Crew Link MVP

Technical Approach Document

March 4, 2019  
Version 0.3

**DOCUMENT CONTROL**

|  |  |  |  |
| --- | --- | --- | --- |
| **Prepared by** | Dharmendra Kumar Singh | **Designation** | Associate Architect |
| **Date** | Feb 20, 2019 | **Signature** |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Approved by** |  | **Designation** |  |
| **Date** |  | **Signature** |  |

**DOCUMENT CHANGE HISTORY**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Date** | **Version** | **Section** | **Description** | **Author** | **Reviewer** | **Approver** |
| Feb 20, 2019 | 0.1 | NA | Microservices Design consideration | Dharmendra Kumar Singh |  |  |
| Mar 4, 2019 | 0.2 | NA | DevOps and Angular front end design considerations | Kiran Chakravarthy |  |  |
| Mar 6, 2019 | 0.3 | NA | Updated Domain Driven Design Architecture as per Feedback recieved | Dharmendra Kumar Singh |  |  |

**TABLE OF CONTENTS**

1 Introduction 5

2 Why Transformation? 6

2.1 Area addressed with Transformation 9

3 Objective 13

3.1 Detailed functional objectives 13

4 Definitions, Acronyms and Abbreviations 15

4.1 Definitions 15

4.2 Acronyms and Abbreviations 15

5 Architectural Representation 16

5.1 Proposed technical architecture – ATPI application framework 16

5.2 Proposed functional Architecture 16

5.3 Technical – Functional relationship 17

5.4 Component level Technical – Functional relationship 17

5.5 Frontend (Angular) Architecture 18

6 Scope 20

6.1 Scope of work 20

6.2 Assumption 20

7 Proposed Technology Stack 21

8 Development Tool 23

8.1 Front End Development 23

8.2 Middleware Development 23

8.3 Backend Development 23

8.4 Source Control 23

8.5 Development Approach 23

9 Non Functional Requirement 26

10 Pre-requisites 26

11 Front end Architecture using Angular 7 28

12 DevOps 30

13 Microservice Technical Architecture 33

14 Frontend Technical Architecture 36

15 Features 37

15.1 Login 37

15.2 Authorization 41

15.3 Customer - Landing Page 42

15.4 Agent - Landing Page 43

15.5 Customer – Raise Travel Request 43

15.6 Customer – Approve Travel Request 44

15.7 Agent – Search Travel Request 44

15.8 Agent – Submit Travel Request 45

15.9 Notification 45

16 Third party integration 46

17 Risks, Constraints, Limitations & Issues 47

17.1 Risks 47

*17.2* Constraints 47

*17.3* Limitations 47

17.4 Issues 47

17.5 Dependencies 47

18 Appendix 48

18.1 Project Standards and Conventions 48

18.2 Development Environment Parameters 48

# Introduction

ATPI CrewLink™ is a travel workflow management tool, specifically designed for the international Shipping and Energy industries. ATPI CrewLink™ will integrate with internal or third-party crew management tools and enables you to manage the entire travel life cycle from scheduling, via travel management to reporting. All in one global platform.

Using ATPI CrewLink™ leads to better control, greater process compliance and optimised traveler safety. With incremental cost savings as a result.

* Operational efficiencies around labor intensive, manual processes and double work
* Cost efficiencies by eliminating disparity of systems and enhanced quality control
* Optimized travel policy compliance
* Visibility of crew or traveler whereabouts and risk control
* Transparency of total cost of travel and opportunity for savings
* Improved forecasting capabilities
* Streamlined communications
* Consolidation of multiple (3rd party) systems in one single global platform
* Automated sourcing of the lowest fare available in the market in accordance with travel policy
* Automated search for the highest possible airline seat availability

# Why Transformation?

The CL transformation considers the following broad categories for migration:

* Cloud Migration
* Scalability: Scalability is a highly significant advantage of cloud technology. Through scalability, you can scale the storage capacity of the infrastructure up or down as per requirement. If you want your infrastructure to grow (upscale and downscale as and when required), but would prefer not to worry about the additional expenses attached to it, you can choose to migrate to cloud technology.
* Increased effectiveness: This gives your company the increased advantage of utilizing it for development and deployment of services, rather than worrying about monitoring and maintaining infrastructure. This leaves the company resources to focus on achieving business goals.
* Lower costs: Since scalability is a strong point of cloud migration, you need not worry about operational costs, because you pay only for what you use. Cloud gives the flexibility to pay by time or resources consumed, depending on the chosen pricing model. This is a boon for small to medium scale businesses, especially when they are trying to control their costs.
* Faster implementation: Application changes can be rolled out faster to the end user.
* Security: Cloud computing security, also known as cloud security is a set of policies, controls and technologies that will protect the data, applications and infrastructure involved in cloud computing. It involves information security, network security and computer security. Security is provided through different layers and protects critical information from data leakage, theft and deletion/manipulation.
* Availability: Cloud providers make resources like infrastructure, servers, storage and other services available for business. This is provided round the clock and with practically no downtime.
* Mobility: The need for mobility makes cloud computing possible. This gives reliability, robustness and scalability to the cloud infrastructure making it possible for the continuous and uninterrupted access to content and applications.
* Disaster recovery: The Cloud Disaster Recovery feature is a backup and restore feature that stores and maintains copies of all the electronic records saved in the cloud computing environment. It is an important security measure, where data can be securely transferred and stored, with user-authentication and password protection.
* Global network: Speed and efficiency are the hallmarks of cloud computing, and this is possible through the impeccable global network feature. There are multiple hubs around the world providing regional infrastructure and resources that gives users access to information within a matter of seconds.
* Loosely Micro service Architecture
* Software built as microservices can be broken down into multiple component services, so that each of these services can be deployed and then redeployed independently without compromising the integrity of an application. That means that microservice architecture gives developers the freedom to independently develop and deploy services.
* Better fault isolation; if one microservice fails, the others will continue to work.
* Code for different services can be written in different languages.
* Easy integration and automatic deployment; using open-source [continuous integration](https://apiumhub.com/tech-blog-barcelona/benefits-of-continuous-integration/) tools such as [Jenkins](https://apiumhub.com/tech-blog-barcelona/best-jenkins-plugins/), etc.
* The microservice architecture enables continuous delivery.
* Easy to understand since they represent a small piece of functionality, and easy to modify for developers, thus they can help a new team member become productive quickly.
* The code is organized around business capabilities.
* Scalability and reusability, as well as efficiency. Easy to scale and integrate with third-party services.
* Components can be spread across multiple servers or even multiple data centers.
* Work very well with containers, such as Docker.
* Complement cloud activities.
* Microservices simplify security monitoring because the various parts of an app are isolated. A security problem could happen in one section without affecting other areas of the project.
* Increase the autonomy of individual development teams within an organization, as ideas can be implemented and deployed without having to coordinate with a wider IT delivery function.
* Component based UI framework
* Component-based architecture that provides a higher quality of code.
* TypeScript: better tooling, cleaner code, and higher scalability.
* RxJS: efficient, asynchronous programming.
* The platform-agnostic philosophy.
* High Performance.
* Google Long-Term Support.
* Work flows and Rules engine
* A [workflow](https://www.cflowapps.com/workflow/) is a streamlined flow of multiple processes which when completed leads to the finished state of a project. The workflow engine is time-bound and is designed to carry out a sequence of tasks in a seamless manner.
* Declarative Programming
* Rule engines allow you to say "What to do" not "How to do it".
* The key advantage of this point is that using rules can make it easy to express solutions to difficult problems and consequently have those solutions verified (rules are much easier to read then code).
* Rule systems are capable of solving very, very hard problems, providing an explanation of how the solution was arrived at and why each "decision" along the way was made (not so easy with other of AI systems like neural networks or the human brain - I have no idea why I scratched the side of the car).
* Logic and Data Separation
* Speed and Scalability
* Centralization of Knowledge
* By using rules, you create a repository of knowlegde (a knowledgebase) which is executable. This means it's a single point of truth, for business policy (for instance) - ideally rules are so readable that they can also serve as documentation.
* Tool Integration
* Explanation Facility
* Rule systems effectively provide an "explanation facility" by being able to log the decisions made by the rule engine along with why the decisions were made.
* Understandable Rules
* DevSecOps
* Cost reduction is achieved by detecting and fixing security issues during the development phases which also increases the speed of delivery.
* Speed of recovery is enhanced in the case of a security incident by utilizing templates and pet/cattle methodology.
* Threat hunting can avoid bad publicity, and therefore can potentially increase sales – it is obviously easier to sell a secure product.
* Improved overall security by reducing vulnerabilities, reducing insecure defaults, and increasing code coverage and automation through the use of immutable infrastructure
* Keeping in step with the frantic innovation intrinsic to cybercrime by effectively managing security auditing, monitoring, and notification systems.
* ‘Secure by design’ principle is ensured by using automated security review of code, automated application security testing, educating, and empowering developers to use secure design patterns.
* Everyone is responsible for security. DevSecOps fosters a culture of openness and transparency, and does so from the earliest stages of development.
* The ability to measure different things which can be seen by everyone – DevSecOps enables a culture of constant iterative improvements.

## Area addressed with Transformation

* UI / UX | Latency | Search | Validation
* Issues with page loading - Hanging of system
* Rig booking in Maps and Calendars
* No spinners are rendering in the application
* Menu item issues
* Incorrect status counts on the menu bar
* Page performance
* Workflow process issues - Sometimes appropriate status is not reflected
* Search issues
* Performance issues in all pages
* Page rendering issues on large bookings
* Synchronization calls while creating the PNRs, by using the auto & manual bookings
* Manual intervention
* Notifications
* Booking alerts
* E-ticket alerts
* General alerts
* Random behavior
* Manual deployments
* Different behavior in environments
* Ticket Window
* Sync issues, Sometimes CrewLink is completely down.
* Synchronization calls while creating the PNR’s by using the auto & manual bookings
* Email alerts issues
* Booking alerts, E- Ticket alerts, General alerts, group email alerts
* Work flow process issues
* Sometimes status are not changed appropriate,
* When user has 1PNR,2Quotes, and when customer approves PNR the Quotes are not moving into the Reject status for flight and hotel bookings
* Search issues
* Search button is not working properly on travel Request booking page.
* Search response is very slow under travel request search booking page
* Improper search results : When user clicks on search button under travel request page, Travel request is directly getting opened instead of producing exact search result.
* In Advanced search, while adding existing passengers: Advanced search link is not working.
* Login issues
* Login page loading is taking more than expected time
* Performance issues in all the pages
* Page loading issues in almost all the web pages, Page refreshing is very slow and error pop-up displays
* Page redirection errors: When user clicks on Customer Region/When user selects Customer region from dropdown in customer setup>Getting HTTP status 500 Error
* Page renderings on big bookings
* Compatibility issues
* Drop down navigation links are overlapping on main navigation,
* Alignment issues
* Filed box is not aligned properly for “Journey Type” filed in Flight – Itinerary Details section.
* Validation issue
* While TR booking, change the customer name and try to click on “Proceed to Booking”, pop-up is displaying with Manual Mod or Semi Auto Mode, after clicking on   manual mode, it is displaying error i.e, mandatory fields to be filled.
* Actual error: Pop-up is opening to select the mode to booking
* Expected Error: Without filling the mandatory fields in the screen, it should not open the pop-up for booking mode (Without giving last name, proceed to booking pop-up is displayed)
* Improper error messages
* Error Description for the errors found is not provided in all the pages.

* Issues with the loading the pages- Hanging the system
* Map & Calendar view issues for the rig booking
* No spinners are rendering in the application.
* Menu item issues
* No significance place for enhancements, same enhancements are here and there example email configurations

# Objective

The key objectives for the transformation is to

* Convert existing monolithic architecture to micro service based architecture.
* Component based UI Design.
* Migration from on premise to Azure cloud.
* Implementation of Devops
* Implementation of GDPR and PbD based development.

## Detailed functional objectives

* Login
* oAuth based and OTP via email
* Client Specific Theme-Management
* Custom theme configuration
* Assets, page and application level theme customization
* User Specific Dashboard
* On-screen snapshot of bookings
* One Place Navigation
* Request processing
* Role based actions reminders
* Notifications
* Customer On-Boarding
* Upload of Customer Organization
* Initial Customer generation with minimal data
* Ability for the customer to update the profile data
* Customer Profile Management
* Ability to quickly add / modify Customer data
* Reminder notifications for completion of profiles
* Master data Audit history
* Master Data improved validations
* Improved and Optimized Customer data
* Booking Management

Voyage planning |Crew Rotation |Booking |Booking Modification |Booking Confirmation |Communication

* Improved UX for Voyage and Crew rotation and planning
* Booking History for minimization of conflict management
* Clone from existing bookings to create new
* New existing Workflow Engine framework implementation
* New existing Rules Engine framework employment
* Map and Calendar based visualization, with re-organized data models
* Message queuing for In-Application communications
* New existing Event hubs for in-app events
* Highly secured data encryption
* PNR Sync
* Elimination of XML/JSON visualization and display in a readable format
* Auto and Manual PNR sync
* Improved On-Page Validation Messages
* Display error in readable format
* In-Application Search Engine
* Reorganized database schema
* Notification
* Office 365 Email Dashboard, with ability to monitor and resend
* Email notifications for various events to be templatized

# Definitions, Acronyms and Abbreviations

## Definitions

|  |  |
| --- | --- |
| **Terms** | **Definition** |
|  |  |
|  |  |

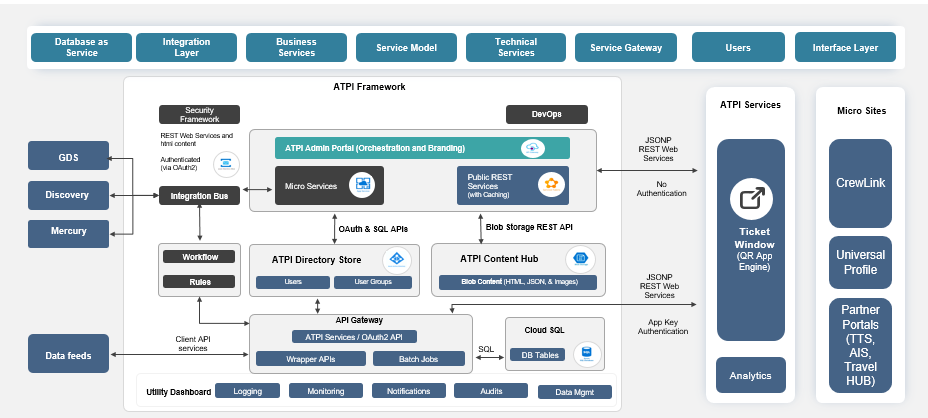
## Acronyms and Abbreviations

|  |  |
| --- | --- |
| **Acronym** | **Meaning** |
| AD | Active Directory |
| API | Application Programming Interface |
| CDN | Content Delivery Network |
| UX | User Experience |
| UI | User Interface |
| GDS | Global Distribution System |
| CL | Crew Link |
| GDPR | General Data Protection Regulation |
| PbD | Privacy by Design |
| SFC | Service Fabric Cluster |
| RG | Resource Group |
| MDS | Master Data Service |
| HBE | Homeport Booking Engine |
| CSR | Client side Rendering |
| SSR | Server side rendering |
| BDD | Behavior driven development |
| DDD | Domain Driven design |
| TDD | Test driven design |

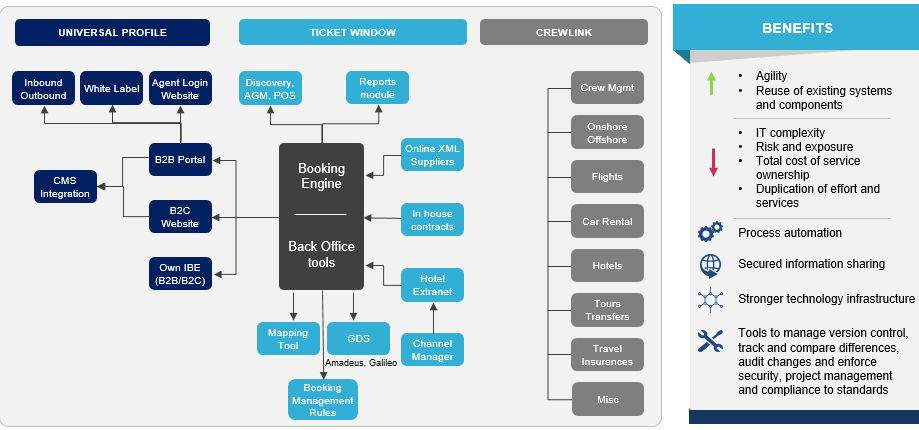
# 

# Architectural Representation

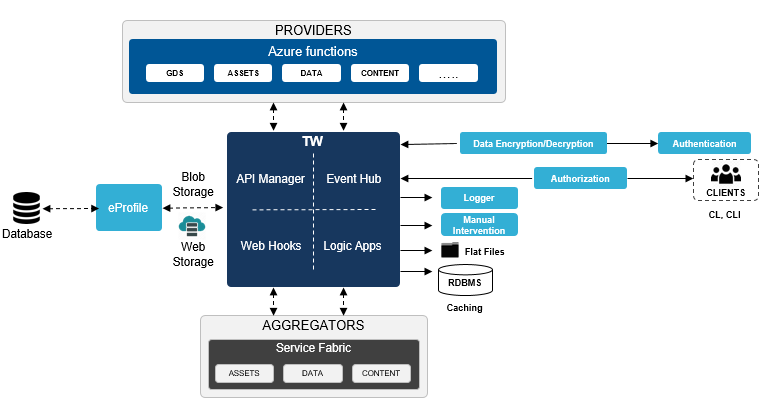
## Proposed technical architecture – ATPI application framework



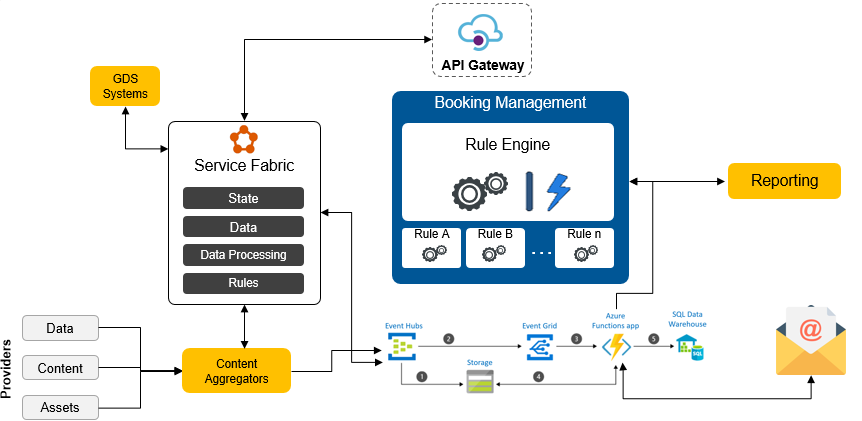
## Proposed functional Architecture



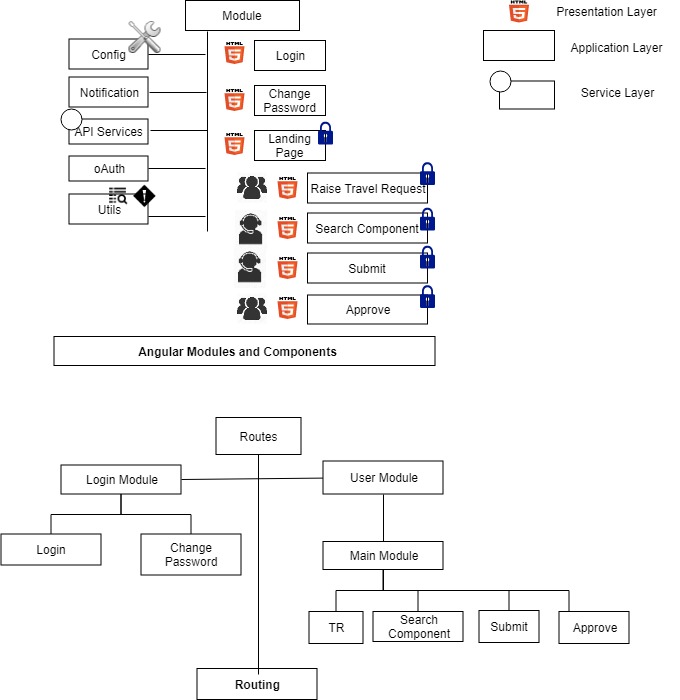
## Technical – Functional relationship

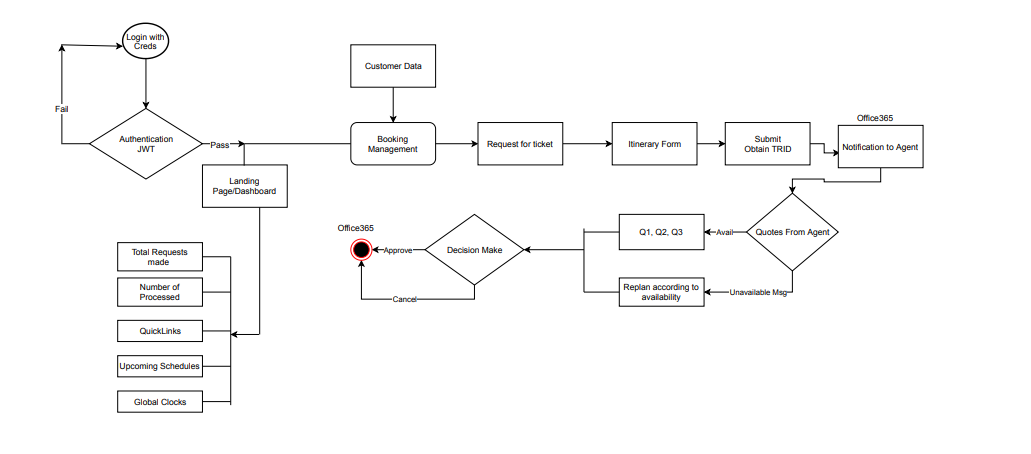


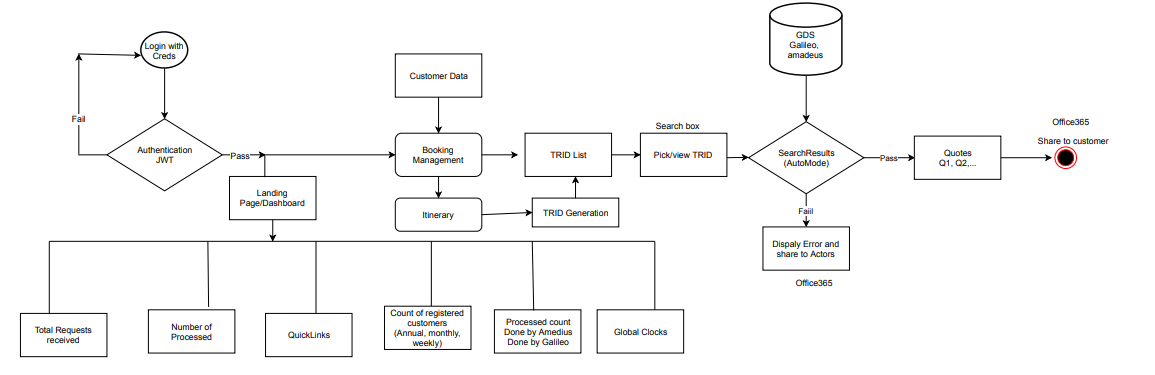
## Component level Technical – Functional relationship



## Frontend (Angular) Architecture



**Customer Process Flow**

**Agent Process Flow**

# Scope

## Scope of work

* UX and UI Development
* Authentication and Authorization
  + Customer and Agent roles one each
* Travel request generation limited to Flights
* GDS integration as provided by ATPI
* Request Processing for GDS
* Email Notifications

## Assumption

* GDS implemented services will be provided by ATPI
* CL MVP backend interacts with the above services to fetch data
* Basic Master Data required for the current scope will be implemented.
* Manual on boarding limited to user login, roles

# Proposed Technology Stack

* Hosted Blob Storage
* Service fabric Clusters
* Load balancing
* API Management
  1. API Gateway Implementation
  2. Gain insights into APIs
  3. Connect to back-end services
  4. Secure and optimize APIs
  5. Documentation using Swagger
* Database implementation
  1. Implementation of MDS
  2. Application specific database will be refactored and normalized based on Micro service architecture
* DB Schema
  1. Normalization \*
  2. Shared. Data in a database are shared among different users and applications.
  3. Permanent or Persistence. Data in a database exist permanently in the sense the data can live beyond the scope of the process that created it.
  4. Validity/integrity/Correctness. Data should be correct with respect to the real world entity that they represent.
  5. Security. Data should be protected from unauthorized access.
  6. Consistency. Whenever more than one data element in a database represents related real world values, the values should be consistent with respect to the relationship.
  7. Non-redundancy: No two data items in a database should represent the same real world entity.
  8. Independence. Data at different levels should be independent of each other so that the changes in one level should not affect the other levels.
  9. Easily Accessible. It should be available when and where it is needed i.e. it should be easily accessible.
  10. Recoverable. It should be recoverable in case of damage.
  11. Flexible to change. It should be flexible to change.
* Email Notifications
  1. Office 365 API Implementation
  2. Office 365 Admin Dashboard
* Logic Apps
  1. simplify and implement scalable integrations
  2. Model and Automate workflow steps visually
  3. Integration across services and protocols
  4. Event based system can trigger any combinations actions, conversions, and condition logic

# Development Tool

## Front End Development

* Node.js version 8.x or 10.x
* npm packages
* npm client
* Visual Studio Code
* Angular CLI

## Middleware Development

* Visual Studio 2017 Professional
* Azure Subscription
* SSMS

## Backend Development

* Sql Server 2014 and above

## Source Control

* Git HUb

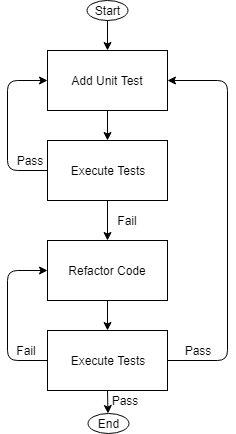
## Development Approach

* Domain Driven Development

Domain Driven Design advocates modeling based on the reality of business as relevant to our use cases. As it is now getting older and hype level decreasing, many of us forget that the DDD approach really helps in understanding the problem at hand and design software towards the common understanding of the solution. When building applications, DDD talks about problems as domains and subdomains. It describes independent steps/areas of problems as bounded contexts, emphasizes a common language to talk about these problems, and adds many technical concepts, like entities, value objects and aggregate root rules to support the implementation.

* Test Driven Development

Test-driven development starts with developing test for each one of the features. The test might fail as the tests are developed even before the development.



**Microservice** : , xUnit.NET - xUnit.NET has taken a pretty unique, modern, and flexible approach to unit testing. It changes terminology, so you no longer define TestFixtures and Tests...you specify Facts and Theories about your code

You are free to extend the framework to support BDD-style Concern/Context/Observation specifications

**AngularJs**: Jasmine & Karma - Jasmine is framework for testing and Karma is the test runner which spawns a web server that executes source code against test code for each of the browsers connected.

**End to End Testing**: Protractor - Protractor is an end-to-end test framework for Angular and AngularJS applications. Protractor runs tests against your application running in a real browser, interacting with it as a user would without depending on other tools for performing the same.

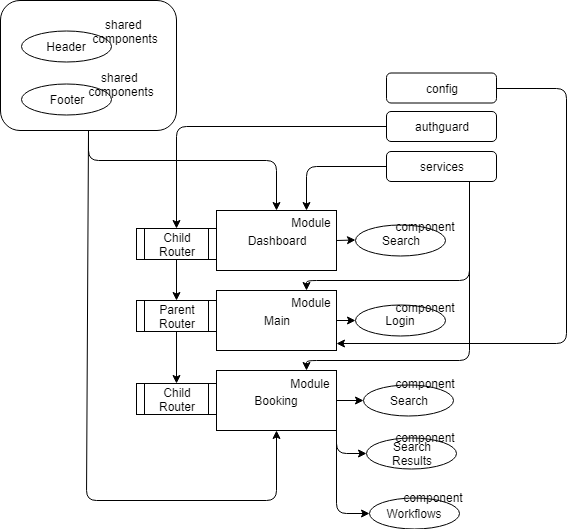
# Non Functional Requirement

* Security
* Logging
* Exception Handling
* Storage
* Configuration
* Performance
* Flexibility
* Disaster recovery
* Accessibility

# Pre-requisites

* Continued support on the following
  + Requirements analysis and design
  + Understanding of production issues
  + Database architecture and reports requirements
  + Requirements gathering and scheduling
  + Access to existing servers / services (IT and Networking)
* Azure platform for SaaS
* Access to existing source code and applications
  + SQL Server Database Access
  + Mercury Web service API’s source code
  + POS Plugin Web service API src / documentation
* Documents pertaining to
  + Current architecture
  + System integration (Application, Infra, Data, DevOps)
  + GDS API Documentation
* Initial knowledge transfer from PO for transition and support until completion of transformation, for application maintenance
* Support from Delphi engineers for migration
* Dedicated availability from Product Owner and key stakeholders, for smooth execution

# Front end Architecture using Angular 7



* Modules:

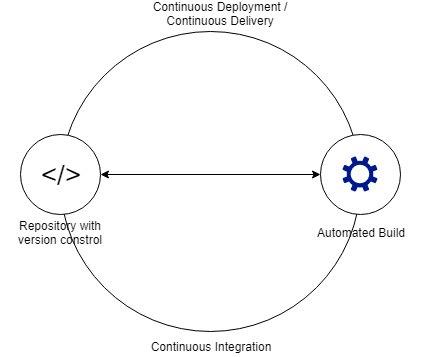
Module organizes an application and extends it with capabilities from external libraries and consolidates components, directives, and pipes into cohesive blocks of functionality, each focused on a feature, application business domain, workflow, or common collection of utilities. Modules can add services to application developed internally or by external sources. Lazy loading of modules asynchronously by router improves the performance of the overall application by only loading the necessary modules required. An entry bootstrap module launches the application.

* Component

A component is a class which is defined to be visible in any elements in the screen or browser. It is the most granular unit in the application usually independent of other components and can be reused in any other application. Each component defines a class that contains application data and logic, and is associated with an HTML *template* that defines a view to be displayed in a target environment. Easy integration and reusability are the major advantages of the components.

* Services and Dependency injection
* Service is any value, function, or feature that an app needs. A service is typically a class with a well-defined purpose. It should do something specific and do it well. The tasks that need be delegated by components like fetching data from the server, validating user input, or logging are handled by services and specific responsibilities. Dependency injection enables to factor application logic into services and make them available to components. For data or logic that isn't associated with a specific view, and that you want to share across components, you create a *service* class. A service class definition is immediately preceded by the @[Injectable](https://angular.io/api/core/Injectable) () decorator. The decorator provides the metadata that allows your service to be *injected* into client components as a dependency. *Dependency injection* (DI) lets you keep your component classes lean and efficient. They don't fetch data from the server, validate user input, or log directly to the console; they delegate such tasks to services.
* Template and data binding: A template combines HTML with Angular markup that can modify HTML elements before they are displayed. String interpolation is one way of data binding where template (HTML) receives the data from the components properties. Property binding is another one way data binding where the modifications are applied at the component level. Event binding is the updation/sending of the value/information of a certain variable from the presentation layer (view) to the component (model). Finally two way data binding is a combination both event and property binding.
* Routing: The Angular [Router](https://angular.io/api/router/Router) NgModule provides a service that lets you define a navigation path among the different application states and view hierarchies in your app.

# DevOps



Devops cycle involves following:

* Source code repositories
* Continuous integration
* Build
* Continuous Delivery or Continuous Deployment

The mode of implementation will involve evolution with following steps:

* Source code will be pushed to the git repositories.
* Merged source code will be pushed to respective Azure services from MSVS
* Merged source code will be pushed to hosted blob storage for CSR from Visual Studio code.
* Azure DevOps will be implemented
  + Integration of git to Azure DevOps
  + Pipelines
    - Tasks
      * Build pipelines
        + Build Automation
        + unit cases
        + Source code analysis in MSVS
      * Release pipelines
        + Automated test cases
    - Jobs

Series of tasks that run sequentially on the same target. Target is either an agent or the server.

* + - Conditions
      * Conditions or steps under which a job will run.
    - Expressions
      * Describe decisions the system should make
    - Variables
      * Data into various parts of the pipeline.
    - Resources

Outside pipe lines

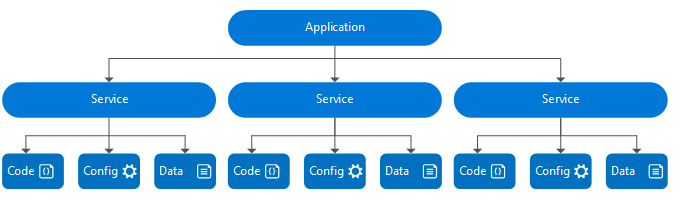
* + - * secure files
      * variable groups
      * service connections
      * agent pools
      * other repositories
      * containers
    - Templates
      * Reusable logic as YAML
    - Scripting
      * Cross platform scripts
    - Library
      * Variable groups
      * Task Groups
        + Standardize and centrally manage deployment steps
      * External Service Connections
      * Secure files
        + Security files
  + Continuous Delivery implementation
  + Considering all the factors continuous deployment will be employed

Infrastructure:

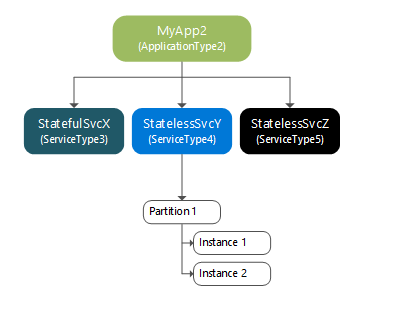
* Micro-service deployment .Net C# Code
  + Service Fabric Clusters
* Angular Deployment
  + CSR Deployment
    - Hosted Blob Storage

**Service Fabric Clusters**

An application is a collection of constituent services that perform a certain function or functions. A service performs a complete and standalone function and can start and run independently of other services. A service is composed of code, configuration, and data. For each service, code consists of the executable binaries, configuration consists of service settings that can be loaded at run time, and data consists of arbitrary static data to be consumed by the service. Each component in this hierarchical application model can be versioned and upgraded independently.



There can be one or more instances of a service type active in the cluster. Replication essentially provides redundancy for the service to be available even if one node in a cluster fails. A partitioned service further divides its state (and access patterns to that state) across nodes in the cluster.



Scaling in Service Fabric is accomplished by:

1. Scaling by creating or removing stateless service instances
2. Scaling by creating or removing new named services
3. Scaling by creating or removing new named application instances
4. Scaling by using partitioned services
5. Scaling by adding and removing nodes from the cluster
6. Scaling by using Cluster Resource Manager metrics

Creating a stateless service requires defining an InstanceCount. The instance count defines the number of instances of the stateless service's application logic that should be running in the cluster. Increasing the number of instances is the recommended way of scaling out a stateless service.

When an instance of a stateless named-service fails, a new instance is created on an eligible node in the cluster. For example, a stateless service instance might fail on Node X and be re-created on Node Y.

**Hosted Blob Storage (CSR)**

Default file and optionally custom 404 page will need to be specified. As the feature is enabled, a container named $web is created if it doesn't already exist.

Files in the $web container are:

* served through anonymous access requests
* only available through object read operations
* case-sensitive
* available to the public web per settings specified

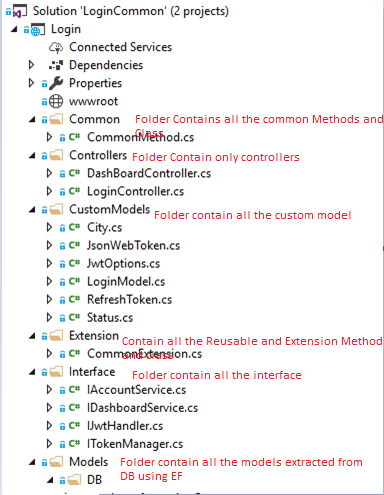
# Microservice Technical Architecture

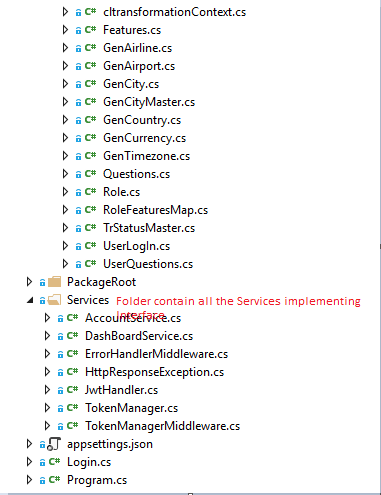
The overall source project structures for micro services and angular based front are depicted below. The non-functional requirements like logging, exceptional handling will be implemented during the development phase. The exiting ng npm modules will be employed to achieve the same in the Angular. For Microservices nLog and application insight will be employed along with generic custom exception handling for error handling.

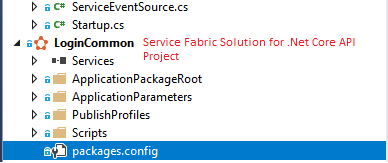
Data security will be achieved using the Azure services for data at rest and transit. Data at rest will be employed using Azure’s Service encryption using service managed keys. Data in transit will be implemented using the best practices suggested by Azure.

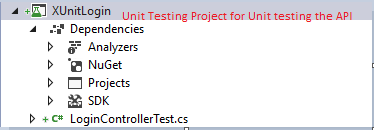
Azure services will be utilized to achieve the data storage, cloud SQL and caching services. The other factors considered are below:

* Configuration
* Performance
* Flexibility
* Disaster recovery
* Accessibility

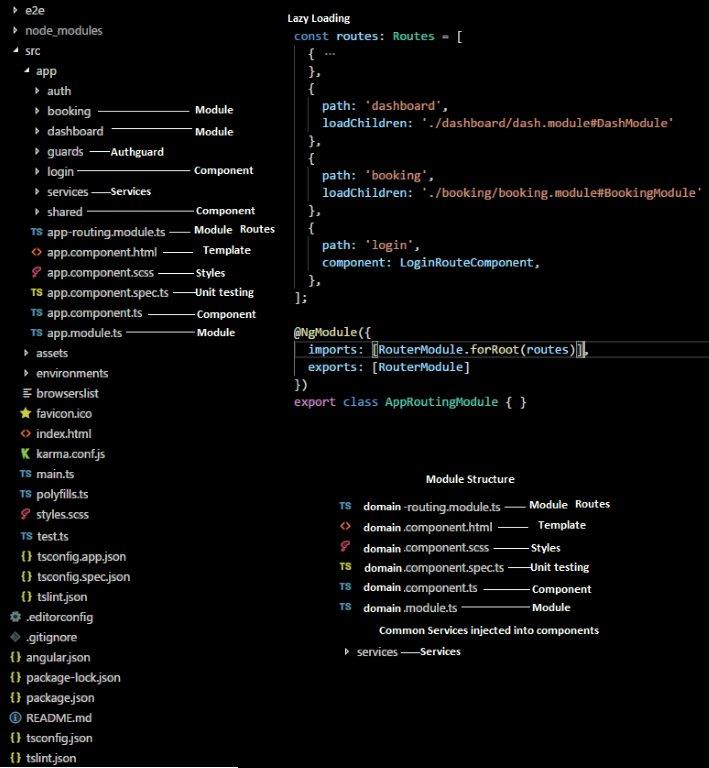








# Frontend Technical Architecture



# Features

## Login

**Access Token:**

* Can be of any format that the Auth Server Likes.
* May have information decodeable by the clients.

**Refresh Token:**

* Can be of any format that the Auth Server likes
* MUST NOT have information readable by the clinets.

**JWT: Set of claims (Protected information) encoded in a json object**

Claims maybe digitally signed and/or encrypted (why would you not?)

Full form JSON Web token.

### Approach:

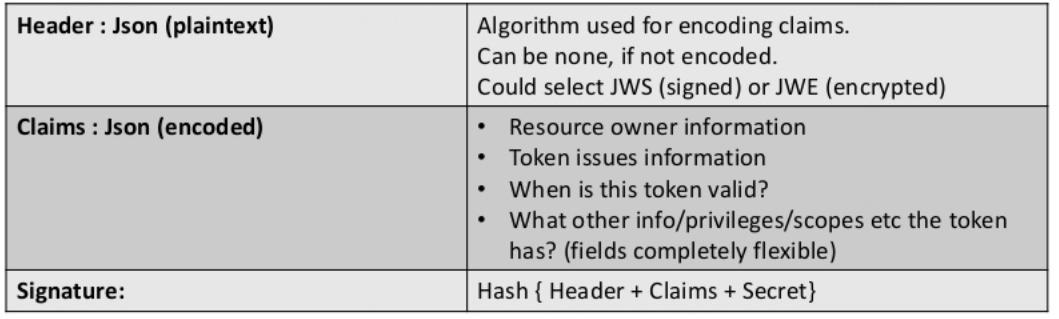
Access Token - Use a public/private key pair

* Others would like to decode your token.
* Public Key should be accessible to all intereseted.

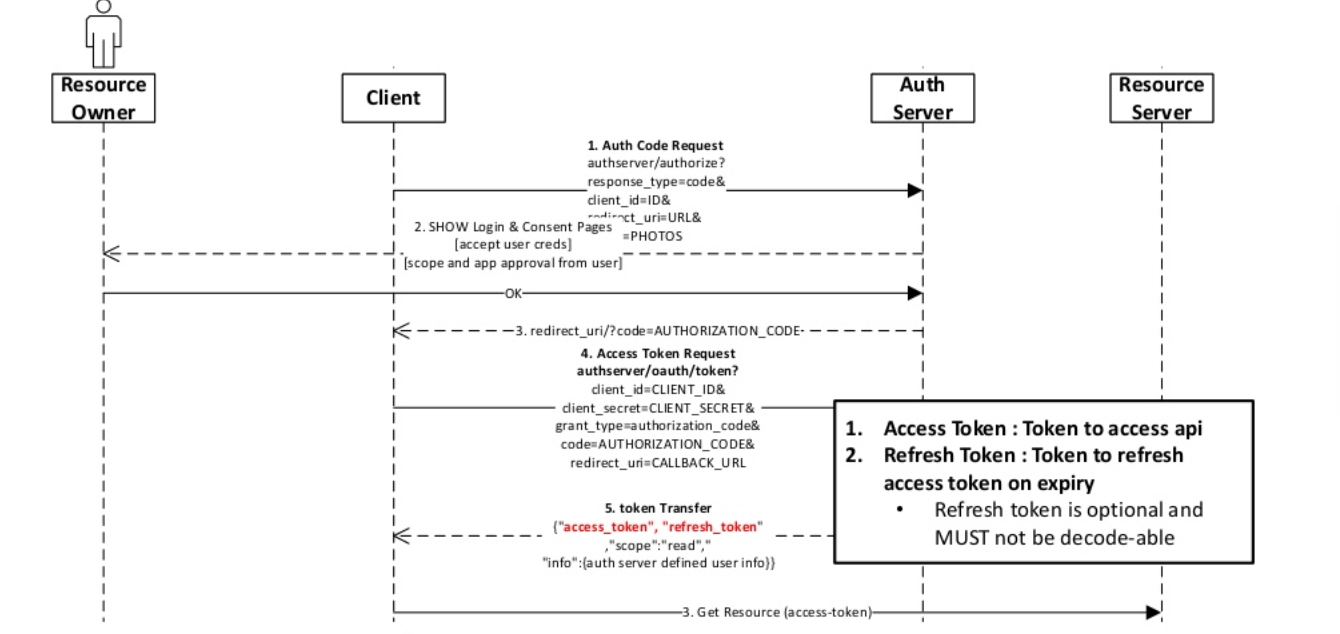
Refresh Token – Use a private/Secret key

* MUST be encrypted using a salt or a private key.
* Public key of refresh token should not be queryable.
* WHY? Refresh token is used to generate an acess token
* No Info useful in it, except for the Auth Server
* Only Auth Server should decode refresh key.

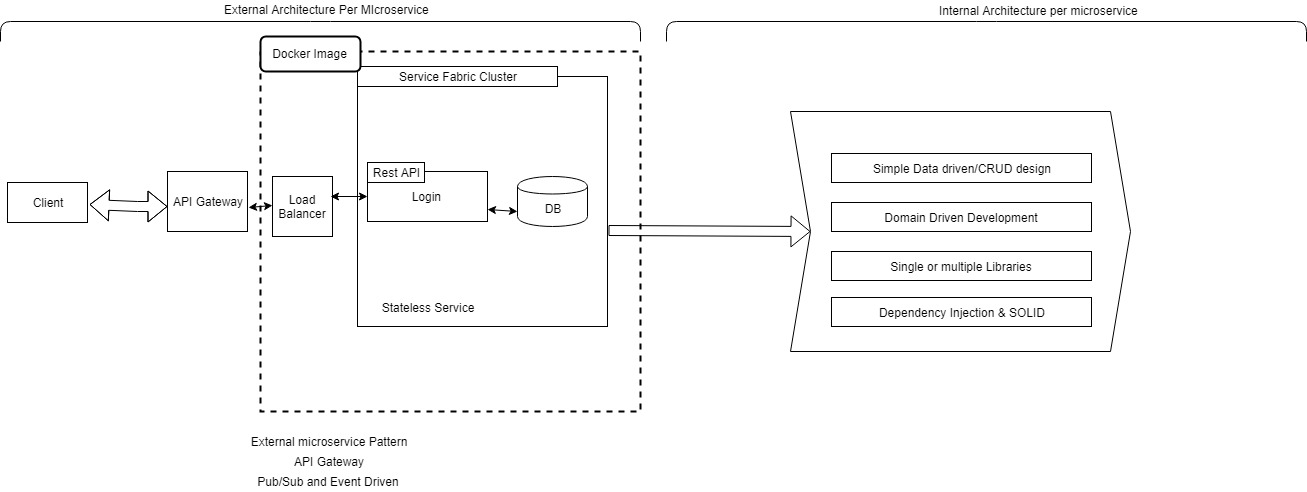
### Purpose



### Sequence Diagram



### Low level architecture

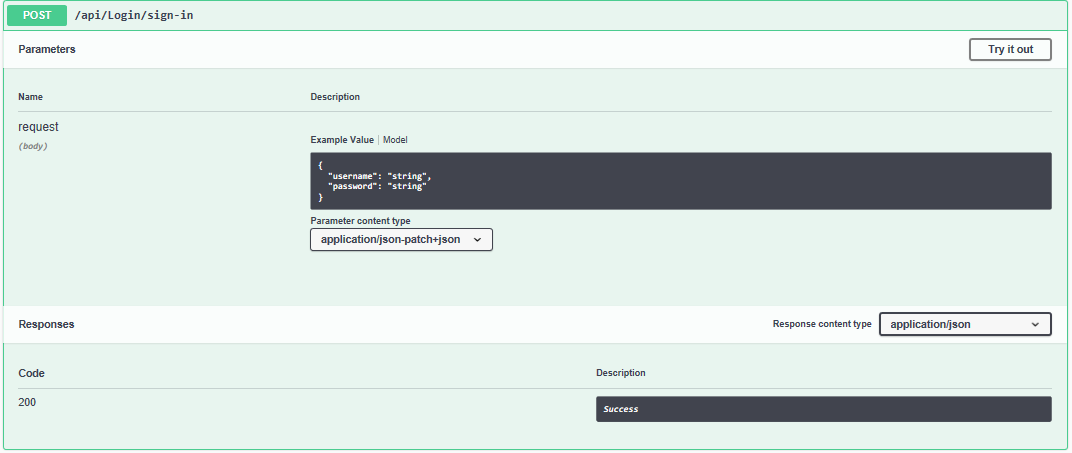


### Database Diagram

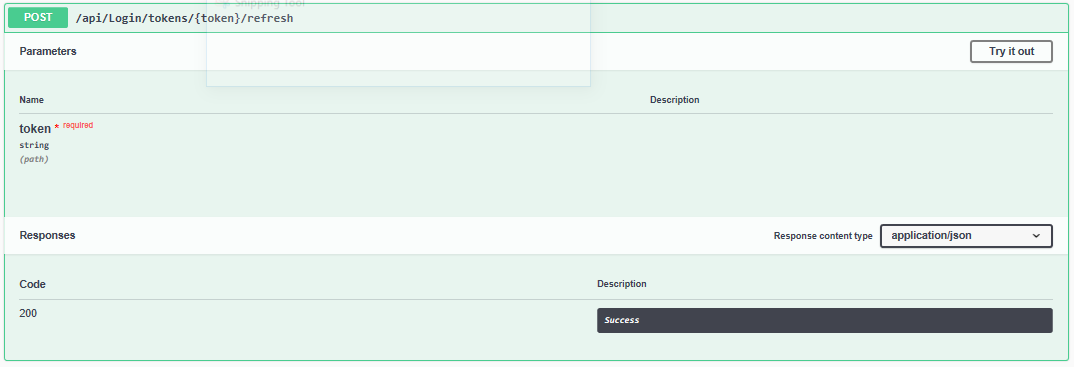


### API Documentation

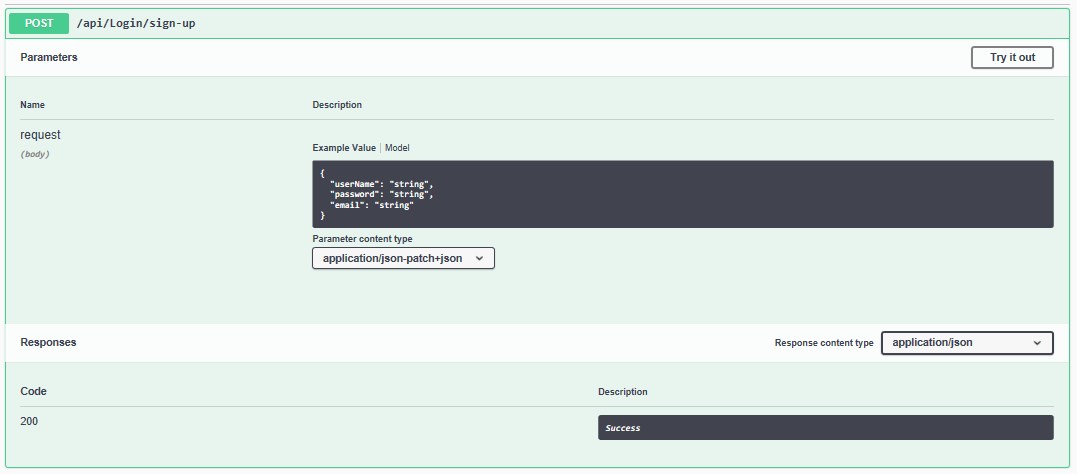
* 1. **/api/Login/sign-in**



* 1. **/api/Login/tokens/{token}/refresh**



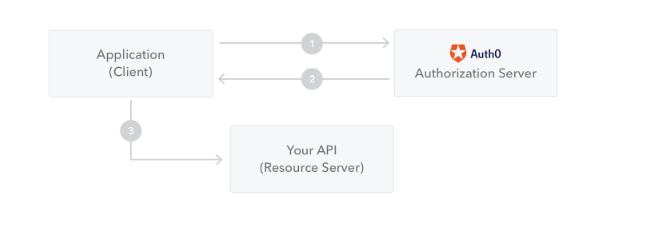
* 1. **/api/Login/sign-up**



### GDPR and PbD Factor

## Authorization

This is the most common scenario for using JWT. Once the user is logged in, each subsequent request will include the JWT, allowing the user to access routes, services, and resources that are permitted with that token.



### Purpose

JSON Web Tokens consist of three parts separated by dots (.), which are:

* Header
* Payload
* Signature

Therefore, a JWT typically looks like the following.

xxxxx.yyyyy.zzzzz Let's break down the different parts.

**Header**

The header typically consists of two parts: the type of the token, which is JWT, and the signing algorithm being used, such as HMAC SHA256 or RSA.

**Payload**

The second part of the token is the payload, which contains the claims. Claims are statements about an entity (typically, the user) and additional data. There are three types of claims: registered, public, and private claims.

**Registered claims:** These are a set of predefined claims which are not mandatory but recommended, to provide a set of useful, interoperable claims. Some of them are: iss (issuer), exp (expiration time), sub (subject), aud (audience), and others.

Public claims: These can be defined at will by those using JWTs. But to avoid collisions they should be defined in the IANA JSON Web Token Registry or be defined as a URI that contains a collision resistant namespace.

Private claims: These are the custom claims created to share information between parties that agree on using them and are neither registered or public claims.

### Approach

Whenever the user wants to access a protected route or resource, the user agent should send the JWT, typically in the **Authorization** header using the **Bearer** schema.

 The server's protected routes will check for a valid JWT in the Authorization header, and if it's present, the user will be allowed to access protected resources. If the JWT contains the necessary data, the need to query the database for certain operations may be reduced, though this may not always be the case.

If the token is sent in the Authorization header, Cross-Origin Resource Sharing (CORS) won't be an issue as it doesn't use cookies.

We will use Private claims by querying database in JWT token.

### Database Diagram

Refer 13.1.5

### GDPR and PbD Factor

TBD

## Customer - Landing Page

TBD

### Purpose

TBD

### Approach

TBD

### Database Diagram

TBD

### GDPR and PbD Factor

TBD

## Agent - Landing Page

TBD

### Purpose

TBD

### Approach

TBD

### Database Diagram

TBD

### GDPR and PbD Factor

TBD

## Customer – Raise Travel Request

TBD

### Purpose

TBD

### Approach

TBD

### Database Diagram

TBD

### GDPR and PbD Factor

TBD

## Customer – Approve Travel Request

TBD

### Purpose

TBD

### Approach

TBD

### Database Diagram

TBD

### GDPR and PbD Factor

TBD

## Agent – Search Travel Request

TBD

### Purpose

TBD

### Approach

TBD

### Database Diagram

TBD

### GDPR and PbD Factor

TBD

## Agent – Submit Travel Request

TBD

### Purpose

TBD

### Approach

TBD

### Database Diagram

TBD

### GDPR and PbD Factor

TBD

## Notification

TBD

### Purpose

TBD

### Approach

TBD

### Database Diagram

TBD

### GDPR and PbD Factor

TBD

# Third party integration

GDS via exiting HBE

# Risks, Constraints, Limitations & Issues

## Risks

## Constraints

<Specify the Constraints>

## Limitations

<Limitations for the design need to be specified>

## Issues

## Dependencies

|  |  |
| --- | --- |
| **Item** | **Description/Reference** |
|  |  |

# 

# Appendix

## Project Standards and Conventions

|  |  |
| --- | --- |
| **Item** | **Description/Reference** |
|  |  |

## Development Environment Parameters

|  |  |  |
| --- | --- | --- |
| **Environment** | **Parameter Name** | **Value** |
|  |  |  |