GitHub Team name :project-today-is-my-bookout-day

No. Name Email ID

Sim Cher Boon (cherboonsim.2017)

Goh Yu Xin (yuxin.goh.2017)

Ambrose Tan (ambrose.tan.2017)

Ng Qing Hui (qinghui.ng.2017)

Shaun John Cheetham (sjcheetham.2016)

1. Background and Business Needs

1.1. Background

369 Airline, herein described as “the airline”, requires a robust enterprise system to handle their day to day operations. The airline has requested a series of use cases to fulfil reservations, baggage, and claims. The airline has also highlighted that the system should be reliable. The following are the needs listed by the airline.

1.2. Business Needs

1. A passenger is able to make reservations for a flight.
2. Airline reservations can be cancelled by the passenger.
3. A passenger is able to perform self check-in for baggage.
4. A passenger is able to search for flights.
5. A passenger is able to report a lost or damaged baggage to the airline.

2. Stakeholders

Stakeholders	**Description**
Airline Product Manager	Responsible for leading the cross-functional team that is responsible for improving it. Sets the strategy, roadmap, and feature definition for a product or product line.
Solutions Architect	Responsible for designing, describing, and managing the creation of the solution to solve the business problems
Solutions Engineer	Responsible for implementing the designed solution and ensuring its technical usability
Passenger	One of the end users who will be interacting with the system as a customer
Customer Service Officer	One of the end users who will be interacting with the system from the perspective of a support officer

3. Key Use Cases

Title: Check-In Baggage

Use Case ID: AR001

Description: Passenger will update the system on the description of their baggage for a particular flight. This is to improve the tracking of the baggage and what should be compensated in the scenario where the baggage becomes lost/damaged.

Actor: Passenger

Main Flow of Events:

1. Passenger will access the CheckInBaggage Frontend Page to check-in their baggage.
2. CheckInBaggage Frontend Page invokes the CheckInBaggage System Service to retrieve information from Baggage database.
3. The CheckInBaggage System Service will invoke the Flight System Service to update information in the Flight database.
4. The CheckInBaggage System Service invokes the Baggage System Service to update information in the Baggage database.
5. Upon completion, the CheckInBaggage Frontend Page will display a successful check in page to the passenger

Alternative Flow of Events:

Check-in not successful

4a. An error message will be sent back to CheckInBaggage Frontend Page and the relevant error message will be displayed to the user.

Pre-conditions: Passenger must have an existing Flight reservation and have an existing account with system.

Post-conditions: Baggage added and status updated.

Use Case Title: Reserve Flight
Use Case ID: AR002
Description: Passenger will search a list of available flights before picking the desired flight. Passenger then reserves the flight. This is a core feature as it provides convenience to the passengers.
Actors: Passenger
Main Flow of Events:

1. Passenger searches for flight based on date, pax, origin, and destination using the Flight Booking Frontend Page.
2. Flight Booking Frontend Page will retrieve details of available flight information based on the search parameters using the Flight Booking System Service.
3. Passenger will reserve the flight by providing additional information and choice of flight to the Flight Booking Frontend Page.
4. Flight Booking Frontend Page will reserve the flight with the given passenger's information using the Flight Booking System Service which invokes the Flight Service. Flight Service will add a reservation to the Flight database and return a success outcome.
5. Flight Booking Frontend Page will display success upon completion of the request made to the Flight Booking System Service.

Alternative Flow of Events:

No flight information found based on the user's search terms

2a. Flight Booking Frontend Page will display that no flight details are found.

Fail to reserve flight

5a. Flight Booking Frontend Page will display a failure to reserve flight.

Pre-conditions: Passenger has an account with the system.
Post-conditions: Flight reservation added and flight availability updated.

Use Case Title: Cancel Flight
Use Case ID: AR003
Description: Passenger will cancel the flight reservation. Instead of going to the airport to cancel the flight or making a call to the service center, passengers can proceed online instead to cancel their flights.
Actors: Passenger
Main Flow of Events:

1. Passenger views the list of existing reservation using the Flight Booking Frontend Page.
2. The Flight Booking Frontend Page invokes the Flight Booking System Service to retrieve reservations and flights made by the passenger through the Flight System Service.
3. Flight System Service retrieves the reservations from the Flight database and returns them to the Flight Booking System Service.
4. Flight Booking System Service returns the list of reservations to the Flight Booking Frontend Page.
5. Flight Booking Frontend Page will displays all the reservations.
6. Passenger will select the reservation that he wants to cancel via the Flight Booking Frontend Page.
7. Flight Booking Frontend Page will call the Flight Booking System Service to cancel the indicated reservation.

8. Flight Booking Frontend Page will display success upon completion of Flight Booking System Service.

Alternative Flow of Events:

No flight reservation made by the user

4a. Flight Booking Frontend Page displays no flight reservations.

Failure to cancel flight reservation made by the user

7a. Flight Booking Frontend Page will display a failure to cancel the flight reservation.

Pre-conditions: Passenger has an account and existing Flight reservation.

Post-conditions: Flight reservation status and flight availability updated.

Use Case Title: Search Flight

Use Case ID: AR004

Description: Passenger will search the database for the best flight to travel to the chosen destination. This is an important feature as it helps to improve the passenger's user experience greatly.

Actors: Passenger

Main Flow of Events:

1. Passenger searches for a flight using the advanced filter option with the parameters (origin, destination).
2. Flight Frontend Page will call the flight service to retrieve the relevant results from the Flight database.
3. The Flight Frontend Page will then display flight results.

Alternative Flow of Events:

No available flight found

2a. Flight Booking Frontend Page will display a blank page.

Pre-conditions: Nil

Post-conditions: Flight results displayed to the user.

Use Case Title: Report Lost/Damaged Baggage

Use Case ID: AR005

Description: Passenger files a Lost/Damaged Baggage report to the airline system to notify the airline about the claim. This helps to improve customer experience as they can report the claim online without the need to file the claim physically.

Actors: Passenger

Main Flow of Events:

1. The passenger reports the lost/damaged baggage to Report Baggage Frontend Page.
2. Report Baggage Frontend Page will call the Report System Service and report the lost/damaged baggage asynchronously to the Issue System Service to add a request of the lost/damaged baggage.
3. Report Baggage Frontend Page will display that the report has been filed to the passenger.

Alternative Flow of Events: None

Pre-conditions: Passenger has an account and existing Flight reservation

Post-conditions: Issue report will be created with the Lost/Damaged Baggage

4. Architectural Decisions

Architectural Decision - Microservices	
ID	AD1
Issue	A monolithic application may result in poor fault tolerance should a part of the application crash. This is because all the services in a given instance will be unavailable as well. Furthermore, modularity will be an issue as our use cases depend on a few base APIs that only need to communicate with the other base APIs.
Architectural Decision	Split our application into RESTful microservices.
Assumptions	Assume that monolithic services, when they crash, results in the entire application being unavailable.
Alternatives	Point-to-point, centralised messaging, centralised process control
Justification	Requires more development time as compared to microservices, messaging API to be implemented on each and every given service and a central messaging server needs to be maintained which adds another potential single point of failure.

Architectural Decision - Continuous Integration/Delivery	
ID	AD2
Issue	Manual pushing and validating of code is time consuming, and negatively impacts the maintainability and portability of the application.
Architectural Decision	Implement TravisCI to automatically test, build, and deploy our application.
Assumptions	Assume that TravisCI will be available as a host for continuous deployment/integration.
Alternatives	Since TravisCI is an externally hosted CI/CD provider, we can opt for a self-hosted CI/CD such as Jenkins instead to increase the amount of control we have over CI/CD, and reduce security/reliability issues from relying on a third-party SaaS provider. We can also manually test and deploy the code ourselves.
Justification	TravisCI is a free and easily configurable system that requires no additional setup on our end, aside from providing it with a YML script file to dictate its behaviour. Given the budget constraints of the project, it would be preferable to rely on an existing free system in order to perform CI/CD.

Architectural Decision - Usage of API Gateway	
ID	AD3
Issue	As we have broken our services into microservices, they are now handled by different servers with different IP addresses and ports. Therefore, we also need a single point of authentication to check if the client consuming the service is authorised.
Architectural Decision	To use an API gateway to protect the endpoints behind the API gateway, and to also implement authentication mechanisms. This was achieved through the use of Lambda functions to control the validation of a user's credentials as well as to issue and verify a user's session token. This token is stored on a REDIS cluster and validated for each protected API call.
Assumptions	We will only need to expose the client services through the API Gateway and not through other means.
Alternatives	Enterprise Service Bus.
Justification	API Gateways are usually outward facing, while Enterprise Service Bus are usually inwards facing. Since our need is to provide authentication and routing functionality for the clients contacting to our services, we have thus chosen the API Gateway.

Architectural Decision - Defence in Depth pattern	
ID	AD4
Issue	There are many ways to access the microservices, security for each microservice may differ.
Architectural Decision	Base services and integration services require a different level of security due to the different data they receive as inputs. The integration service will need to focus more on the user inputs, ensuring that the values exist within our system before calling the base services. Whereas the Base services will focus on input by the integration services ensuring that SQL vulnerabilities cannot be exploited. Additionally, we also implement the use of AWS Web Application Firewall (WAF) to mitigate attacks like DDOS and cross-site scripting. This combination of defensive coding and WAF results in multiple layers of protection before reaching the Base services.
Assumptions	Each layer of Architecture will have a different security focus.
Alternatives	Single point of defense

Justification	A single point of defense would create a situation where should it fail, the internal systems would be unprotected. This means that there is a need to ensure that this single point is operational constantly, this would make maintaining or improving that point difficult. Defence in Depth would offer more security when the single point of defense fails to detect intrusions.

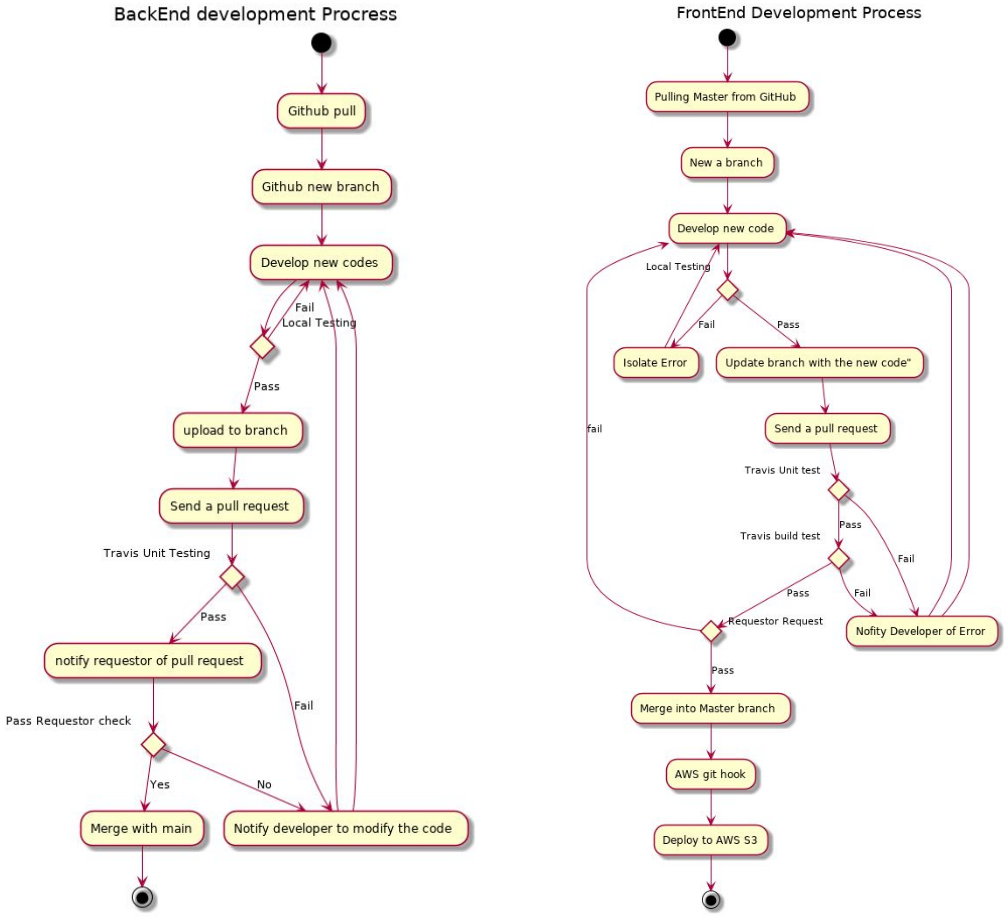
Architectural Decision - Database storage integration pattern	
ID	AD5
Issue	We need to store data pertaining to flights, reservations, and baggage items in a manner to preserve high availability and modularity. Furthermore, it should store data in a reliable fashion, while the microservices should be able to retrieve data consistently from the storage medium.
Architectural Decision	Implement a MySQL database to share data between the base microservices and their linked databases.
Assumptions	Only the base microservices will need to retrieve data from the databases.
Alternatives	File transfer, peer-to-peer sharing
Justification	We chose database storage for its portability where it can be adapted easily and scaled to meet the needs as required. Furthermore, it maintains reusability as compared to file transfer and peer-to-peer sharing which may not provide strong consistency of data. Database storage also allows for robust fault tolerance in the form of database replication as compared to file transfer and peer-to-peer sharing.

Architectural Decision - Usage of shared classes across all microservices	
ID	AD6
Issue	Due to the splitting up of the logic into microservices, some similar code is likely to be shared across multiple instances e.g. database retrieval logic. Therefore, we need to have good reusability and modularity of codes across microservices.
Architectural Decision	Copy interfaces and utility classes across all microservices, applying design patterns e.g. Factory, Builder, as well as Spring Beans to deal with the creation of each object.
Assumptions	Assume that when copying the code over, there is no further modification to the code required.
Alternatives	Create a common module that can be packaged and included in each microservice.
Justification	Requires more time and expertise to do so. In the long run, this would improve the modularity, testability, and reusability of code but would take more time than simply copying over the files to each microservice directory.

Architectural Decision - Gradle	
ID	AD7
Issue	We need to ensure installability and adaptability in both local environment, for testing, as well as within the remote environment for staging/production. Furthermore, a build tool should also be introduced for testability such that a single command can execute the requisite testing for the microservices.
Architectural Decision	Employ the use of Gradle as a build automation system to deal with dependency installation and testing.
Assumptions	N/A
Alternatives	Maven, Apache Ant, manual library configuration
Justification	Gradle, as a tool is shown to be faster than Maven and Apache Ant, improving the speed at which our team can debug and test our microservices. Furthermore, compared to Apache Ant and Maven, Gradle runs on domain specific language instead of XML, which helps analyzability and modifiability of build configurations via the reduced verbosity compared to XML.

5. Development View

5.1. Activity Diagram



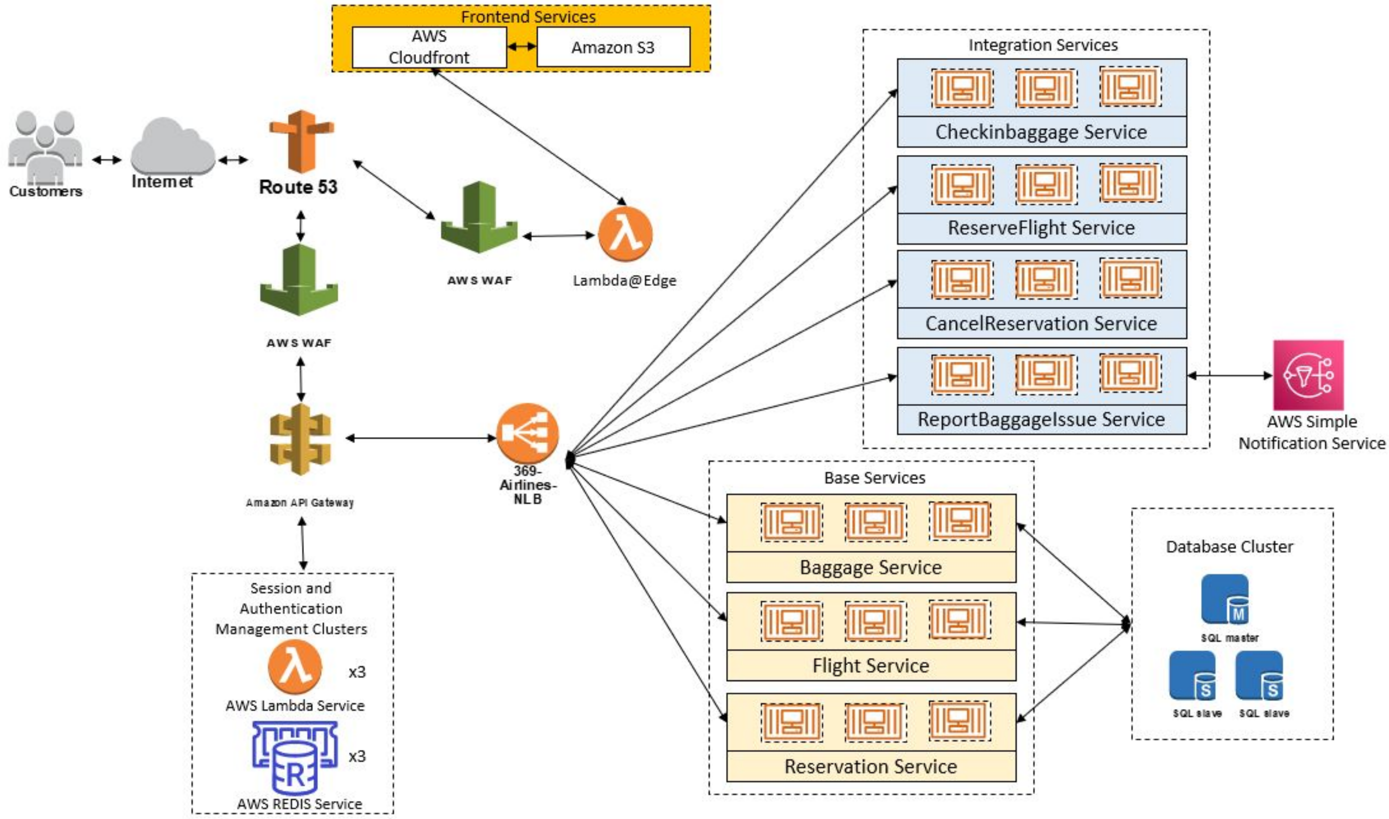
5.2. Deployment Strategy

Our team uses a combination of Travis CI, Gradle, as well as NPM in order to manage deployments. Versioning is done via Git (could be improved with the usage of versioning tags). Continuous integration was achieved for the backend via running of tests for base API services using Travis CI, ensuring that endpoints do not break during pull requests.

For the backend, we opted to Dockerize and set up each microservice manually due to requiring configurations to match the endpoints of the AWS endpoints instead of local endpoints.

For our frontend, continuous delivery was set up, where a VueJS production build would be automatically generated by Travis CI and deployed automatically to Amazon S3 to be hosted statically.

6. Solution View



Our Enterprise Solution is deployed on Amazon Web Service (AWS). The adoption of AWS removes the need to maintain the infrastructure, instead we can focus on the deployment patterns and integration patterns.

We adopted a microservice architecture for the deployment of our services. This will ensure that the services are decoupled, making it easier to identify and fix faulty services. Because our services are modular, changes in a service has minimal impact on the other services and testing can be done individually.

Our services are also designed to be reusable, the same configurations and deployment package for each service can be reused to deploy a subsequent instance. This is done through the use of Container Services (Shown in diagram as dotted boxes with the orange service).

Refer to the *detailed_solution_view.pdf* for a more detailed view.

6.1. Integration Endpoints

Source System	Destination System	Protocol	Format	Communication Mode
Cancelation Service	Flight Service	HTTPS	JSON	Synchronous
Cancelation Service	Baggage Service	HTTPS	JSON	Synchronous
Cancelation Service	Reservation Service	HTTPS	JSON	Synchronous
CheckInBaggage Service	Reservation Service	HTTPS	JSON	Synchronous
CheckInBaggage Service	Baggage Service	HTTPS	JSON	Synchronous
ReserveFlight Service	Reservation Service	HTTPS	JSON	Synchronous
Flight Search Service	Flight Service	HTTPS	JSON	Synchronous
Flight Service	MySQL	JDBC	SQL	Synchronous
Baggage Service	MySQL	JDBC	SQL	Synchronous
Reservation Service	MySQL	JDBC	SQL	Synchronous
Report Lost Baggage	Baggage Service	HTTPS	JSON	Synchronous
Report Lost Baggage	AWS SNS	HTTPS	JSON	Synchronous
AWS SNS	AWS Email Subscriber	SNS Messaging Protocol	Message	Asynchronous
AWS Email Subscriber	Email	SMTP	IMF - RFC5322	Synchronous

6.2. Hardware/Software/Services Required

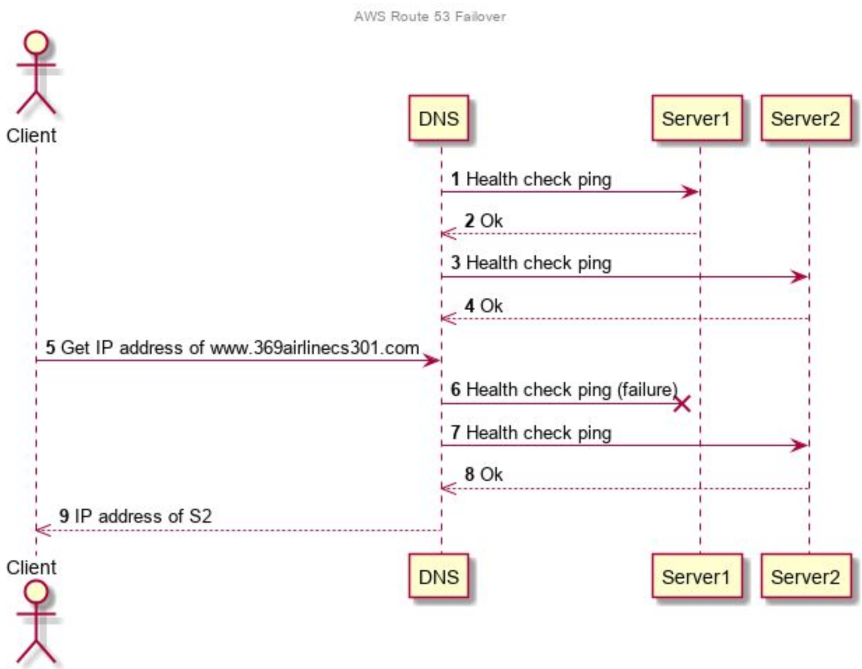
No	Item	Quantity	Buy/Lease	License	Cost
1	S3 Buckets	2	Per GB Usage	Proprietary	$0.02/GB (link)
2	Network Load Balancer (NLB)	1	Hourly + Per GB Processed	Proprietary	$0.025/hr (link) + $0.006 per LCU-hour
3	Application Programming Interface (API) Gateway	1	Per million requests	Proprietary	$4.25/million requests (link)
4	Relational Database Service	3	Monthly	Proprietary	(link)
5	Web Application Firewall	1	Monthly	Proprietary	$5/month+ (link)
6	CloudFront	1	Per GB Usage	Proprietary	Up to $0.114/GB (link)
7	Elastic Cloud Compute (EC2) On Demand Instances (t3.nano)	2	Hourly	Proprietary	$0.0052/hr (link)
8	Elastic Container Service (Fargate)	21	Hourly	Proprietary	$0.04048/hr (link)
9	AWS Redis (cache.t2.micro)	3	Hourly	Proprietary	$0.017/hr (link)
10	AWS Route 53 (hosted zones)	2	Traffic bandwidth + Hosted Zone	Proprietary	$40/hosted zone + incoming traffic cost (link)
11	Simple Notification Service (SNS)	1	Per 100k emails	Proprietary	$2.00/100k emails (link)
12	Travis CI	N/A	Monthly	Open source - MIT License	N/A

13	Apache Tomcat	N/A	N/A	Open source - Apache License 2.0	N/A
14	Gradle	N/A	N/A	Open source - Apache License 2.0	N/A
15	Spring	N/A	N/A	Open source - Apache License 2.0	N/A

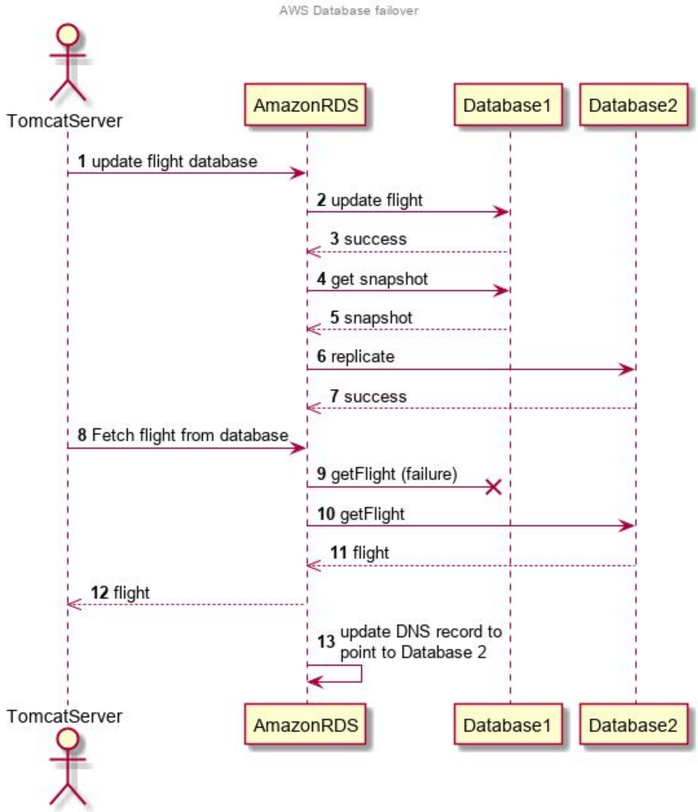
7. Availability View

Node	Redundancy	Clustering			Replication(if applicable)			
		Node Config	**Failure Detection**	**Fail-over**	**Repl. type**	**Session State Storage**	**Db Repl. Config**	**Repl. mode**
AWS Route 53	Horizontal Scaling	Active-Active	Ping	DNS	N/A	N/A	N/A	N/A
AWS Database	Horizontal scaling	Active-Passive	Events	DNS	DB	Database	Master-slave	Asynchronous
AWS Redis	Horizontal scaling	Active-Passive	Events	DNS	DB	Database	Master-slave	Asynchronous
AWS Elastic Container Service	Horizontal and Vertical scaling	Active-Active	Ping	DNS	Session	N/A	N/A	N/A
AWS EC2 instance	Horizontal Scaling	Active-Active	Ping	DNS	N/A	N/A	N/A	N/A

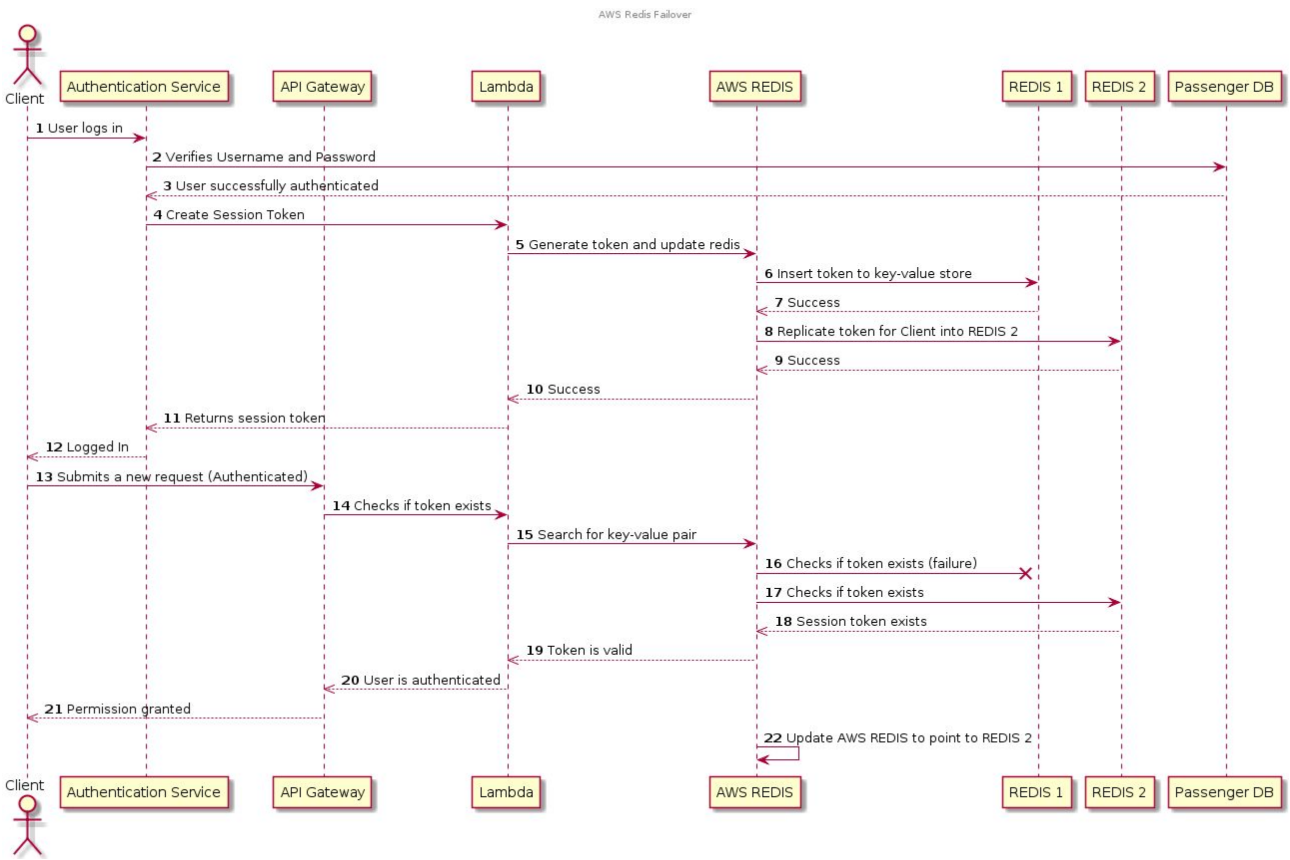
7.1 AWS Route 53 Sequence Diagram for Failover



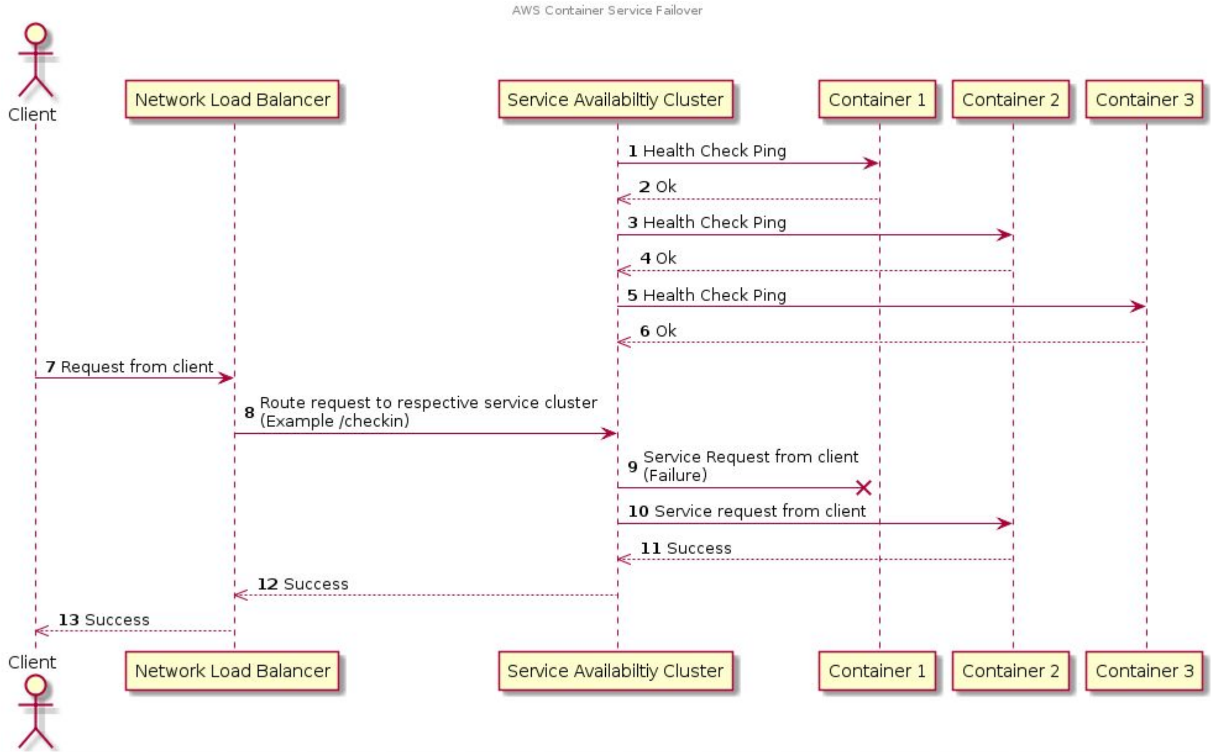
7.2 AWS Database Sequence Diagram for Failover



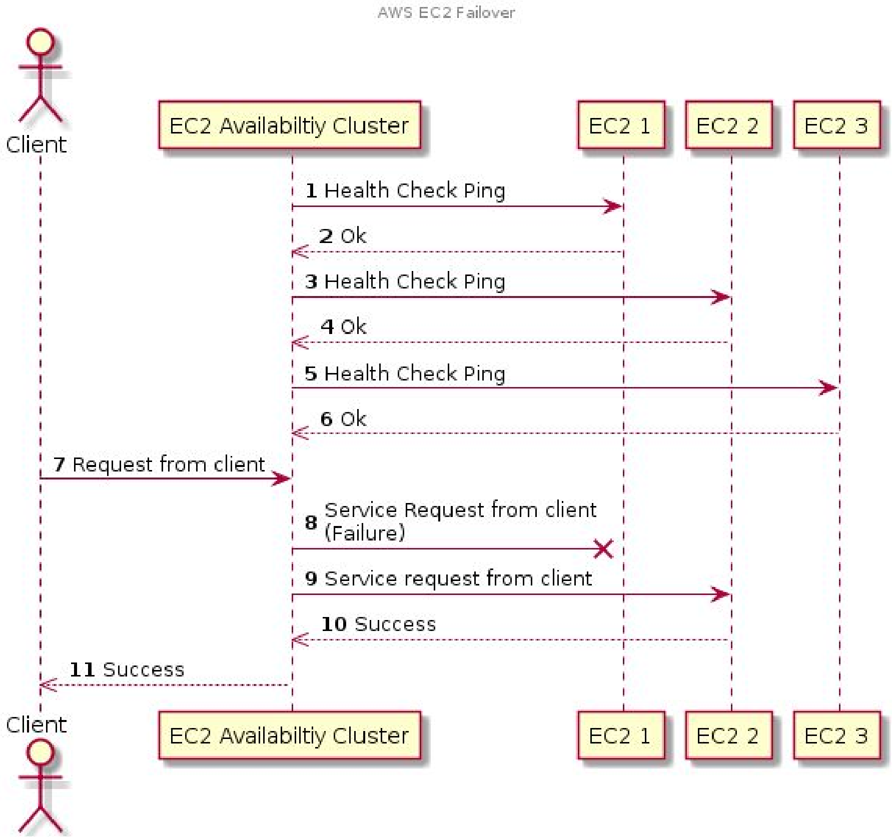
7.3 AWS REDIS Sequence Diagram for Failover



7.4 AWS Container Service Sequence Diagram for Failover



7.5 AWS EC2 Sequence Diagram for Failover



8. Security View

No	Asset/Asset Group	Potential Threat /Vulnerability pair	Possible Mitigation Controls
1	Databases	SQL Injection/Unsafe creation of SQL statements. Affects the Confidentiality of data stored in database.	Prepared Statements (implemented via Spring), escaped characters (not implemented)
2	Client	Cross-site scripting (XSS) / XSS Injection. Affects the Integrity of the client's browser document object model through injection of scripts.	Set Content-Security-Policy headers (Implemented via Lambda@Edge) & Through WAF rules that inspects incoming traffic to filter out potential XSS attacks
3	Servers	Denial-of-Service (DoS) attack / Distributed-Denial-of-Service (DDoS) attack. Affects the availability of the services hosted on the server.	Rate limiting (implemented via AWS Web Application Firewall), upstream filtering (not implemented)
4	Client	Cross-site request forgery (CSRF). Affects the integrity of the request made.	Add CSRF token requests to all actions (not implemented), challenge-response tests (not implemented)
5.	Servers	Server-side request forgery (SSRF). Affects the confidentiality of the system being exploited on.	Sanitize user requests to ensure that only permitted URL schemes are allowed. (not implemented)
6.	Client	Man-in-the-middle attack / Session token hijacking. Affects the confidentiality and integrity of the requests made by the client.	Serve all requests over HTTPS to encrypt traffic before it reaches the client/server (implemented via CloudFront).
7.	Client	SSL Stripping attack / Accessing a secure site via HTTP. Affects the confidentiality and integrity of the requests made by the client.	Add HSTS header and redirect all traffic from HTTP to HTTPS (implemented via Lambda@Edge), certificate pinning (not implemented)

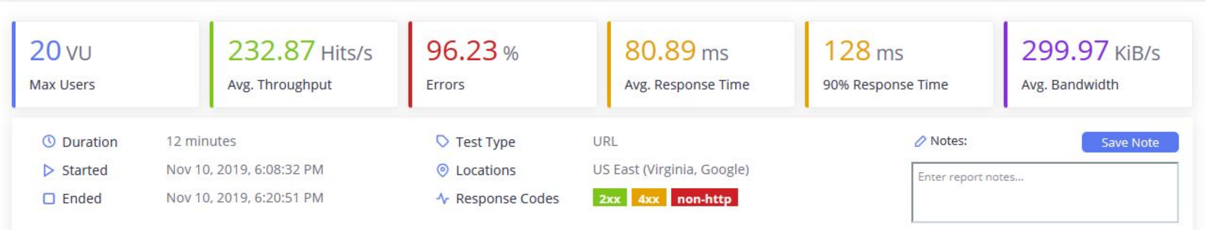
8.	Servers	Unauthorised scanning and probing.	The WAF parses the application's access logs to detect suspicious behaviour from individual IP addresses. If a suspicious behaviour is detected, that IP address will be blocked for a period of time.
9.	Servers	IP addresses with poor reputations access the services for dubious intentions.	Queries are made on an hourly basis to a third-party reputation list that returns a range of IP addresses that have been determined to be suspicious. These IP addresses are then blocked.
10.	Servers	Bad bots and content scrapers	The WAF detects bad bots and scrapers and directs that traffic to an API Gateway endpoint. This endpoint is pointed at a Lambda function that will then add the source IP to the list of blacklisted IP addresses.

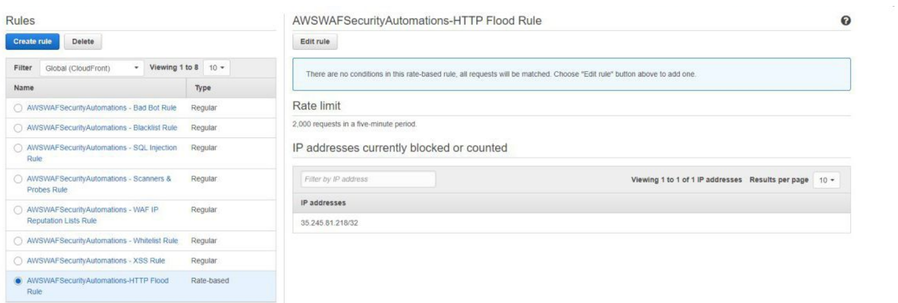
*Refer to Solution View for security components, and the *detailed_solution_view.pdf* for a more detailed view.

9. Performance View

No	**Description of the Strategy**	**Justification**	**Performance Testing (Optional)**
1	Static website caching	Since the pages used in our frontend are static, the use of cache will shorten the time to fetch the contents. The cache is stored in Cloudfront for future fetching. Since the DNS is pointed to the cloudfront, the client will always be served either a new content, or a cache of a previously served content.	No Cache: 10ms Cache: 3ms
2	Load balancing	In event of high load, we are able to effectively distribute requests via the network load balancer (NLB). The requests will then reach each service cluster, and the service cluster will choose at round robin which instance to pass on the request to.	NIL
3	Stress testing the website	To simulate high traffic on the website to ensure that website will carry on as per normal without any of the services failing.	*See Below

Stress testing findings





The use of a stress testing tool resulted in a triggering of an IP Ban on the stress testing tool due to the WAF rules implemented. Throughout the testing, the homepage was still accessible, and no degradation of services was observed.

*Refer to Solution View for performance view, and the *detailed_solution_view.pdf* for a more detailed view.