ITSA Project Report

GitHub Team Name: 神啊救救我吧

Team Members:

1. Chan Yu Siang
2. Gan Min Xuan Darren
3. Khoo Zhi Tao Sebastian
4. Muhammad Syukri Bin Rahiman
5. Yong Weng Kiat

Background and Business Needs

The HR Management System is one of the systems intended for use by employees in a company, Shen Ah Pte Ltd.

As Shen Ah Pte Ltd continues with their rapid expansion plans, having increased in size from 2 employees to over 50 employees in the span of a year, the HR Management System was built to transition the manpower management away from a Google Sheets-based system, thus enabling the human resource department to scale up their capacity quickly. For example, employees currently have to submit leave applications through a Google Form, afterwhich a HR executive reviews the request and updates the manpower and payroll system, before sending the employee a confirmation email. This labour-intensive process takes at least 3 working days to complete. The HR Management System therefore takes these time-intensive processes into account and represents a paradigm shift in how the company manages their employees.

In addition, the HR Management System must be easily accessible over the Internet as some of Shen Ah Pte Ltd's employees work from home. Ultimately, the HR Management System would represent an all-in-one solution for all manpower management needs for Shen Ah Pte Ltd.

Stakeholders

Stakeholder	Stakeholder Description
HR	HR will be responsible for adding and maintaining the accuracy of employee data on the HR Management System
Accounts	Responsible for generating and verifying the accuracy of employee's monthly payslips
Managers	Managers will use the HR Management System to view and approving

	upcoming employee leave requests
Employees	Employees will use the HR Management System to view and apply for leave

Key Use Cases

Users login	
Use Case ID	UC001
Description	User enters website URL into the browser, web pages are served by an **HR Web Application on Apache.** User logins to the system using his company login credential, the web application, through the **API Gateway running Flask, integrates with Main Service on Spring Boot** for authentication. Upon a user login, the web application generates a random session key and stores it in its database. The frontend will store the session key in a session. This allows the user to access authorized-users-only webpages.
Actors	Employee, API Gateway, HR Web Application, Main Service, Employee Service
Main Flow of events	1. User attempts logins to HR system application using their company registered account 2. Authentication is successful
Alternative Flow of events	1. User attempts logins to HR system application using their credentials 2. Authentication is unsuccessful
Pre-conditions	Users already have a company account
Post-conditions	Employee redirected to the main page of HR system application upon successful authentication

Employee applying for leave	
Use Case ID	UC002
Description	Employee enters the leave application page served by the **HR Web Application** running on **Apache** and the system invokes a call to the **Main Service** to retrieve employees

	record to be displayed on the page. The employee will then proceed to select his leave options such as start and end date and press the submit button. Upon pressing submit, the **Leave endpoint on Main Service** will trigger an insert record into its **database**. The approving manager will also receive and inbox message stating that a new leave request have been made.
Actors	Employee, Employee Microservice, Main Service, HR Web Application, API Gateway
Main Flow of events	1. Employee views the calendar and select the duration of the leaves 2. Employee select the type of leave 3. Employee clicks the submit button and wait for approval from his/her manager 4. Manager receives
Alternative Flow of events	1. Employee views the calendar and select the duration of the leaves
Pre-conditions	Employee is registered in the company's system
Post-conditions	Employee will receive a message in his/her inbox notifying the results of the leave request

Approval of leave by manager	
Use Case ID	UC003
Description	Department Manager enters the **leave approval page** served by the **HR Web Application** and the system uses the **API Gateway** to invoke a **call to the Main Service** to retrieve all upcoming *pending* and *confirmed* leaves in the manager's department. The manager then proceeds to approve or reject the pending leave applications listed, which then makes another call through the **API Gateway** to the **Main Service** to update the employee's leave status and remaining leave balance. At the same time, notifies the employee that their leave application status has been updated via the built-in app inbox
Actors	Department Manager, Main Service, HR Web Application,

	API Gateway
Main Flow of events	1. Manager views the leave records table containing the leave application records 2. Manager clicks on the approve button to approve the employee's leave application
Alternative Flow of events	1. Manager clicks on the reject button to reject the employee's leave application
Pre-conditions	Employee submitted a leave application
Post-conditions	Employee's leave balance is deducted accordingly based on whether the request was approved or rejected.

Create and assign new employees to their respective departments	
Use Case ID	UC004
Description	Using the web interface served by the **HR Web Application on Apache**, the HR manager clicks on 'add a new employee' button and fills up the new employee's details. Upon clicking the confirm button, the system uses the **API Gateway** to call the **Employee Microservice** to add the new employee into the **database**
Actors	HR Manager, Employee Microservice, API Gateway, HR Web Application
Main Flow of events	1. HR Manager navigates to employee management page 2. Click on add new employee buton 3. Enter employee information 4. Click on the confirm button
Alternative Flow of events	-
Pre-conditions	-
Post-conditions	User is created and may login with the newly created credentials.

View Employee's performance and update accordingly to their manager's feedback	
Use Case ID	UC005
Description	The Accounts Manager enters the payroll page served by the **HR Web Application** running on **Apache.** The Accounts Manager then search for the employee and click search. The system will then use the **API Gateway** to call the **Employee Microservice** to retrieve the data of the Employee from the **database**. The HR manager will review the feedback given by the employee's manager and enter the performance evaluation into the system. The system will then use the **API Gateway** to invoke **Main Service** to add/update the employee's performance. The payroll table will show the updated pay after appraisal.
Actors	Department Manager, Employee Microservice, Main Service, HR Web Application, API Gateway
Main Flow of events	1. Department Manager searches for the employee under his department whose performance evaluation he/she would like to view 2. Clicks on the "search" button 3. Department Manager updates the performance evaluation sheet. 4. Clicks on the "submit" button 5. The payroll of the employee is generated successfully
Alternative Flow of events	-
Pre-conditions	Employee has worked in the current month Employee's manager has submitted his/her feedback to the HR department
Post-conditions	Employee's performance evaluation has been updated and his/her payroll is generated.

Key Architectural Decisions

Architectural Decision - Separation of employee functionality into microservice	
ID	A-001
Issue	As the company is expanding, the new HR system needs to be designed for scalability, modularity and maintainability. If we were to encapsulate all functionality into a monolithic application, it would make future changes to the existing codebase or the addition of new features extremely difficult as the entire codebase has to be retested and redeployed. In addition, the resource usage of a single monolithic app would result in a single point of failure when the application goes down.
Architectural Decision	We chose to separate out one of the features - employee management out from the original monolithic application and turn it into a microservice that exposes REST APIs related to employee management.
Assumptions	-
Alternatives	Monolithic application
Justification	Considering that the company is expecting to grow rapidly, a microservice architecture will make it easier to (horizontally and vertically) scale individual modules should the need arise

Architectural Decision - Builder Design Pattern for creating new Employees	
ID	A-002
Issue	As we separate our employee to another microservice, When creating a new employee, we also need to create their leave count, payroll and session in the Main-Application database which is separated from the Employee Service. Hence, there are different parts to “build” when creating an employee.
Architectural Decision	Builder Design Pattern to build the complex Employee object
Assumptions	-
Alternatives	Instead of doing a builder, have our Employee Microservice

	contain the logic for invoking API endpoints on the Main Service directly when creating an Employee.
Justification	We decided to use Builder as it helps us to create the complex Employee object. By using a builder pattern, we are able to remove or add any building parts without modifying existing codes used to create the employee and will only need to update the builder to call the necessary building parts. On the other hand, we are also loosely coupling our employee microservice and main application.

Architectural Decision - API Gateway (Facade)	
ID	A-003
Issue	The frontend would be directly calling our API endpoints and a change in the endpoint would require a change in the frontend code as well due to the tight coupling. Security concerns of exposing our API endpoints.
Architectural Decision	Encapsulate all endpoint calling into a facade API Gateway to only allow a single point of entry for clients to the API endpoints
Assumptions	API Gateway has access to all other endpoints through the internet or intranet
Alternatives	Have the Web Application directly invoke the API endpoints
Justification	The API Gateway would serve as the single point of entry for clients to our microservice APIs. This helps with enhancing security by preventing unauthorized usage of the API even if users know our HTTP endpoints. The facade design pattern also hides the logic and calling order of APIs, as well as their intermediate outputs when executing complex operations (such as those for the leave application). Lastly, the API Gateway lets us decouple the frontend web application from the backend API. When changes to APIs and endpoints are made, the only change that needs to be made is on the API Gateway as opposed to changing every single call on the frontend if we were to directly have the frontend call the APIs.

Architectural Decision - Usage of AWS Cloud	
ID	A-004
Issue	The team was given the freedom to choose our tech stack and deployment environment, being able to choose between local deployment, cloud, or a mix of both. The choice would affect future decisions such as: development strategy, CI/CD pipeline, availability, security and performance designs. Therefore it was important that the right environment that everyone was comfortable with was chosen
Architectural Decision	Our team chose to deploy our backend on cloud using AWS instead of doing a local implementation
Assumptions	-
Alternatives	Do a local deployment using Git and/or Docker
Justification	Firstly, this was chosen for the ease of development. By hosting our environment on the cloud, we do not have to worry about configuring local environments and compatibility differences between different local development environments. Another reason for choosing AWS was for the ease of availability design. With features like Elastic Beanstalk offering quickly configurable load-balancing, SSL, automatic scaling and failover, it made things easier and quicker to develop and deploy

Development View

Development Strategy

We built iterative and functionally, where each member is assigned specific tasks to work on concurrently. This meant that we could not all push to Master as this would almost definitely result in conflicts. As such, we branched a ‘Development’ branch from Master, and individual team members then further branched from the Dev branch into their own branches as we worked on the use cases and functionality. When the individual functionality is completed, it is tested against the code in Development. Once that is done, the code will then be pushed into GitHub for automated deployment and testing through our CI/CD pipeline.

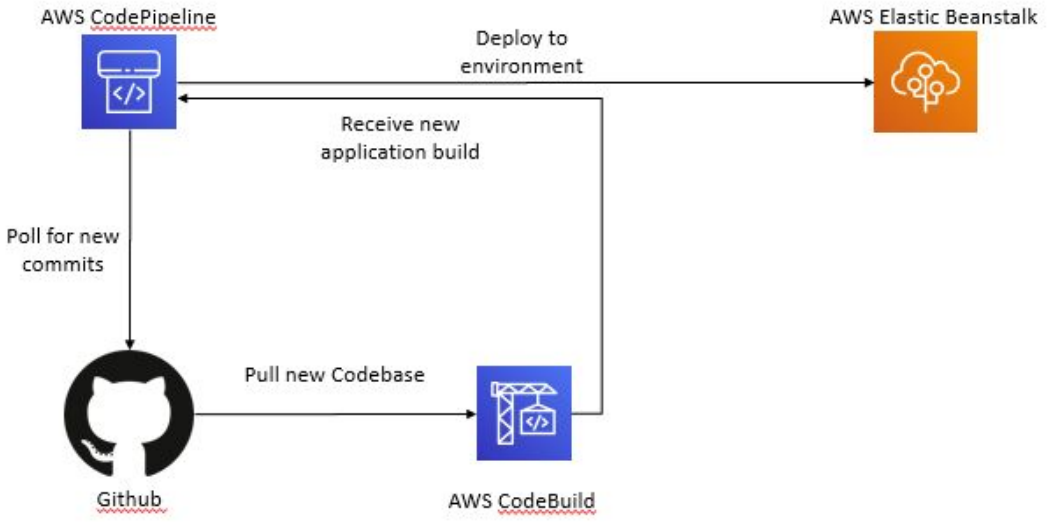
Testing

Newly pushed code for the API Gateway is automatically tested using AWS CodeBuild. A Pytest test case is run to ensure basic CRUD functionality of the employee endpoint on the API Gateway Once the test case passes, the Flask project is built into a zip file and automatically deployed according to the deployment details outlined below.

Deployment

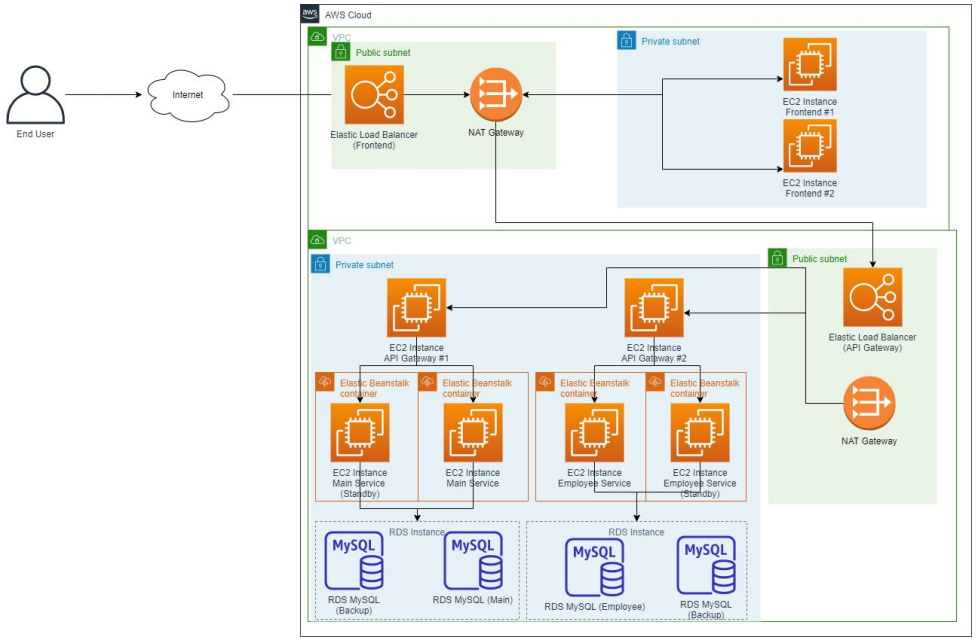
Continuous Deployment is achieved using AWS CodePipeline. AWS CodePipeline automatically polls the team's Github repository for new commits pushed onto the 'development' branch. Once a new commit is detected, it automatically uses CodeBuild to load the newly pushed code for Employee Microservice and Main Service, and automatically packages and builds the JAR file for deployment. The JAR file is automatically deployed onto Elastic Beanstalk through CodePipeline onto both main and standby application environments with zero downtime. The same is done for the Python Flask API Gateway, but without the compiling stage since Python does not require a build/compile stage and instead is directly copied from version control to deployment.

CI/CD Pipeline Flow



Solution View

Overall Architecture



Maintainability

In order to have better maintainability of the code in the future, the team attempted to incorporate Gang of Four design patterns where appropriate. We have implemented a Facade design pattern in our API Gateway, encapsulating the logic of complex API calls and obfuscating the inner workings of the backend code running on Java Spring-Boot.

The builder design pattern will also help in our maintainability as we can maintain parts of the builder codes that needs to be maintained. In the event of having to create an employee differently, we can always create another builder method to handle the building of the new requirements. Builder (part a,b,c,d) introduce new part and need to change part -> create new builder that takes in Builder (part, a,c,d,e)

Integration Endpoints

Source Sys.	Destination Sys.	Protocol	Format	Comms. Mode
Web Server*	Load Balancer (API Gateway)	HTTPS	JSON	Synchronous
Load Balancer (API Gateway)	API Gateway	HTTP	JSON	Synchronous
API Gateway	Main Service	HTTP	JSON	Synchronous

API Gateway	Employee Microservice	HTTP	JSON	Synchronous
Employee Microservice	Employee Database	JDBC	MYSQL	Synchronous
Main Service	Main Database	JDBC	MYSQL	Synchronous

Software Required

Component	Environment	Quantity
HR Web Application*	Apache on Ubuntu 18.04	2
API Gateway	Flask on Ubuntu 18.04	2
Employee Microservice	Spring Boot on Ubuntu 18.04	2
Main Service (Monolith)	Spring Boot on Ubuntu 18.04	2

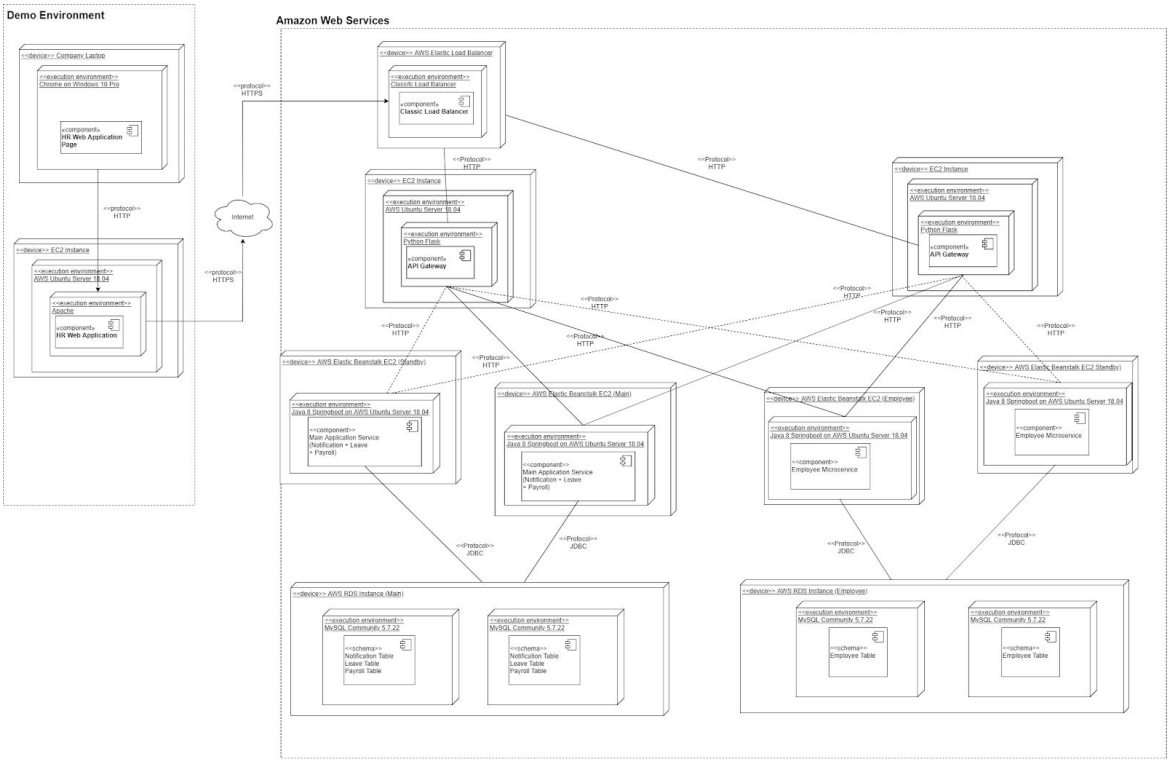
* for the demo, we will be using a local deployment instead, but for a full deployment it will be on the cloud as well

Cloud Infrastructure Required

Following the AWS high-level architecture, the table lists the components needed for implementation

No.	Item	Quantity
1	Elastic Load Balancer	2
2	EC2 Compute Instance	6
3	RDS MySQL DB Instance	2 (With multi-zone support)
4	NAT Gateway	2

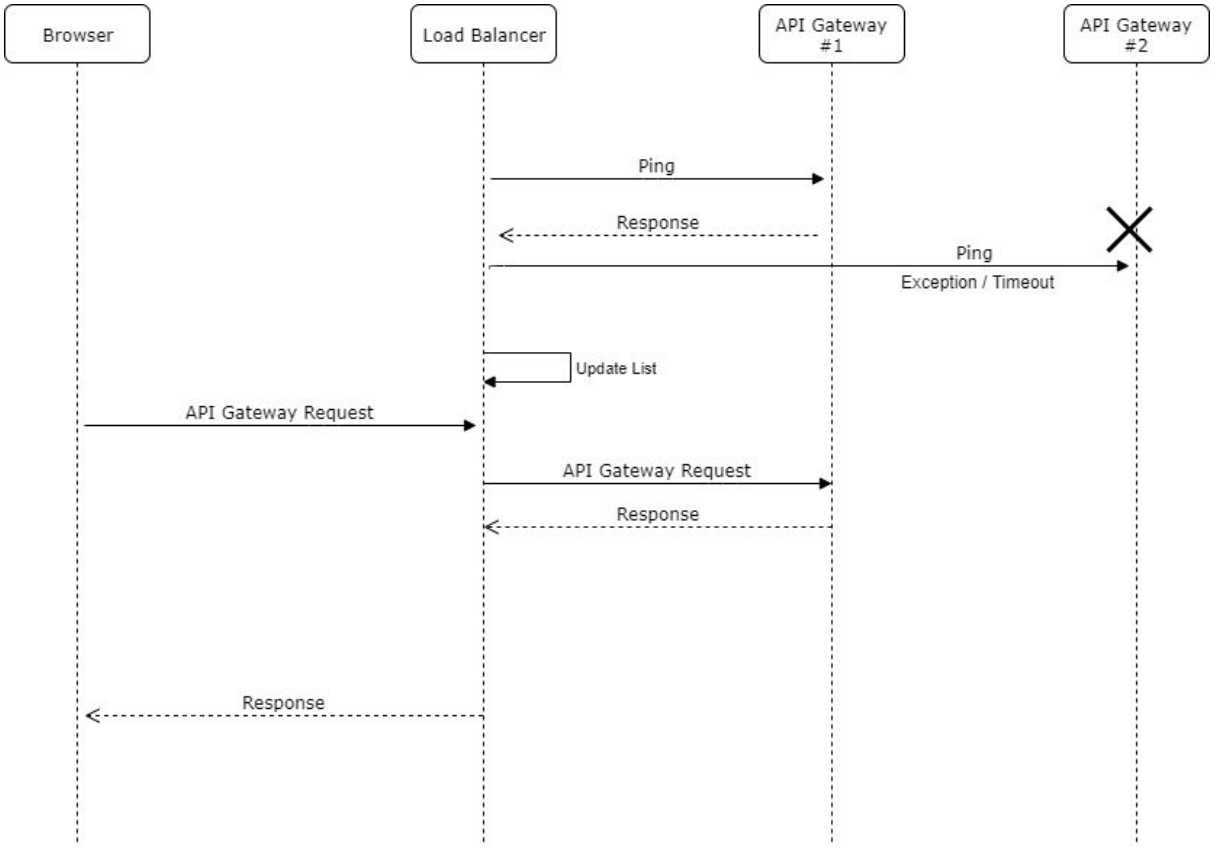
Deployment Diagram



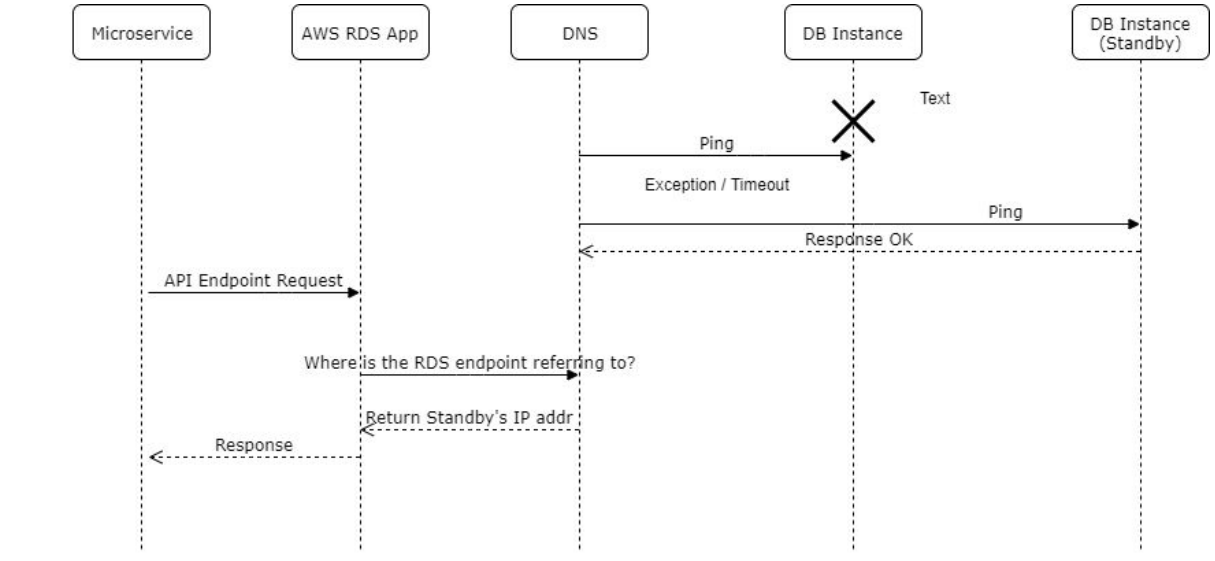
Availability View

Node	Redundancy	Clustering			Replication			
		Node Config	Failure Detection	Failover	Repl. Type	Session State Storage	DB Repl. Config	Repl. Mode
API Gateway	Horizontal Scaling	Active-Active	Ping	Load-balancer	Session	Database	-	-
Employee microservice	Horizontal Scaling	Active-Passive	Ping	Client (API Gateway)	-	-	-	-
Main Service	Horizontal Scaling	Active-Passive	Ping	Client (API Gateway)	-	-	-	-
Database	Horizontal Scaling	Active-Passive	Ping	DNS	DB	-	Master - Slave	Synchronous

Load Balancer Failover

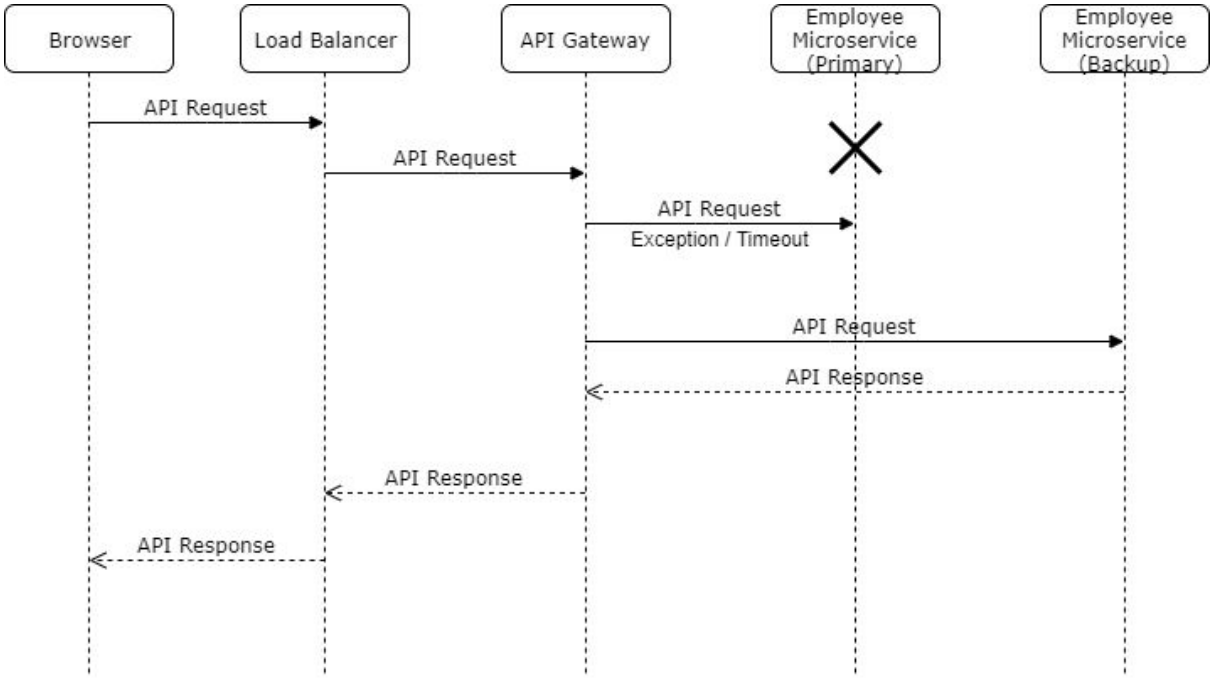


DB Failover



For AWS RDS, once a failover happens, the original master becomes the new standby

Microservice Failover



Security View

No.	Asset/Asset Group	Potential Threat/Vulnerability Pair	Possible Mitigation Controls
1	Data in Database	SQL Injection	Input validation, escape special characters,
2	Data in Database	CSRF	Session token
3	API Gateway and Main Backend, Data in Database	Main in the Middle Attack	HTTPS / SSL

Input Validation (Both Frontend and Backend)

We made extensive use of Spring Boot's built-in input validation features. For example, when we specify path and query parameters, we can set them to non-String data types. When an SQL injection is attempted using ' or 1 = 1' or other variants, it would cause an exception in the code that we can choose to handle or discard. Further input validation and parameter binding can also be done to prepared SQL statements in order to protect against SQL injection.

Authentication and usage of session token

Hashed tokens ensure that authentication is one-way and almost impossible to replicate especially by reverse engineering or brute force

Restricting API calls to only POST

Restricting API calls to POST methods instead of GET masks the URL and reduces the understanding a potential attacker might get of our architecture just from examining our URLs

HTTPS/SSL between client and API Gateway

In order to prevent man-in-the-middle / eavesdropping attacks, we have used a domain name (reused from ESM class last semester), and secured it with an SSL certificate issued from AWS. This provides an encrypted tunnel for information flow between clients and the API Gateway.

Access Control

Network traffic into our architecture is limited using AWS built in Virtual Private Cloud (VPC) **Access Control Lists** and individual instance **Security Groups**. 2 VPCs are used. The first VPC hosts our frontend HR Web Application and acts as a DMZ and the second VPC acts as our internal network.

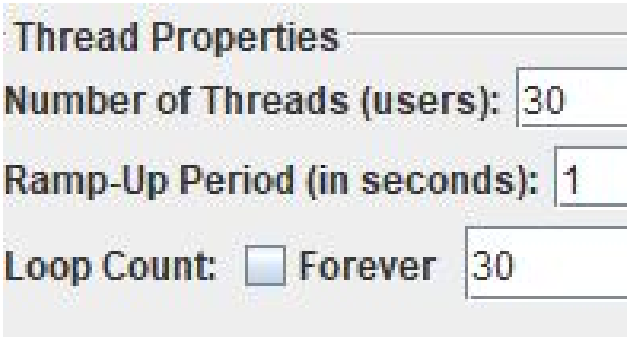
For external-facing load balancers (API Gateway and Web Application), traffic from all IPv4 addresses are allowed, but only through port 443 (HTTPS). For internal API servers, only inbound traffic allowed is on port 80 (HTTP)

Device	Direction	Type (If applicable)	Protocol	Port Range	Source (Inbound) / Destination (Outbound)
Load Balancer (Web Server)	Inbound	HTTPS	TCP	443	0.0.0.0/0
	Outbound	-	TCP	> 1024	0.0.0.0/0
Web Server	Inbound	HTTP	TCP	80	Load Balancer (Web Server)
	Outbound	-	TCP	> 1024	0.0.0.0/0
Load Balancer (API Gateway)	Inbound	HTTPS	TCP	443	0.0.0.0/0
	Outbound	-	TCP	> 1024	0.0.0.0/0
API Gateway	Inbound	HTTP	TCP	80	Load Balancer (API Gateway)
	Outbound	HTTP	TCP	80	0.0.0.0/0
Main Service	Inbound	HTTP	TCP	80	API Gateway (Through NAT Gateway)
	Outbound	-	TCP	> 1024	API Gateway
	Outbound	MySQL	TCP	> 1024	MySQL RDS Instance (Main)
Employee Microservice	Inbound	HTTP	TCP	80	API Gateway (Through NAT Gateway)
	Outbound	-	TCP	>1024	API Gateway
	Outbound	MySQL	TCP	>1024	MySQL RDS Instance (Employee)

Performance View

No.	Description	Justification	Performance Testing
1	Employee info caching on the API Gateway for improved performance	When a user logs into the HR system, naturally they would require employee data. Therefore, instead of always pulling employee data from the database through API Gateway to Employee Microservice, the API Gateway keeps a cache of all employee info in memory and serves this info unless a change was made to the employee database	When doing load testing in **JMeter**, the employee endpoint with caching done consistently exhibits about roughly 30% better response time as compared to other endpoints that return a similar amount of data

JMeter Performance Testing



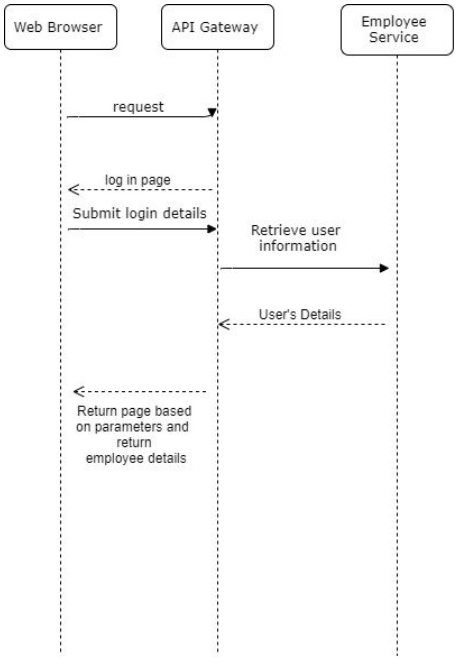
Label	# Samples	Average	Min	Max	Std. Dev.	Error %
get all employees	1800	70	15	2857	177.54	0.00%
get inbox	1800	103	23	2352	99.63	0.00%
get leave record	1800	104	25	2121	78.84	0.00%
get payroll	1800	98	24	1215	46.19	0.00%
TOTAL	7200	94	15	2857	112.46	0.00%

Performance benchmarking was done using JMeter, with 30 concurrent users at once with 30 requests each, for a total of 900 requests for each run. This was done over 2 runs in order to ensure that the first result was not a fluke

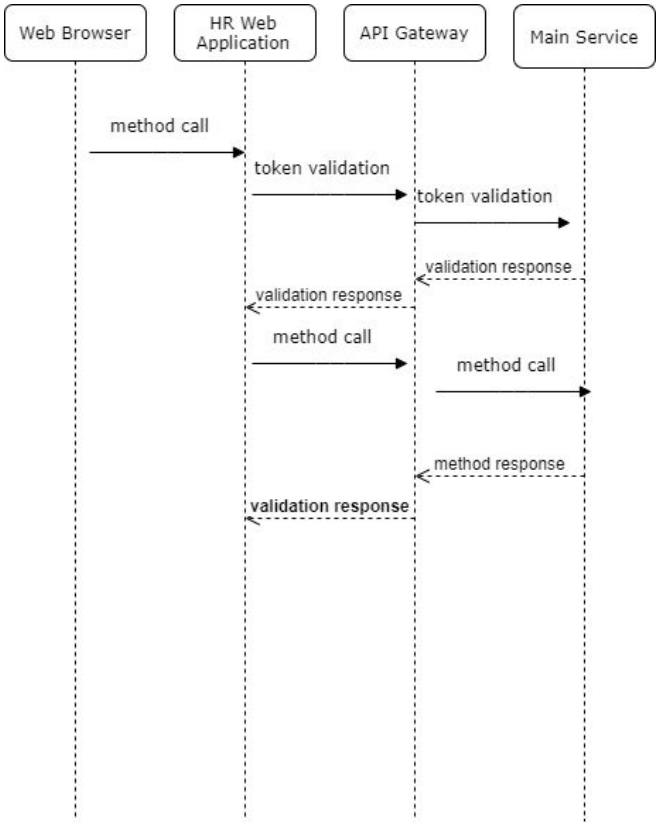
Appendix

Sequence Diagram for use cases

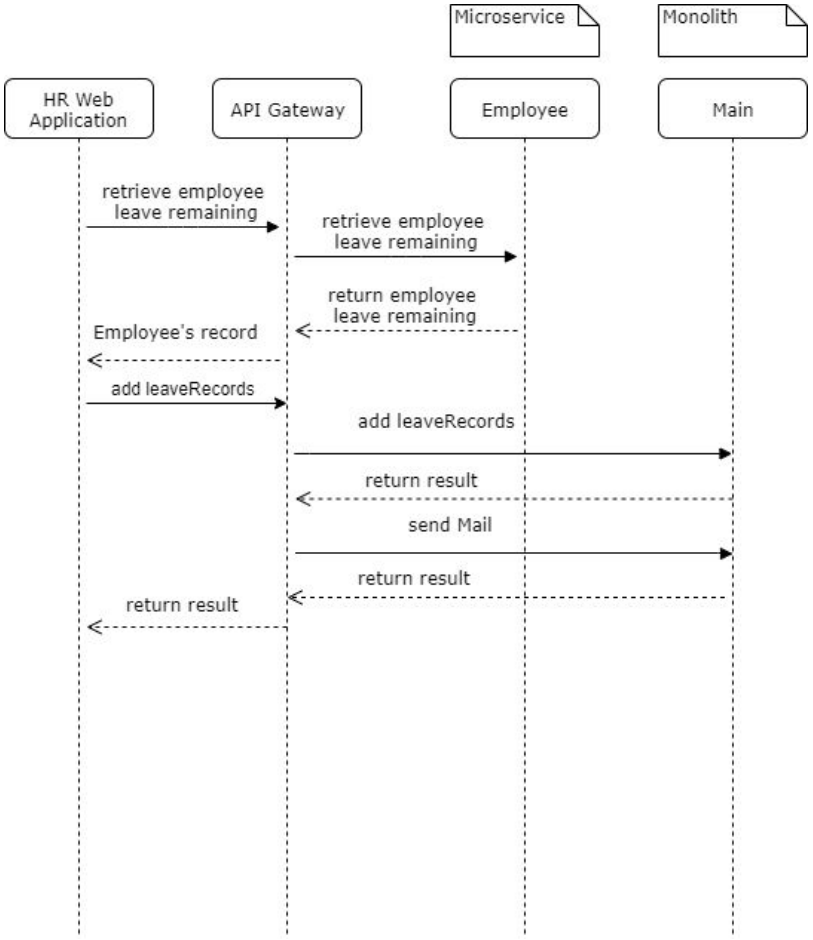
UC001 - Users login



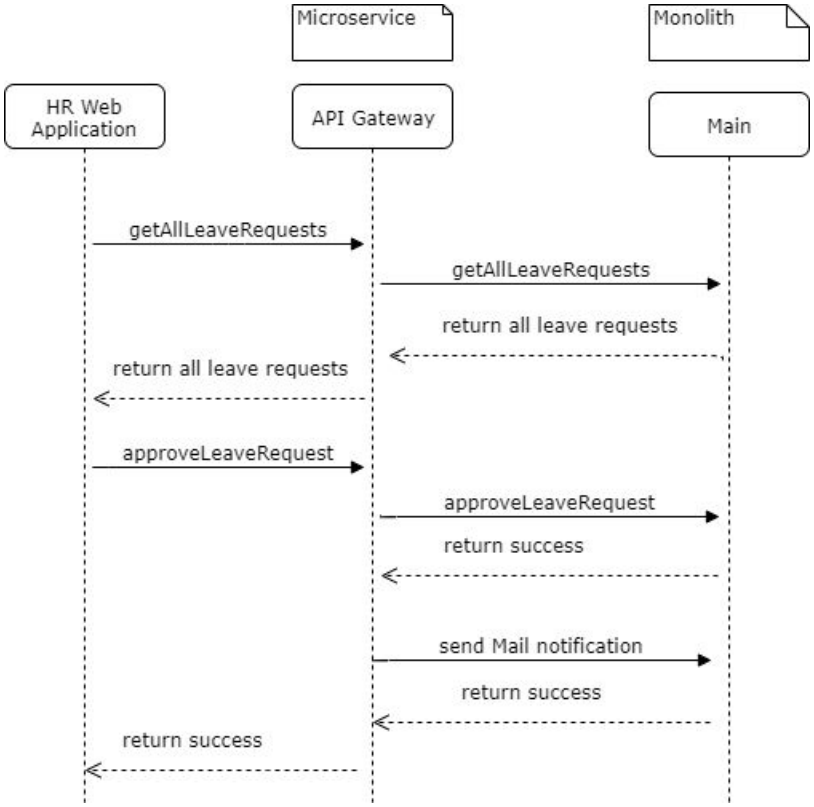
Token Validation after successful login, to be invoked before any method call.



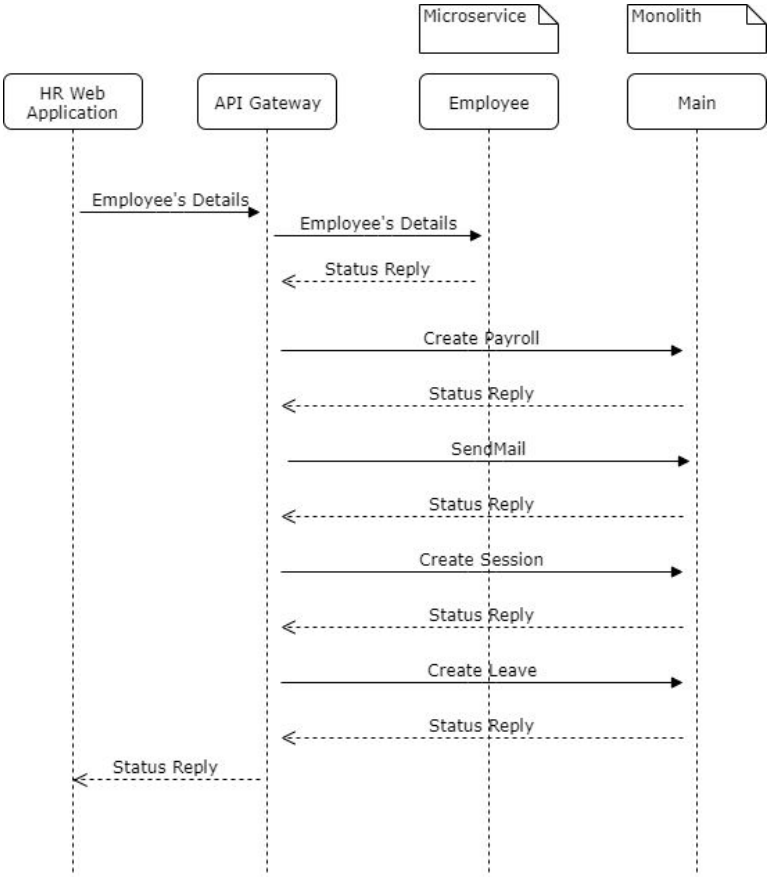
UC002 - Employee applying for leave



UC003 - Approval of leave by manager



UC004 - Create and assign new employees to their respective departments



UC005 - View Employee's performance and update accordingly to their manager's feedback