CS301 - IT Solution Architecture

AY 2021/22 Term 1

Project Proposal

Prepared for:
Professor OUH Eng Lieh

Done by:
Team 8 (Github: G2team8)

Name	Student ID
Ho Jing Yi	01375797
Lee Sean Jin	01337831
Lim Zhong Zhen Timothy	01348521
Soh Bai He	01375548
Tan Song Yuan Daniel	01332737

Content Page

Background and Business Needs

COMO Group is a company headquartered in Singapore, with a wide variety of systems across different business units. At the heart of COMO Group lies ComoClub with an app that allows customers to redeem or purchase exclusive experiences and items across COMO's Business Units. This requires the application to leverage on and integrate with the existing systems in COMO.

Currently, each business unit system is separated from one another and there is no middleware to manage integration between these systems. A lack of integration creates information silos which creates inefficiencies and redundancies across the business.

Our proposed system architecture includes a middleware which serves to integrate and consolidate existing systems. It simplifies the connection between the app to multiple 3rd party APIs like Stripe, Memberson and 7Rooms. It is also designed to handle future integrations from other Business Units and to onboard new external APIs easily.

Stakeholders

Stakeholder	Stakeholder Description	Permissions
ComoClub's ops managers	In charge of liaising with the partner merchants of Club21, curate and upload pictures and descriptions of the experiences to Memberson CRM.	Read, Write
ComoClub's Maintenance team	In charge of scalability, availability and data security of our proposed middleware.	Read, Write
ComoClub's Developers	In charge of developing ComoClub's Club 21 application	Read, Write
AWS	Proposed middleware cloud platform provider	N.A
App Users	Users of the Club21 application	N.A.
7rooms, Memberson CRM	External API providers	N.A.

Key Use Cases

Use Case Title: Login	
Use Case ID	1
Description	This use case describes how a user logs into the ComoClub app.
Actors	Users, Memberson CRM
Main Flow of Events	1. User enters his/her name and password. 2. The app validates the entered credentials and logs the user into the system.
Alternative Flow of Events	If the user enters an invalid name and/or password, the app displays an error message. The user fails to login.
Pre-conditions	-
Post-conditions	The user is now logged into the app and directed to the app's homepage. If the use case was unsuccessful, the state is unchanged.

Use Case Title: View details of an experience	
Use Case ID	2
Description	This use case describes how a user select and views details of an experience (from various 3rd party endpoints)
Actors	Users, 7Rooms Booking, Memberson CRM

Main Flow of Events	1. User select one of the experiences from the curated list of experiences 2. The app shows the details of the selected experience
Alternative Flow of Events	Cannot connect to 3rd party api
Pre-conditions	User is currently at the home page
Post-conditions	User to be at the experience's detail page

Use Case Title: Book an experience with COMO points	
Use Case ID	3.1
Description	This use case describes how a user books an experience with their available COMO points.
Actors	Users, 7Rooms Booking, Memberson CRM
Main Flow of Events	1. User clicks on "Book Now" button 2. User choose their desired date and time as well as the number of tickets to be purchased 3. User selects "Pay with COMO credits" 4. App verifies user's number of available points against the points required for redemption. 5. If points are sufficient, the app shows a summary of the user's booking details and requests the user to confirm purchase details. 6. User clicks on the "confirm payment" button. 7. The app reserves the booking slots for the user and sends a booking confirmation to the user's email
Alternative Flow of Events	Users do not have enough COMO credits and have to pay with a credit card.
Pre-conditions	User to be at the experience's detail page
Post-conditions	Booking successful and booking confirmation email sent.

Use Case Title: Book an experience with credit card	
Use Case ID	3.2
Description	This use case describes how a user books an experience with credit card
Actors	Users, 7Rooms Booking, Memberson CRM, Stripe
Main Flow of Events	1. User clicks on "Book Now" button 2. User choose their desired date and time as well as the number of tickets to be purchased 3. User selects "Pay by Credit Card" 4. The app shows a summary of the user's booking details and requests the user to confirm purchase details. 5. User clicks on the "confirm payment" button. 6. The app redirects the user to an external webpage for credit card confirmation. 7. Upon successful payment, the app reserves the booking slots for the user and sends a booking confirmation to the user's email
Alternative Flow of Events	Payment process failed
Pre-conditions	User to be at the experience's detail page
Post-conditions	Booking successful

Use Case Title: View details of an item from the “Redemptions” tab	
Use Case ID	4
Description	This use case describes how a user selects and views the details of an item available for redemption.
Actors	Users, MembersonCRM
Main Flow of Events	1. User selects an item. 2. App shows the details of the item and the number of points that the user currently has available.
Alternative Flow of Events	Connection to 3rd party api failed
Pre-conditions	User is on the “Redemptions” tab.
Post-conditions	User is viewing the details of an item available for redemption.

Use Case Title: Redeem an item with COMO credits	
Use Case ID	5.1
Description	This use case describes how a user redeems an item with their available points.
Actors	Users, Memberson CRM
Main Flow of Events	1. User clicks on “Redeem Now” button 2. User selects item quantity 3. User selects “Pay with COMO credits” 4. App verifies user’s number of available points against the points required for redemption. 5. If points are sufficient, the app shows a summary of the user's redemption details and requests the user to confirm purchase details. 6. User clicks on the “confirm payment” button. 7. App processes the user's redemption (item is reserved for the user) and shows a success message.
Alternative Flow of Events	Users do not have enough COMO credits and have to pay with a credit card.
Pre-conditions	User is viewing the details of an item available for redemption.
Post-conditions	Item redemption successful

Use Case Title: Redeem an item with credit card	
Use Case ID	5.2
Description	This use case describes how a user redeems an item with their credit card.
Actors	Users, Memberson CRM, Stripe
Main Flow of Events	1. User clicks on “Redeem Now” button 2. User selects item quantity 3. User selects “Pay by Credit Card” 4. The app shows a summary of the user's redemption details and requests the user to confirm purchase details. 5. User clicks on the “confirm payment” button. 6. The app redirects the user to an external webpage for credit card confirmation. 7. Upon successful payment, the app processes the user's redemption (item is reserved for the user) and shows a success message.
Alternative Flow of Events	Payment process failed
Pre-conditions	User is viewing the details of an item.
Post-conditions	Item redemption successful

Quality Attributes

Maintainability

- Microservices

The applications in our architecture run as independent, ephemeral microservices that increase the modularity and portability of services.

Loose coupling of applications and services across AWS through the use of load balancers & API gateway further increases the modularity of the services used in our architecture, this increases the ease of maintainability of the containerised applications.

Amazon ECS is a fully managed container orchestration service that will be used to easily orchestrate and scale the containers running in the cluster. Operational complexity of having to manually manage each microservice will be reduced through the use of Amazon ECS.

- Design patterns

The facade design pattern will be utilised through an API Gateway that abstracts and encapsulates services running in the ECS cluster. Through this design pattern, we are able to increase the portability of our microservices, allowing easier maintenance of each microservice.

The external configuration store pattern will also be utilised through storing runtime container configurations such as the external API endpoints in Amazon DynamoDB which is a highly available noSQL database. This provides a centralised storage where containers will retrieve their runtime configurations from. This further provides easier management and control of configuration data, and for sharing configuration data across applications and application instances. Sensitive data such as API tokens and credentials that need to be shared across applications/ instances will be stored in a centralised location which is AWS Secrets Manager that each microservice will have access to.

- Infrastructure as code

Terraform will be used to provide infrastructure as code where it manages and provisions infrastructure through code instead of through manual processes. This allows replication of AWS resources to be done across multiple environments and regions rapidly and easily.

- Continuous integration & continuous delivery

CI/CD pipelines will be set up using GitHub actions, Amazon EventBridge and AWS Lambda which will automate deployment from code to container. The pipeline will integrate testing on a VM before building the image to be run by ECS.

- AWS managed services

In addition to AWS Fargate and AWS ECS, other AWS managed services such as AWS SQS, Amazon DynamoDB etc will be utilised as well. Some of these services include serverless computing solutions such as AWS Lambda & AWS Fargate which will reduce operational overhead of having to manage the server running our applications. This increases the maintainability of the applications as developers only need to focus on maintaining the source code.

Availability

- Redundancy

Redundancy will be achieved through active-active load balanced applications running in an autoscaling group in multiple AWS availability zones behind a load balancer. In the case where an availability zone goes down, there will still be instances of the applications in another availability zone running to handle incoming requests.

- AWS managed services

Our architecture focuses on utilising AWS managed services and serverless services whenever possible. This guarantees a much higher level of availability as compared to having to run these services on our own. E.g. AWS S3 guarantees 99.99% availability over a given year. This level of availability may not be achievable if we manage our own storage service locally.

- Failover plan

AWS managed application load balancers will be utilised to automatically perform failover duties when instances in an availability zone go down.

In the scenario where the external APIs go down, each middleware container will push the incoming requests from the user to an AWS SQS Queue that will store requests in order when it detects that the external APIs are not available. Through this implementation, we aim to provide diminished service to the user instead of having no service at all when business critical external APIs become unavailable.

In the scenario where our middleware containers perform aggregated API calls to the external APIs and a subset of these requests fail, the container instance will push the failed request details to the AWS SQS Queue to be re-tried again later. This provides diminished availability of functionality instead of unavailability when aggregated API requests in the middleware fail.

In the event of a disaster, the team has chosen to adopt the backup and restoration method in another AWS region (Sydney) to reduce costs. The trade off will be low Recovery Point Objective (RPO) and Recovery Time Objective (RTO). To allow for better RPO and RTO, Amazon CloudWatch will be used to create alarms to ensure that the disaster recovery process can begin at the soonest time to reduce downtime. To facilitate this, cross region replication of ECR images will be implemented together with Terraform. During a disaster, the Terraform code will replicate infrastructure and services into the new region. Images will be pulled from the DR region's ECR to quickly recover our microservices. DynamoDB will be configured to use global tables which will create replica tables in the DR region to ensure data availability.

Security

- VPC Encapsulation

After traffic passes through the AWS API Gateway, all traffic will be contained within the VPC. This ensures that traffic does not traverse the public internet. This reduces the attack surface where malicious attackers are able to intercept traffic to compromise confidentiality of the requests made to our services.

VPC endpoints services (AWS Private Link) will be used with AWS services such as DynamoDB to allow our services running in the VPC to access DynamoDB with no exposure to the public internet. Traffic between the VPC and the AWS service does not leave the Amazon network.

Since the middleware microservices will require the need to call external API endpoints over the internet, a NAT gateway will be provisioned in the public subnet to allow outgoing traffic from private subnets. Services outside the VPC will not be able to access our microservices directly over the internet.

- Encryption

Data at rest will be encrypted in Amazon DynamoDB using AWS managed keys created in AWS Key Management Service (AWS KMS).

SSL certificates generated by AWS Certificate Manager (ACM) will be used to encrypt data in transit when requests are made to our Route53 domain.

- Authentication

Our architecture ensures end to end user authentication where only registered users in Memberson CRM are able to access our services.
AWS Cognito will be integrated with AWS Amplify to secure frontend pages and will also be integrated with AWS API Gateway to secure API endpoints with OAUTH 2.0 authorisation. This prevents our endpoints from being publicly accessible. Our middleware microservices will authenticate to external API endpoints on behalf of the user.

- AWS Web Application Firewall (WAF)

WAF can be integrated with Amazon CloudFront and Amazon API Gateway to protect our web application and APIs against common web exploits and bots that may affect availability, compromise security, or consume excessive

resources. It also gives us control over how traffic reaches our applications through security rules that control bot traffic and block common attack patterns, such as SQL injection or cross-site scripting.

Performance

- Autoscaling

Autoscaling policies such as scheduled autoscaling can be applied together with target tracking autoscaling policies to further to forecasted peak periods to reduce the strain of the servers to allow clients to continue to use our services as per normal.

- AWS managed services

To allow for deeper integration with AWS services, Amazon ECS was chosen as the container orchestration service over alternatives such as EKS. The maturity of ECS brings about smooth integration with Amazon SQS and AWS Secrets Manager to carry out business functionality more efficiently which will reduce latency between requests. Our architecture uses mostly AWS managed or serverless services where possible which automatically scales to meet the demand during peak periods. This allows our services to continue performing optimally during high stress.

- Caching

AWS Cloudfront will be used as a CDN to serve static pages on the frontend. It offers a multi-tier cache by default that improves latency and lowers the load on origin servers when the object is not already cached at the Edge. Amazon DynamoDB Accelerator (DAX) can also be used to implement in memory caching to allow for our microservices to retrieve commonly accessed data from DynamoDB as the origin.

Proposed Services and Cost

- **Production environment**

Activity / Hardware / Software / AWS Service	Description	Cost (USD)
Amazon API Gateway	HTTP API requests units (exact number), REST API request units (exact number), Cache memory size (GB) (None), WebSocket message units (thousands), Average message size (32 KB), Requests (per hour), Requests (10000 per day)	1.29
DynamoDB on-demand capacity	Average item size (all attributes) (1 KB), Data storage size (0.1 GB)	0.03
DynamoDB Accelerator (DAX) clusters	DAX nodes (1 instances of type t3.small)	44.53
Amazon Virtual Private Cloud (VPC)	Number of NAT Gateways (2)	86.26
Amazon Virtual Private Cloud (VPC)	Number of interface VPC endpoints per region (1)	18.99
Amazon Simple Queue Service (SQS)	DT Inbound: Internet (0 GB per month), DT Outbound: Not selected (0 TB per month), Standard queue requests (million per month), FIFO queue requests (1 million per month)	0.50
Amazon CloudWatch	Number of Custom/Cross-account events (10), Standard Logs: Data Ingested (1 GB), Vended Logs: Data Ingested (1 GB), Logs Delivered to S3: Data Ingested (1 GB), Number of Lambda functions (4), Number of requests per function (60 per hour)	11.49
AWS Secrets Manager	Number of secrets (12), Average duration of each secret (30 days), Number of API calls (30000 per day)	9.36
Amazon Cognito	Advanced security features (Enabled), Number of monthly active users (MAU) (300)	15.00
Amazon Elastic Container Registry	DT Inbound: Internet (1 TB per month), DT Outbound: Not selected (0 TB per month), Amount of data stored (50 GB per month)	5.00

AWS Web Application Firewall (WAF)	Number of Web Access Control Lists (Web ACLs) utilized (1 per month), Number of Rules added per Web ACL (7 per month), Number of Rule Groups per Web ACL (per month), Number of Managed Rule Groups (per month), Number of Rules inside each Rule Group (per month)	12.60
AWS Fargate	Average duration (1 days), Amount of ephemeral storage allocated for Amazon ECS (20 GB), Number of tasks or pods (8 per day)	89.97
Amazon Route 53	Hosted Zones (1), Basic Checks Within AWS (2), Number of domains stored (2)	2.90
Application Load Balancer	Number of Application Load Balancers (1)	49.06
Amazon Elastic Container Registry	DT Inbound: Not selected (0 TB per month), DT Outbound: Not selected (0 TB per month), Amount of data stored (50 GB per month)	5.00
AWS Lambda	Number of requests (100000)	0.00
Total		**351.98**

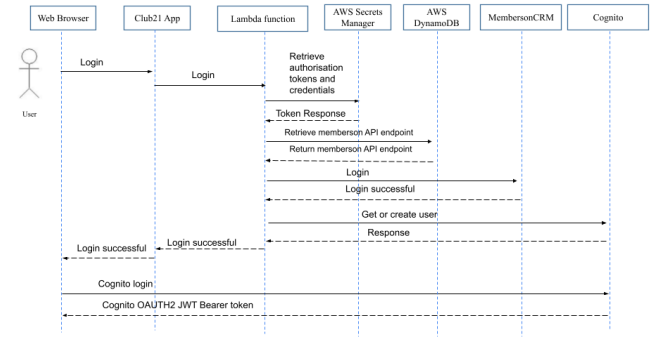
- **Deployment environment**

Activity / Hardware / Software / AWS Service	**Description**	**Cost (USD)**
Amazon Route 53	Hosted Zones (2), Basic Checks Within AWS (1)	1.40
Amazon API Gateway	HTTP API requests units (exact number), REST API request units (exact number), Cache memory size (GB) (None), WebSocket message units (thousands), Average message size (32 KB), Requests (per month), Requests (1000 per month)	0.00
DynamoDB on-demand capacity	Average item size (all attributes) (1 KB), Data storage size (0.1 GB)	0.03
Amazon Virtual Private Cloud (VPC)	Number of NAT Gateways (1)	43.36
Amazon Virtual Private Cloud (VPC)	Number of interface VPC endpoints per region (1)	9.54
Amazon Simple Queue Service (SQS)	Data transfer cost (0), Standard queue requests (million per month), FIFO queue requests (1 million per month)	0.50
Amazon CloudWatch	Number of Metrics (includes detailed and custom metrics) (10), Standard Logs: Data Ingested (1 GB), Vended Logs: Data Ingested (1 GB), Number of Custom/Cross-account events (3)	4.40
AWS Secrets Manager	Number of secrets (12), Average duration of each secret (30 days), Number of API calls (100 per day)	4.82
Amazon Cognito	Advanced security features (Enabled), Number of monthly active users (MAU) (1)	0.05
Amazon Elastic Container Registry	DT Inbound: Internet (50 GB per month), DT Outbound: Not selected (0 TB per month), Data transfer cost (0), Amount of data stored (50 GB per month)	5.00
AWS Fargate	Average duration (1 days), Number of tasks or pods (4 per day), Amount of ephemeral storage allocated for Amazon ECS (20 GB)	44.98
AWS Lambda	Number of requests (50000)	0.00
Total		**114.08**

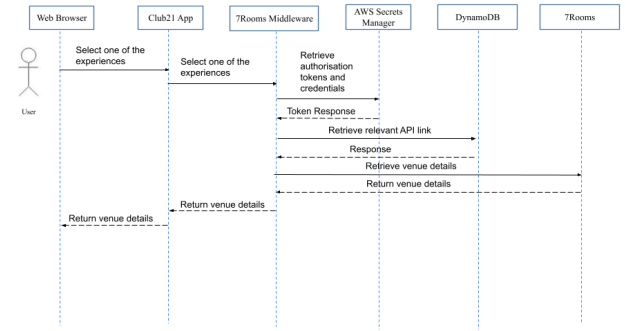
Views

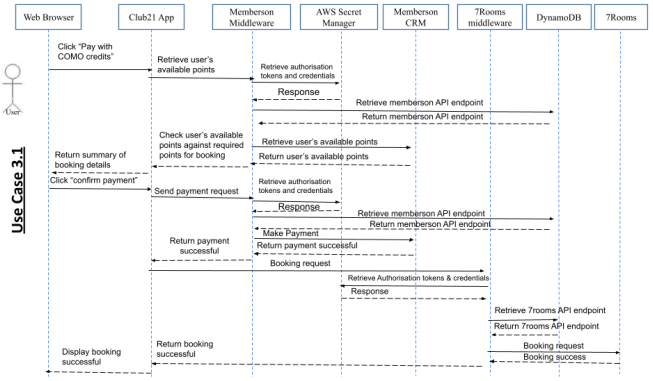
Sequence diagram

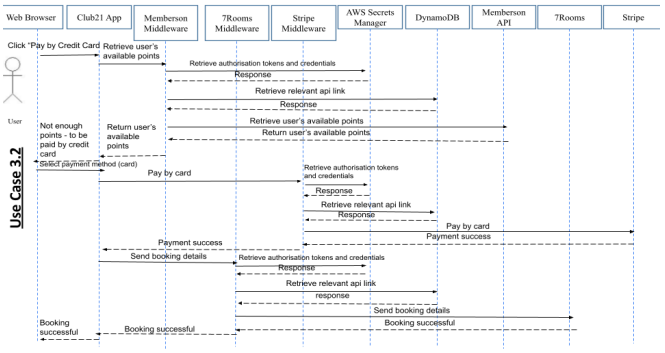
Use Case 1: Login



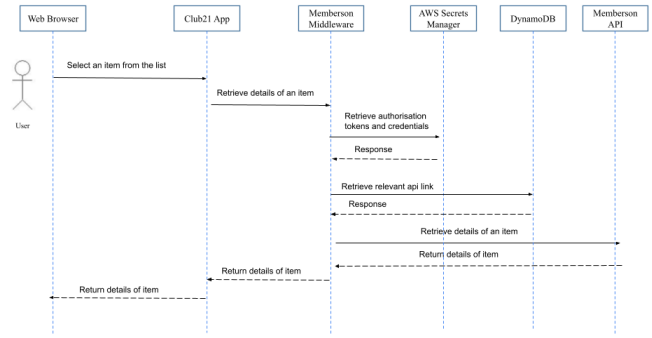
Use Case 2

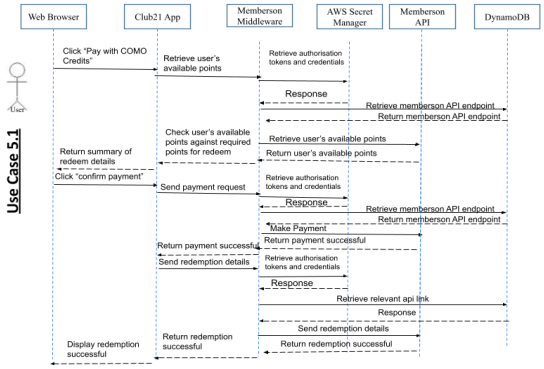


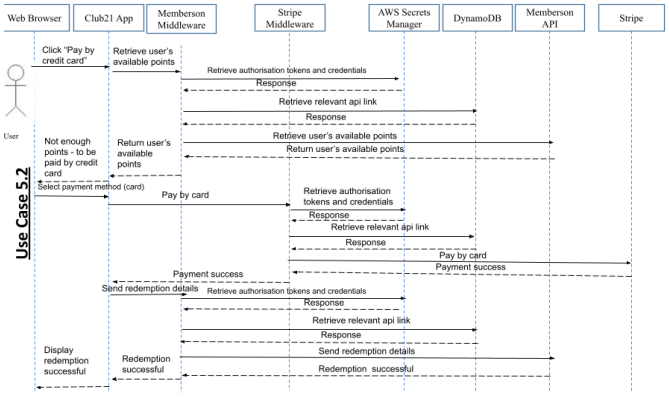




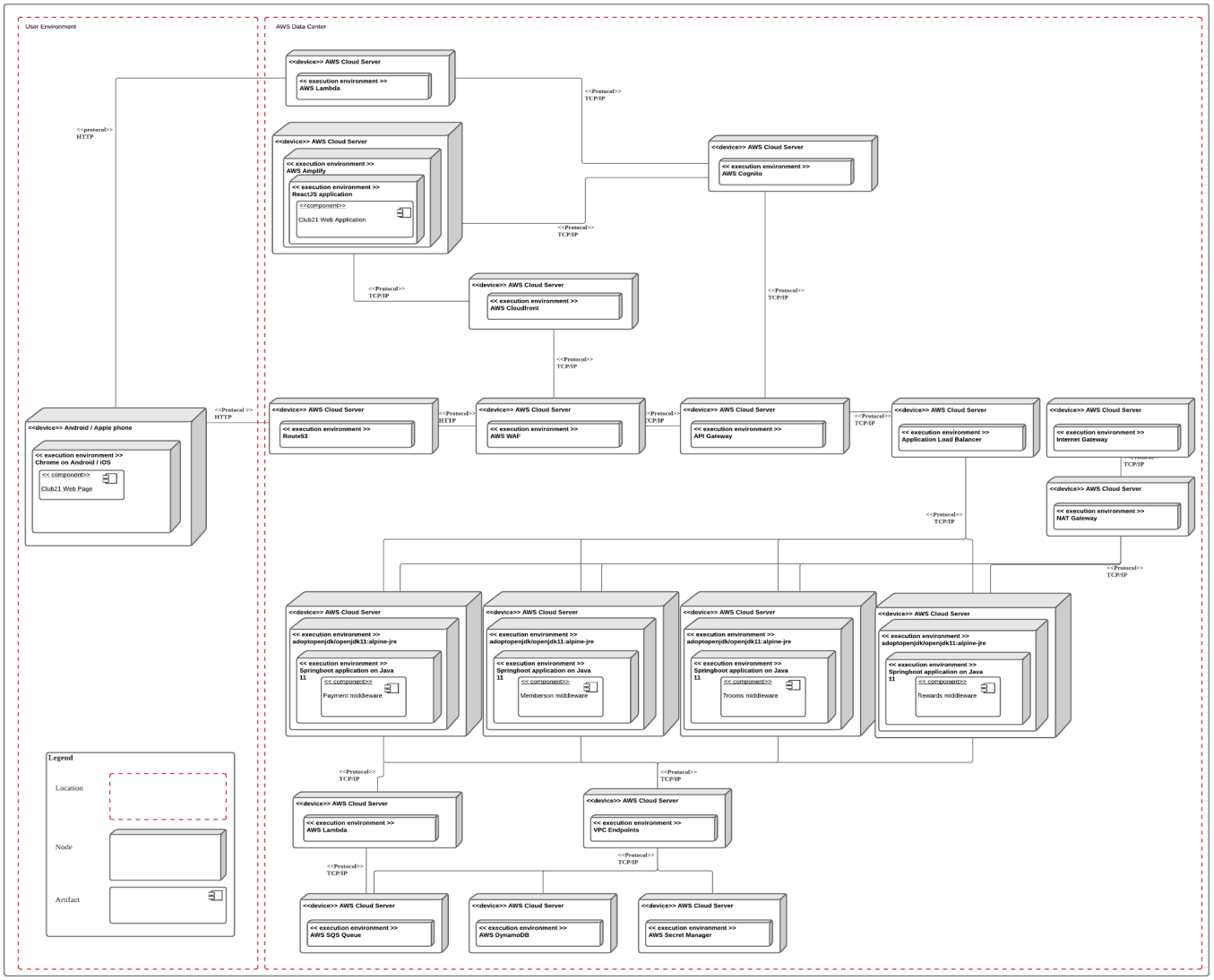
Use case 4





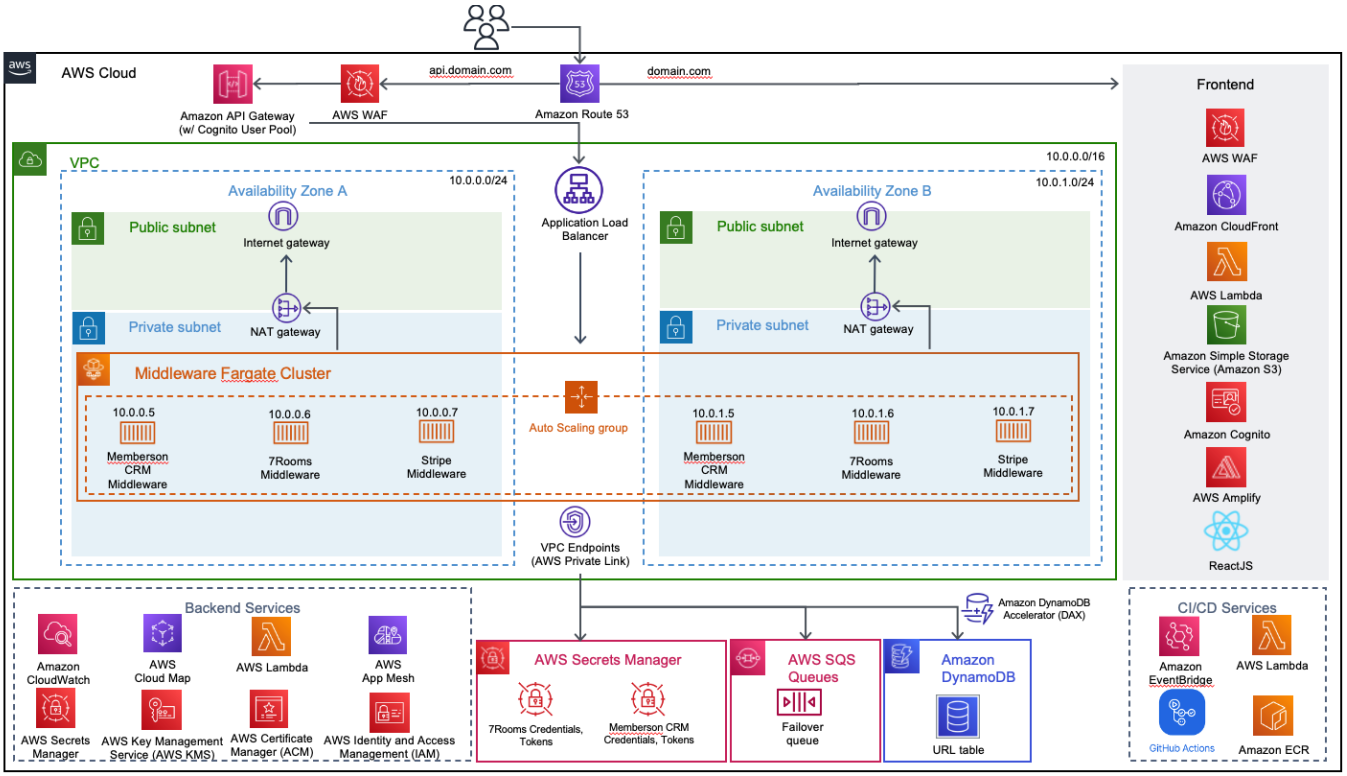


Deployment view


User Environment
AWS Data Center
<<device>> AWS Cloud Server
<< execution environment >>
AWS Lambda
<<protocol>>
HTTP
<<Protocol>>
TCP/IP
<<device>> AWS Cloud Server
<< execution environment >>
AWS Amplify
<< execution environment >>
ReactJS application
<<component>>
Club21 Web Application
<<device>> AWS Cloud Server
<< execution environment >>
AWS Cognito
<<device>> AWS Cloud Server
<< execution environment >>
AWS Cloudfront
<<device>> Android / Apple phone
<< execution environment >>
Chrome on Android / iOS
<< component>>
Club21 Web Page
<<Protocol >>
HTTP
<<device>> AWS Cloud Server
<< execution environment >>
Route53
<<device>> AWS Cloud Server
<< execution environment >>
AWS WAF
<<device>> AWS Cloud Server
<< execution environment >>
API Gateway
<<device>> AWS Cloud Server
<< execution environment >>
Application Load Balancer
<<device>> AWS Cloud Server
<< execution environment >>
Internet Gateway
<<device>> AWS Cloud Server
<< execution environment >>
NAT Gateway
<<device>> AWS Cloud Server
<< execution environment >>
adoptopenjdk/openjdk11:alpine-jre
<< execution environment >>
Springboot application on Java 11
<< component>>
Payment middleware
<<device>> AWS Cloud Server
<< execution environment >>
adoptopenjdk/openjdk11:alpine-jre
<< execution environment >>
Springboot application on Java 11
<< component>>
Memberson middleware
<<device>> AWS Cloud Server
<< execution environment >>
adoptopenjdk/openjdk11:alpine-jre
<< execution environment >>
Springboot application on Java 11
<< component>>
7rooms middleware
<<device>> AWS Cloud Server
<< execution environment >>
adoptopenjdk/openjdk11:alpine-jre
<< execution environment >>
Springboot application on Java 11
<< component>>
Rewards middleware
<<device>> AWS Cloud Server
<< execution environment >>
AWS Lambda
<<device>> AWS Cloud Server
<< execution environment >>
VPC Endpoints
<<device>> AWS Cloud Server
<< execution environment >>
AWS SQS Queue
<<device>> AWS Cloud Server
<< execution environment >>
AWS DynamoDB
<<device>> AWS Cloud Server
<< execution environment >>
AWS Secret Manager
Legend
Location
Node
Artifact

AWS diagrams

Production environment



Development environment