# Overview

Intuit supports below products for Small Business owners:

1. Quickbooks (QB) for Accounting
2. QB Payroll for paying small business employees
3. QB Payments for small business to receive payments from their customers
4. TSheets for time-tracking

Customers can subscribe to any/all of the above products and by doing so, maintain a shared/common business profile across all of the products. Customer can update their business profile from any of the above product workflows. Design and build a Business Profile service where every profile Create/Update request needs to be approved by all the subscribed products of the user.

# Requirements

## Functional Requirements

1. As a user, I should be able to create my business profile from any of the subscribed product website.
2. As a user, I should be able to update my business profile from any of the subscribed product website.
3. As a system, I should validate the user update requests with all subscribed products and only update when all the requests have been validated.

## Non-Functional Requirements

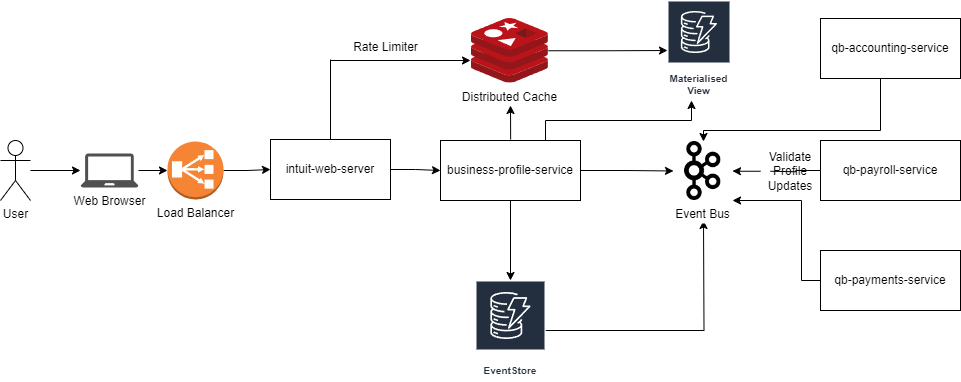
* Scalability: The system should be scalable horizontally. New instances of Web Servers can be added behind the Load Balancer. Kafka, Redis, and DynamoDB offer scalable solutions by default.
* Availability: By using the Load Balancer and distributed storage solutions like DynamoDB and Kafka, the system aims for high availability.
* Security: SSL termination at the Load Balancer, encryption at rest for data stores, and secure access controls for services.
* Monitoring and Logging: Integrate with monitoring solutions like Prometheus, Grafana, and logging solutions like ELK Stack or AWS CloudWatch.

# Assumptions

1. Business Profile Service is only responsible for managing the profile details of the service. The business credentials and product subscription are managed by different microservices.
2. The updated profile will not reflect unless it is approved by all subscribed products.
3. All the Authentication and Authorization is already handled at the BFF level.

# High Level Design

## System Diagram



## Components

1. **Load Balancer:** In a high load distributed environment, the Load balancer will ensure that the requests are highly available. It will help to efficiently distribute incoming web traffic across multiple instances of Web Servers.
2. **intuit-web-server:** Serve as the entry point for client requests, handle authentication, authorization, rate limiting, and route requests to the Business Profile Microservice and other downstream services.
3. **business-profile-service**: Provide isolated business profile operations and serve data from the underlying data storage systems. It handles requests related to creation, updation, and retrieval of business profiles.
4. **Redis Cache:** In-memory data structure store, used as a cache. It stores the business profile data that is frequently accessed. The business-profile-service fetches data from Redis for faster retrieval and puts data into Redis after fetching from or updating the DynamoDB materialized view table.
5. **DynamoDB (Materialized View):** Store the latest state of business profiles for quick CRUD operations.
6. **DynamoDB (Event Log Table):** Captures and stores the event logs pertaining to business profile changes. Any new event in this table is transmitted to the event bus and eventually reflects on the materialized view after approved validation.
7. **Kafka (Event Bus):** Serve as the central event bus for asynchronous messaging and event-driven architecture component.

## Data Flow

1. **User Request**: A user sends a request to access or modify a business profile.
2. **Load Balancer**: The request reaches the Load Balancer, which routes it to one of the available BFF instances.
3. **Intuit-web-server (BFF)**: It authenticates the user and authorizes the request based on user roles or permissions. Once validated, it routes the request to the business-profile-service.
4. **business-profile-service**:
   * For reads: It checks if data is available in Redis. If not, fetch from DynamoDB and store it in Redis for future use.
   * For writes/updates: It make necessary updates in the event log DynamoDB. It then produces a message to Kafka topic to notify of the change. Update the relevant record in DynamoDB table and cache in Redis.

## Important Design Choices

### CQRS and Event-Sourcing Pattern

As per the requirement, any change to the business profile needs to be validated by all the product systems the business is subscribed to. Rather than all the validations being processed synchronously, the commands can be handled asynchronously via the Event-Sourcing patter with CQRS.

Additionally, the system has different performance characteristics for read and write operations. This will help scale the systems independently. In our use case, instead of updating the record directly, we store event logs representing the changes. This helps prevent concurrent updates from causing conflicts as it avoids directly updating the resultant datastore. It also manages an audit log for all the changes made that allows for rollbacks if required.

### Redis Cache over In-Memory Cache

#### Pros:

* Redis offers optional persistence, which allows the data to survive server restarts. In case of in-memory cache, data is lost on instance restart.
* Redis can be configured to run in a distributed setup to handle larger datasets and provide higher availability.
* Redis provides mechanisms to periodically save data on disk.
* It is easier to manage stale data in Redis cache. Any updates on the Datastore can be handled. In case of in-memory cache, it would be difficult to manage stale data on each of the instance running the service.

#### Cons:

* Requires the setup and management of an external service.
* Even though Redis is fast, there's an added network latency compared to in-application in-memory caching.
* There will be additional costs associated with infrastructure
* Redis is memory-bound, meaning the amount of data it can store is limited by the server's memory. Storing vast amounts of data can become expensive.

### Backends for Front-end Pattern

Currently, we only have a single web application. If in future we want to expand to another application types (mobile application etc.,) this will help isolate the authentication/authorization requirements for the different application types. This will also enable better customization specific tot the user interface, including masking specific data.

This also allows for fine-tuning the behaviour and performance without affecting other front-end application types. As a result, it will be smaller, less complex, and likely faster than a generic backend that tries to satisfy the requirements for all interfaces

# Low Level Design

## Create/Update a Business Profile

### API Contract

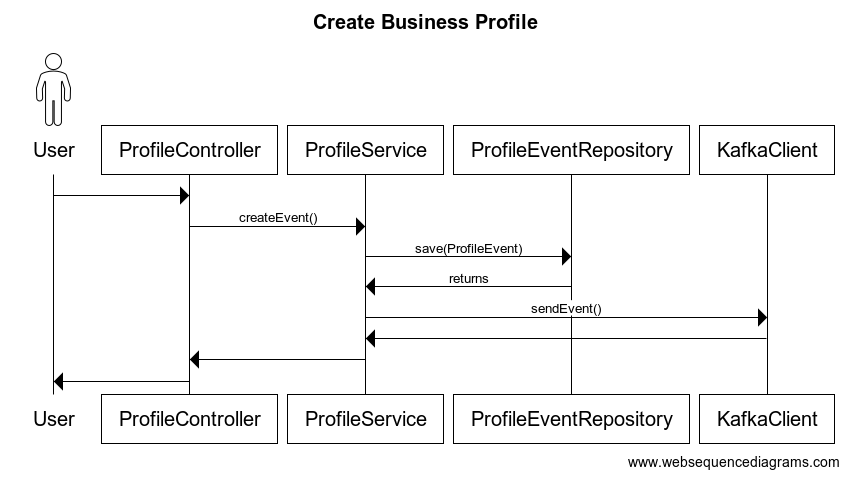
* **API Route:** POST /v1/profile/event

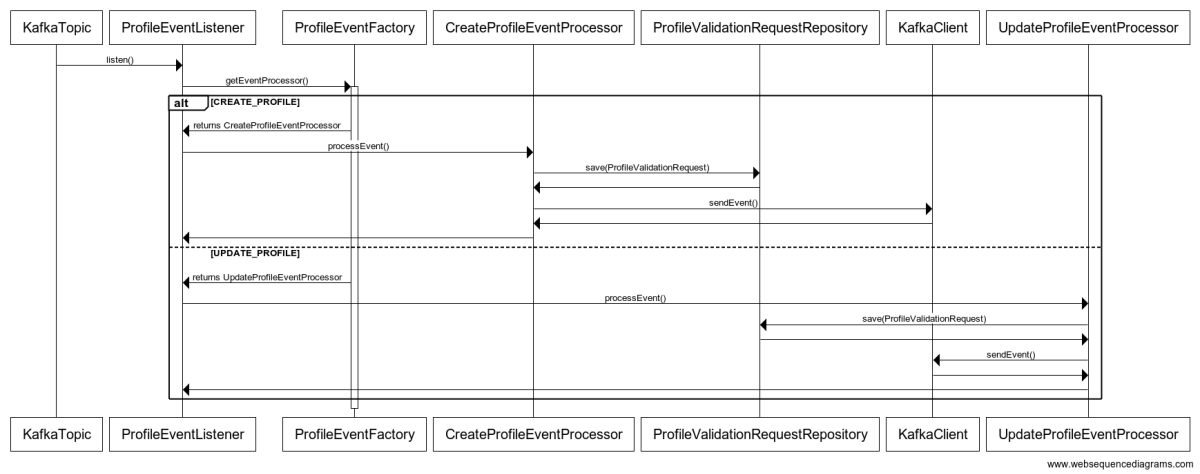
#### Request Parameters

* **Request Header:** X-Business-Id

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Description** |
| eventType | EventType | Enum:- CREATE\_PROFILE/UPDATE\_PROFILE |
| Payload | ProfileDetails | Business profile attributes |
| Source | Product | Enum:- ACCOUNTING/PAYROLL/PAYMENTS |

### Sequence Diagram





### Flow

* 1. User invokes the createEvent API.
  2. The service saves the event payload in ProfileEvent DynamoDB Table
  3. Sequentially, it sends the event to business-profile-event topic.
  4. ProfileEventListener consumes the event, create an instance of CreateProfileEventProcessor or UpdateProfileEventProcessor based on eventType in payload.
  5. For CREATE\_PROFILE event, the event processor checks if a validation request already exist for the business. If it exists, throw an exception else, store the validation request in ProfileValidationRequest table.
  6. For UPDATE\_PROFILE event, the event processor stores the validation request in ProfileValidationRequest table.
  7. After storing the request, the event processor sends a Kafka event to topic business-profile-validator
  8. The events are then consumed by the different product service the performs the validation.

## Get Business Profile

### API Contract

* **API Route:** GET /v1/profile

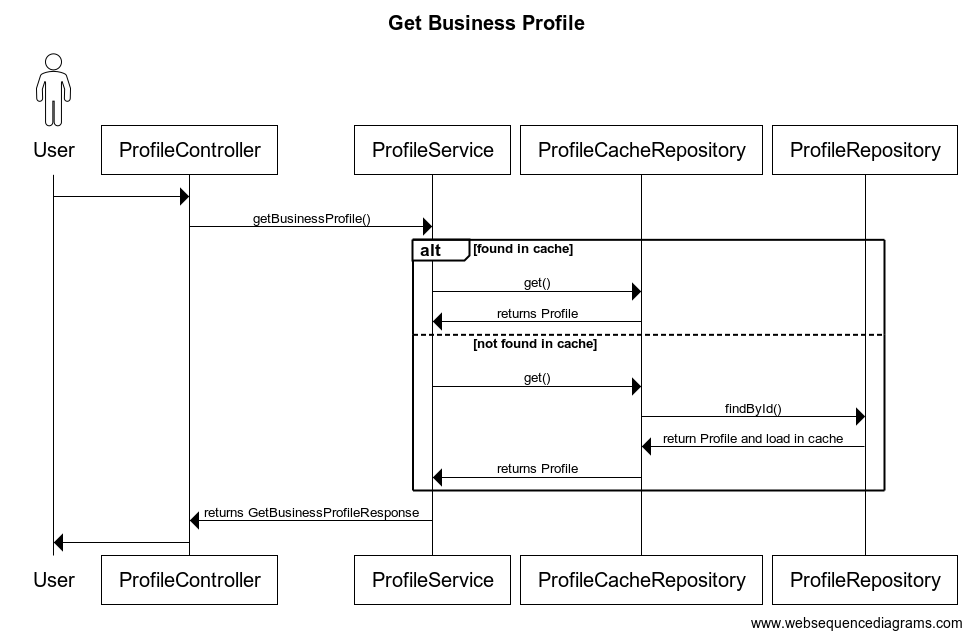
#### Request Parameters

* **Request Header:** X-Business-Id

#### Response Parameters

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Description** |
| id | String | Business Id |
| companyName | String | Name of the business |
| legalName | String | Legal name of the business |
| businessAddress | Address | Address of the business |
| legalAddress | Address | Legal address of the business |
| email | String | Email address of the business |
| website | String | Website of the business |

### Sequence Diagram



### Flow

* 1. User invokes the getBusinessProfile API.
  2. The service fetches the Profile from the Redis cache.
  3. If the Profile with given business id does not exist in cache, it fetches the Profile from the DynamoDBTable.
  4. It loads the cache with the Profile data and returns it to the user.
  5. If the Profile does not exist in the database as well, it returns Not Found response.

## Get Profile Update Validation Status

### API Contract

* **API Route:** GET /v1/profile/update/status

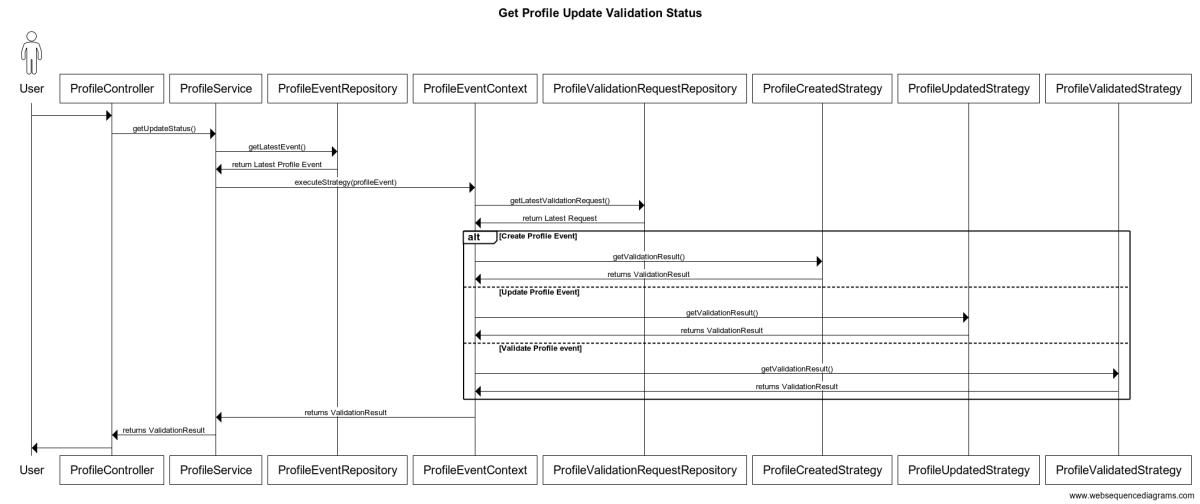
#### Request Parameters

* **Request Header:** X-Business-Id

#### Response Parameters

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Description** |
| event | EventType | Type of recent event (CREATE\_PROFILE/UPDATE\_PROFILE) |
| status | ValidationStatus | Enum: NO\_STATUS/IN\_PROGRESS/ACCEPTED/REJECTED |
| errors | List<String> | List of error messages, if any |

### Sequence Diagram

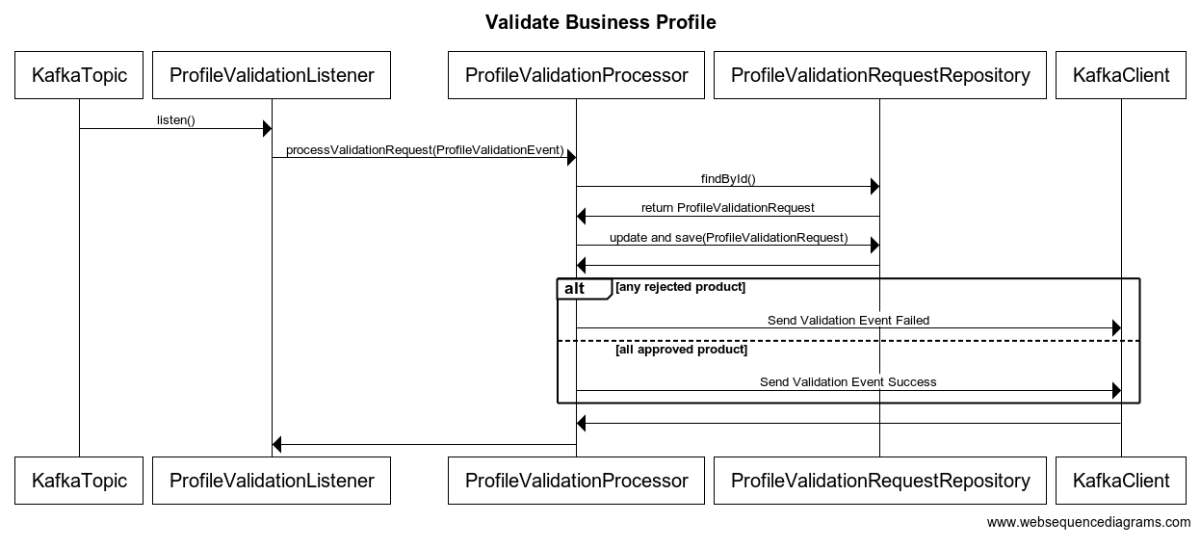
****

### Flow

* 1. User invokes the getUpdateStatus API.
  2. The service fetches the latest event from the ProfileEvent DynamoDBTable for the given business id.
  3. It then fetches the latest validation request from the ProfileValidationRequest table.
  4. Based on the event type validation request, it fetches the validation result from the specific event strategy.
  5. If the latest validation request is in progress, it checks if the request has been timed out. If the request is timed out, it returns a rejection response.

## Validate Business Profile

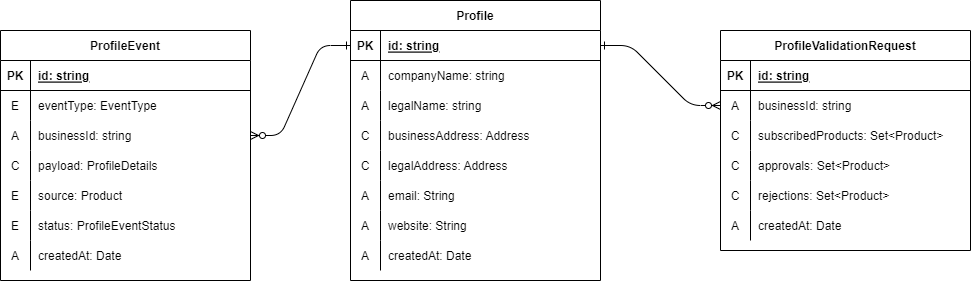
### Sequence Diagram



### Flow

* 1. ProfileValidationListener consumes the event from the profile-validation-event with the event payload.
  2. It then fetches the ProfileValidationRequest from the Table for the given request id.
  3. If the Validation Status is failed from the event payload and it is part of the subscribed products, the request is rejected.
  4. IF the Validation Status is successful from the event payload and it is part of the subscribed product, it is added to the list of approvals in the validation request. If all the subscribed products have been approved, the validation moved to successful status.
  5. If the validation is successful, it sends and even to update the Profile in DB.
  6. If the validation is failed, it sends a failure event.s

# Data Store and Model



## Table: Profile Event

* Primary Index
  + Partition Key: id
  + Sort Key: createdAt
* Global Secondary Index
  + Partition Key: businessId
  + Sort Key: createdAt

|  |  |  |
| --- | --- | --- |
| **Column Name** | **Datatype** | **Type** |
| id | String | PartitionKey |
| eventType | EventType |  |
| businessId | String | GSI: PartitionKey |
| payload | ProfileDetails |  |
| source | Product |  |
| status | ProfileEventStatus |  |
| createdAt | Date | SortKey |

## Table: Profile

* Primary Index
  + Partition Key: id
  + Sort Key: createdAt

|  |  |  |
| --- | --- | --- |
| **Column Name** | **Datatype** | **Type** |
| id | String | PartitionKey |
| companyName | String |  |
| legalName | String |  |
| businessAddress | Address |  |
| legalAddress | Address |  |
| email | String |  |
| website | String |  |
| createdAt | Date | SortKey |

## Table: ProfileValidationRequest

* Primary Index
  + Partition Key: id
  + Sort Key: createdAt
* Global Secondary Index
  + Partition Key: businessId
  + Sort Key: createdAt

|  |  |  |
| --- | --- | --- |
| **Column Name** | **Datatype** | **Type** |
| id | String | PartitionKey |
| businessId | String | GSI: PartitionKey |
| subscribedProducts | Set<Product> |  |
| approvals | Set<Product> |  |
| rejections | Set<Product> |  |
| createdAt | Date | SortKey |

# Launch Plan

* **Containerization**: Use Kubernetes to manage containerized applications, ensuring that they are scalable and highly available.
* **Internationalization**: Deploy the application in different AWS regions closer to the target audience to reduce latency.
* **Testing**: Apart from the unit tests, focus on integration tests, load testing, stress testing, to ensure the application's robustness in production scenarios.
* **Monitoring**: Setup proper monitoring and alarms based on key metrics thresholds on JVM/environment metrics, latency or availability metrics, and business metrics

# Appendix

## ProfileDetails Data Model

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Description** |
| companyName | String | Name of the business |
| legalName | String | Legal name of the business |
| businessAddress | Address | Address of the business |
| legalAddress | Address | Legal address of the business |
| email | String | Email address of the business |
| website | String | Website of the business |

## Address Data Model

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Description** |
| lineOne | String | Address Line 1 |
| lineTwo | String | Address Line 2 |
| city | String | Name of the city |
| state | String | Name of the state |
| zipCode | String | Zipcode of the business location |
| country | String | Name of the country |