BSDSESM1KU-20201

DevOps, Software Evolution and Maintenance, MSc (Spring 2020)

Alex Ardelean - aard@itu.dk

George Dobrin - gedo@itu.dk

Petrut Jianu - peji@itu.dk

Theo Meyer - tmey@itu.dk

Table of Contents

Introduction **4**

System Perspective **5**

System Architecture and Design 5

Figure 1: MVC interactions diagram 6

Figure 2: MiniTwit context diagram 7

Figure 3: User login & tweets sequence diagram 8

Dependencies 9

Technologies 9

Tools 10

Subsystem Interaction 11

Figure 4: Subsystem interaction overview 11

Results 12

Figure 5: Sentry Dashboard at the time of writing 12

Figure 6: GTmetrix scores for our MiniTwit client 14

Figure 7: Google Lighthouse audit scores 14

Weekly Tasks 15

Process Perspective **17**

Team Interaction 17

Figure 8: Automated GitHub Project board 18

CI/CD 18

Heroku CI 18

Figure 9.1: Heroku automatic deploys with CI pass for the simulator 19

Figure 9.2: Heroku Pipeline with our applications and their related GitHub branches19

GitHub Actions 20

Figure 10.1: Successful run of the Node.js CI Action 20

Figure 10.2: Successful run of the Release Action 21

Repository Organization 21

Git and GitHub Setup 22

Figure 11: GitHub Community profile 22

Monitoring and Logging 23

Figure 12.1: Heroku Dashboard - Events and Memory Usage (1 day) 23

Figure 12.2: Heroku Dashboard - Response Time and Throughput (2 hours) 24

Figure 12.3: Heroku Dashboard - Dyno Load and Garbage Collection (2 hours) 25

Figure 12.4: Heroku Dashboard - Overall Response Time and Throughput (1 day) 25

Figure 13.1: Atlas Dashboard - Overall performance metrics 26

Figure 13.2: Atlas Dashboard - close up view of the Opcounters graph 26

Figure 14: Papertrail setup with predefined search options 27

Figure 15: Sentry project statistics, no events for our MiniTwit 28

Security 28

Clustering 29

Lessons Learned Perspective 31

Operation 31

Evolution and Refactoring 32

Figure 16.1: Users table index with over 100 hits per minute 33

Figure 16.2: Messages table compound index 33

Figure 17.1: Heroku metrics dashboard after implementing optimizations 34

Figure 17.2: Heroku metrics dashboard after implementing optimizations 34

Maintenance 36

Figure 18: Latest graph for Group L 36

Figure 19: Group L errors reported by the simulator 37

Table 1: Severity classification of an incident 38

Figure 20.1: First Sentry reported incident, isolated to the unfollow action 38

Figure 20.2: Pagination incident #1 observed after a deploy, resolved immediately 38

Figure 20.3: Pagination incident #2, caused by an improper fix of the first one 39

Figure 20.4: Logical error that caused the unfollow function to never terminate 39

Figure 20.5: Our MongoDB instance ran out of storage space 39

Conclusions 40

DevOps Report

Group L - Blanao Systems

Introduction

This is the final project report of Group L (Blanao Systems) for the “DevOps, Software Evolution and Maintenance” ITU course. This paper covers everything we did and the reasons behind why we did it in that specific manner over the Spring semester of 2020.

Our goal was to migrate an old application into a modern environment while also improving the development process. For this, we used DevOps techniques such as Continuous Integration and Deployment, as well as system monitoring to quickly spot problems before they become disasters. Furthermore, the security enhancements and the readiness for scalability were also important parts that we had focused on.

Our client system can be accessed at this link.

This paper presents an overview of our project from three different perspectives: system design, technical details and development process, and -last but not least- the learning perspective.

System Perspective

System Architecture and Design

In order to adhere to the web standards, we designed our MiniTwit system following the Model-View-Controller design pattern. The model is the database, the view is the client (the front end) and the server (the backend) is the controller. In order to check our reasoning for the technological stack, please refer to the dependencies part.

In the MVC applications, the whole system is seen as a part that represents the user interface and a part that contains the application logic. Another important aspect of an MVC system is the separation of concerns - each part of the model, view, controller can be easily decoupled and others can take the place. For example, if the view part contains too many bugs, or if it needs to be redesigned, it can be decoupled or altered without impacting the model or the controller. In our view, the most important pros of the MVC design pattern in our use case are the following:

- Suited for the fast development of big applications - because of the separation of concerns -> development can be done independently, parallelly.
- After launching the system, if we observe that some parts of the application are not well implemented (i.e. bugs arise), we can quickly intervene, without affecting the other parts of the system.
- Adding features will be easily implemented, with minimal risk of introducing bugs to current functionality since the system is compartmentalized.

Figure 1 illustrates the way model-view-controller systems work: the user interacts with the view which makes use of the controller to manipulate the model.

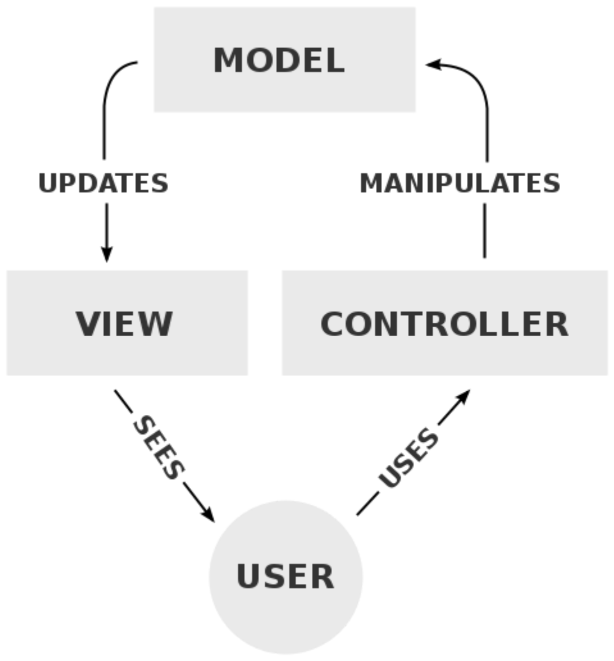


Figure 1: MVC interactions diagram

The context diagram (Figure 2) defines the boundaries of our system. It contains both internal and external components that interact with one another. The internal parts are, in fact, the model, view, and controller. On the other hand, Heroku, which is used primarily for hosting and logging the runtime of our frontend and backend parts, is an external system. The component repository, Github is used by us, the developers.

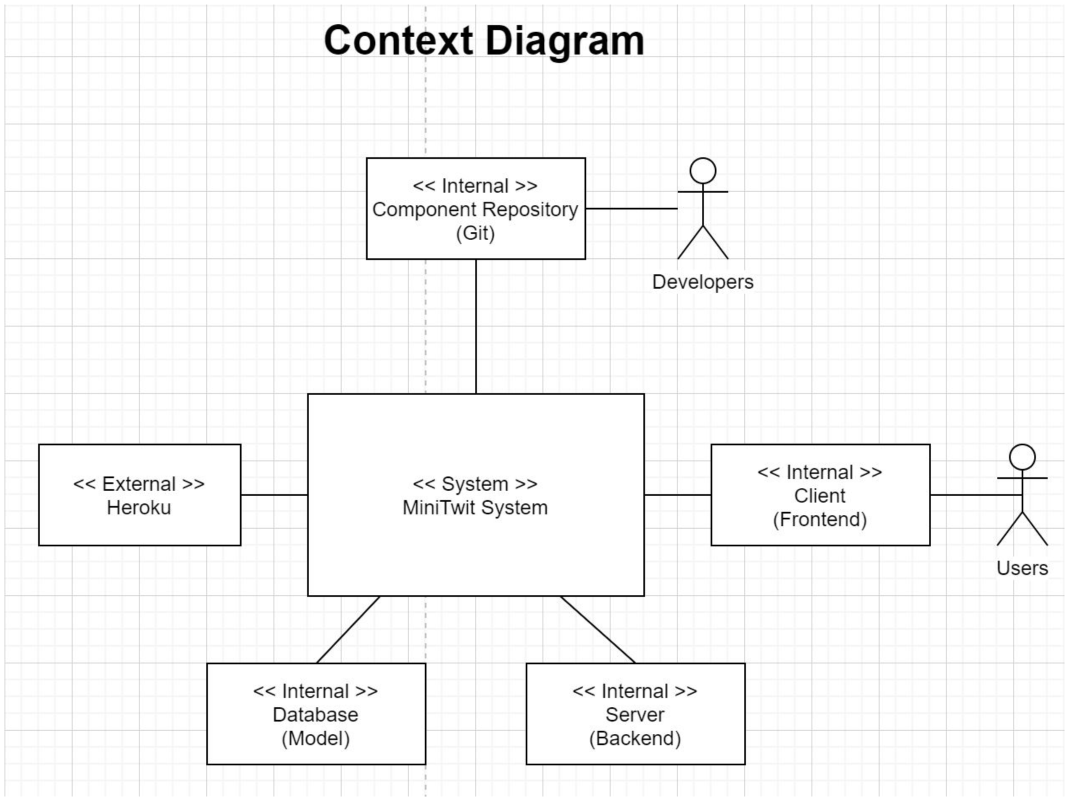


Figure 2: MiniTwit context diagram

In order to better clarify how the three main parts (backend, frontend, database) of our system interact, figure 3 illustrates a sequence diagram for the following scenario: a user wants to authenticate by using his credentials and then he desires to check his public feed. Firstly, he enters the credentials in the client part which sends the request to the backend. The backend redirects the request to the database which verifies them and responds accordingly. If the credentials are correct, the user can authenticate. Otherwise, the user is denied access. Then, after the user accesses the dashboard (main feed), a request is being sent from the frontend to the backend which asks for the database to find all the relevant tweets. The database responds accordingly and finally, the user can check his public feed.

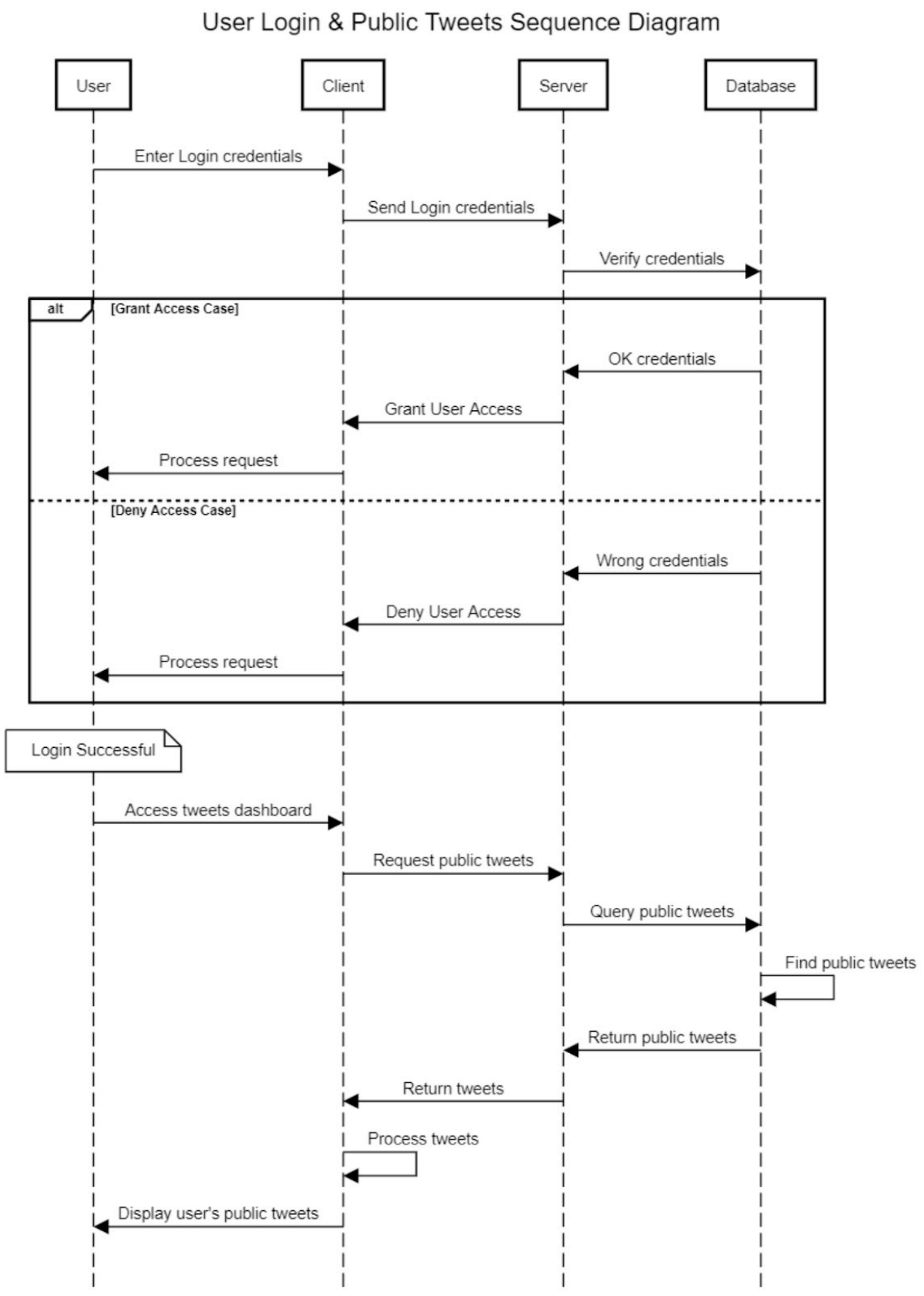


Figure 3: User login & tweets sequence diagram

Dependencies

Our dependencies and the thinking process behind our technology choices are also described in our GitHub repository to increase transparency (StackChoices.md). Lightweight libraries are generally preferred over complex frameworks in order to keep the overhead of the system low and ease future development tasks. This is discussed in more detail in the **Operation** section of this report.

The common theme with our system is simplicity: we try to achieve all DevOps goals with minimal clutter. The entire system is built using JavaScript and includes only popular and well-supported JavaScript libraries, minimizing maintenance costs.

The same philosophy applies to our choice of tools, only platforms that could serve multiple functionalities were included. Heroku and GitHub especially come with lots of built-in functionality and easy to integrate plugins. For example, after deploying MiniTwit to Heroku, we already had an advanced monitoring dashboard, a CI/CD system integrated with our GitHub repository, virtualized and scalable containers (Dynos), and live logging.

Technologies

- **Node** - Enables server-side JavaScript.
- **Express** - Backend framework for our APIs.
- **Mongoose** - DB abstraction layer that connects to the Mongo database.
- **React** - Frontend framework for the MiniTwit client. React was our first choice for the front end because it has good performance, is easy to integrate with the rest of the stack. It also has a strong ecosystem and community, which makes it a better solution that Vue.js or Angular.
- **Semantic** - UI library. Unlike its competitor, Semantic already has a built-in template for social media posts and news feed which makes it a no-brainer.

- **Ava** - Framework for writing readable tests.
- **ESLint** & **Prettier** - Static analysis tools that check code complexity and format.
- **Throng** - Clustering multiple workers for auto scaling and load balancing.

Tools

- **Heroku**
    - Dashboard - Monitoring
    - Heroku CI - CI/CD
    - Dynos - Virtualization and autoscaling
    - Papertrail - Logging
- **GitHub**
    - Actions - CI/CD, automation
    - Issues, PRs, Projects, Releases, Milestones - Team organization
- **Mongo Atlas**
    - Distributed database
    - Database monitoring
- **Sentry**
    - Error monitoring
    - System status

Subsystem Interaction

As previously stated, our MiniTwit system contains three main parts: the database - developed by using MongoDB and hosted on their cloud service, the front end - developed in React.js and hosted on Heroku and the backend - developed in node.js and split into two parts: the API for the simulator (implemented according to the simulator requirements) and the API used by the client (implemented according to the already existing python miniTwit application), both of them hosted by Heroku. The usual flow of the system is the following: the user initiates a request by using the frontend, which then triggers the controller to manipulate the model. The model would reflect the changes and then the user would be able to see the desired outcome. On the other hand, when it comes to the simulator API, an external system would use the API which would manipulate (most likely create) the model with new changes (for example, registering new users, followers, and messages). Figure 4 highlights these system interactions.

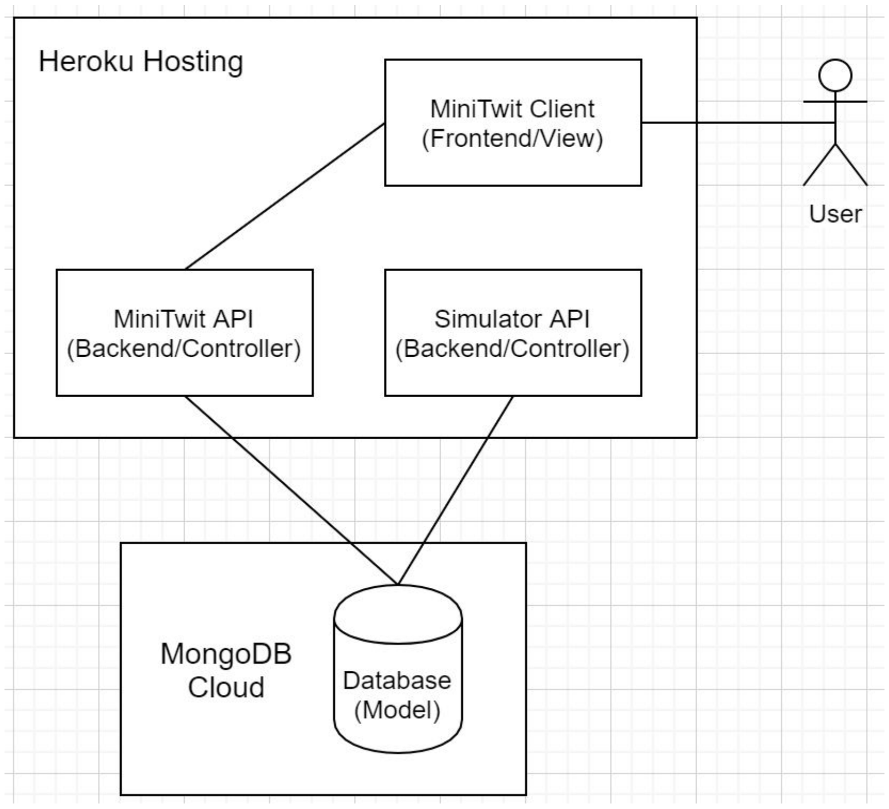

Heroku Hosting
MiniTwit Client (Frontend/View)
User
MiniTwit API (Backend/Controller)
Simulator API (Backend/Controller)
MongoDB Cloud
Database (Model)

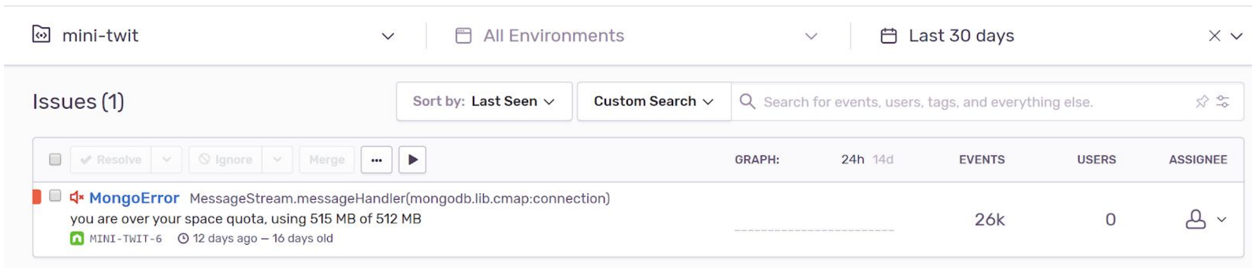
Figure 4: Subsystem interaction overview

Results

In order to properly evaluate the current state of our system, we inspired our assessment from the TIOBE Quality Assessment which suggests that software code quality should be assessed by taking into consideration the reliability, testability, and maintainability of the system. We will be using a grading system where 1 is the minimum score and 5 is the maximum score.

Reliability takes into consideration the stability of the system and responds to questions similar to: “how often does the system crash?” or “Are there any null references existent in the code?”.

In our case, in order to assess the reliability of our system, we can make use of Sentry - an error logging application integrated into our MiniTwit systems. Whenever a run-time error occurs, Sentry notifies the developers accordingly. As it can be seen in Figure except for the MongoError (which showed up because we used the free tier for our MongoDB cloud database - we had only 512MB of space available to use and the collection data surpassed the threshold), in the last 30 days, no other errors popped up.



MongoError MessageStream.messageHandler(mongodb.lib.cmap:connection)
you are over your space quota, using 515 MB of 512 MB
MINI-TWIT-6 12 days ago – 16 days old
26k 0

Figure 5: Sentry Dashboard at the time of writing

During the entire period of this course, we discovered five incidents that usually affected isolated parts of the application, so we consider them only *Severity-1* (more details follow in the ***Maintenance*** section). This is proof of the strength of our system’s reliability. However, it can be argued that, indeed, there is an error. Conclusively, we rate our system’s reliability a 4 out of 5.

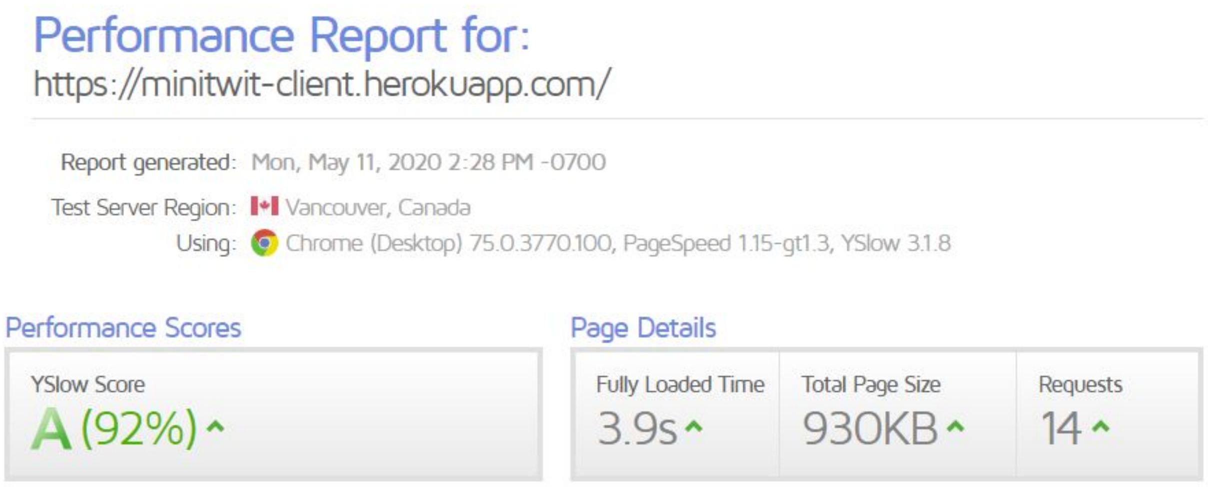
Testability is concerned with the system’s ability to be tested whether it meets the intended requirements. In our case, during the development, unit testing has been set up by making use of Ava - a node.js test runner with detailed error output. We estimate a 90% coverage of source code is automatically tested. There are some edge cases that are yet to be implemented and thus, the testability score for our system is 3/5. In order to verify our implementation of unit testing please refer to the test.js file in the production_api branch of our Github repository.

Maintainability is defined as the degree to which an application is understood, repaired, or enhanced. In our case, because we designed our system adherent to the MVC design pattern, it is out of the box “plug and play” meaning that each part of the frontend, backend and database can easily be decoupled and replaced whenever is needed.

Moreover, this separation of concerns can prove to be really useful in fixing bugs (for example, if bugs are found in the backend, it is highly unlikely that the frontend codebase would be affected) as well as in the implementation of new features (for example, adding a new feature in the frontend doesn’t necessarily mean a change in the backend codebase). This is also supported by the hands-on experiences that we had.

As discussed in the **Maintenance** section, nearly all of our incidents were resolved before they could have any significant impact on our users. Conclusively, we rate our system’s maintainability with 5/5.

Beyond this, when it comes to best practices for our front end, according to both Google Lighthouse and GTmetrix, our client application offers top-notch scores as it can be observed in figures 6 and 7.



Performance Report for:
https://minitwit-client.herokuapp.com/

Report generated: Mon, May 11, 2020 2:28 PM -0700
Test Server Region: Vancouver, Canada
Using: Chrome (Desktop) 75.0.3770.100, PageSpeed 1.15-gt1.3, YSlow 3.1.8

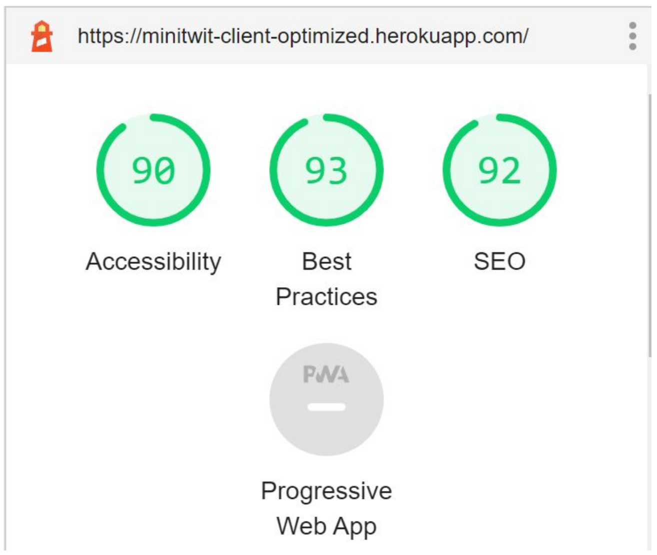
Performance Scores

YSlow Score
A (92%)

Page Details

Fully Loaded Time	Total Page Size	Requests
3.9s	930KB	14

Figure 6: GTmetrix scores for our MiniTwit client



https://minitwit-client-optimized.herokuapp.com/

Accessibility	Best Practices	SEO	Progressive Web App
90	93	92	–

Figure 7: Google Lighthouse audit scores

Weekly Tasks

All weekly tasks have been completed successfully, as presented in the list below, which also includes links to the significant Pull Requests and commits.

1. Refactor MiniTwit to Python3
   a. Feature Mapping
2. Migrate MiniTwit to JavaScript & describe the Distributed Workflow
   a. Introduced Heroku, Node.js, Express, MongoDB, Ava
   b. [Refactor] Develop backend + database
   c. Use a framework to migrate old tests
3. Implement the API for the MiniTwit Simulator & DB Abstraction
   a. Introduced Mongo Atlas, Mongoose, React
   b. Introduce a DB abstraction layer (ORM)
   c. Move the UI to React using a front-end library
4. Create a CI/CD pipeline
   a. Introduced Heroku CI, GitHub Actions
5. Simulator start & Maintenance
   a. Introduced Sentry
6. Monitoring & Architecture Presentation
   a. Using Heroku Dashboard
   b. https://docs.google.com/presentation/d/16-qJESyxr7EPDUeY5y9oys2cQJl7Y1fK5C2dmoQxqkM/edit?usp=sharing

7. Subsystem Interface & Static Analysis Tools
    a. Using Postman
    b. Introduced ESLint, Prettier
8. Logging & SLA
    a. Using Heroku Papertrail
    b. SLA: https://docs.google.com/document/d/1IthVPXtvveWKN7ODbhaocV2EN0tR8GYEwyGzqLBGEJY/edit?usp=sharing
9. Monitoring SLA & Security
    a. Security Report from Blanao Systems (Group L)
    b. Average response time is higher than expected
10. Clustering
    a. Introduced Throng.js
    b. Setup Heroku for high availability

Process Perspective

Team Interaction

How do you interact as developers?

How is the team organized?

Roles, communication channels, best practices

We communicate over…

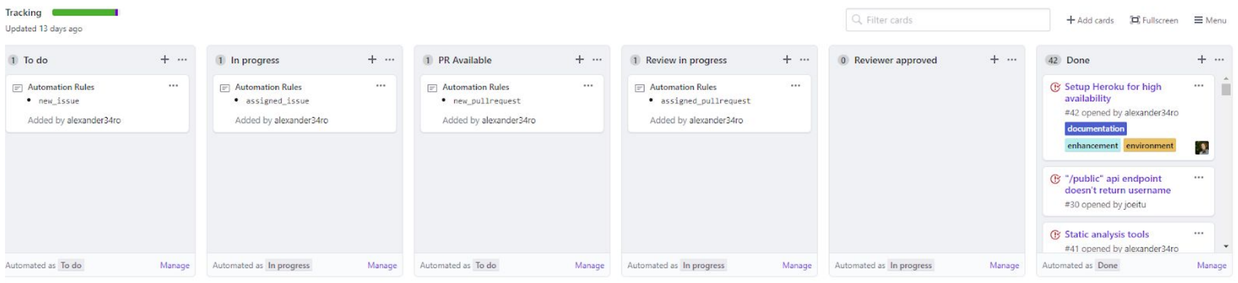
We use a GitHub Project to organize work…

When somebody submits a Pull Request, they wait for a review from somebody else…

We used two main tools for our team interaction: Slack and GitHub. Slack was used for casual and non-essential communication. More important discussion took place on the GitHub issue tracker, as it allows us to split them into different threads and link them to git actions like commits and pull requests. The Github issue tracker was also used to assign team members to issues, and review pull requests before merging.

Tasks were listed into a GitHub Project board to better organize work. An automation script would add a new task to the To do column of the GitHub Projects kanban board when an issue is created. After an issue is assigned to a team member it will move it to the In progress column, and so on. The kanban board is split into the following sections:

- To do
- In progress
- PR Available
- Review in progress
- Reviewer approved
- Done



1 To do
Automation Rules
- new_issue

Added by alexander34ro
Automated as To do — Manage

1 In progress
Automation Rules
- assigned_issue

Added by alexander34ro
Automated as In progress — Manage

1 PR Available
Automation Rules
- new_pullrequest

Added by alexander34ro
Automated as To do — Manage

1 Review in progress
Automation Rules
- assigned_pullrequest

Added by alexander34ro
Automated as In progress — Manage

0 Reviewer approved
Automated as In progress — Manage

42 Done
Setup Heroku for high availability
#42 opened by alexander34ro
documentation enhancement environment

"/public" api endpoint doesn't return username
#30 opened by joeitu

Static analysis tools
#41 opened by alexander34ro
Automated as Done — Manage

Figure 8: Automated GitHub Project board

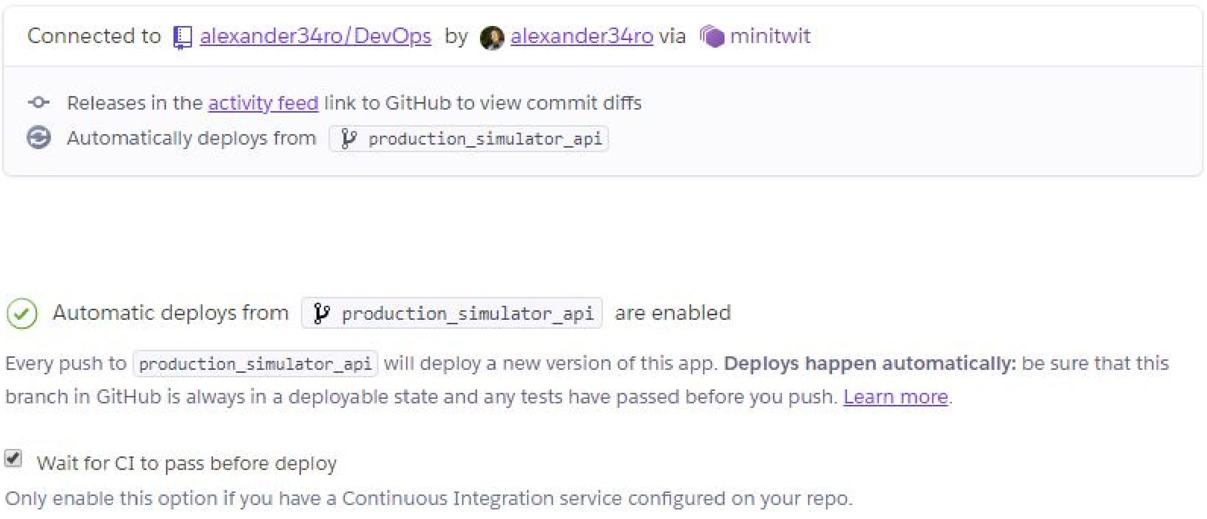
CI/CD

In order to accomplish a well written code base which would eventually reach our target users as quickly and safely as possible, we used the following tools to construct the Continuous Integration/Continuous Deployment pipeline. The reason for choosing them is their exceptional sinergy and reduced overhead. As Heroku is able to connect directly to GitHub and check the results of Continuous Integration obtained there before deploying our apps, all we needed to do was integrate some GitHub Actions that could run our test suite and checks.

Heroku CI

The way Heroku works is that it links directly to our Git repository, creating a remote heroku branch for deployments. The manual deployment process is realised by pushing a version of our code to this branch, which in turn triggers Heroku to build and deploy the application in a virtualized Dyno container/s.

This process would add extra steps and chances for human errors, so instead, we chose to go with the automatic deploys feature, which links the Heroku repository with our GitHub repository and gives us better control over the deployment pipeline. This way, every time we push to a production branch, the same code gets deployed to the Heroku repository for that branch/app, but not before the test suite is run successfully.



Connected to alexander34ro/DevOps by alexander34ro via minitwit

Releases in the activity feed link to GitHub to view commit diffs

Automatically deploys from `production_simulator_api`

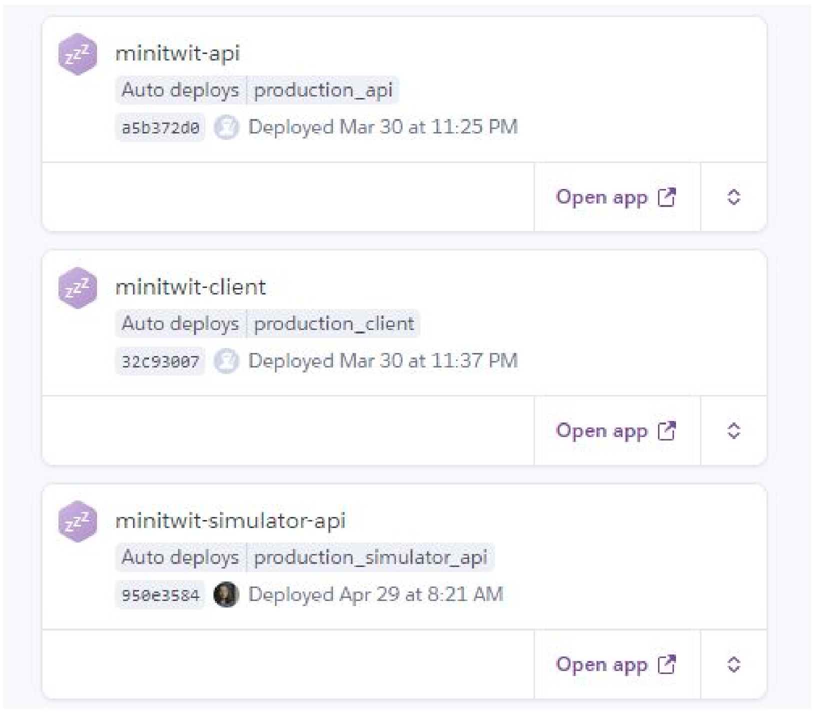
Automatic deploys from `production_simulator_api` are enabled

Every push to `production_simulator_api` will deploy a new version of this app. **Deploys happen automatically:** be sure that this branch in GitHub is always in a deployable state and any tests have passed before you push. Learn more.

- [x] Wait for CI to pass before deploy

Only enable this option if you have a Continuous Integration service configured on your repo.

Figure 9.1: Heroku automatic deploys with CI pass for the simulator



minitwit-api
Auto deploys | production_api
a5b372d0 Deployed Mar 30 at 11:25 PM
Open app

minitwit-client
Auto deploys | production_client
32c93007 Deployed Mar 30 at 11:37 PM
Open app

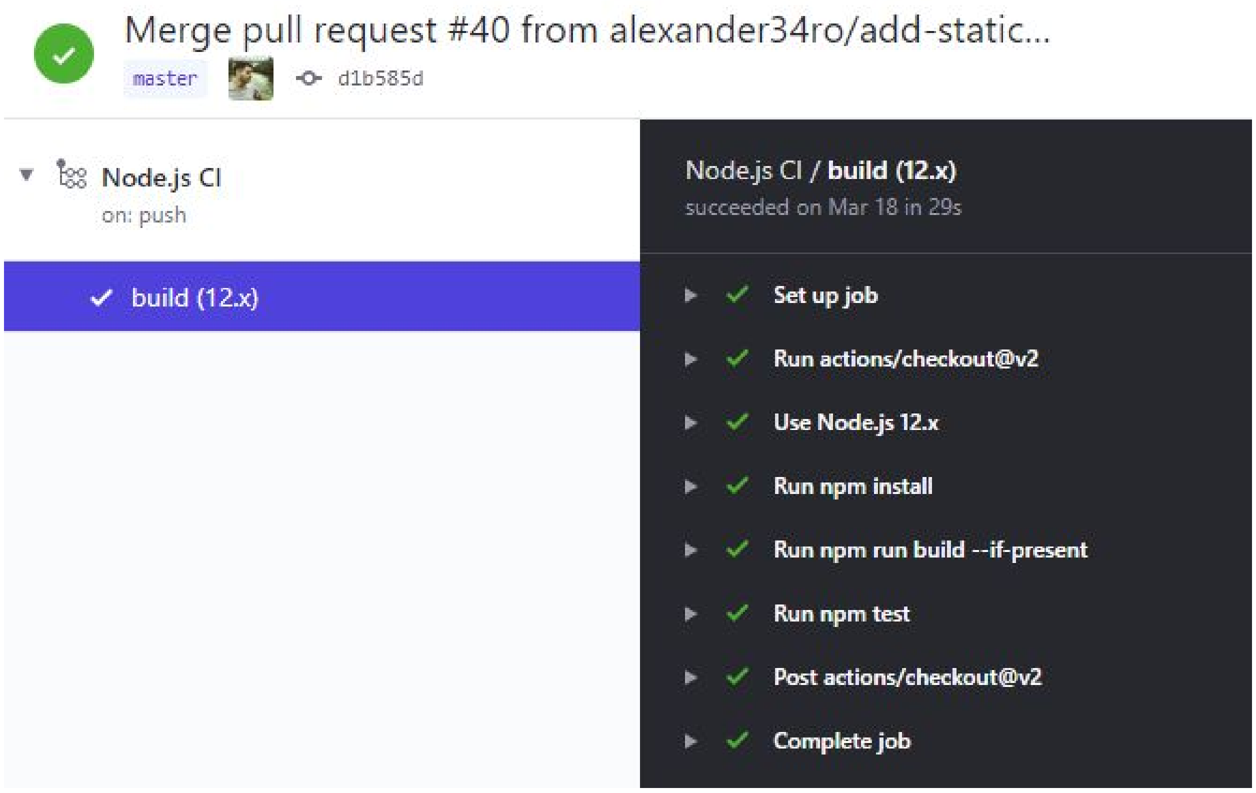
minitwit-simulator-api
Auto deploys | production_simulator_api
950e3584 Deployed Apr 29 at 8:21 AM
Open app

Figure 9.2: Heroku Pipeline with our applications and their related GitHub branches

GitHub Actions

GitHub Actions is one of the platform's most awaited features. It brings the ability to create scripts that are executed by GitHub and alter or operate on part of a repository.

For our CI/CD pipeline, we chose the most popular Node CI script we could find on the GitHub Actions Market and customized it for our environment's needs. This means setting it up to use the latest version of Node.js and creating an npm "test" script for running our test suite with Ava and our static analysis tools. This Action runs every time there is a new commit to a branch and it changes the CI status of the respective Pull Request.



Merge pull request #40 from alexander34ro/add-static...

master d1b585d

Node.js CI
on: push

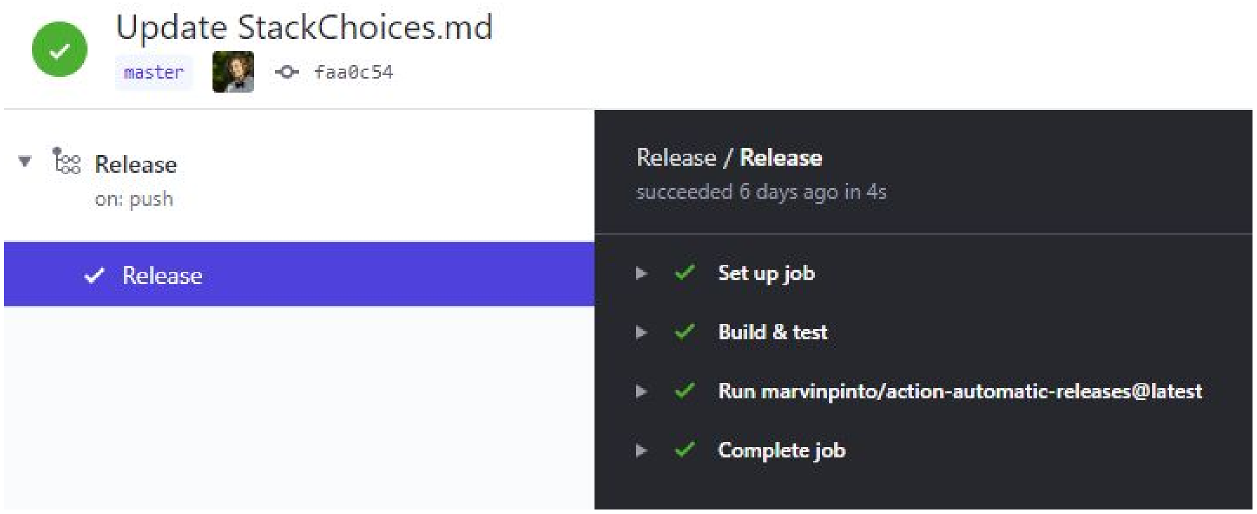
✓ build (12.x)

Node.js CI / build (12.x)
succeeded on Mar 18 in 29s

✓ Set up job
✓ Run actions/checkout@v2
✓ Use Node.js 12.x
✓ Run npm install
✓ Run npm run build --if-present
✓ Run npm test
✓ Post actions/checkout@v2
✓ Complete job

Figure 10.1: Successful run of the Node.js CI Action

Another GitHub Action that is part of our pipeline is the Release Action which automatically creates release builds for our application. This is done solely to meet course requirements and create evaluation artifacts, but it is not a part of our CI/CD pipeline because of our direct link between GitHub and Heroku.


Update StackChoices.md

master faa0c54

Release
on: push

Release

Release / Release
succeeded 6 days ago in 4s

Set up job

Build & test

Run marvinpinto/action-automatic-releases@latest

Complete job

Figure 10.2: Successful run of the Release Action

Repository Organization

Because the project was not expected to grow very large and in order to keep things simple we used a mono repository, storing everything in a single public GitHub Repository. There are three folders on the root of our repository. We keep database backups in backups. The refactored MiniTwit application using Python 3 can be found inside remote_code. Our three JavaScript applications are placed inside refactore_remote_code.

We make use of submodules for our three apps, that are also integrated with git and Heroku so that changes made to the subfolder of a project get deployed to the corresponding server. For example, the code for the simulator is contained in the simulator_api folder and, when code is pushed to the production_simulator_api branch rooted in this folder, it gets automatically deployed to the Heroku repo for the simulator.

We use templates to better organize Issues and Pull Requests and to guide external contributors that would like to commit to our repository. We also welcome contributors with a Code of Conduct and Contributing Guidelines. We encourage transparency by making all our decisions public and available in the Stack Choices file.

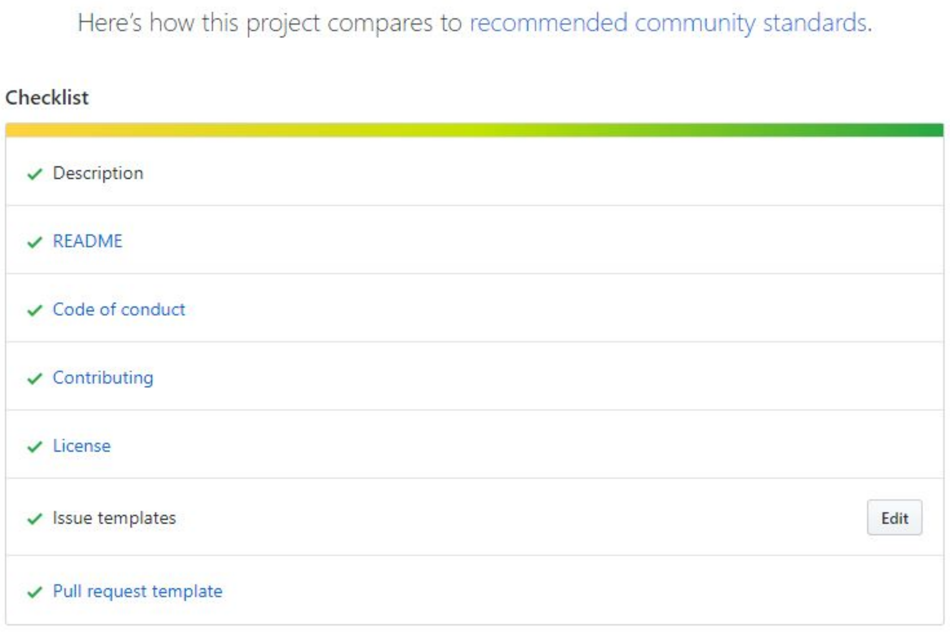
Git and GitHub Setup

As a branching model, although it is not the most robust way of branching, due to the development team being small and everybody working on separate parts of the project we decided on a feature-branching model. In simpler words: we create a branch, develop the feature, test it, and then merge the branch.

Each open task had its own Issue to facilitate progress tracking. These ranged from weekly tasks to simple documentation Issues and can be viewed here. To have a better overview of our progress, we also took advantage of the integrated Kanban board feature from GitHub. Furthermore, we took it a step further by installing a robot that automatically moves the cards based on certain actions, such as assigning to a developer.

Details about CI/CD and analysis tools can be found in the CI/CD subchapter.

The work that went into developing our GitHub repo is also illustrated in our fully complete Community profile.


Here's how this project compares to recommended community standards.

Checklist

- ✓ Description
- ✓ README
- ✓ Code of conduct
- ✓ Contributing
- ✓ License
- ✓ Issue templates
- ✓ Pull request template

Figure 11: GitHub Community profile

Monitoring and Logging

Heroku’s built-in dashboard was our choice as a monitoring solution. This is both due to its powerful features, but also because it requires no additional integration and maintenance costs on our end.

There are several metrics that were deemed relevant for monitoring purposes. At the top of our dashboard we have the Events sections displaying deploys and Dyno scaling in blue, Heroku outages in orange, and server errors in red. Below it, we keep a close eye on the memory usage of our application, since our clustered Node instances can scale up to a large number of nodes with high RAM requirements.

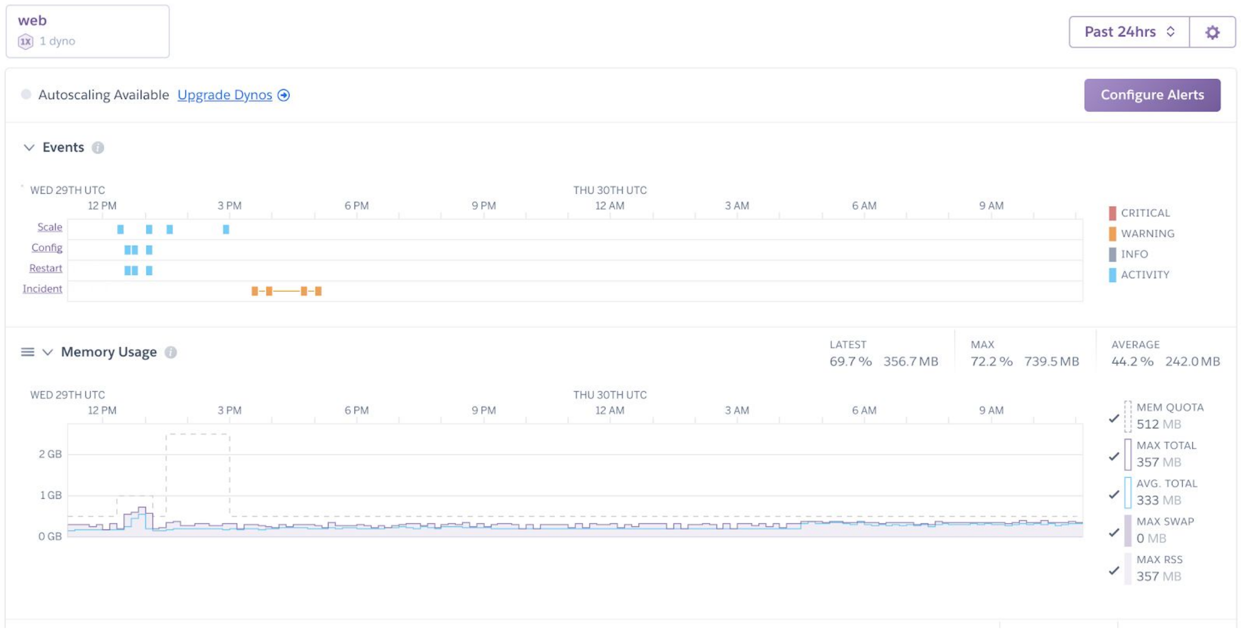


Figure 12.1: Heroku Dashboard - Events and Memory Usage (1 day)

Next, we look at the response time and throughput of our app. We chose these metrics because they give us insight into how responsive our application is to the users.

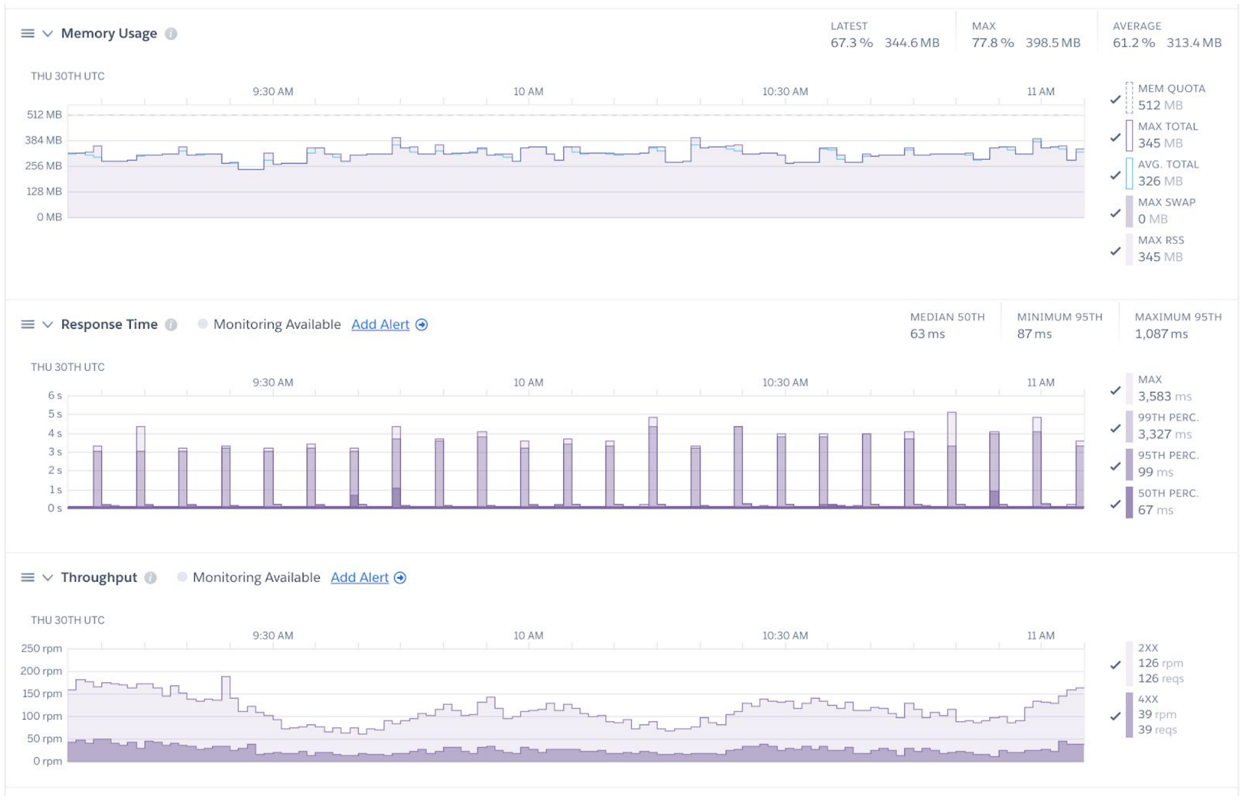


Figure 12.2: Heroku Dashboard - Response Time and Throughput (2 hours)

Last, we check the load on our Dyno containers and the performance of the Node.js system. An interesting observation we made based on these metrics is that every time there is a garbage collection event in the JavaScript engine, our response time spikes dramatically for a few moments. We tried to find a solution for this problem, but at the moment it seems to be a known JavaScript issue.

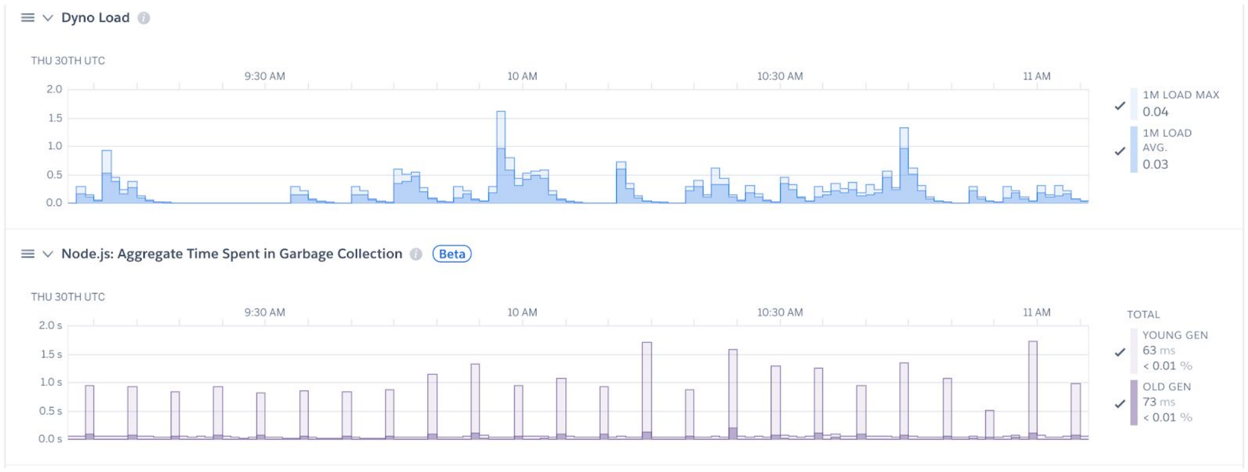


Figure 12.3: Heroku Dashboard - Dyno Load and Garbage Collection (2 hours)

In a more positive light, overall, our results are quite solid, averaging a less than 100ms response time for the 95th percentile and an average throughput of over 100rpm.

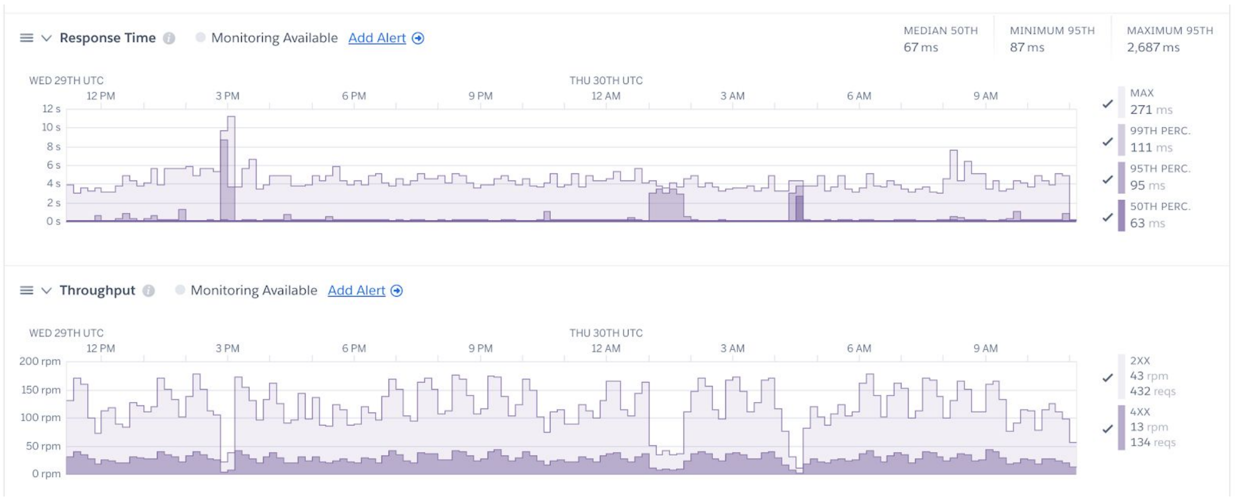


Figure 12.4: Heroku Dashboard - Overall Response Time and Throughput (1 day)

MongoDB Atlas also provides us with a dashboard with the relevant database metrics for our shards. An easily searchable dashboard with fine granularity options gives us vision into the number of operations processed every second, the average speed of the network interface, the number of active connections at any time, and the disk size of the database.

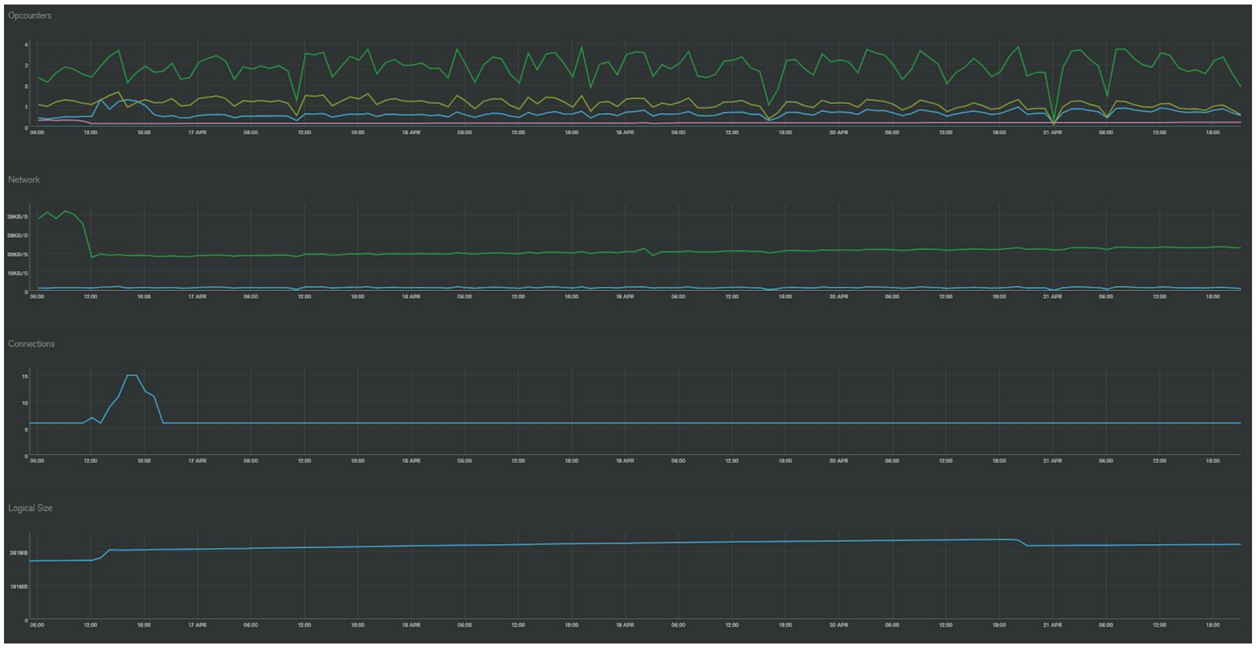


Figure 13.1: Atlas Dashboard - Overall performance metrics

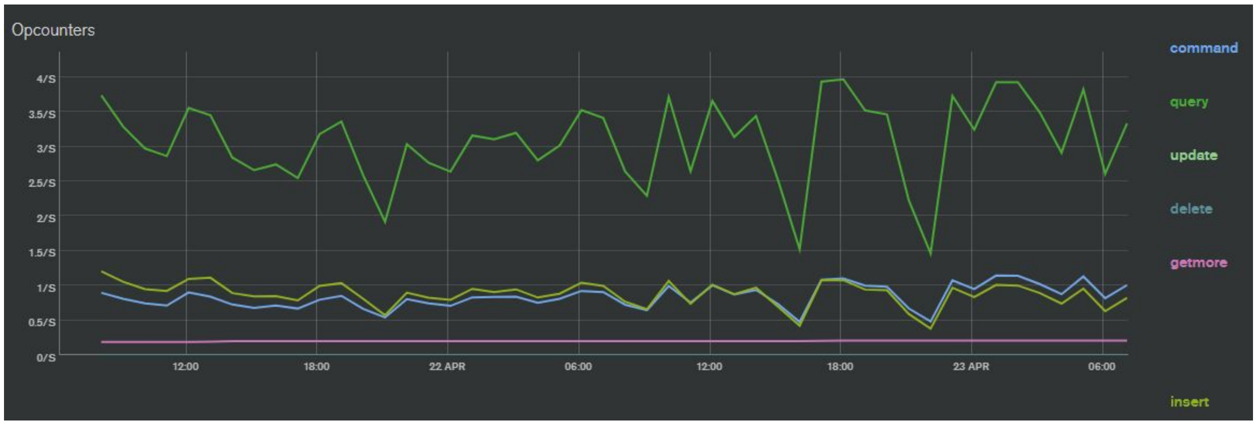


Figure 13.2: Atlas Dashboard - close up view of the Opcounters graph

For logging, Papertrail was chosen due to its direct integration as a Heroku add-on and its highly feature-rich free tier package. It came with a default setup that included filters for all of Heroku's events, like Deploys and Dyno changes, but we also added our own searches to easily find error messages and request data (Figure 14).

Since Papertrail gives us a generous monthly quota, we decided to log data regarding all relevant request parameters and the associated server response. Searching through logs to find error messages was generally not necessary because we chose to integrate Sentry for error tracking and alerting.

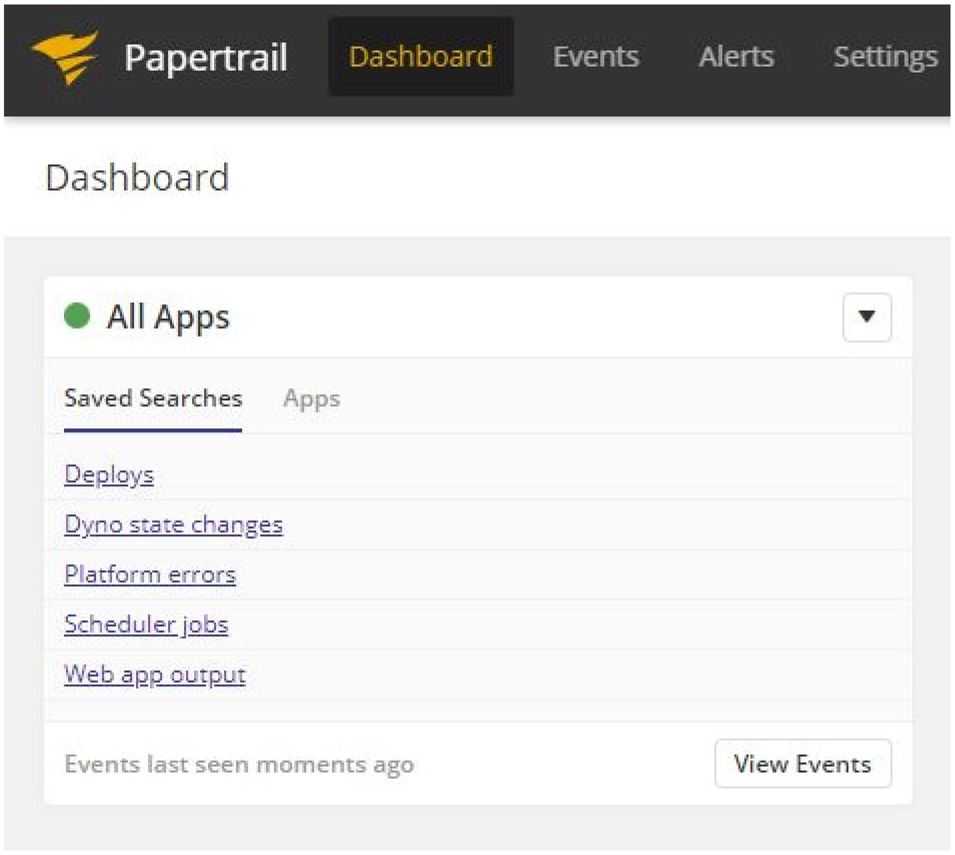
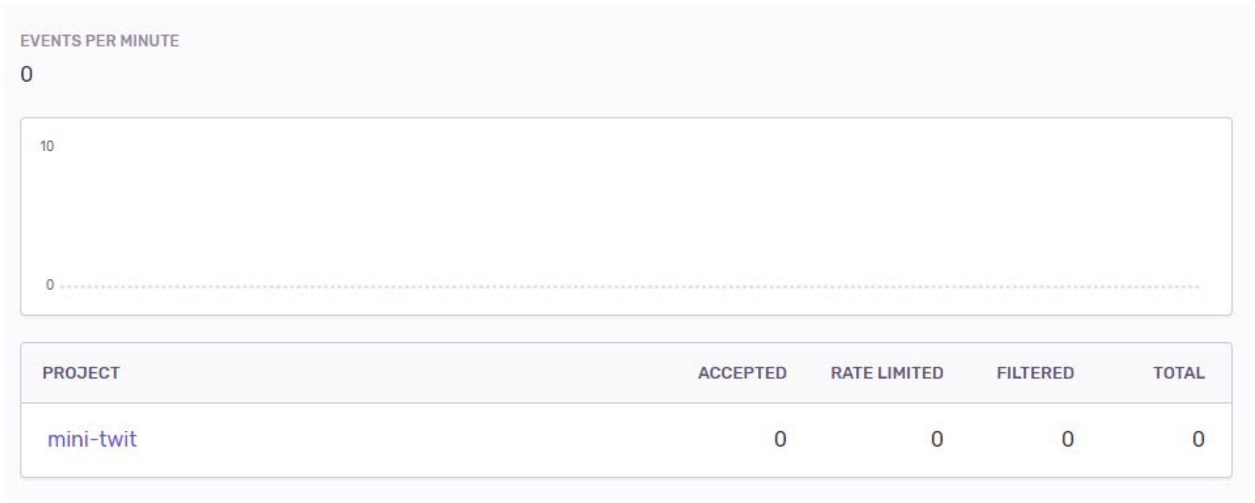


Figure 14: Papertrail setup with predefined search options

Another important tool in our DevOps process has been Sentry, an error tracker that analyzes the logs from our application and reports errors back to all subscribed developers. This makes incident detection simple since we are alerted via email whenever an error is detected. Together with the error message, Sentry also includes the stack trace, request parameters, event history, user agent data, and many more.

Using Sentry also gives us an overview of the state of our application based on the number of events that occured in the past two weeks. This is illustrated in Figure 15 where we can see that our app hasn't had any incident during this period.



PROJECT	ACCEPTED	RATE LIMITED	FILTERED	TOTAL
mini-twit	0	0	0	0

Figure 15: Sentry project statistics, no events for our MiniTwit

Security

Following the security report from our fellow colleagues, we realized we had deep problems with the response time of the application and had to update our SLA to more realistic numbers. We also had to implement patches that, amongst others, included pagination and database indexes to reduce the response time.

Our protection mechanism stood tall against injection attacks, cross-site scripting, and cookie jacking, and other common attacks. Unfortunately, our system was vulnerable to clickjacking and was briefly taken down by a DDOS attack. These vulnerabilities made us reconsider the time we invest in our system's security and are now high-priority open issues in our backlog.

On a more theoretical side, we have a low risk of being vulnerable to malicious code being deployed since any code that is pushed into the master branch and passes the tests is automatically deployed. In order to mitigate this risk, any changes made to the repository must be reviewed by at least one other team member - which, in a team of four, means that 50% of parties involved would have to be compromised. This is highly unlikely and extremely difficult to protect against, but shall the team increase we will implement additional security measures.

Regarding confidentiality - our users' data is protected against MITM attacks not only via traffic encryption via SSL and HTTPS, but we also took preventive measures and we are storing the hashed versions of the passwords in the database. However, our own source code is not confidential due to the requirements of this project and the fact that we believe in open-source programming.

Furthermore, we are constantly monitoring news regarding vulnerabilities in our NPM dependencies via Dependabot and NPM's builtin security checker. We also set up GitHub alerts when vulnerabilities are discovered in our dependencies.

In conclusion, we are aware that our security mechanism has overlooked some situations, and fixing this is a high priority for us.

Clustering

In order to enable clustering for our application, we integrated Throng.js to support Node's Clustering API. We used a middleware instead of connecting directly to the API because Throng.js is a small and well-supported library, and it drastically simplifies the process of creating multiple nodes for our application to run on.

For scaling and load balancing, we went with Heroku's builtin auto-scaling feature since it comes bundled with premium Dynos and can be easily controlled from the Dashboard. This is, however, a choice that we wouldn't have made if the cost wasn't an issue.

Heroku’s auto-scaling uses the desired average response time for the 95th percentile of requests as a control parameter for scaling the Dyno formation up or down. Because it uses response time, it relies on our application having a very small variance in response time. This is not adequate in our case since the JavaScript garbage collection events can temporarily spike the response time.

This is also not suitable for real-life scenarios where load and response time can fluctuate wildly, especially compared to the controlled environment of the simulator. A better option would have been to go with a Heroku third-party paid add-on that scales based on queuing time instead of overall response time.

Clustering also had a positive impact on performance, as discussed in the **Evolution** section.

Lessons Learned Perspective

Operation

The main difference between this project and other academic software development projects is the long-term thinking/mentality that is necessary. While those projects were required to run for the short period that they were evaluated, the DevOps project needs to be available for the entire duration of the course and theoretically much more. This means that every decision needs to be analyzed based on the long term impact it will have on the system and this means that we have to assume increased responsibility for the decisions we are making.

This is a complex and difficult task because every potential solution comes with its own trade-offs and it is the job of the Operation Engineer to determine which options should yield the best results during the lifespan of the project. The main aspects to take into consideration are:

- How will this change impact product maintenance
- How will it alter the development of new features
- How will performance/running costs be affected by this change
- How much rework will this require in the future (technical debt)

The main tradeoff we had to consider while building and maintaining this project is that some solutions might be easier to use and integrate but they are generally less robust in functionality. Another way of seeing this is that the former solutions are more lightweight and don't come with the extra functions that can slow you down. We generally agree with the second point of view and choose to integrate only lightweight dependencies in our application since they make maintenance easier, add less overhead when adding new features, and usually require fewer resources to run.

This is the kind of thought process that was used to decide between developing the refactored front-end with React, Angular, or Vue. We chose to go with React because it is a lightweight library, while Angular and Vue can be considered fully-fledged frameworks that come with a lot of redundant (at least for our use case) features. (Tracking Issue)

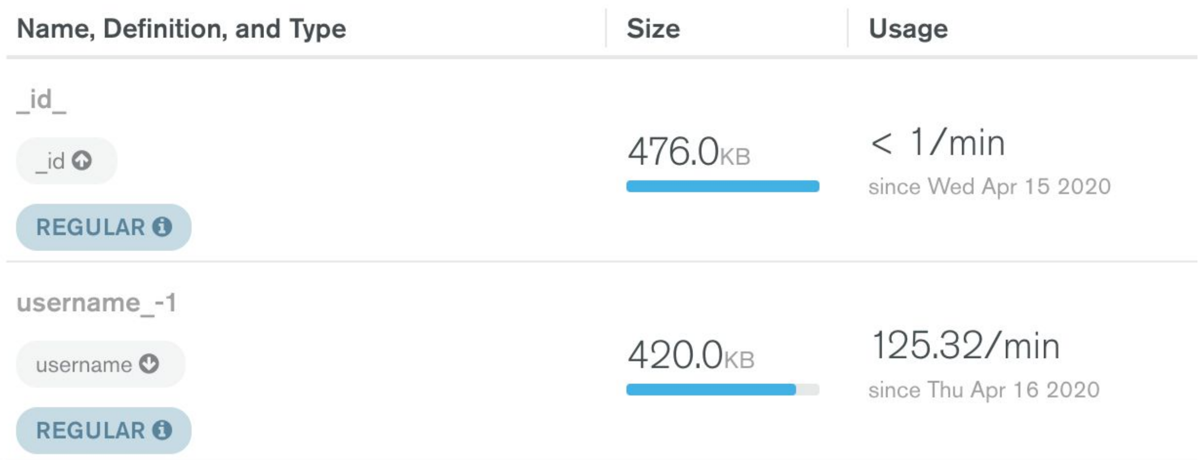
We understand that our philosophy is not perfect, and this can be exemplified in our choice of a database. We decided to choose MongoDB because it is the solution that adds the least overhead, in large part because it is designed to work with JavaScript in the JavaScript way: pure JSON and no schema. This convinced us to forgo PostgreSQL and its rock-solid performance which would have significantly increased the number of requests that we can serve in a minute. The difference in performance might not be that big in a real-life scenario since for this project we were using the free tier database on MongoAtlas which is severely limited compared to what industry workers expect from MongoDB. (Tracking Issue)

This philosophy can also be observed in our choice of dependencies. We believe that using a library to solve a problem might usually be an easy solution for the moment, but it brings with it the debt of having an additional dependency. Using a library that is no longer compatible with the project or updating it introduces breaking changes, can result in maintenance costs that could have been easily avoided. An example is the recent JavaScript crash from April 2020 when **over 3.4 million** projects were affected by a change in a one-line JavaScript library.

Evolution and Refactoring

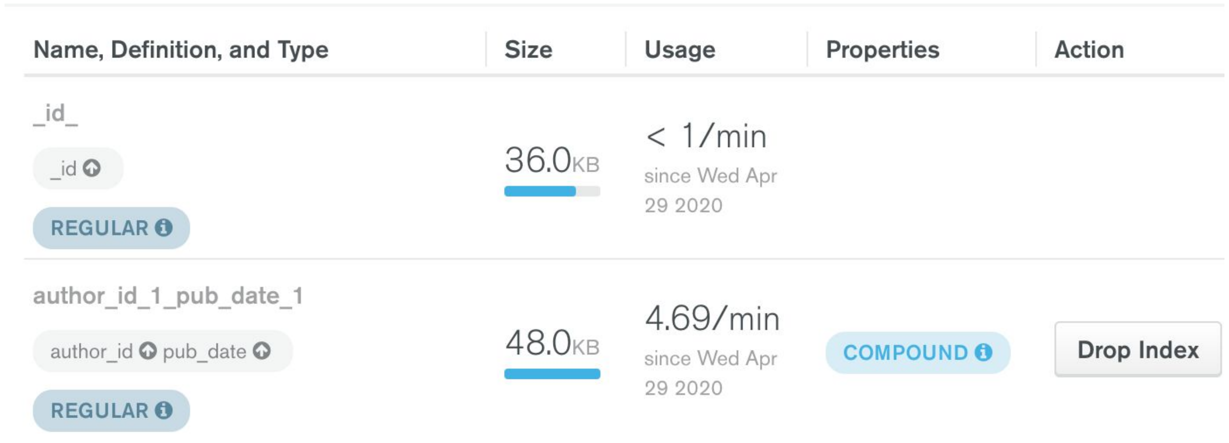
The latest sent and reported “latest_ids” graph made available to us reflected the fact that our simulator API is not as performant as it could be. In order to address these performance issues we implemented multiple optimization steps with regards to different parts of our system.

Firstly, we implemented a couple of indexes to our MongoDB database in order to improve the speed of data retrieval operations. For example, for the users table, we applied an index which optimizes the queries for username in a decreasing order - everytime the simulator makes a request, the user details are queried, so the index proves to be beneficial.



Name, Definition, and Type	Size	Usage
_id_ _id REGULAR	476.0KB	< 1/min since Wed Apr 15 2020
username_-1 username REGULAR	420.0KB	125.32/min since Thu Apr 16 2020

Figure 16.1: Users table index with over 100 hits per minute



Name, Definition, and Type	Size	Usage	Properties	Action
_id_ _id REGULAR	36.0KB	< 1/min since Wed Apr 29 2020		
author_id_1_pub_date_1 author_id pub_date REGULAR	48.0KB	4.69/min since Wed Apr 29 2020	COMPOUND	Drop Index

Figure 16.2: Messages table compound index

Additionally, another step taken into optimising the performance of the API is the implementation of pagination for both backends (simulator API and main API) responses. (Please refer to this commit as an example).

For example, a request to https://minitwit-simulator-api.herokuapp.com/msgs?p=5 would return the page 5 of all messages.

Figures 17.1 and 17.2 illustrate the dramatic impact that our changes have had upon the system performance. The grey label represents the commit which introduced pagination. The indexes on the database and the clustering have been applied at the same time as pagination.

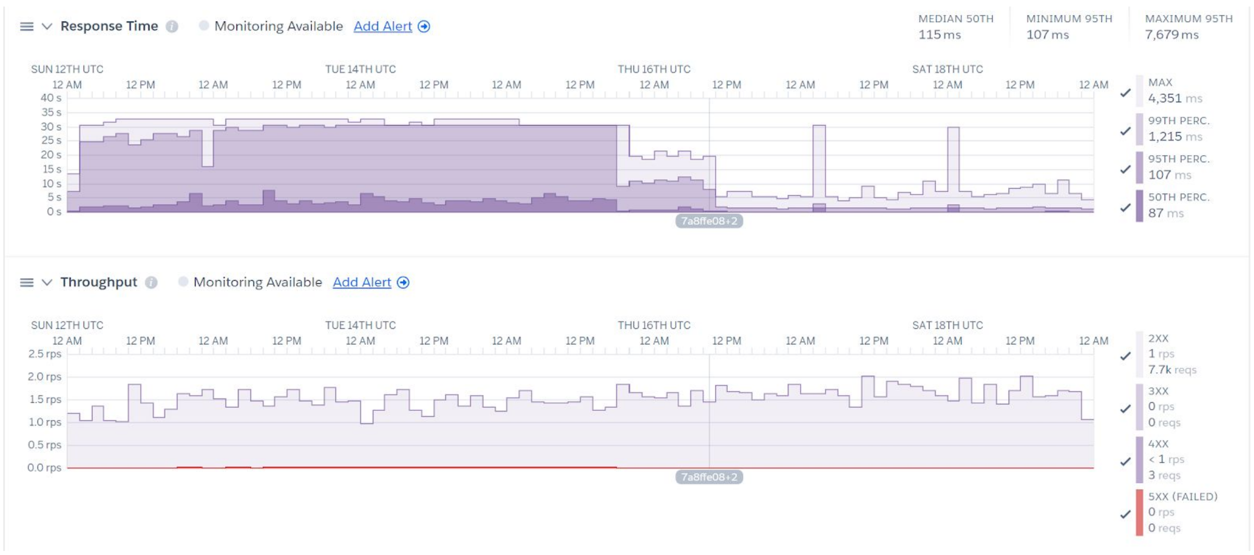


Figure 17.1: Heroku metrics dashboard after implementing optimizations

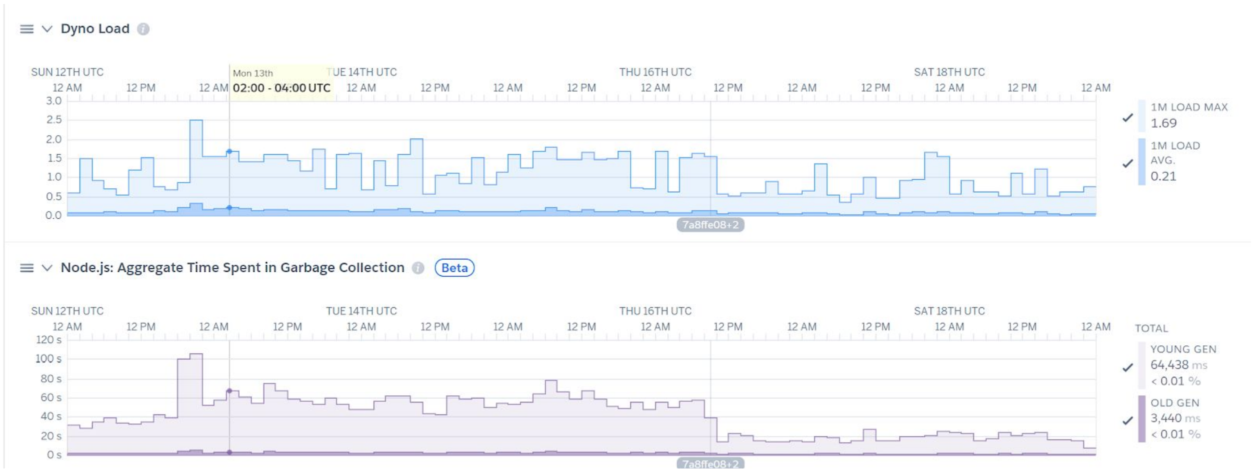


Figure 17.2: Heroku metrics dashboard after implementing optimizations

System performance also saw an increase after implementing clustering. This is because Node.js instances are limited in terms of memory and resource consumptions, so the non-clustered app was not even close to getting a high load on the system provided by Heroku.

As described in the Issue [Setup Heroku for high availability](), we tried multiple formations in terms of Dyno size and cluster node count, as well as the automatic scaling feature of Heroku and, based on our analysis, our systems seems to perform best while keeping a low cost on the Hobby Dyno with a cluster of 4 nodes. This keeps the system consumption at about 300mb of RAM while recording performance peaks over 180 requests per minute.

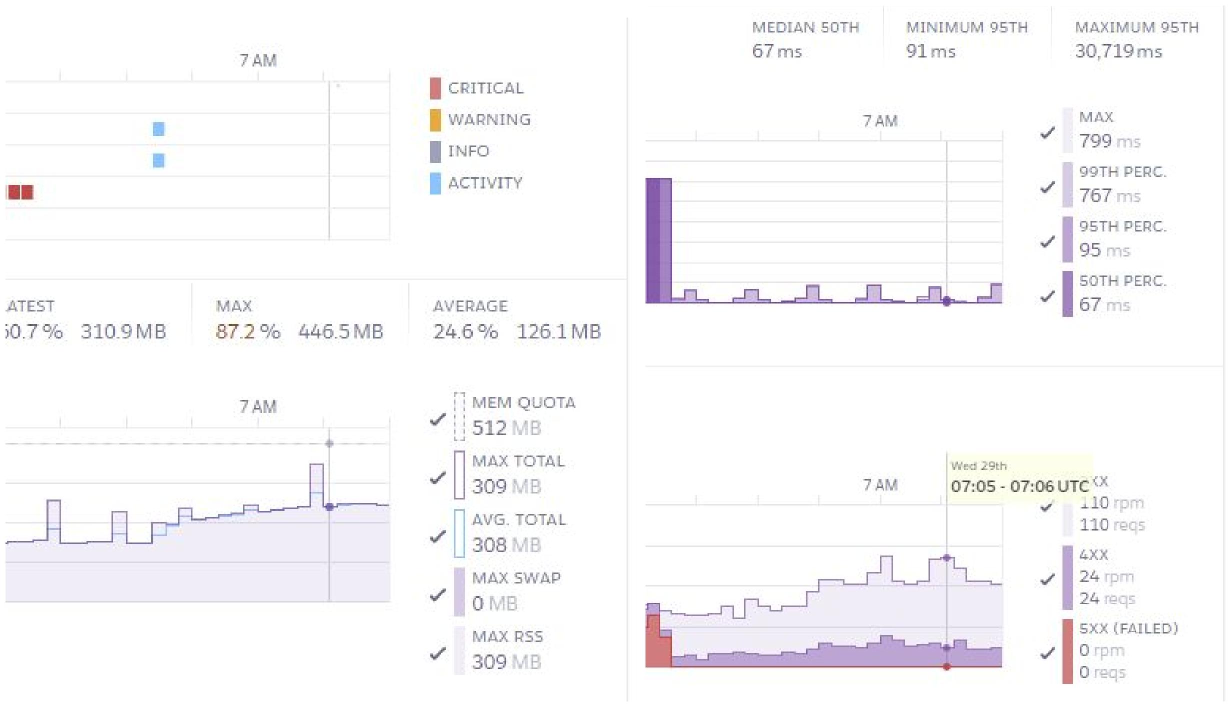


Figure 17.3: Performance enhancements after introducing clustering

Maintenance

The maintenance phase of the project was treated with extreme seriosity. We decided early on to integrate Sentry as an error tracker and this allowed us to easily detect failures and correct them in a timely manner.

We had two early-stage bugs related to the implementation of the simulator that were quickly addressed. This is visible on the line graph below as the absence of data points at the beginning of the plot. There are also 2 spikes observable on the line graph. The first one is due to a short Heroku outage in the European server region. The last one is due to the fact that our simulator was killed before the other groups.

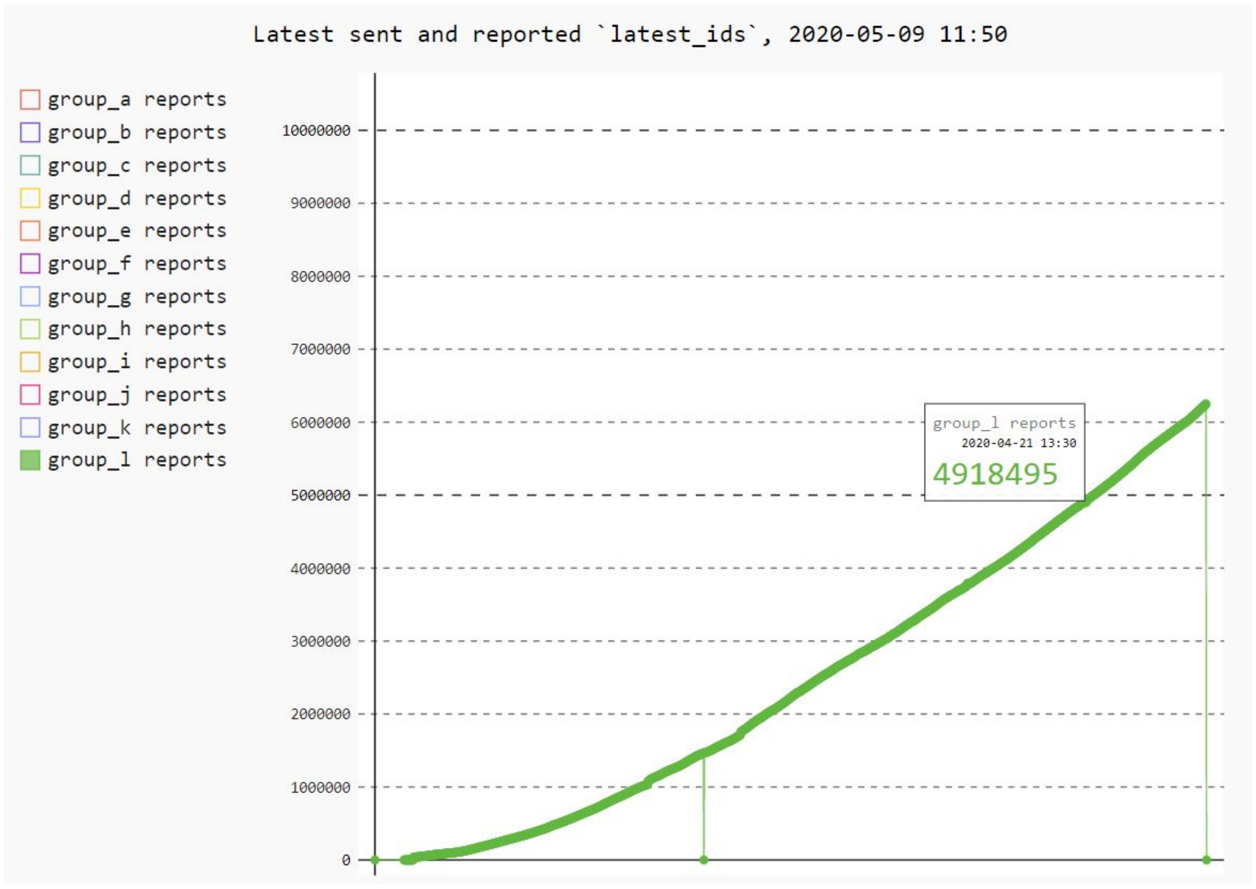


Figure 18: Latest graph for Group L

Observing the errors chart, we can see that, due to the early errors we missed on registering some important users. As a consequence, we were also throwing errors on following and messaging those users and the errors started to pile up. Those bugs were quickly patched and, after the simulator was restarted, our application stabilized and requests were no longer generating errors.

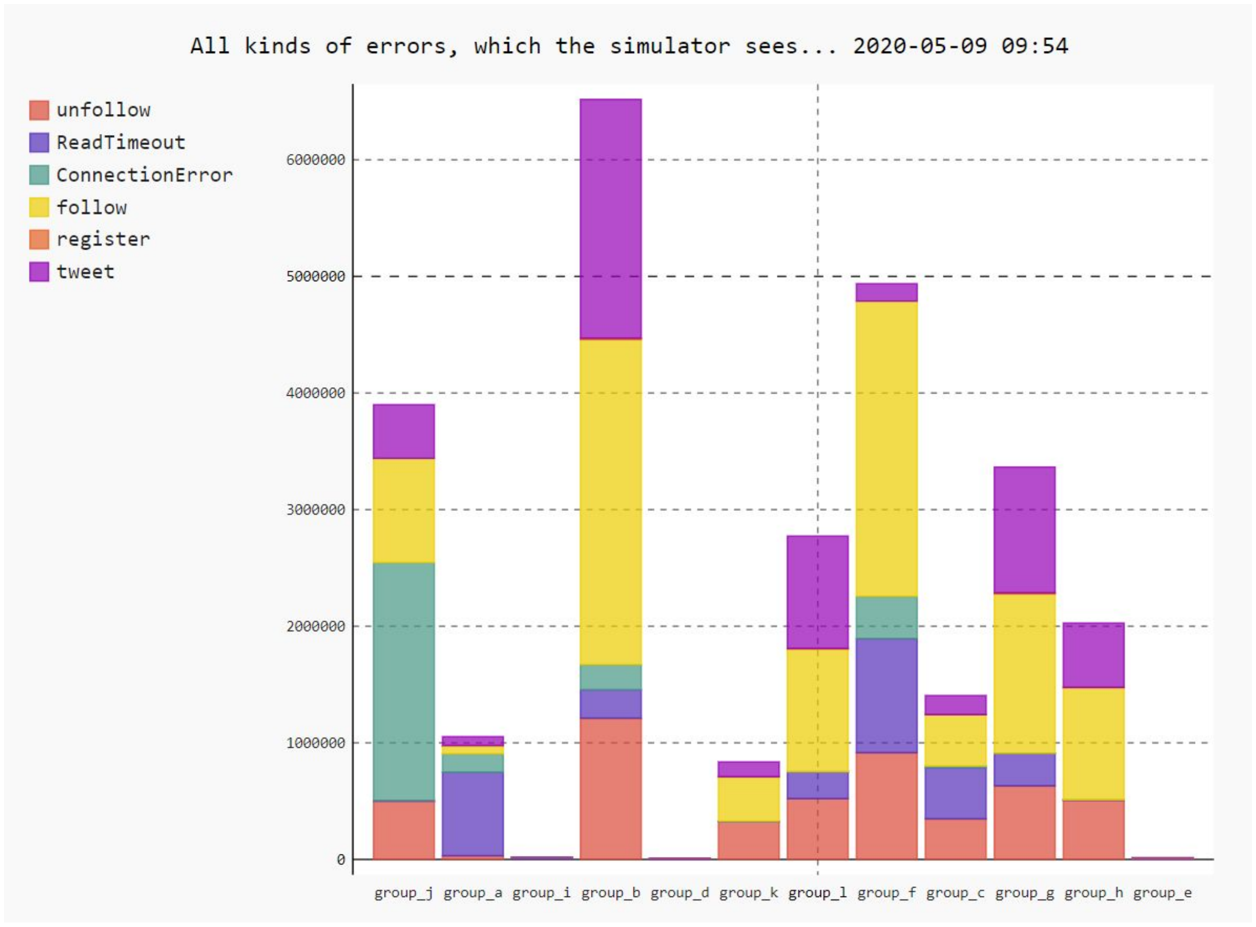


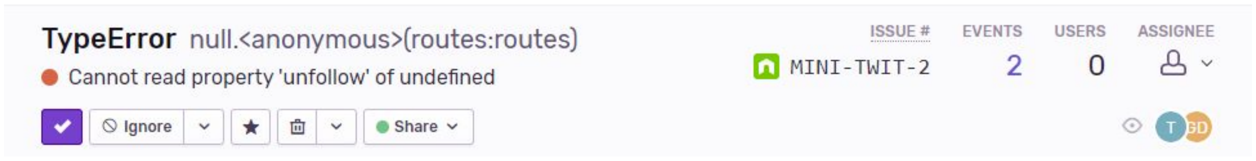
Figure 19: Group L errors reported by the simulator

Using Sentry we managed to identify a total of 5 incidents that affected our system during the course of the simulator. We use the ontology from **Table 1** to evaluate the severity of an incident in an objective manner. This allows us to more easily communicate about incidents and decide on the most appropriate course of action to resolve them.

Severity-0 (Minor)	Severity-1 (Most incidents)	Severity-2 (Urgent)	Severity-3 (911)
Issues that don't directly impact product usability.	Incidents that impact product usability but don't bring it to a halt or affect more obscure functions.	A feature is down for a large number of customers. Basic functionality is down for a small subset of customers.	Basic functionality is not usable for important customers.

Table 1: Severity classification of an incident

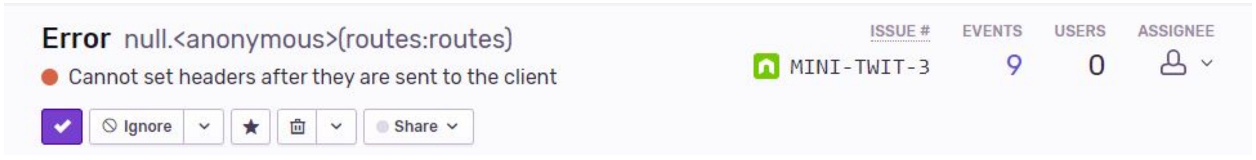
The first incident observed was considered Severity-1 since it was affecting users trying to unfollow other users. We know from our monitoring dashboard that this is not such a popular functionality of the application, but we still tried to patch is as fast as possible. This resulted in only two events being recorded for this incident.


TypeError null.<anonymous>(routes:routes)
Cannot read property 'unfollow' of undefined

ISSUE #	EVENTS	USERS	ASSIGNEE
MINI-TWIT-2	2	0	

Figure 20.1: First Sentry reported incident, isolated to the unfollow action

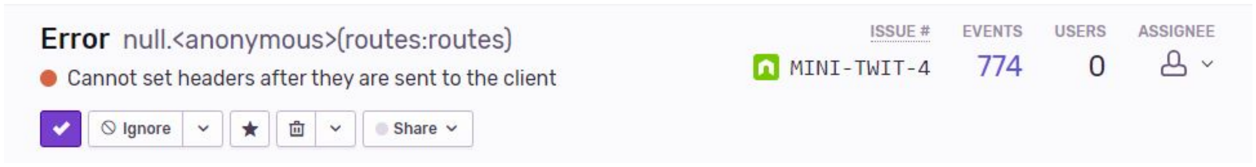
The second incident was cataloged as Severity-2 due to its potential to affect a large number of users if left unchecked. It was observed right after deployment and we were able to patch it immediately.


Error null.<anonymous>(routes:routes)
Cannot set headers after they are sent to the client

ISSUE #	EVENTS	USERS	ASSIGNEE
MINI-TWIT-3	9	0	

Figure 20.2: Pagination incident #1 observed after a deploy, resolved immediately

The fix that was deployed then ended up causing more harm than good. It masked the error rather than fix it and this is how we came to the third incident. It was still patched relatively quickly resulting in less than 30 minutes of downtime for the messages function.

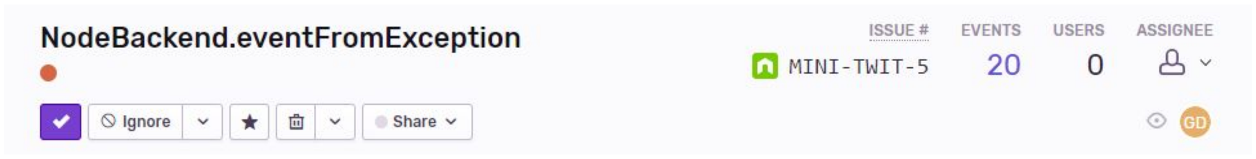

Error null.<anonymous>(routes:routes)

● Cannot set headers after they are sent to the client

ISSUE #	EVENTS	USERS	ASSIGNEE
MINI-TWIT-4	774	0	

Figure 20.3: Pagination incident #2, caused by an improper fix of the first one

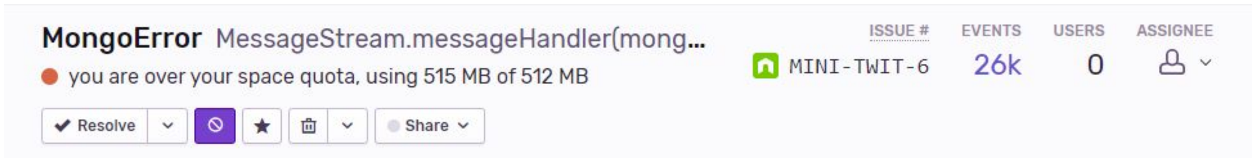
Incident number four was also observed after deployment and patched quickly. It was considered Severity-0 because even though the server returned an error in the end, users were seeing the page rendered correctly.


NodeBackend.eventFromException

ISSUE #	EVENTS	USERS	ASSIGNEE
MINI-TWIT-5	20	0	

Figure 20.4: Logical error that caused the unfollow function to never terminate

Our last observed error was an issue with our free tier MongoDB database. Only 512MB were available for database storage and our system run out of space right at the end of the simulator. This was considered Severity-3 but required a longer time to be fixed due to the intrusiveness of a database upgrade.


MongoError MessageStream.messageHandler(mong...

● you are over your space quota, using 515 MB of 512 MB

ISSUE #	EVENTS	USERS	ASSIGNEE
MINI-TWIT-6	26k	0	

Figure 20.5: Our MongoDB instance ran out of storage space

Conclusions

What began as a simple legacy application slowly turned into a complex system built with modern frameworks, tools and techniques that allow easier maintenance and faster deployment of improvements, new features, and fixes. Working hands-on with the DevOps principles resulted in a transparent, maintainable application, and many lessons learned, as detailed in this report.

Our developer team now understands not only what does “DevOps” truly mean, but also when and how it should be used. Furthermore, we are able to see that the time invested into building the right pipeline comes with great returns in the long-run.

While we are far from being experts in this field, we can consider ourselves ready to tackle larger scale, long-running projects - at least by choosing the right tools and practices for our problem and setting the right work culture in the team.