Migration to Microservices

Overview:

The project involved migration of a legacy vacation rental and property management system, to a transition to a Microservices architecture. The system integrates transactions related to property bookings and performs various related tasks. The primary function is to fetch bookings from multiple platforms (like Operto, Trip advisor etc.) and provide a solution to manage these bookings via a PWA platform where the staff can work on tasks assigned to them, create issues related to the property, upload images/videos, etc. The application portal interacts with QuickBooks accounting software to manage invoices, estimates, sales orders, and payrolls for the staff, customers, and vendors. It also provides features for new bookings based on customized rules.

Client details:

Name: Confidential | **Industry:** Real Estate | **Location:** USA

Technologies:

ColdFusion, MSSQL Database, Symfony, REST, GIT, CI/CD, React, Redux, Workbox, Service Workers, QuickBooks, Aglio documentation.

Project Description:

This project enabled the client to migrate from legacy system to Microservices architecture. This was implemented due to the following reasons:

- High product growth
- To address concerns related to the product's technology stack wherein Microservices architecture helps to keep the application modules independent of each other.
- All the original code components were interconnected, interdependent and tied up with a single codebase. This would have led to performance issues later on. The application is built on two servers (ColdFusion/Lucee & React/Symfony/LAMP). They share a centralized Microsoft SQL Server database. Some salient features of the application:

- The ColdFusion and LAMP Clients interact via a shared domain cookie (a JWT Token). This token is created on the ColdFusion server using a shared secret which is passed onto the LAMP server and verified using the same shared secret.

Microservice Architecture:

- The backend is set up with Microservices architecture. The setup uses an API layer that creates all of the endpoints for communicating to other Microservices, along with some internal routes as well for handling authentication and permissions.

- The API layer communicates with other Microservices through RabbitMQ. The project on GitHub is set up for local development using Docker. The Docker compose file includes the following services:

    1. API layer (built-in express)
    2. PostgreSQL (is exposed on the Docker network for all other Microservices to use as well)
    3. RabbitMQ (exposed on the same Docker network)

Interservice Communication:

- Requests that are made from the API Gateway layer to a Microservice, those that requires a response are made using an RPC request through RabbitMQ.

- While any jobs that need to be run asynchronously can be sent to the Microservice using Work Queues on RabbitMQ.

- A messenger service handles the RPC requests and sends tasks on a Work Queue for each framework which can be created and added to a base repository for that framework.

- Each Microservice has its routes defined in the API Gateway layer and all the communication is done via Messenger & all the Microservices have their dedicated DB for DB transactions.

API Layer:

- The API Layer is the layer that handles all incoming requests. It also passes the necessary data on to RabbitMQ to the correct queue.

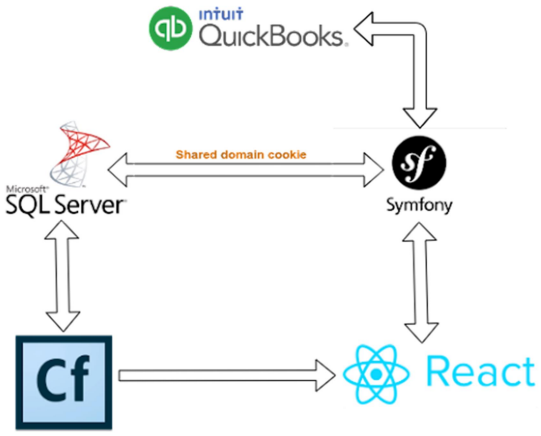
- It handles storing the permissions required for routes and if the routes are protected or not. The middleware handles the Permissions which is added through the verify token middleware whereby a list of required permissions is passed to the middleware.

- For Microservice routes, there is a protected property in the method object. If the protected property is set to true, the authenticateMicroRoute middleware will ensure that the logged-in user has the required permissions. If the member does not have the required permissions, the API Layer will return a 403 error.

- Controllers handle the registering of internal routes whereby the app and the optional prefix pass through the constructor of that controller. The index file in the internal routes folder handles the registering of all internal routes. Managing members, Roles, and Permissions are also handled here.

- The index file in the micro route folder handles the external routes, for communicating to other Microservices. The JSON file stores the Routes for each Microservice.

Flow Diagram:



Architecture: