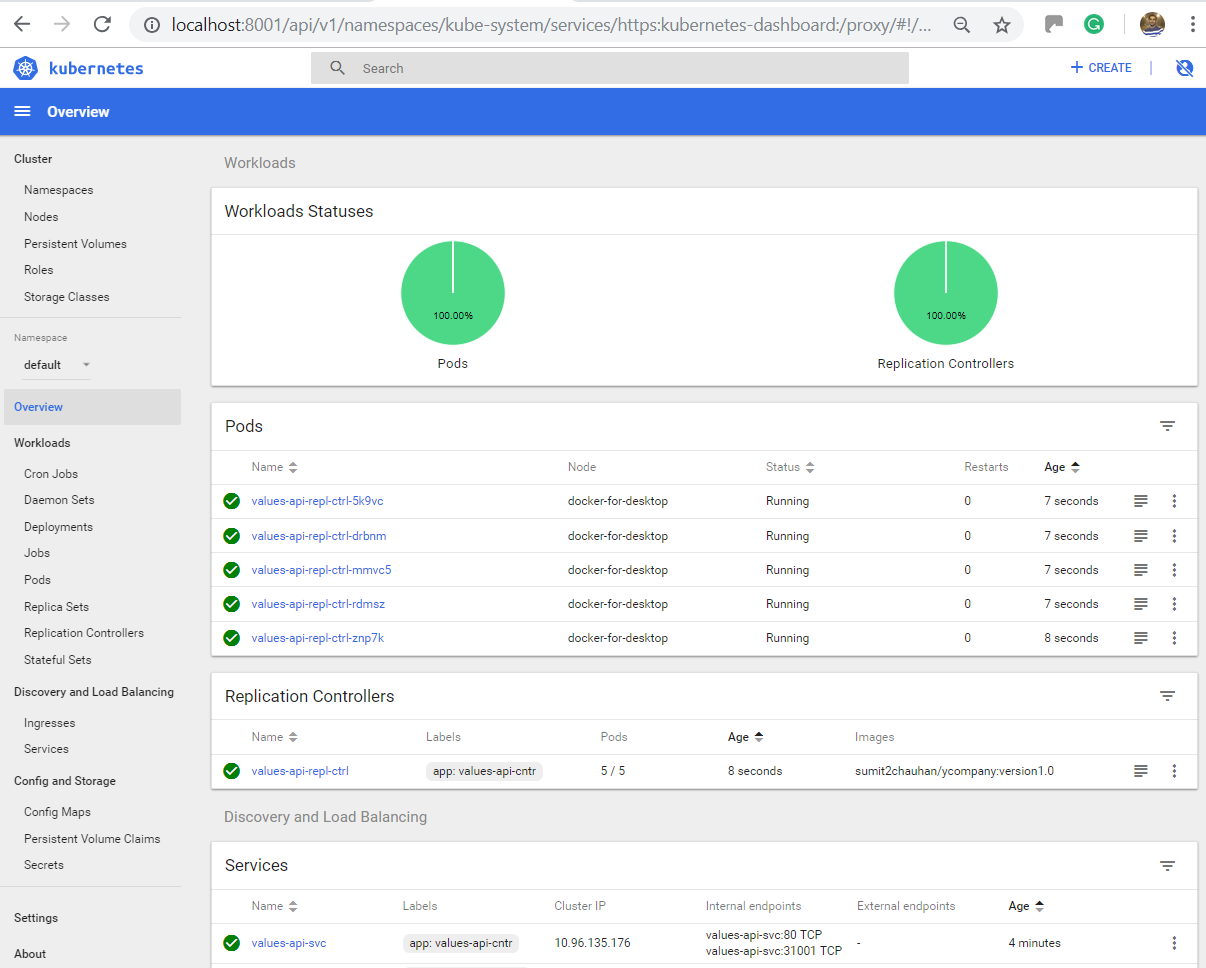
## Requirements Traceability Matrix

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

## Technology POCs

### Kubernetes based service orchestration

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



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 |
|  |  |  |  |