

CS301: IT Solution Architecture

AY 2019-20 Semester 1

Faculty: OUH Eng Lieh

**Final Project Report**

**Team Drop All Databases;--**

Chua Pei Si

Kidmann Daniel Goh

Lim Huan Sen

Lee Wei Yuan

**Content Page**

[**Background & Business Needs** 1](#_Toc24336938)

[**Stakeholders** 1](#_Toc24336939)

[**Key Use Cases** 1](#_Toc24336940)

[**Development View** 9](#_Toc24336941)

[**Solution View** 11](#_Toc24336942)

[Deployment Diagram 11](#_Toc24336943)

[Overall View 12](#_Toc24336944)

[**Integration Endpoints** 15](#_Toc24336945)

[**Software/Services Required** 16](#_Toc24336946)

[**Availability View** 16](#_Toc24336947)

[**Security View** 17](#_Toc24336948)

[**Performance View** 18](#_Toc24336949)

[**Appendix** 20](#_Toc24336950)

# 

# **Background & Business Needs**

Trading App is a secure online trading platform, providing users with the latest stock prices and related news, and the ease of trading financial instruments at the click of a button. Our platform uses the latest social messaging service, Telegram, to send notifications on successful purchases.

# **Stakeholders**

Internal:

1. Project Manager: Lee Wei Yuan
2. Development Team: Lee Wei Yuan, Lim Huan Sen, Chua Pei Si, Kidmann Goh
3. Quality Assurance: Lim Huan Sen
4. CTO: Professor Ouh Eng Lieh

External:

1. Users
2. Telegram (Social Messaging Service)
3. News API Provider (Latest Stock-related News)
4. Heroku MongoDB (Cloud Database Provider)
5. Heroku Postgres (Cloud Database Provider)
6. AWS (Cloud Platform Provider)

# **Key Use Cases**

|  |  |
| --- | --- |
| **Login to Exchange Web Application** | |
| **Use Case** | 1 |
| **Description** | User logs in to Trading App Client with his/her username and password |
| **Actors** | User, Trading App Client Server, Trading App Client, Apigee, Trading App Backend Serve, Trading DB |
| **Main Flow of events** | 1. User accesses Trading App Client via url and is redirected to login page 2. User enters username and password and submits 3. Trading App Client posts the username and password in a JSON payload via HTTPS to the API Gateway’s login endpoint 4. Apigee (API Gateway) proxies this request to the Trading App backend which validates the username and password. 5. If valid credentials are provided, the Trading App backend returns the user’s information as a json response. 6. Apigee receives this json response and attaches a JSON Web Token (JWT) to the header for the client to receive. This JWT must be provided by the Trading App Client in subsequent requests for the Apigee to perform authentication. |
| **Alternative Flow of events** | Trading App Backend Server down   1. 1Apigee acts as a load balancer between two ec2 instances of the Trading App backend. Should an instance fail, the other would still be alive to serve client requests. |
| **Pre-conditions** | 1. Trading App Client Server is live 2. At least one Trading App Backend Server is live |
| **Post-conditions** | 1. User is logged into the Trading App Client and able to perform the following business use cases. |

|  |  |
| --- | --- |
| **User views historical instrument prices** | |
| **ID** | 2 |
| **Description** | User views historical instrument prices on Trading App Client |
| **Actors** | User, Trading App Client, Apigee, Trading App Backend Server, Trading DB |
| **Main Flow of events** | 1. User accesses Trading App Client chart page and clicks on a financial instrument to view its prices. 2. The Trading App Client calls the endpoint on Apigee to fetch the instrument’s prices, attaching the JWT received in use case 1 (login) in the request header. 3. Apigee extracts this request header and compares the requested resource against the user’s claims in the received JWT. 4. If the user is allowed to access the requested resource, Apigee proceeds to proxy the request to the Trading App backend server. 5. Otherwise, Apigee responds with an HTTP 403 (Unauthorized) Error, denying access to the requested resource. 6. Steps 3 to 5 are performed on every request to the Trading App Backend Server. |
| **Alternative Flow of events** | Trading App Backend Server down   1. Apigee acts as a load balancer between two ec2 instances of the Trading App backend. Should an instance fail, the other would still be alive to serve client requests. |
| **Pre-conditions** | 1. User is logged in 2. Trading App Client Server is live 3. Trading App Backend Server and database is live |
| **Post-conditions** | 1. User receives a transaction success / failure message on the client. 2. User’s balance is updated in the Trading App database, reflecting the changes in her available balance and owned stocks from the purchase of the stock. |

|  |  |
| --- | --- |
| **User purchases instruments and receive telegram notification** | |
| **Use Case** | 3 |
| **Description** | User purchases an instrument and receive a telegram notification |
| **Actors** | User, Trading App Client, Trading App Backend Server, Apigee, Trading DB, Messaging Microservice, Messaging DB, CloudAMQP, Telebot Service, Redis Job Store, Telegram API |
| **Main Flow of events** | 1. The Trading App Client will send a JSON request to Apigee to the Trading App Backend Server 2. Trading App Backend server will add the trade details to the Trading DB 3. The Trading App Backend server checks if the User has enabled notifications. If he has, a notification will be sent to him 4. The Trading App Backend server sends a message containing details of the user’s transaction and the amount spent 5. The Messaging Microservice adds a job in the Redis Job Store to publish a message to the CloudAMQP queue, and responds with the status to the Trading App Backend server 6. A job (that runs in 1-minute intervals) on the Telebot microservice consumes messages from the queue 7. After consuming all the messages, it can in 1 minute, the Telebot service retrieves the user’s Telegram chat ID from the Messaging DB and then sends telegram messages to the user by invoking the Telegram API |
| **Alternative Flow of events** | Database is down   1. Trading DB hot standby kicks in and acts as failover |
| **Pre-conditions** | 1. User is logged in 2. Trading App Backend Server and Trading DB is live 3. User has enabled notifications |
| **Post-conditions** | 1. User will see a transaction success / failure page 2. Trade data is updated in the database 3. User receives a notification on Telegram on the transaction details |

|  |  |
| --- | --- |
| **User views instrument’s latest news** | |
| **Use Case** | 4 |
| **Description** | User accesses the chart page and selects an instrument to view news headlines related to it. |
| **Actors** | User, Apigee, Trading App Client, News Microservice, News DB, News Provider’s REST API (external) |
| **Main Flow of events** | 1. User selects an instrument on the chart page. 2. Trading App Client invokes the News microservice’s GraphQL endpoint to fetch news related to the user’s selected instrument 3. The News microservice receives this request and validates it against the written GraphQL schema 4. If valid, it makes a request to the news provider’s REST API, and stores the response into a MongoDB database (News DB) which will act as a failover in future requests for the same keyword if the news provider is unavailable. 5. The response from the news provider’s REST API is sent back to the Trading App Client in the format it requested for (specified in its original GraphQL request). 6. If the GraphQL request is invalid, the News microservice responds with an error message. |
| **Alternative Flow of events** | News provider REST API is down   1. News Microservice fetches and stores the news it fetches from the news provider’s API periodically. In the event of the news provider API being unresponsive, the News microservice will switch to serve news feed from its database. |
| **Pre-conditions** | 1. User is logged in to the Trading App Client 2. News Microservice is live 3. News Provider’s REST API is live |
| **Post-conditions** | User is able to view the news pertaining to her selected stock |

|  |  |
| --- | --- |
| **User enables Telegram’s notifications** | |
| **Use Case** | 5 |
| **Description** | User visits his profile page and enables notifications. |
| **Actors** | User, Apigee, Trading App Client, Trading App Backend Server, Trading DB, Telebot service, Telegram API, Messaging |
| **Main Flow of events** | 1. User clicks on enable notifications in profile page. 2. Trading App Client sends a request to the Trading App Backend server that the user wants to enable notifications 3. Trading App Backend updates the Trading DB that the user wants to enable notifications 4. Depending on whether the update was successful or not, the Trading App Backend returns a status message to the Trading App Client 5. While steps 2 to 4 are happening, the User will find the bot on Telegram, and send a registration message to it. 6. The Telegram API will send a message to the Telebot service via a webhook 7. When the Telebot service receives the request from the Telegram API, it will add a job to the Redis Job Store to notify the user that he has registered through the Telegram API |
| **Alternative Flow of events** | Heroku server crashes   1. If the server crashes, any pending jobs will still be on the Redis Job Store. When the app is restarted, the jobs will be executed again. |
| **Pre-conditions** | 1. User logged in 2. All services (actors) to be up and running |
| **Post-conditions** | 1. User opt-in for notifications is updated on Telebot microservice database. 2. User receives a notification on Telegram that he has registered for notifications |

**Key Architectural Decisions**

|  |  |
| --- | --- |
| Architectural Decision: **API-Driven Architecture** | |
| ID | 1 |
| Issue | 1. Backend & Frontend are tightly coupled 2. Lack of extensibility |
| Architectural Decision | An API-Driven Architecture allows for Split Stack Development, where the backend and frontend can be decoupled and remove any development dependencies imposed on one another as long as an interface is agreed upon. This also enables parallel development in the team. |
| Assumptions | None. |
| Alternatives | Monolithic Application |
| Justification | This provides extensibility to develop different client applications that consume the API, such as a mobile application. This is compared to a monolithic web application, where a refactoring step (to build more tightly coupled business logic on top of the existing monolith for the mobile app to call) is required to allow integration of a mobile application. |

|  |  |
| --- | --- |
| Architectural Decision: **Facade** using API Gateway (Apigee) | |
| ID | 2 |
| Issue | 1. Complexity of the Underlying System Code 2. Security (Multiple Entry Points) |
| Architectural Decision | An API gateway sits in front of our APIs and acts as a single point of entry for all our microservices. The single-entry point has several benefits:   1. It means a smaller attack surface, making the application more secure. 2. The API gateway acts as a reverse proxy and wraps the different backend APIs for the frontend to call, with the gateway handling request routing. If there are any changes in backend APIs, configuration changes only need to be made on the API gateway and the frontend will not be affected. 3. Common functionalities across all APIs can be implemented on the gateway, such as traffic monitoring and authentication. |
| Assumptions | None. |
| Alternatives | Direct client-to-backend/microservice communication |
| Justification | API Gateway minimises the number of requests to the back end (spaghetti architecture) by the client and reduces cross-cutting concerns like security and authorization. It is also more convenient to designed for the needs of different client applications (e.g. Mobile apps). |

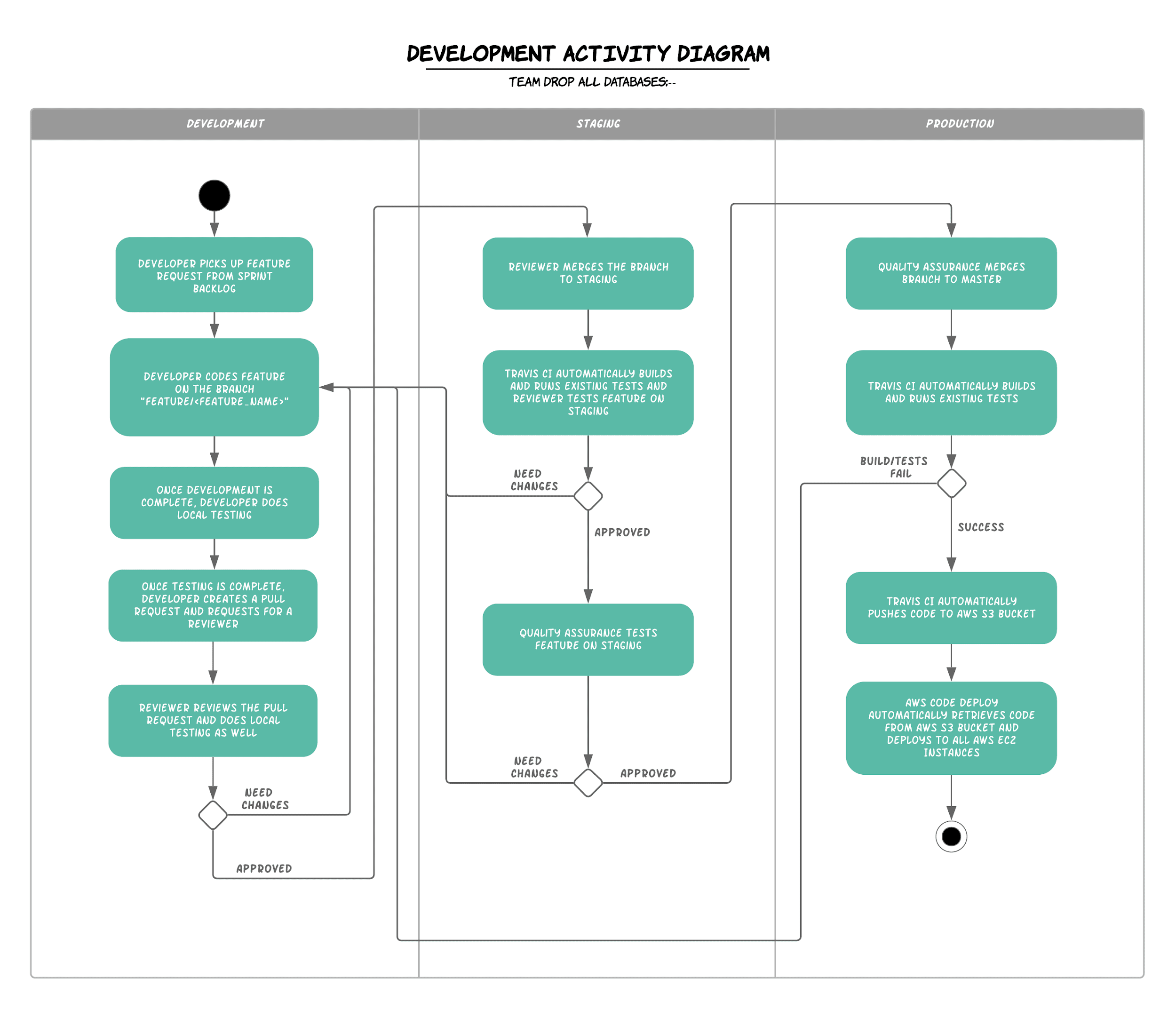
|  |  |
| --- | --- |
| Architectural Decision: **Microservice Architecture** (Messaging and News Microservices) | |
| ID | 3 |
| Issue | 1. Avoid complexity in testing and development in a Monolithic Application 2. Tightly coupled modules |
| Architectural Decision | A Microservice Architecture allows us to break a large application into loosely coupled modules that communicate through APIs. This allows separate teams to develop and test each microservice without any dependencies on another. |
| Assumptions | None. |
| Alternatives | Monolithic Application |
| Justification | Easier to scale, maintain and test with a microservice architecture |

|  |  |
| --- | --- |
| Architectural Decision: **Factory** **Method** for Logger | |
| ID | 4 |
| Issue | Decouple creation of logger objects |
| Architectural Decision | The Apache log4j2 module internally implements a factory method to create logger objects specific for each class. This method is only called upon only if our class needs it and Apache log4j2 will create a logger that logs events tied to the class. |
| Assumptions | None. |
| Alternatives | Implementation our own logger as a Singleton. |
| Justification | There is no need for the Singleton Design Pattern as our logs are not written to a single file where write can only be by one logger at a time. Instead, our instances log to a cloud-hosted management service, PaperTrail, at the same time with no clashes. |

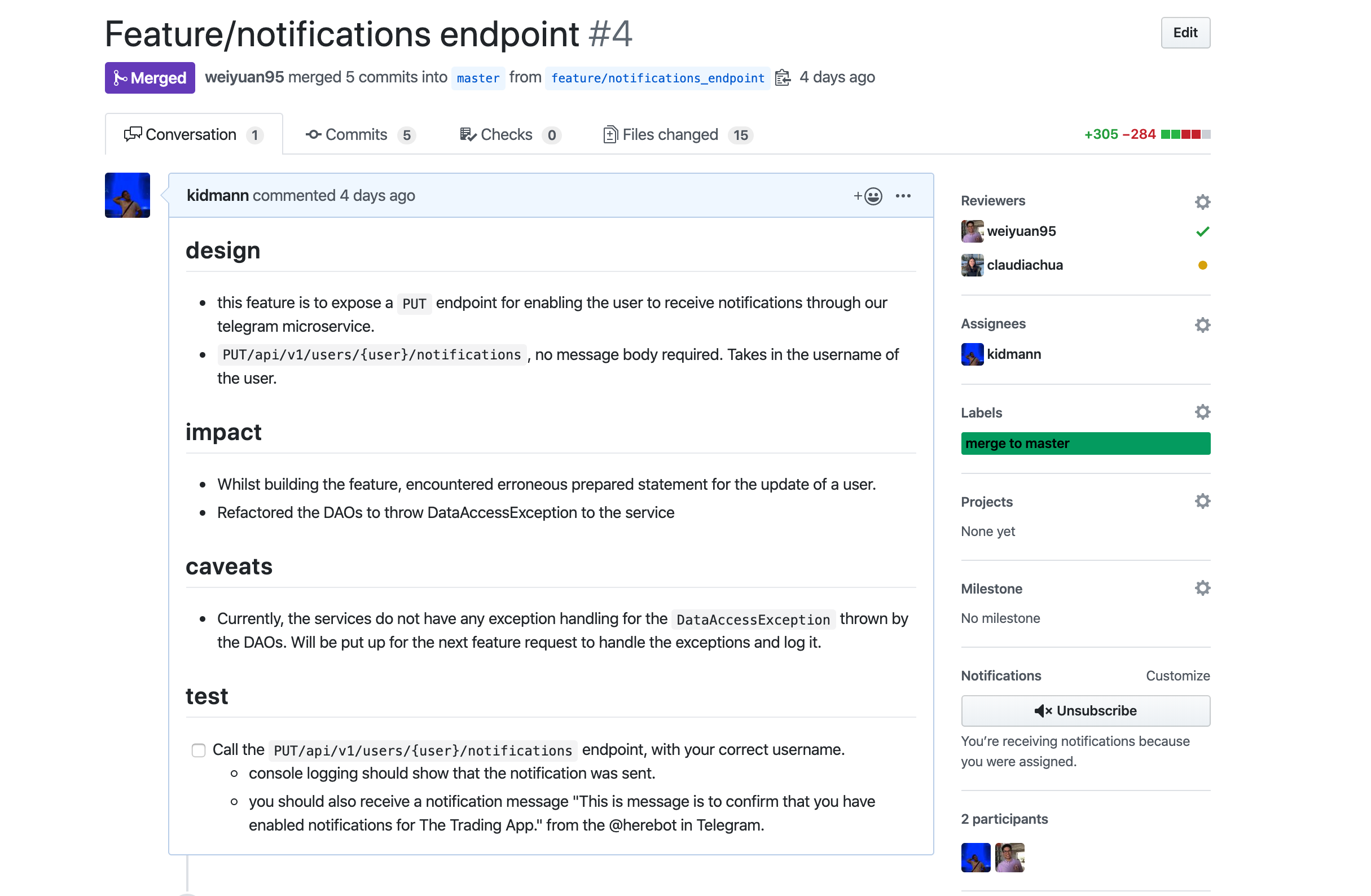
|  |  |
| --- | --- |
| Architectural Decision: GraphQL **Adapter** for REST API endpoints | |
| ID | 5 |
| Issue | A news API that we want to consume is currently served over REST. |
| Architectural Decision | Wrapping an existing REST endpoint with a GraphQL Adapter as we want the application to be able to consume GraphQL and REST endpoints. |
| Assumptions | Application will be consuming GraphQL APIs in the future. |
| Alternatives | None. |
| Justification | To simulate a situation where an existing third-party service exists in a protocol type that we do not want. Since we might have multiple services using that service, we can have a microservice wrapping that service. Our microservice will expose the endpoints that we want in the right protocol. |

# 

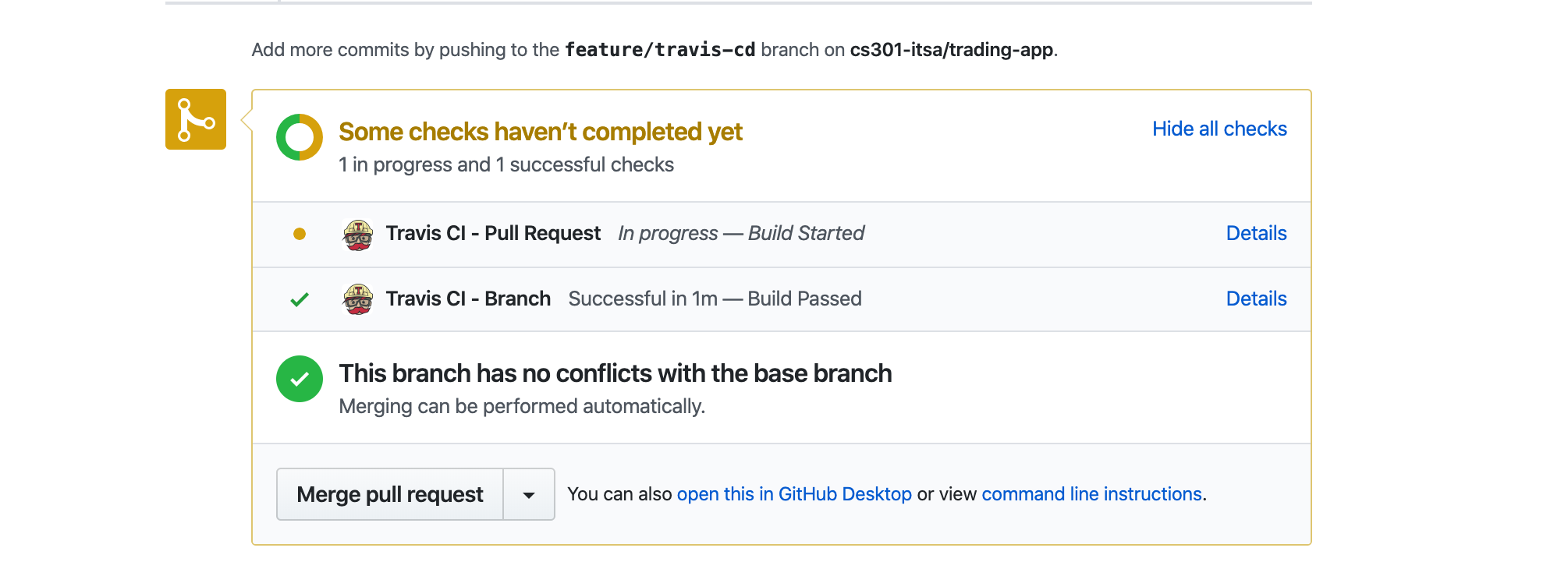
# **Development View**



**Pull Request Template**

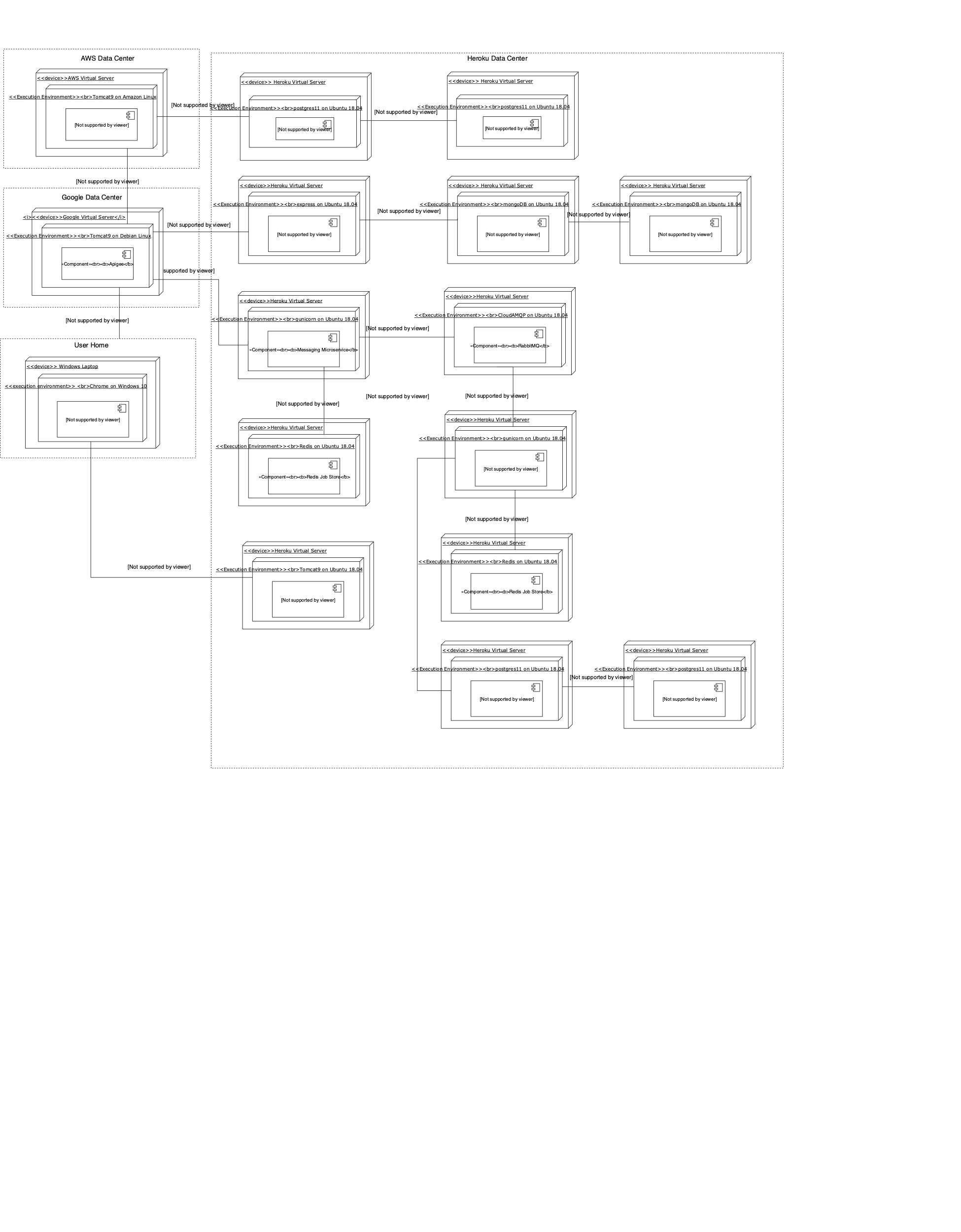


**TravisCI Integration**

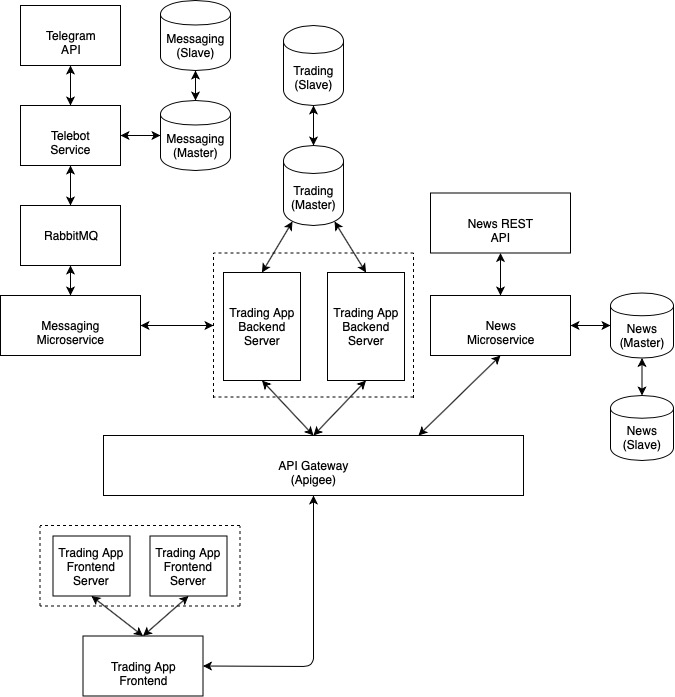
****

# **Solution View**

## Deployment Diagram



## Overall View

****

Ease of maintainability achieved with:

**Design Patterns**

**Facade Pattern (API Gateway)**

With the API gateway acting as a **single point of entry** for any frontend clients, the frontend clients will only need to know the address of the gateway because the gateway will be responsible for routing the requests to the relevant microservices. **This eases maintainability in the frontend** as any changes in the backend APIs will not affect the frontend clients. Configuration changes only need to be made on the API gateway.

However, this means that changes in the backend API will require changes on the API gateway. With this in mind, we need to ensure that updating the API gateway has to be as lightweight as possible.

**Factory Pattern (Apache Log4j2 Module)**

The Apache Log4j2 module that we have chosen for logging, internally implements a factory method for creating loggers specific for each class by passing the Class name as a parameter to the method. This allows us to abstract the instantiation of objects without exposing instantiation logic.

**Adapter Pattern (GraphQL)**

With the advent of GraphQL — a potential successor of REST — many institutions are beginning to expose GraphQL APIs. By creating a GraphQL wrapper, client applications can leverage on GraphQL’s powerful capabilities on existing REST APIs. Furthermore,

**Architectural Styles**

**API-Driven Architecture**

The business logic and operations is handled by the **Spring Boot REST API (Trading App)**. Being a REST API, it is completely stateless, and persists data into a database. This allows **horizontal scalability**, as increasing the number of nodes in the clustered environment will have little impact on the client. Application state is maintained client-side with a Single Page (Web) Application.

**Client-managed State**

The traditional approach to web applications with multiple pages (multi-page applications) is to have application state maintained on the server side with sessions. This way of maintaining state gives rise to issues in a clustered environment, where client sessions typically need to be stickied with the server they made first contact with. Although there are well-established solutions to this problem, the team decided on building the web application as a **Single Page Application (SPA)** — in which state is maintained on the client rather than the server — as the solution.

An SPA differs from the traditional MPA in that it is served as a single static file only containing browser-executable code (Javascript, HTML and CSS). This code is responsible for fetching data (in our case, from API endpoints) and dynamically rendering pages with HTML5 features to simulate the existence of multiple pages when in fact, it is simply DOM elements that are being manipulated by javascript. Performance of SPAs are typically significantly faster than MPAs given that all browser-executable code is requested only once, and data to populate the different views are fetched from lightweight REST APIs. This contrasts with traditional MPAs where the client, upon request of a page, has to wait for the server to generate the page (HTML) after fetching the relevant data from a database.

**High extensibility with decoupled clients**

By adopting an API-driven architecture, the application is **extremely extensible**, as new and existing systems can be integrated with relatively ease. For example, native applications (mobile or desktop) can be developed and use the REST APIs provided by our backend right off the bat. This contrasts against having traditional multi-page web applications (jsp, php, mvc frameworks, etc.), where considerable developer effort is required to extract the functionalities of the core application to a separate REST API which native clients can call.

**API Gateway**

By leveraging on an API gateway to proxy requests from client applications to the server, some non-business functionalities can be abstracted away from the core application’s services. Some examples include performance monitoring and authentication. This improves maintainability given that the functionalities need not be built and implemented on each service and adding another layer of complexity.

**Cloud Architecture**

Two providers were used for server resources on the cloud - **Heroku (PaaS)** and **AWS EC2 (IaaS)**.

Cloud-based logging was also implemented with **PaperTrail (SaaS).** Logs from the deployed applications on the cloud are actively sent to **PaperTrail**, which aggregates all the logs in one place. (Refer to Figure 2 in the Appendix.)

**Ease of deployment**

Both **Heroku** and **EC2** have integrations with **TravisCI**. This allows us to have continuous integration and deployment. By pushing/merging to the Master branch on Github, **TravisCI** automatically builds the application and runs the unit/integration tests. If successful, **TravisCI** pushes the code to an **AWS S3 bucket**. **AWS CodeDeploy** pulls the code and deploys it to our servers.

**Maintainability**

* Code changes are easily applied. As mentioned above, only a single push is necessary to re-deploy applications on **Heroku** and **EC2**. This can be scaled to multiple instances.
* Logs from deployed instances are pushed to **PaperTrail**. This allows developers to view aggregated logs from different deployments, instead of using SSH to enter into the different deployed servers to view the logs
* A lot of deployment logic is abstracted away from developers by the service providers. For example, spinning up a new instance on AWS just takes a few clicks, and adding it to the deployment group (for automatic deployment) takes another few clicks.

**Performance**

* Both **AWS** and **Heroku** provide **auto-scaling abilities**. Being part of the cloud, the service providers can easily spin up new instances to serve a sudden influx of requests. However, we did not actually implement auto scaling as it is a paid service.

With an API-driven architecture that sits on the cloud, clients can be thin, since most of the ‘heavy lifting’ is done on the cloud servers.

## 

# **Integration Endpoints**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Source System** | **Destination System** | **Protocol** | **Format** | **Communication Mode** |
| Browser | Trading App Client Server | HTTPS | JSON | Synchronous |
| Trading App Client (on browser) | Apigee | HTTPS | JSON | Synchronous |
| Apigee | Trading App Backend Server | HTTPS | JSON | Synchronous |
| Apigee | Messaging Microservice | HTTPS | JSON | Synchronous |
| Messaging Microservice | RabbitMQ | AMQP (TCP) | Text/Binary | Synchronous |
| Messaging Microservice | Redis Job Store | RESP Protocol (TCP) | Text/Binary | Synchronous |
| RabbitMQ | Telebot Microservice | AMQP (TCP) | Text/Binary | Synchronous |
| Telebot Microservice | Redis Job Store | RESP Protocol (TCP) | Text/Binary | Synchronous |
| Telebot Microservice | Telegram API | HTTPS | JSON | Synchronous |
| Telebot Microservice | Messaging DB | Postgres Wire Protocol (TCP/IP) | Text/Binary | Synchronous |
| Trading App Backend Server | Trading DB | Postgres Wire Protocol (TCP/IP) | Text/Binary | Synchronous |
| Apigee | News Microservice | HTTPS | JSON | Synchronous |
| News Microservice | News DB | MongoDB Wire Protocol (TCP/IP) | JSON | Synchronous |
| News Microservice | News Provider REST API | HTTPS | JSON | Synchronous |

# 

# **Software/Services Required**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **No** | **Item** | **Quantity** | **License** | **Buy / Lease** | **Cost (Optional)** |
| 1 | VueJS | 1 | Open-sourced | - |  |
| 2 | Spring Boot | 2 | Open-sourced | - |  |
| 3 | Amazon Linux | 2 | Proprietary | Lease | Free Tier |
| 4 | Apache Maven | 2 | Open-sourced | - |  |
| 5 | Ubuntu | 1 | Open-sourced | - |  |
| 6 | Travis CI | 1 | Open-sourced | - |  |
| 7 | Apigee | 1 | Proprietary | Lease | $100 / month (estimated) |
| 8 | Heroku Web Server | 3 | Proprietary | Lease | Free Tier |
| 9 | Redis | 2 | Open-sourced | - |  |
| 10 | Telegram API | 1 | Open-sourced | - |  |
| 11 | AWS EC2 Instances | 2 | Proprietary | Lease |  |
| 12 | Nginx | 2 | Open-sourced | - |  |
| 13 | Heroku MongoDB | 1 | Proprietary | Lease | $200 / month |
| 14 | Heroku Postgres DB | 2 | Proprietary | Lease | $200 / month |
| 15 | CloudAMQP | 1 | Open-sourced | Lease | $99 / month |
| 16 | Papertrail | 1 | Proprietary | Lease | $18 / month |

# **Availability View**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Node** | **Redundancy** | **Clustering** | | | **Replication** | | | | Implemented? |
| Node config | Failure detection | Failover | Repl. type | Session state storage | DB Repl. Config | Repl. mode |
| Trading App Client Server | Horizontal Scaling | Active-active | Ping | Load- balancer | - | - | - | - | No (Paid service) |
| Trading App Backend Server | Horizontal Scaling | Active-active | Ping | Load-  balancer | - | - | - | - | Yes |
| Trading DB (Heroku Postgres) | Horizontal Scaling | Active-active | Ping | DB | DB | Client | Master-slave | Synchronous | No (Paid service) |
| Messaging DB (Heroku Postgres) | Horizontal Scaling | Active-active | Ping | DB | DB | Client | Master-slave | Synchronous | No (Paid service) |
| Messaging Microservice | Horizontal Scaling | Active-active | Ping | Load-  balancer | - | - | - | - | No (Paid service) |
| News DB (Heroku MongoDB) | Horizontal Scaling | Active-active | Ping | DB | DB | Client | Master-slave | Synchronous | No (Paid service) |
| News Microservice | Horizontal Scaling | Active-active | Ping | Load-  balancer | - | - | - | - | No (Paid service) |

# **Security View**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **No** | **Asset/ Asset Group** | **Potential Threat/Vulnerability Pair** | **Implemented Controls** | **Possible Mitigation Controls** |
| 1 | Data | SQL / Unsanitised user input (integrity, confidentiality) | User input sanitisation | Cloudflare Web Application Firewall |
| 2 | JSON Web Token | Cross-Site Scripting (XSS) / Unsanitised user input (integrity, confidentiality) | User input sanitisation, external javascript, refresh token implementation | Cloudflare Web Application Firewall |
| 3 | Services | Distributed Denial of Service (DDOS) / Application (availability) | Load-Balancing | Cloudflare DDOS protection |
| 4 | Data | Man-in-the-middle Attack / Packets (integrity, confidentiality) | HTTPS enforced with HSTS header |  |
| 5 | Data | Click-jacking / User  Interface (integrity, confidentiality) | X-Frame-Options HTTP header set to **deny** |  |

# **Performance View**

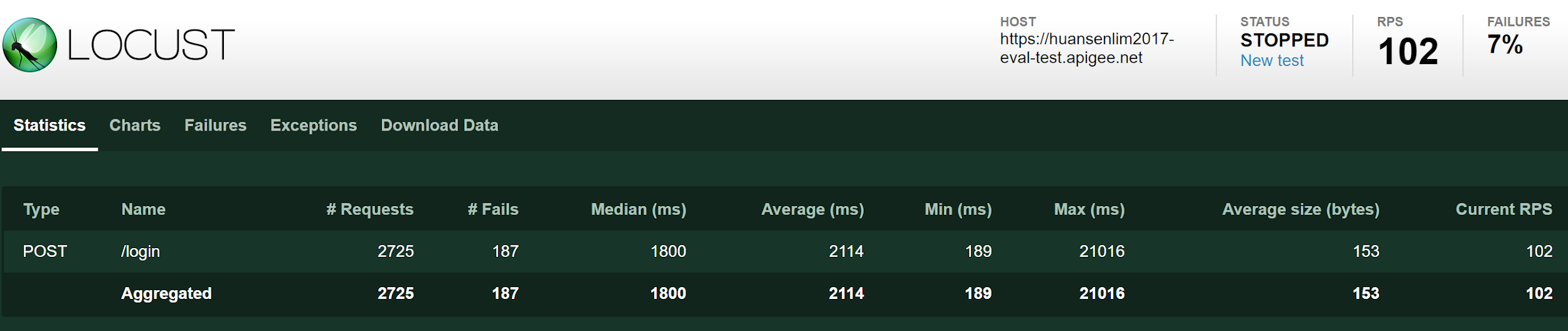
1000 users simulated, 50 hatch rates

(50 new users are spawned per sec, up to 1000 concurrent users)

Proxied and round-robin load balanced by Apigee across two servers

Successful requests per second: 102

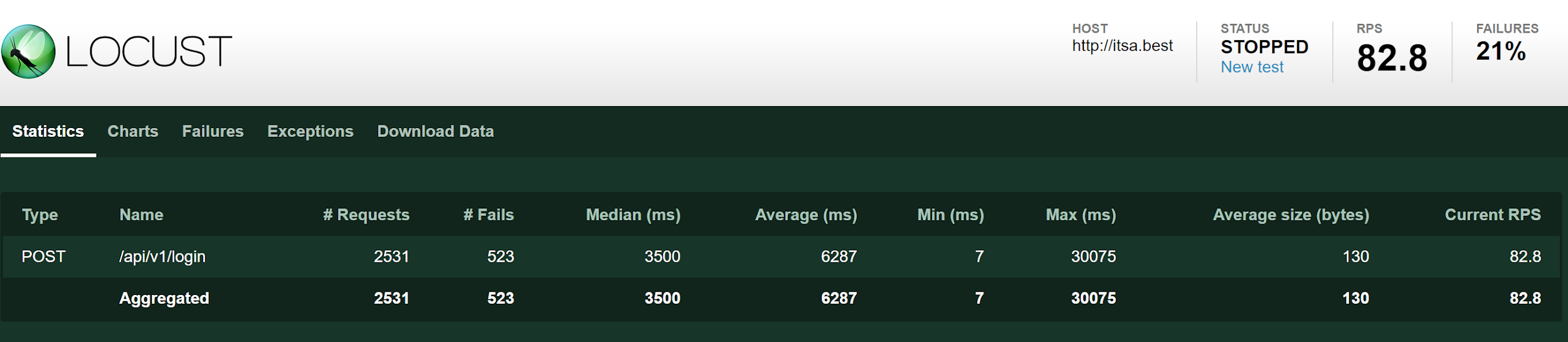
Failure Rate: 7%



Direct ping on itsa.best

Successful requests per second: 82.8

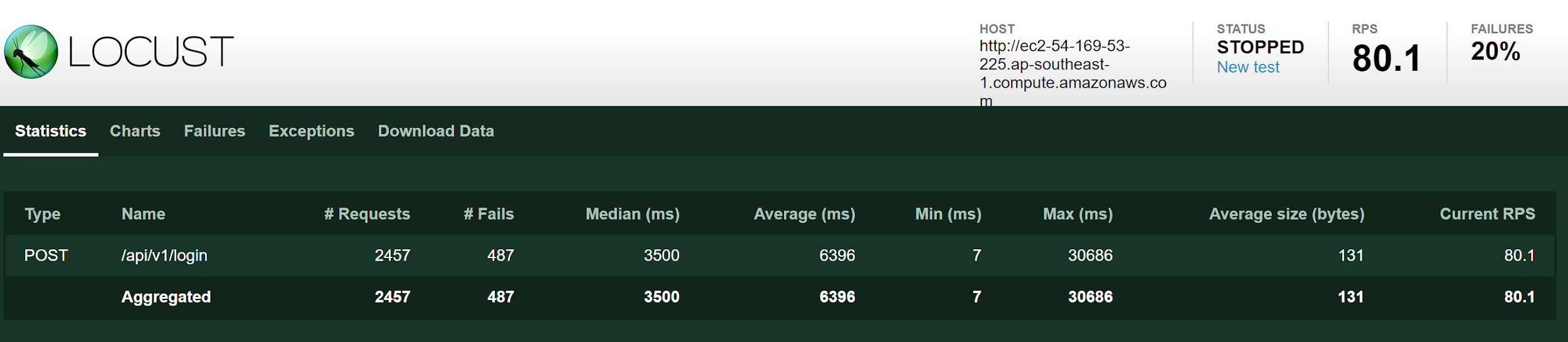
Failure Rate: 21%



Direct ping on slave server

Successful requests per second: 80.1

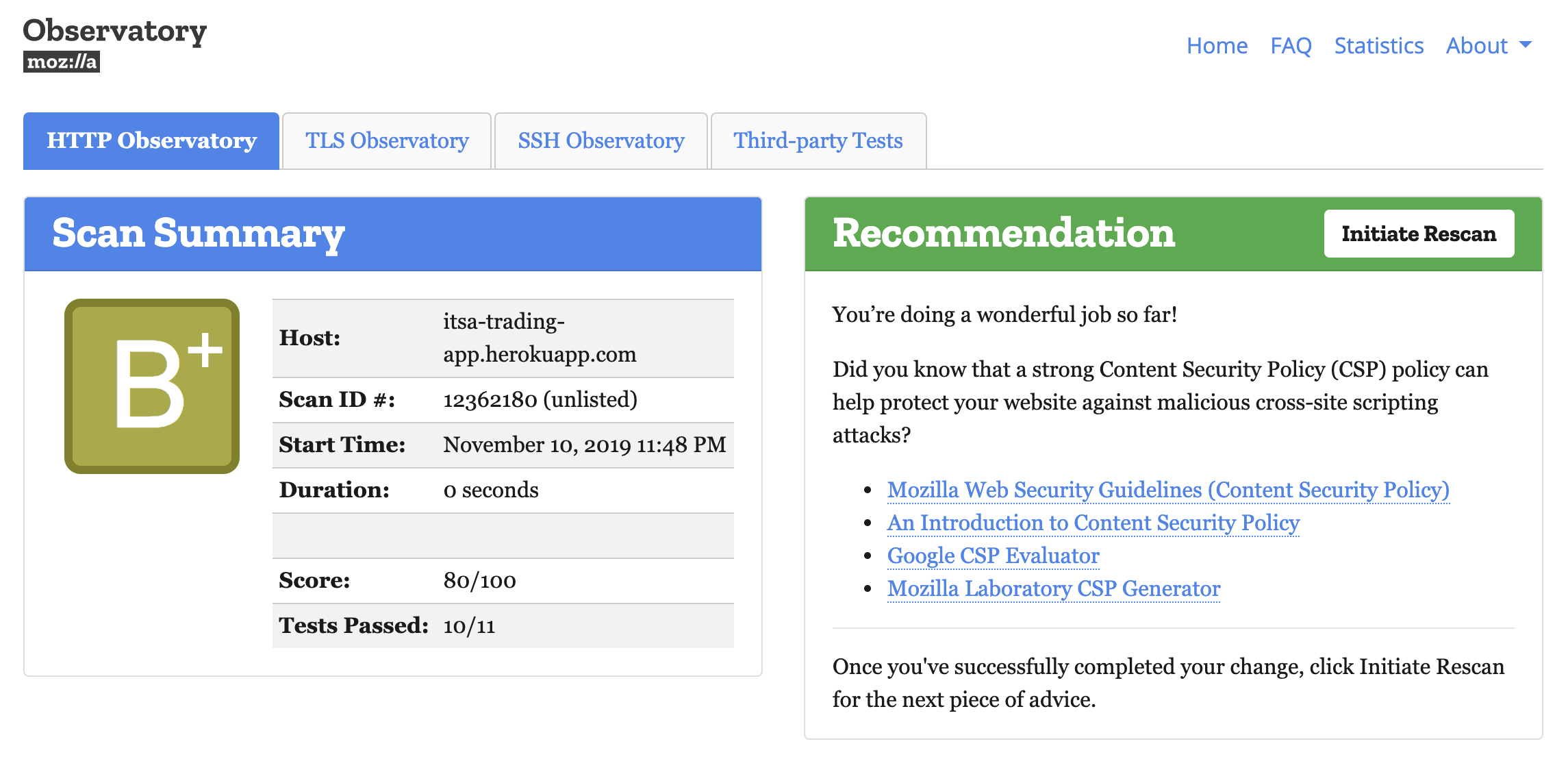
Failure Rate: 20%



|  |  |  |  |
| --- | --- | --- | --- |
| **No** | **Description of the Strategy** | **Justification** | **Performance Testing (Optional)** |
| 1 | Load balancer on Apigee (implemented) | Running two instances of the application across two servers allows for more requests to be served. We do not run two applications on a single server since that would be a single point of failure | Locust - an open-source load-testing framework built with Python |
| 2 | Client-side eager loading (implemented) | JSON payloads holding Instrument price data to populate the main chart are large in size, and requests for them require an average of about 1 to 2 seconds to complete. To allow the Trading App Client to appear faster, it fetches the data from the Trading App Backend for a select number of instruments, caching the data in the browser’s memory, ready to be loaded on the user’s click. | - |

# **Appendix**

**Fig 1**



Content-security-policy header cannot be implemented properly due to the **highcharts.js** library used in the single page application <https://github.com/highcharts/highcharts/issues/6884>.

**Fig 2**

