# Error 404

1. Li Jian Wei (jianwei.li.2017)
2. Lim Jia Wei (jiawei.lim.2017)
3. He Yicheng (yicheng.he.2017)
4. Yan Zifan (zifan.yan.2017)
5. Chia Wei (wei.chia.2017)
6. Fong Kin Shing (ksfong.2017)

## Background and Business Needs

The business requires a system for its human resource management, which can manage employees' details, calculate their final salary, manage their leaves and generate HR related reports. The Human Resource Management System (HRMS) is an application that can meet all the above needs. The HRMS must be accessible by all 100 employees from their work laptop and always up to date so that they are always accessing the latest data.

The work laptop is sometimes brought home by the users; hence, the data must be accessible even after leaving the company premise. The application also must be easy to learn and use. All employees are assigned to a Laptop running the Windows OS.

The current HRMS is a monolithic application with a local database. Employees can only apply for leave from a designated work laptop. Updating of data on multiple work laptops with the application is tedious. The application can only be used in the company's premise. The business wishes to transform this application into one that fulfils the operational needs without compromising the security of the company's data.

Our group will decompose the monolithic application into microservices while retraining most of its functionality. Most microservices will be hosted on the cloud with their respective databases. Our group adopts a high availability architecture with a primary and secondary environment for all load-balanced instances. The secondary databases are passive and retrieve data from the primary database, which is the single source of truth. We also added 2-factor authentication, firewall and access control to enhance application security and safeguard the integrity of the data. The application client also comes with some usability features such as saving last state user inputs and remember my username feature.

Initial Application:
https://github.com/HamzaYasHin1/HR-Management-System-in-Java-using-swing-framework

## Stakeholders

| Stakeholder | Stakeholder Description |
|---|---|
| Human Resource | Human Resource Department oversees managing manpower and will use the HRMS to add, remove, or update employee's information, calculate an employee's pay, generate a report and payslip from the system. |
| Employees | Employee will use HRMS to apply for leave and view their employee information. |
| Management | Manager will access HRMS to handle employee leave request, view employee profile, documents and reports. |

| | |
|---|---|
| IT Department | The IT department oversees developing and maintaining HRMS. The IT department is also responsible for designing IT architecture for the system to best suit the business needs. |
| Amazon Web Services | The Amazon Web Services is a service provider that the business use for hosting of microservices and databases on the cloud. |

## Key Use Cases

| **Use Case Title – User Login into Application Client** | |
|---|---|
| **Use Case ID** | 1 |
| **Description** | The use of **OTP** (One Time Password) as a **2 Factor Authentication** to authenticate the user logging in adds an extra layer of security to our microservices and databases. A standalone application client is vulnerable. Hence, the authentication of the user is important to ensure **confidentiality** and data **integrity** as data are accessible only to those authorized to have access and the system prevents unauthorized access to, or modification of, computer programs or data. To a certain extent, the **authenticity** of the user can also be proved.<br>User role is identified upon logging in to enforce access control. This feature is implemented in the **Application Client** itself to determine which panel the logged-in user can access. Access Control also ensures data **integrity**.<br>All user logged in will have a new unique SessionID generated. The SessionID identifies and validates the login. When the same user, identified by the UserID logs in again, the SessionID will be replaced and the previous SessionID tied to the same UserID will no longer be valid. This prevents multiple logins from the same user, which eliminates the chance of data state mutation. It also restricts the number of concurrent logins possible to the number of users available. This, in turn, will ensure that the **resource utilization** will not exceed the server capacity and thus, meet the requirement. |
| **Actors** | Human Resource, Employee and Management |
| **Main Flow of Events** | 1. The user enters login credentials<br>2. The system **calls the User microservice** (persist in **a tomcat instance**) through the **API Gateway**.<br>3. **User microservice** retrieves data from a **database** to authenticate the login credentials.<br>4. The **User microservice** returns the login status to the system.<br>5. The system **calls the User microservice** (persist in **another tomcat instance**) to perform 2-factor authentication.<br>6. **User microservice invokes the Nexmo API** for OTP.<br>7. OTP code is sent to the user's mobile phone.<br>8. When the user enters the OTP Code, the system calls the **User microservice**.<br>9. **User microservice** invokes the **Nexmo API** to validate the OTP code.<br>10. The system stores UserID and Role in-memory **cache** upon logging in for access control and login validation on every panel in the system. |

| | |
|---|---|
| | 11. User successfully logged in |
| | 12. System calls the **Session microservice** to generate a new SessionID for the user that has just logged in.<br>13. A new thread will be created to call the **MQ microservice** to retrieve notification. |
| **Alternative Flow of Events** | Remember User<br>9. System stores Username in Local Data for the Remember User feature.<br>10. User successfully logged in |
| | Invalid User<br>1. User attempts to log in with invalid credentials<br>9. User failed to log in<br>10. Application client displays the error message |
| | Invalid OTP Code<br>6. User keyed in the wrong OTP code while logging in<br>9. Application client prompts the user to re-enter OTP code |
| **Pre-condition** | NIL |
| **Post-condition** | User is successfully logged into the system. |

| | |
|---|---|
| **Use Case Title - User Add Employee** | |
| **Use Case ID** | 2 |
| **Description** | The **last state** of the new employee **data** is stored in the **database** so that even if the user accesses the application on a different work laptop, the last changes made will still be retrievable. This piece of state data is tied to a UserID and can only be updated with both valid UserID and SessionID. Hence, it ensures data **integrity** as it prevents unintended data modification. Retrieval of the last state of new employee data also improves **operability** as the user do not have to retype again.<br><br>The updating of the State Data from the Application Client is performed through a secondary **thread** on the Application Client using Java Schedule Executor Service. Thus, ensuring **performance** as the user can interact with the application with no potential **time** delay caused by the updating of state data.<br><br>The master database is also replicated on slave databases and always on standby to failover as soon as a failure is detected. This ensures that the database is highly **available** with minimal downtime. |
| **Actors** | Human Resource |
| **Main Flow of Events** | 1. The user clicked the Add Employee button.<br>2. The system calls the **Session microservice** to check if the SessionID is still valid.<br>3. The system calls the **Session microservice** to retrieve the "Last State" of the Add Employee input fields from a **database**.<br>4. The system displays the last state of the input fields. |

| | |
|---|---|
| | 5. The system will run a **new thread** to capture the change in the state of the input fields every minute and calls the **Session microservice** to store the change in the **database.**<br>6. The user submits the new employee details<br>7. The System calls the **Employee microservice** (persist in **a tomcat instance**)<br>8. Employee microservice updates data into the **database** |
| **Alternative Flow of Events** | Application Client crashed and User logins again<br>6. Application crashes<br>7. User reopen the application and access the Add New Employee panel<br>8. The application calls the **Session microservice** to retrieve the last state stored in the **database** by the **userID**. |
| **Pre-condition** | User is logged in and role allows the addition of new employee |
| **Post-condition** | Employee is successfully added. |

| | |
|---|---|
| **Use Case Title – Auto Updating of the Application Client** | |
| **Use Case ID** | 3 |
| **Description** | The Application Client has high **modifiability** as it can be remotely updated if there is an internet connection. |
| **Actors** | Human Resource, Employee and Management |
| **Main Flow of Events** | 1. The user clicked the shortcut desktop icon to run the Launcher (HRMS.exe)<br>2. The launcher checks the latest version of the application client through the website (version.html)<br>3. The launcher compares the version with the existing client version.<br>4. The launcher downloads the latest client and unzips it.<br>5. The launcher starts the client application. |
| **Alternative Flow of Events** | The client is already the latest<br>4. The launcher starts the client application. |
| **Pre-condition** | Assigned with a work laptop that runs Windows OS. |
| **Post-condition** | User successfully launch the latest updated client. |

| | |
|---|---|
| **Use Case Title - User Update Employee Detail** | |
| **Use Case ID** | 4 |
| **Description** | The Employee Data is **pre-fetched** and loaded with the JPanels. This improves **time behaviour** performance when the user is using the application.<br><br>The data are validated before the system call the Employee microservice. Next, the AWS Web Application Firewall also inspects the parameters passed through the HTTP |

| | |
|---|---|
| | request. Finally, the microservice uses a Prepared Statement to perform the SQL query. These 3 lines of defence are put in place to prevent SQL Injection. |
| **Actors** | Human Resource |
| **Main Flow of Events** | 1. The System calls the **Employee microservice** (persist in **a tomcat instance**)<br>2. Employee microservice retrieve data from a **database** to display on the Employee List panel.<br>3. User access the Employee List panel.<br>4. User selects the employee to edit.<br>5. The System display the individual employee details on the employee details edit panel.<br>6. The user makes changes.<br>7. The user submits the updated employee details<br>8. The System calls the **Employee microservice** (persist in **a tomcat instance**)<br>9. Employee microservice updates data into the **database** |
| **Alternative Flow of Events** | Invalid Updates<br>● Duplicate username<br>● Duplicate email<br>● Duplicate phone number |
| **Pre-condition** | User is logged in and role allows editing of employee details |
| **Post-condition** | Employee details are successfully updated. |

| **Use Case Title - Approve/Reject Leave Application** | |
|---|---|
| **Use Case ID** | 5 |
| **Description** | The Management have access rights to view and approve or reject the pending leave applications. Upon approving or rejecting the leave application, a notification message will be queue to the UserID of the employee who applied for the leave.<br>Upon the next login of the employee, the messages will be polled from the AWS MQ on a secondary thread. This is to improve performance so that the user can proceed to use the application, while the secondary thread gets all the notification messages and display them. |
| **Actors** | Management |
| **Main Flow of events** | 1. User access the Leave Application panel<br>2. The system calls the **Leave microservice** (persist in **a tomcat instance**)<br>3. Leave microservice retrieves data from a **database**<br>4. System displays the pending leave applications.<br>5. User approves or rejects the leave<br>6. The system calls the **Leave microservice** to update the leave application status in the **database**. |

| | |
|---|---|
| | 7. The system calls the **MQ microservice** to queue an application status message (eg. Leave has been approved) to a queue that has the userID of the employee that took the leave as the queue name.<br>8. System displays success message. |
| **Alternative Flow of Events** | 6. System displays no pending leave when no pending leave is retrieved. |
| **Pre-condition** | User is logged in and role allows approving/rejective rights |
| **Post-condition** | Leave application status are updated to approved or rejected. |

## Key Architectural Decisions

| **Architectural Decisions - Microservice Architecture** | |
|---|---|
| **ID** | 1 |
| **Issue** | To ensure availability and performance when the user uses the HRMS. |
| **Architectural Decision** | We moved the application functions out of the monolithic application into microservices that are hosted on cloud. This is so that the services can be reused for other IT applications that the company is using (eg. User microservice can be reused for Outlook, Skype). Additionally, each individual service can be scaled differently based on the operational requirement. The User service will be used by several IT Applications, whereas the Employee Service is used only by HRMS. Hence, the services are scaled differently in terms of server bandwidth to ensure performance satisfaction. |
| **Assumptions** | The company also uses user details for other IT services such as Outlook and Skype. |
| **Alternatives** | We also considered the Service Oriented Architecture. |
| **Justification** | We rejected the SOA as the services we are dealing with are fine-grained in nature. We only have a few services and they only belong to one business function – the Human Resource. Thus, we rejected the SOA as we think it makes this architecture style redundant if we regroup the services that we separated into a Human Resource Service Group.<br><br>Additionally, as we only have a few services, we can benefit from the modularity of the Microservice Architecture. We will be able to analyse the issue, test the services and deploy quickly without suffering from the messy management of large quantity of microservices. |

| **Architectural Decisions - Cloud Architecture** | |
|---|---|
| **ID** | 2 |
| **Issue** | To meet the business requirement of having the HRMS accessible from anywhere. |
| **Architectural Decision** | We moved the data onto Cloud to make it more accessible and the user can now access the data if there is connection to the internet. We chose both the **Outsourced Private and** |

| | |
|---|---|
| | **Public Cloud** model. The Cloud service is outsourced to AWS. We subscribed to Private Cloud for our microservices and Public Cloud for the Gateway. Private Cloud is secure as the public cannot access the cloud directly. |
| **Assumptions** | The business now requires the application to be accessible anywhere. |
| **Alternatives** | Onsite Private Cloud |
| **Justification** | The company is a medium-sized company that does not have both the IT Capacity and Money to manage an On-Site Private Cloud. Hence, we are outsourcing it to AWS as a mitigation strategy. |

| **Architectural Decisions - Stand-Alone Application Client** | |
|---|---|
| **ID** | 3 |
| **Issue** | High availability of the application client. |
| **Architectural Decision** | All users can access the HRMS through the company laptop with the application client installed. The Stand-Alone Application Client will be auto-updated on launch if there is a newer version. |
| **Assumptions** | All users of the HRM are employees of the company and assigned to a laptop that can run the HRMS. |
| **Alternatives** | Hosting the Web Client Application on a web server. |
| **Justification** | In terms of availability, the **Stand-Alone Client is much more available** as the availability of the Client is dependent on the user's laptop. If the user laptop or the client crashes, only that particular user is affected, whereas for a Web Client; if the webserver crashes, all users cannot access the client. |

| **Architectural Decisions - Clustering & Load Balancing of Servers (Active-Active)** | |
|---|---|
| **ID** | 4 |
| **Issue** | To ensure high availability and performance of the microservices. |
| **Architectural Decision** | We chose to cluster multiple instances of the same microservice for load balancing. This reduces the load on each instance and hence, **improve the performance** when the application client makes requests to the microservices. This also improves the availability as the gateway to the instances continuously check for the health of each instance and load balance the incoming traffic to the instance that is healthy. |
| **Assumptions** | All instances are configured the same and have the same computing power. |
| **Alternatives** | Active - Passive redundancy |
| **Justification** | As the company is a medium-sized business that does not have a lot of additional funds for technology, We decided that **Active-Active** is much more **value for money** as all |

| | |
|---|---|
| | servers that we are paying for are actively in use to **raise** both the **Performance** and **Availability** of the application.<br>For Active – Passive, we are essentially paying to have a standby server that will not be utilized until the Active goes down. In terms of scalability, increasing another node to an Active-Active setting will raise the performance as the Load Balancer can now spread the load to one more additional server, while the **Active – Passive setting has no benefit in terms of Performance**. |

| **Architectural Decisions - Relational Database Service (Active - Passive)** | |
|---|---|
| **ID** | 5 |
| **Issue** | Ensure the availability of the data. |
| **Architectural Decision** | We subscribed to the AWS RDS for database replication. The RDS automatically replicate the primary database on our secondary database. It also comes with a failover feature that will kick in when it detects that our primary database has failed. |
| **Assumptions** | None. |
| **Alternatives** | Active-Active Database Replication. |
| **Justification** | The main issue with Active-Active is that there is no **single source of truth**. When both databases have the same field updated at the same time, it will result in data conflict and it will be time-consuming to manually resolve the conflicts. Having conflicting data also **threatens the data integrity.** |

| **Architectural Decisions - Tiered Architecture** | |
|---|---|
| **ID** | 6 |
| **Issue** | Segregation of the presentation, business logic and data to secure the data. |
| **Architectural Decision** | We have 2 main Tiers that are further broken down to more tiers. In the **Client Tier**, we have the Application Client that mainly serves as the **Presentation Tier**, which consists of User Interface that the user interacts with and **Business Logic Tier** which holds some of the business logic.<br>In the **Server Tier**, we have the main **Business Logic Tier,** which consists of an API gateway and microservices that are held on the public subnet and private subnet respectively. They coordinate with the surrounding tier, move and process data. There is also the **Data Tier** that holds the data in the database.<br>The segregation of the Data and most of the Business Logic from being directly accessible on the Application Client is to improve the **confidentiality and ensure the integrity** of the data. This architecture can be seen in our Network Diagram, which is designed to be highly secured. The segregation also improves maintainability as the different segments are separated and can be changed independently. |

| | |
|---|---|
| **Assumptions** | None. |
| **Alternatives** | Use Architecture without or lesser tier/ segmentation. |
| **Justification** | If the alternative is implemented, any change will directly affect the other components, and everything must be changed together. For example, a change in the Business Logic will require a change in the Application Client. However, by segmenting the Presentation and Business Logic into separate tiers, we can make changes to the Business Logic without changing the Application Client. This improves the maintainability of our application. |

| **Architectural Decisions - Token Authorization** | |
|---|---|
| **ID** | 7 |
| **Issue** | Ensures the integrity of the API location |
| **Architectural Decision** | We implemented a token authorization session service. After the user successfully logs in, a security token is sent to the user's Application Client. Every time the Application Client invokes the microservice, the token must be passed through the header to make a valid request. Microservices are not directly accessible as they are hidden in VPC subnet (with no public IP). Application Client can only communicate with the gateway when authorised. |
| **Assumptions** | None. |
| **Alternatives** | OAthu 2 |
| **Justification** | OAuth 2.0 allows users to log into an application by signing in through third parties credential which trusts OAthu. In our case, we are not utilizing third party log in credentials and hence, we are not using OAuth. Instead, we uses SessionID as a security token. |

| **Architectural Decisions – Zuul API Gateway** | |
|---|---|
| **ID** | 8 |
| **Issue** | Create a single point of entry for all microservices. |
| **Architectural Decision** | API gateway will validate each arriving request, proxy/route them to the specific microservices. It will also authenticate each inbound request. The ribbon module in the gateway also provides load balancing features for microservices. It also provides a filtering function in which we can write our own filtering rules. |
| **Assumptions** | None. |
| **Alternatives** | AWS API Gateway |
| **Justification** | Zuul gateway is easier to implement as no code modification is required for the existing spring boot microservice. However, for AWS gateway to work, it requires the AWS |

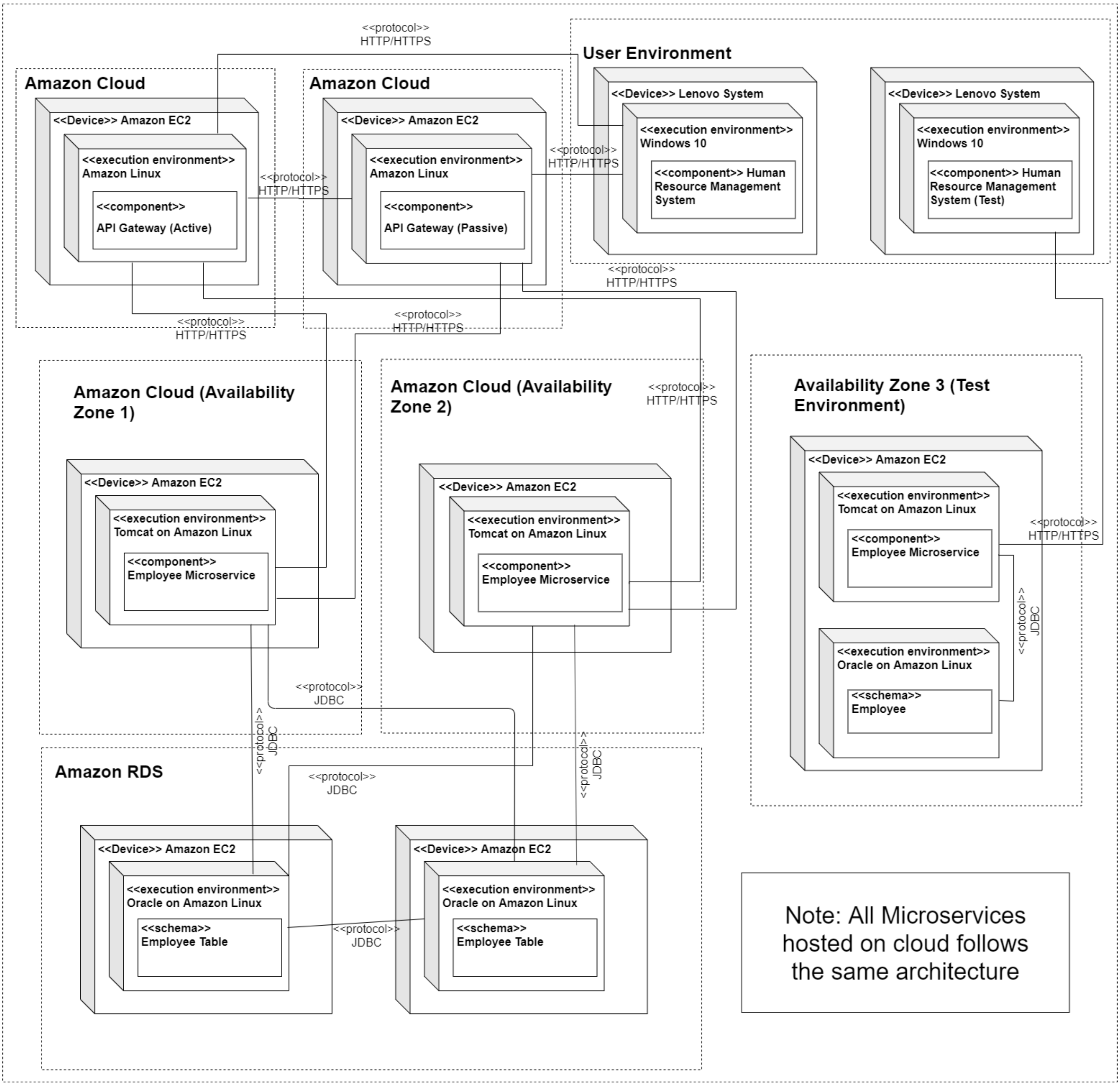
| | |
|---|---|
| | lambda, and a jar file to be uploaded to S3. It is a huge inconvenience when the microservices are being updated. Zuul is open source and free, while the AWS Gateway and lambda incur a cost. |

# Deployment View

## Work Distribution

| Name | Role | Job Scope |
|---|---|---|
| Yan Zifan | Infrastructure | Manages the servers and databases. Ensure availability of the infrastructures in both test and production environment. |
| Li Jian Wei | Security | Establish the security of the processes and data. |
| He Yicheng | Microservices Developer | Develops the microservices. Ensure the maintainability and functionality of the codebase. |
| Fong Kin Shing | Microservices Developer | |
| Chia Wei | CICD & Quality Assurance | Set Up the automated CICD process. Monitor the execution of test cases and deployment to the Test Environment. |
| Lim Jia Wei | Client Application Developer | Develops the microservices. Ensure the maintainability and functionality of the codebase. Ensure the performance of the client. |

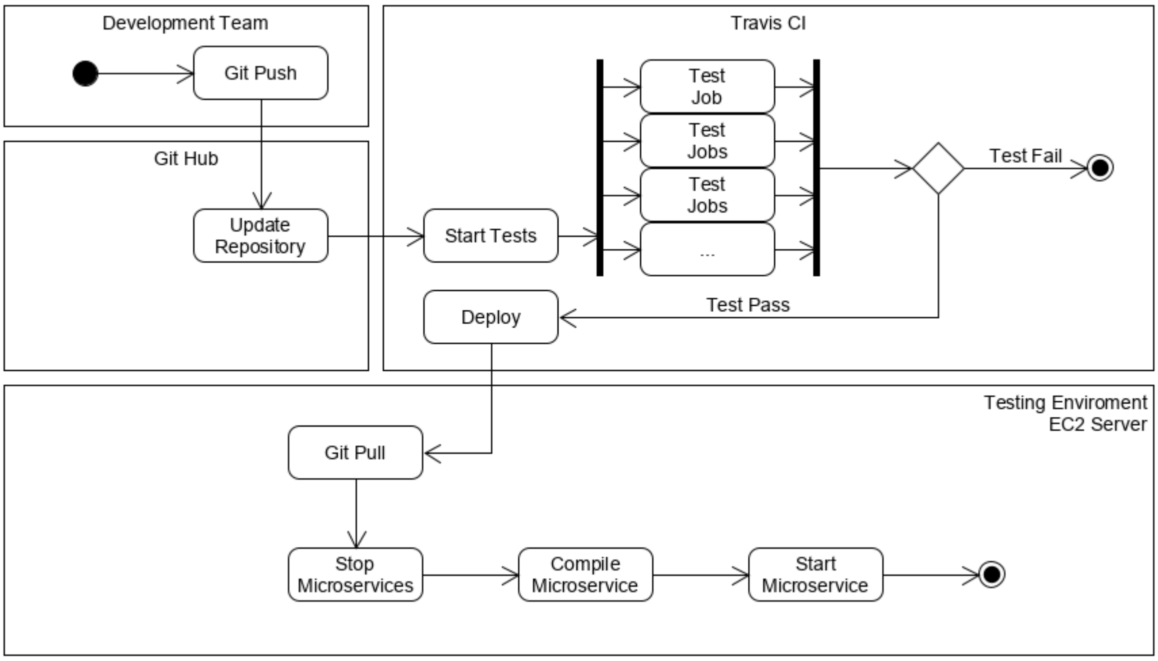
## Deployment Diagram



## Microservice Deployment Automated Testing

Once the development team has finished updating the microservices, they commit and push to Git. Git helps with versioning of the code. Upon pushing into Git, Travis CI will detect the update and automatically pull from Git and build each microservice concurrently (known as a Job). Each job is isolated, compiled and tested (using maven test) and once all jobs have successfully built, it will deploy (using HTTP to a test environment) for further testing. Otherwise, it will simply fail, and Travis CI will notify the user via email

The Test Environment, once it receives the Deploy command from Travis CI, will automatically pull from Git, halt all microservices and compile them before starting them for local testing.



## Microservice Deployment to Production

After adequate testing of the microservices, it will be manually deployed to the production server.

## Application Deployment

For the Client Application, it will be compiled into an executable file (hrms.exe) and then passed to the dedicated business user for User Acceptance Test. Upon completion of the UAT, the hrms.exe together with the jar libraries it uses will be zipped and uploaded onto a dedicated web server. The version.html on the webserver will also be changed to reflect the newest update version.

# Solution View

## Design Pattern

**Singleton** – The Session Class in the Application client is written with the Singleton Design Pattern as only one instance of Session is required for each login.

**Builder** – The EmployeeBuilder Class hides the complex creation of each Employee Object. Every time the method Load() is called, all employee details will be retrieved from the database through the microservice,the EmployeeBuilder will take in result of the API call to create the individual Employee Object and put them into the Employee ArrayList.

**Factory** – The JSONAPI Factory eases the process of calling JSON API. HTTPURLCONNECTION requires multiple lines of code to successfully make the request. Using the Factory Pattern will reduce the number of repeated codes that have to be written every time a JSON API call is made. As there are multiple types of JSON API Calls (eg. GET, POST, PUT, DELETE), the Factory Pattern is used to defer instantiation to subclasses.

**Facade** – The API Gateway is a single point of entry to our microservices. It is a façade that hides the IP of our services from the request coming in and authenticates them before allowing access to our services, providing a layer of security. The API Gateway directs the incoming traffic to the correct microservice, hence reducing the difficulty of managing multiple services. It also checks the health of each service and load balance the incoming traffic. This ensures the availability and performance of our service.

## Auto Client Updater

The Client is updated automatically upon starting of the client launcher. This eases the process of updating the Client as it can be done remotely without any human assistance. The Launcher will check if the existing application version is the same as the latest version stated on the webpage (version.html). If it is different, the Launcher will download the latest Client (zip file) from our website. It will proceed to unzip it and launch the client application (hrms.exe).

## Remember My Username

We added the option to remember the username as a Usability Feature. The username is stored at Local Data and it will be retrieved loaded on the Login Panel whenever the Application Client is opened.

## Microservice Architecture

Breaking down to microservices improves maintainability as it modularises the functionality of the initial Application Client. It is easier to make changes, test and each microservice can be deployed independently. Failure in an individual microservice will be isolated and not affect the other services. This also makes analysing of the root cause issue easier.

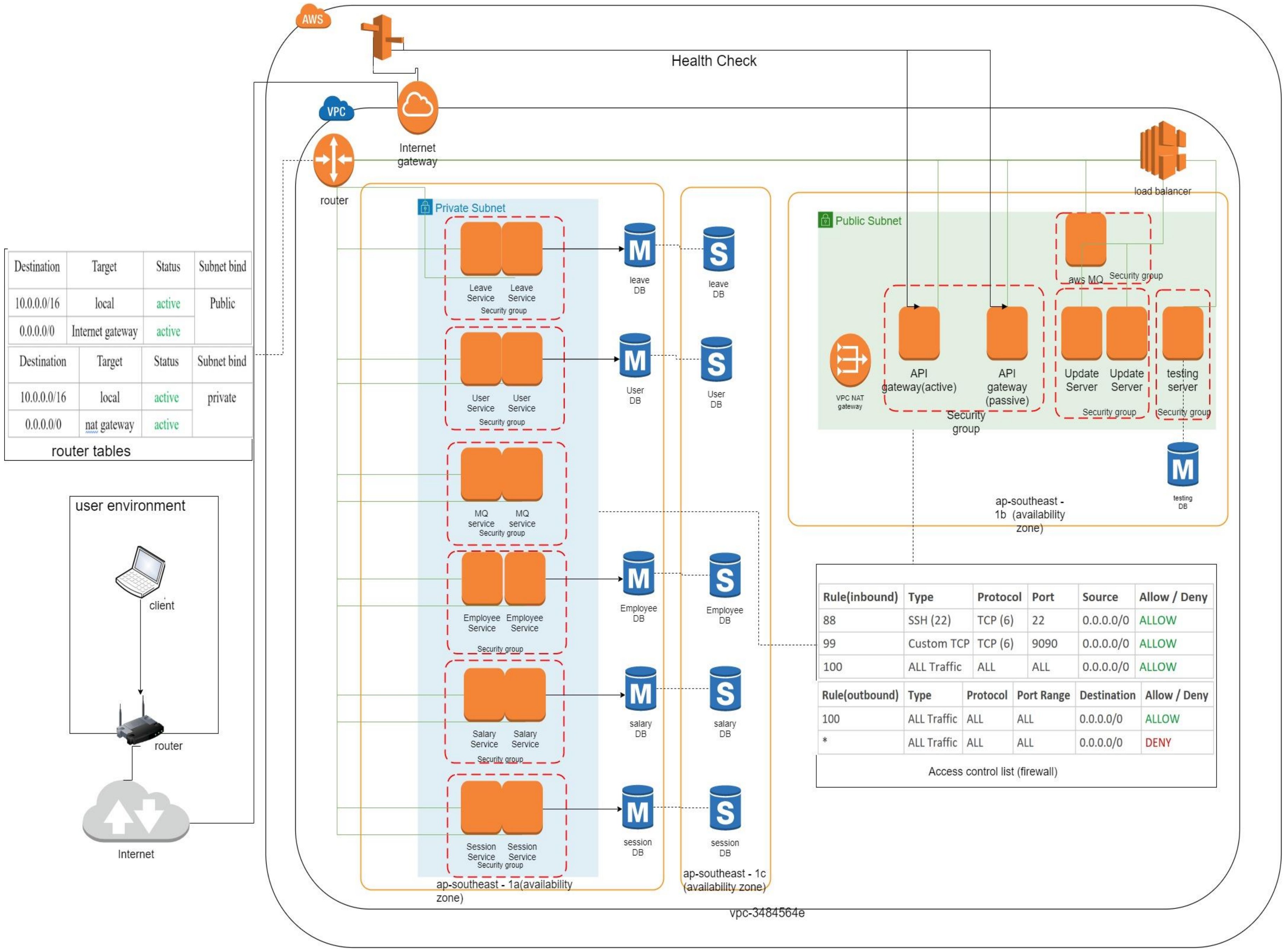
## Client Service Architecture

Initially, all user data are pulled from the database to be used for the login authentication feature of the monolithic application. Sophisticated users can reverse engineer our client application code and get all user information. Hence, we moved the authentication of the user logging into the client application to the User microservice and the Client no longer is able to retrieve all user details from the database. Moving the authentication to a service will improve the security of the data that can be assessed by the client application.

## Tiered Architecture

In our Tiered Architecture, the Presentation Layer is largely separated from the Business Logic Tier. Hence, changes in Business Logic that are made in the microservice/gateway will not affect the Application Client. This form of modularization improves maintainability. In addition, since authentication is done in the gateway, which is separated from the application, it also improves the security because even when the Application Client is compromised, the Business Logic (microservice and gateway) and most importantly the Data is still safe.

## Network Diagram



| Destination | Target | Status | Subnet bind |
|---|---|---|---|
| 10.0.0.0/16 | local | active | Public |
| 0.0.0.0/0 | Internet gateway | active | |

| Destination | Target | Status | Subnet bind |
|---|---|---|---|
| 10.0.0.0/16 | local | active | private |
| 0.0.0.0/0 | nat gateway | active | |

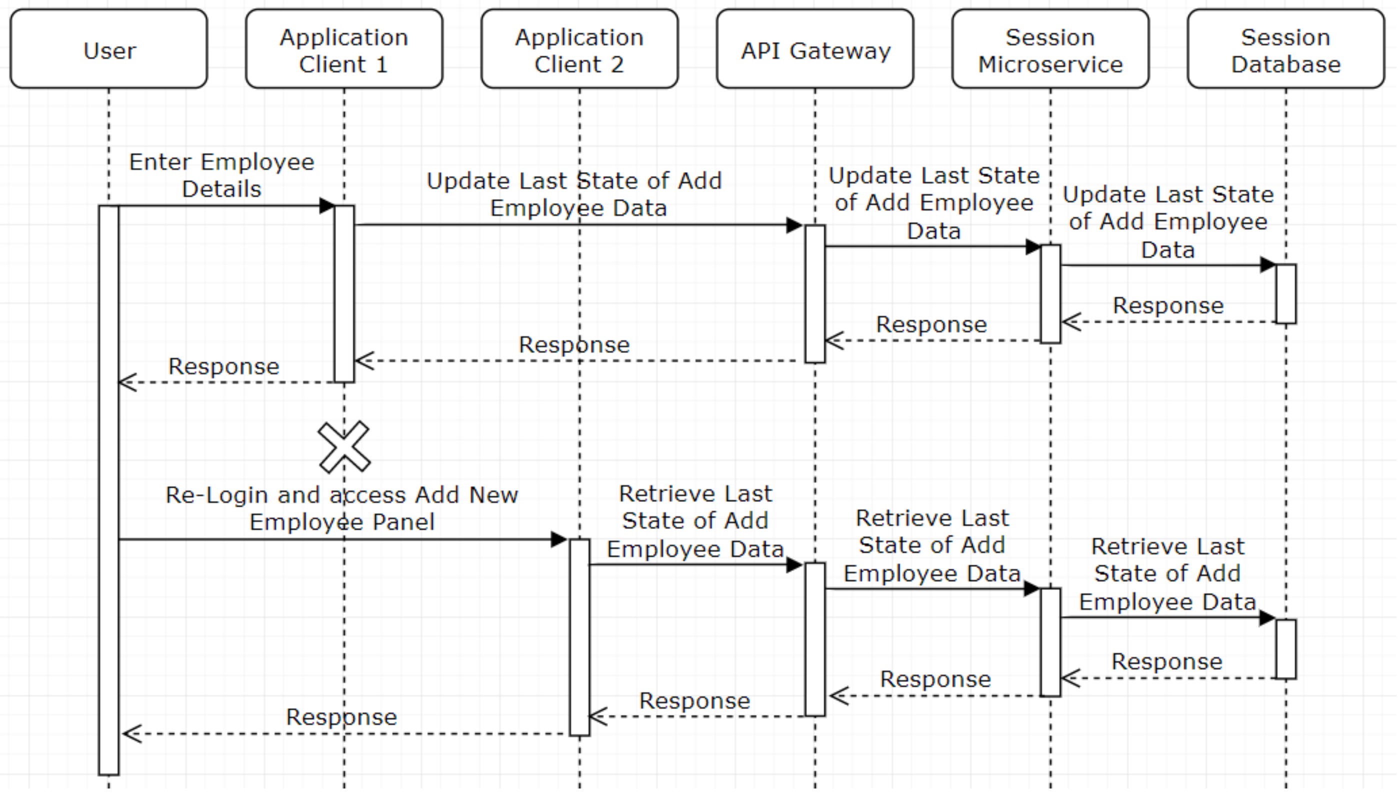
router tables

| Rule(inbound) | Type | Protocol | Port | Source | Allow / Deny |
|---|---|---|---|---|---|
| 88 | SSH (22) | TCP (6) | 22 | 0.0.0.0/0 | ALLOW |
| 99 | Custom TCP | TCP (6) | 9090 | 0.0.0.0/0 | ALLOW |
| 100 | ALL Traffic | ALL | ALL | 0.0.0.0/0 | ALLOW |

| Rule(outbound) | Type | Protocol | Port Range | Destination | Allow / Deny |
|---|---|---|---|---|---|
| 100 | ALL Traffic | ALL | ALL | 0.0.0.0/0 | ALLOW |
| * | ALL Traffic | ALL | ALL | 0.0.0.0/0 | DENY |

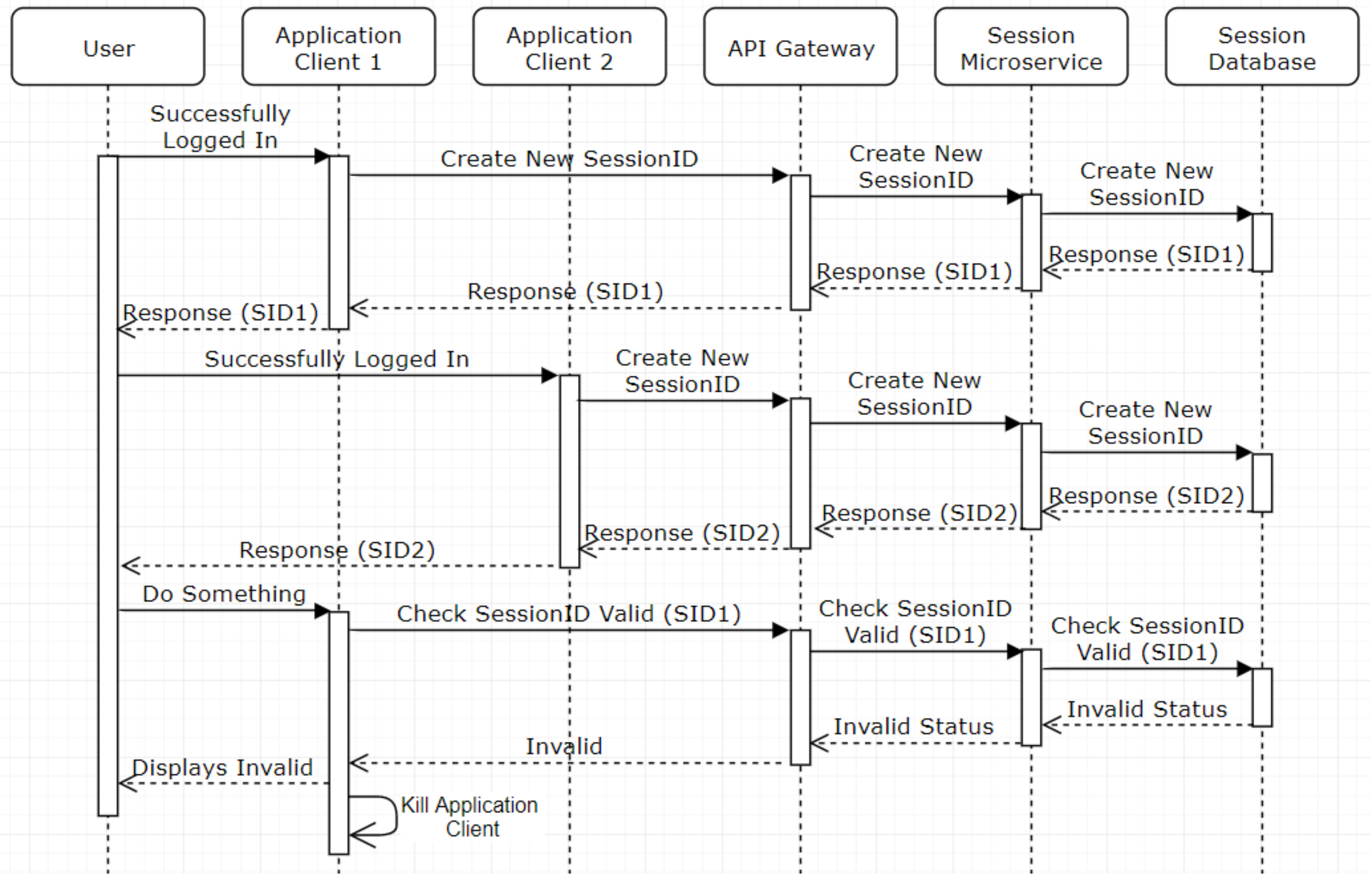
Access control list (firewall)

## State Management


User
Application Client 1
Application Client 2
API Gateway
Session Microservice
Session Database
Enter Employee Details
Update Last State of Add Employee Data
Update Last State of Add Employee Data
Update Last State of Add Employee Data
Response
Response
Response
Response
Re-Login and access Add New Employee Panel
Retrieve Last State of Add Employee Data
Retrieve Last State of Add Employee Data
Retrieve Last State of Add Employee Data
Response
Response
Response
Response

When the user types in anything on the Add New Employee Panel, the last state of the changes made will be stored in the database at an interval of 1 minute. In the event that the Application Client crashes, the user can re-login to the Application Client and the last state saved in the database will be restored.

## Session Management



Every login is tied to the UserID and SessionID. When the **same UserID** logins again, the associated SessionID (**SID1**) will be replaced by a new SessionID (**SID2**) in the database. Thus, when the user attempts to perform any activity that requires the microservices on the **first login (SID1),** the activity will be **halted**, and the client application will be **killed** with a multiple login warning. This is because **SID1** is **invalid** and only the login with **SID2** and the **same UserID** can proceed with activities that interact with the microservices.

## Integration Endpoints

| Source System | Destination System | Protocol | Format | Communication Mode |
|---|---|---|---|---|
| Application Client | Gateway | HTTPS | JSON | Synchronous |
| API Gateway | Application Client | HTTPS | JSON | Synchronous |
| API Gateway | Microservice | HTTP | JSON | Synchronous |
| Microservice | API Gateway | HTTP | JSON | Synchronous |
| Message Microservice | Amazon MQ | JMS | Text | Synchronous |
| Amazon HQ | Message Microservice | JMS | Text | Synchronous |

## Hardware/Software/Framework/Services Required

### Hardware

| No | Item | Quantity | License | Buy / Lease | Cost (Optional) |
|---|---|---|---|---|---|
| 1 | EC2 Instances | 18 | N.A. | Leasing Monthly | $164.39 |

### Software

| No | Item | Quantity | License | Buy / Lease | Cost (Optional) |
|---|---|---|---|---|---|
| 1 | Tomcat | 18 | N.A. | N/A | $0 (Comes with Springboot) |
| 2 | Linux Kernel 4.14 | 18 | NA | NA | $0 (Comes with EC2) |
| 3 | Java Runtime Environment | 18 | NA | NA | $0 |

*Every EC2 Instances is running Tomcat on Linux OS with Java Runtime Environment.

### Framework/Platform

| No | Name | Component |
|---|---|---|
| 1 | Spring Boot | JPA |
| 2 | Netflix Open Source | Zuul |

| | | |
|---|---|---|
| | | Ribbon |
| | | Eureka |

*The EC2 instances that are hosting the Microservice comes with Spring Boot.
*The EC2 instances that are hosting the API Gateway comes with Zuul, Ribbon, Eureka and Spring Boot.

**Services**

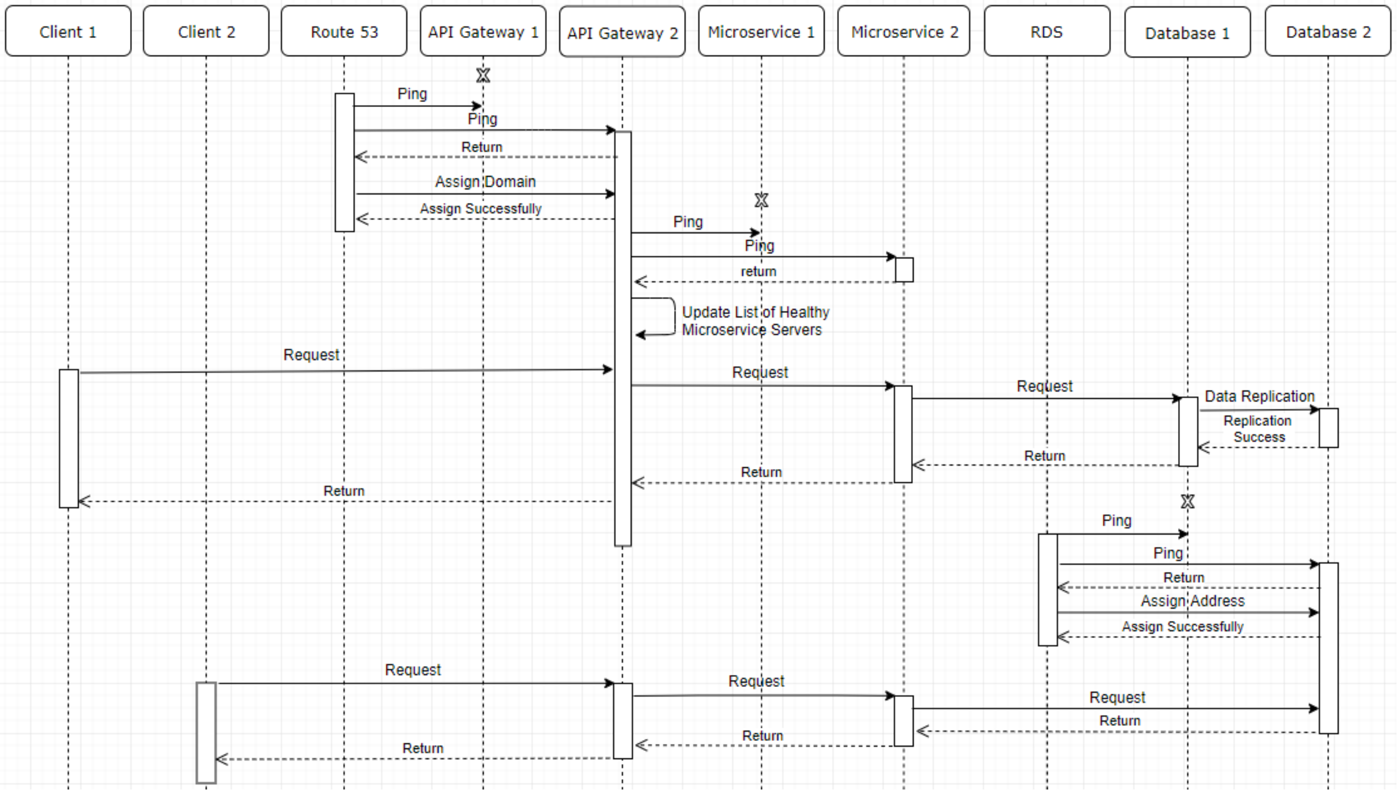
| No | | Item | Quantity | License | Buy / Lease | Cost (Optional) |
|---|---|---|---|---|---|---|
| 1 | AWS | RDS | 6 | N.A. | Leasing monthly | $305.13 |
| 2 | | MQ | 1 | N.A. | Leasing monthly | $10.69 |
| 3 | | Route 53 | 1 | N.A. | Leasing monthly | $0.50 |
| 4 | | AWS Shield | 1 | N.A. | Leasing monthly | $3000 |
| 5 | | Elastic IP | 6 | N.A. | Leasing monthly | Free |
| 6 | | VPC | 1 | N.A. | Leasing monthly | $0.045/GB |
| 7 | | CloudWatch | 18 | N.A. | Leasing monthly | $37.8 |

## Availability View

| Node | Redundancy | Clustering | | | Replication (if applicable) | | | |
|---|---|---|---|---|---|---|---|---|
| | | Node Config. | Failure Detection | Failover | Repl. Type | Session State Storage | DB Repl. Config | Repl. Mode |
| User Microservice | Horizontal | Active-Active | Ping | Load-Balancer | RDS | - | Master-Slave | Synchronous |
| Employee Microservice | Horizontal | Active-Active | Ping | Load-Balancer | RDS | Database | Master-Slave | Synchronous |
| Leave Microservice | Horizontal | Active-Active | Ping | Load-Balancer | RDS | Database | Master-Slave | Synchronous |
| Message Queue | Horizontal | Active-Passive | Ping | Load-Balancer | - | - | - | - |
| Session Microservice | Horizontal | Active-Active | Ping | Load-Balancer | RDS | - | Master-Slave | Synchronous |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| API Gateway | Horizontal | Active-Passive | Ping | Route53 | - | - | - | - |
| Salary Microservice | Horizontal | Active-Active | Ping | Load-Balancer | RDS | - | Master-Slave | Synchronous |
| Testing Microservices | - | - | - | - | - | - | - | - |

## Failure Detection and Fail-Over



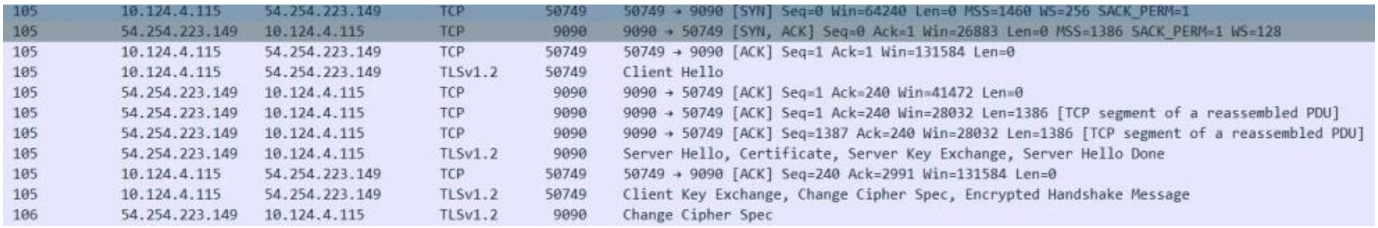
The **Route 53** checks on the health of both **API Gateway 1** and **API Gateway 2** at an interval of 1 minute. The Route 53 checks by **pinging** the API Gateway. If API Gateway 1 is down while 2 is up, **Route 53 will point the domain to API Gateway 2**. Subsequent HTTP request entering the domain will be routed to API Gateway 2.

The **Zuul API Gateway (ribbon module) load balances** the incoming request using **Round Robin**. It continuously **pings** all the servers that it knows of and **update a list of healthy servers** to route the request to. When a request comes in, it will be routed to a server that is **alive** and of the required Microservice.

**The AWS Relational Database Service** with **Multi-AZ** provides failure support for our databases. It automatically provisions and maintains a synchronous standby replica of our primary database in a different availability zone. The **primary database is replicated on a standby replica** to provide **high availability** and **data redundancy**. When the primary database goes down, Amazon’s failover technology will kick in and point the domain to the replica database. We can access the database through the endpoint provided by AWS.

## Security View

| No | Asset / Asset Group | Potential Threat/ Vulnerability Pair | Possible Mitigation Controls |
|---|---|---|---|
| 1 | Data | Threat: Man-In-The-Middle-Attack<br>Vulnerability: Unencrypted network tunnel | Implemented HTTPS (Based on SSL certificate) Communication (Prevent attackers from seeing any information the client submits to the gateway) between Client and Gateway<br>*SSL certificate which uses PKI framework and asymmetric cryptography to encrypt the message/information. |



```
105   10.124.4.115     54.254.223.149   TCP       50749   50749 → 9090 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
105   54.254.223.149   10.124.4.115     TCP        9090   9090 → 50749 [SYN, ACK] Seq=0 Ack=1 Win=26883 Len=0 MSS=1386 SACK_PERM=1 WS=128
105   10.124.4.115     54.254.223.149   TCP       50749   50749 → 9090 [ACK] Seq=1 Ack=1 Win=131584 Len=0
105   10.124.4.115     54.254.223.149   TLSv1.2   50749   Client Hello
105   54.254.223.149   10.124.4.115     TCP        9090   9090 → 50749 [ACK] Seq=1 Ack=240 Win=41472 Len=0
105   54.254.223.149   10.124.4.115     TCP        9090   9090 → 50749 [ACK] Seq=1 Ack=240 Win=28032 Len=1386 [TCP segment of a reassembled PDU]
105   54.254.223.149   10.124.4.115     TCP        9090   9090 → 50749 [ACK] Seq=1387 Ack=240 Win=28032 Len=1386 [TCP segment of a reassembled PDU]
105   54.254.223.149   10.124.4.115     TLSv1.2    9090   Server Hello, Certificate, Server Key Exchange, Server Hello Done
105   10.124.4.115     54.254.223.149   TCP       50749   50749 → 9090 [ACK] Seq=240 Ack=2991 Win=131584 Len=0
105   10.124.4.115     54.254.223.149   TLSv1.2   50749   Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
106   54.254.223.149   10.124.4.115     TLSv1.2    9090   Change Cipher Spec
```

| | | | |
|---|---|---|---|
| 2 | Services | Threat: Malicious HTTP Requests<br>Vulnerability: API Gateway exposed to public. Allowing HTTP Request. | Implemented Token Authorization verification (used to verify the caller is a valid user before allowing it to use other services).<br>Microservices are not directly accessible as they are hidden in VPC subnet (with no public IP). They are accessible only via Gateway.<br>Microservices are only allowed to talk to Gateway (This is set via AWS security group). |
| 3 | Server | Threat: Attackers might try to disrupt normal traffic of our server by conducting a distributed denial-of-service (DDoS) attack.<br>Vulnerability: API Gateway exposed to public. Allowing HTTP Request. | Implemented Load Balancer which has an off-loading function that defends our server against distributed denial-of-service (DDoS) attack. It does this by shifting attack traffic from one server to all other servers to prevent overloading servers, optimize productivity, and maximize uptime.<br>AWS shield which provides always-on detection and automatic inline mitigation that minimize our application downtime and latency which protect our application against DDoS attack. |
| 4 | Data | Threat: SQL Injection which executes malicious SQL statements to control our database server behind the gateway. It is also | Reduce our attack surface. We did this by removing all unnecessary database functionality to prevent hackers from taking advantage of it.<br>Implemented user access control with appropriate privilege such that only admin-level privileges can access the |

| | | | |
|---|---|---|---|
| | | possible for attackers to use SQL injection vulnerabilities to bypass our security measures, go around authentication and authorization of the application and retrieve the content of the entire SQL database.<br>Vulnerability: Lack of input validation | database directly, all normal users need to go through microservices such as “Employee service” to access the database.<br>MySQL by default will be able to validate user input with functions such as “mysql_real_escape_string()” to ensure dangerous characters such as ` are not passed to a SQL query in data. We also enable input validation at “User service” for more layers of security.<br>Both API Gateway and microservice will validate the user’s input. |

## Performance View

| No | Description of the Strategy | Justification | Performance Testing (Optional) |
|---|---|---|---|
| 1 | The Microservices are load balanced at the API Gateway using Round Robin. | Spreading the requests to multiple servers will ensure the **capacity** to allow 100 concurrent users to access the services within 2 seconds **time response** is met. | |
| 2 | The updating of Last State Data and loading of Notifications in the Client Application is Multi-Threaded. | Processing non-time-critical data in the background on another thread while running the main client application on the main thread will **reduce the delay** in loading of the main application, hence, allowing the user to continue using the application with minimal delay. | |
| 3 | The Session class within the Application Client is implemented with the Singleton Design Pattern | Instantiating the Session Class only once and calling it whenever it is required will **remove the possibility** of instantiating a second time and **wasting resources**. There should also be only one instance of Session per login to the client. | |
| 4 | The Employee Data is Pre-Fetched and loaded on the Employee List Panel when a user from the Human Resource logins. | Pre-fetching and loading the Employee List upon user login ensures a smooth transition to the Employee List View as the user do not have to wait for the Employee List to load on the panel.<br>As the company continues to grow, there will be even more employees. Hence, it can potentially impact the loading time of the entire Employee List. | |