



Facultade de Informática

UNIVERSIDADE DA CORUÑA

TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN ENXEÑARÍA DO SOFTWARE

**Analysis, design and implementation of a
Node.js Web Application for personal link
management using a single link in social
networks.**

Student: Antón Valladares Poncela

Director: Adrián Carballal Mato

A Coruña, September of 2021.

To my mom, for putting up with me all these years.

Acknowledgements

I would like to thank all the staff of the faculty for all their time and the things I have learned from them, but specially, and in no particular order, Javier Paris, Adrian Carballal, Miguel Rodríguez Rubio, Antonino Santos and Cristina Costoya.

Abstract

The main objective of this project is the development of a Web Application, using a modern technology stack and a scalable system architecture, to provide Social Network users a place to store all their important links for their followers to access in a seemingly transparent manner from their user profiles.

This is in some cases necessary as some social networks, of which Instagram is the prime example, block the usage of links/URLs anywhere in their application, except for the usage of a single link as a user's Webpage inside their profile information.

The project will also focus on providing useful insights and analytics to the profile owner based on the metadata collected from visitors' clicks on the link collection.

Resumo

El objetivo principal de este proyecto es el desarrollo de una Aplicación Web, utilizando un *stack* de tecnologías moderno, y una arquitectura de sistema escalable, para proveer a los usuarios de las Redes Sociales de un lugar donde almacenar todos sus links importantes para que sus seguidores puedan acceder a ellos de una manera aparentemente transparente desde sus perfiles de usuario.

Esto en algunos casos es necesario, ya que algunas redes sociales, de las cuales Instagram es el mejor ejemplo, bloquean la utilización de links en cualquier sitio de la aplicación, excepto la utilización de un único link como la página web del usuario dentro de su perfil de usuario.

El proyecto también tiene como objetivo proporcionar al usuario de nuestra aplicación con datos y analíticas basadas en los metadatos recolectados de los clicks en la colección de links de la aplicación.

Keywords:

- FullStack Development
- REST API
- MERN Stack
- Analytics
- Data Processing
- React Web Application

Palabras clave:

- Desarrollo FullStack
- API REST
- MERN Stack
- Analíticas
- Data Processing
- Aplicación Web React

Contents

1	Introduction	1
2	Context	5
2.1	The problem	11
2.2	Example	14
3	Tools and Methodology	17
3.1	Methodology	19
3.2	Tools and development environment	22
4	Backend and Data Storage	27
4.1	Analysis	29
4.1.1	Data Storage and Persistence	29
4.1.2	Backend Architecture	30
4.1.3	Mongoose and Models	30
4.1.4	Service layer	31
4.1.5	Routers	31
4.1.6	Analytics and Task Queue	32
4.2	Design and Implementation	33
4.2.1	Data Storage and Persistency	35
4.2.2	Model Layer	37
4.2.3	Service Layer	39
4.2.4	Routing layer	41
4.2.5	Task Queue and Analytics	42
4.2.6	Other Libraries and Tools	44
5	Frontend	47
5.1	Analysis	47

5.2	Design and Mockups	49
5.2.1	Libraries and Tools	53
5.3	Implementation	53
5.4	Results	55
5.4.1	User profile from a visitor’s perspective	55
5.4.2	New user registration	56
5.4.3	User’s view of his own profile and analytics	58
5.4.4	Modifying profile information and uploading a new profile picture	61
5.4.5	Creating a new link	62
6	Testing and Continuous Integration	65
6.1	Testing Tools	65
6.2	Tests and Coverage	66
6.3	Continuous Integration	67
7	Conclusions	71
	List of Acronyms	75
	Glossary	77
	Bibliography	79

List of Figures

1.1	Instagram early UI layout	1
1.2	WSJ Article on the 1 Billion USD Operation	2
1.3	A Thomson Reuters building billboard welcomes Facebook, Inc. to the Nasdaq	3
2.1	2013 - 2018 Instagram Monthly Active Users	6
2.2	Instagram’s CEO Admits to copying Snapchat features to an employee	7
2.3	Instagram debuts its new video streaming feature.	8
2.4	An Instagram post by Victoria’s Secret containing buyable products	10
2.5	Linking to a Telegram profile from the website field of a user profile, is automatically detected and the action is rejected.	12
2.6	Setting that, if explicitly disabled by the streamer, allows other users to post links in that chat.	13
2.7	Music Services by Millions of Users. 2018-2019, © Statista.	14
3.1	Graph detailing the generic agile approach to Software Engineering.	17
3.2	Schematic showing a generic Kanban process.	18
3.3	An example Kanban Card in the In Progress section of our project.	20
3.4	The Automated Kanban Board inside our Github Repository.	21
3.5	Graphical User Interface of MongoDB Compass	25
4.1	General Diagram of the project’s system architecture	27
4.2	Backend Diagram of the project’s system architecture	28
4.3	Example router receiving a POST request to register a new user.	32
4.4	Main Structure of the project	33
4.5	Diagram of the resulting MongoDB collections.	35
4.6	Manual creation of the Clicks Time Series Collection using MongoDB Compass	36
4.7	All three collections once initialized.	37
4.8	Contents of the model folder	38

4.9	Mongoose Schema used to model the User entity	38
4.10	Pre-save middleware function for validating, salting and hashing passwords. A method for validating a passwords is also created.	39
4.11	Example of user service methods <i>getPublicUserData</i> and <i>getAllUserData</i> , with their respective JSDocs.	40
4.12	Diagram showing the Authentication and Authorization process using JWTs.	41
4.13	Implementation of the PassportJS strategy to check a user provided JWT. . . .	41
4.14	Link router accepting POST request to store metadata related to a click on a link	42
4.15	All the application's routers being setup in the main <i>app.js</i> file.	42
4.16	Analytics processing task being scheduled every 5 minutes on the main BullMQ Worker. This is done from the main ExpressJS <i>app.js</i> file.	43
4.17	Analytics pipeline for computing the total clicks by date for a given user. . . .	43
4.18	Analytics pipeline for computing the top countries generating clicks on a user's profile.	44
4.19	Combined and error logs stored in JSON format inside the project.	44
4.20	Setup of the Winston logging files and their individual configuration.	45
4.21	Private Dependabot security alert on one of our projects nested dependencies.	45
4.22	Pull Request from Dependabot to update to the newest version of BullMQ. . .	46
5.1	Mockup of the Mobile version of a user's profile from a visitor's perspective.	49
5.2	Mockup of the Mobile version of the login page.	50
5.3	Mockup of the Mobile version of the registration page.	51
5.4	Mockup of the Mobile version of a user profile, from that user's perspective. .	52
5.5	Main starting point for the React application. Contexts are initialized and Routes are specified. States are also used.	54
5.6	React <i>useState</i> and <i>useEffect</i> hooks being used to dynamically query the REST API or generate components.	54
5.7	Profile page from a visitor's perspective, on a mobile device.	55
5.8	Profile page from a visitor's perspective, on a desktop computer.	56
5.9	Registration screen on a mobile device.	56
5.10	Registration screen, on a desktop computer. Email validation failed because it didn't match the email Regular Expression	57
5.11	Private view of a user profile. Navigation dropdown menu opened.	58
5.12	Private view of a user profile on a desktop device. All private menu options visible on the navigation bar. First analytics graph visible.	59
5.13	Private view of a user profile. Daily clicks interactive graph visible.	59

LIST OF FIGURES

5.14	Private view of a user profile. Interactive graph showing the top countries that have visited that profile is being tapped on.	60
5.15	File picker popup when selecting the option to upload a new profile picture for a given user.	61
5.16	Desktop interface for creating a new link on your user profile.	62
5.17	Mobile interface for creating a new link on your user profile.	62
6.1	Jest test run output via console.	66
6.2	Jest testing syntax, as well as integration with testing extension Jest Runner. .	67
6.3	Jest coverage report output showing the number of times each line was covered.	67
6.4	node.js.yml Workflow configuration file to use the Github Actions pipeline. .	68
6.5	In the Workflows tab we can see the state and result of all finished runs of our Node.js workflow. We can also see how much time they took to run, and if they were successful or failed.	69
6.6	For every run, we can see the result of each of the phases of the pipeline, so as to analyze where the pipeline failed if it did.	69
6.7	Workflows from master branch being run, on a pull request that targets master as destination branch.	70

List of Tables

Introduction

SOCIAL NETWORKS have been steadily and exponentially rising in popularity since the first appearances of what we can consider the predecessors of modern-day social networks, with the launch of Myspace in 2003.

Nowadays, social networks are a huge, continually growing, intricate networks of users, content and data, that span way more features and functions than their original purpose. The prime example of this is Instagram.

Instagram started as an American Photo Sharing social network, which originally went by the name Burbn, and which was at the beginning very similar to Foursquare. After its founders, Kevin Systrom and Mike Krieger noticed that this was not a good idea, they decided to pull their application together around the idea of a photo sharing platform. They renamed the application to Instagram, which is a mixture of *Instant Camera* and *Telegram*.

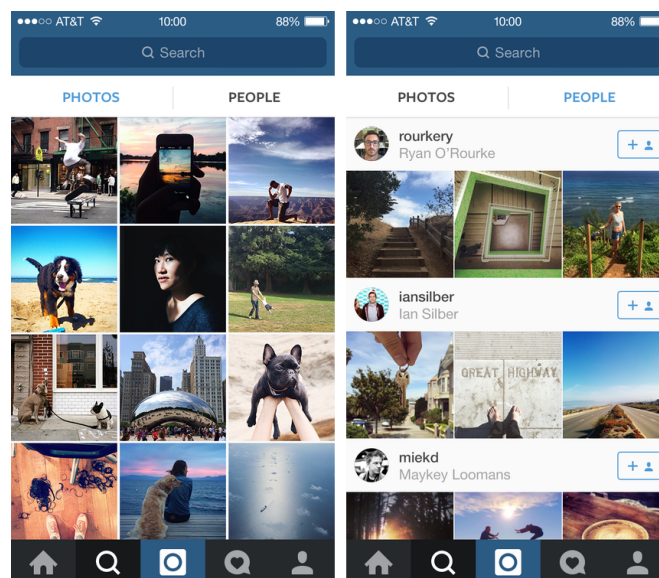


Figure 1.1: Instagram early UI layout

The next few years, Instagram (as well as other social networks which were growing at that time, such as Facebook and Twitter), closed investment funding rounds one after another, each of them for exponentially higher amounts of money. In 2012 for example, a series A funding round was closed from a group of important investors, one of which was none other than Jack Dorsey himself, the then Co-founder of Twitter which is now also the CEO of payment processing platform *Square*.

On April of 2012, Instagram delivered an adaptation of its application for Android Phones, which was downloaded several million times in just the span of the first day.

The turning point for Instagram was on the 9th of April of 2012, just six days later after their first Android release, when *Facebook, Inc.* bought the social network for more than 1 Billion USD (300 Million USD in cash, and 23 million shares of Facebook stock).

Insta-Rich: \$1 Billion for Instagram

Facebook Inks Its Biggest Deal Ever; Neutralizes Threat from a Hot Photo Start-Up



Instagram founders Mike Krieger, left, and Kevin Systrom.

BLOOMBERG NEWS

Figure 1.2: WSJ Article on the 1 Billion USD Operation

Very shortly after the operation was completed, and *Facebook, Inc.* had become the complete owner of Instagram, Facebook went public on the Nasdaq via an Initial Public Offering (IPO), which was historical because it was one of the biggest in internet history, having an initial valuation of over 104 Billion USD.



Figure 1.3: A Thomson Reuters building billboard welcomes Facebook, Inc. to the Nasdaq

It's more than obvious by the initial valuation, that at the time of Instagram's buyout, the social network wasn't one of Facebook's biggest assets, accounting for just 1% of the Internet Giant's value. Of course, later the public would end up realizing that the buyout of Instagram was in fact due to Facebook's fear of having a competitor to its main platform.

Chapter 2

Context

AFTER the acquisition by Facebook, Instagram continued growing extremely fast. In 2015, just 3 years after the buyout, Instagram's Monthly Active Users, had skyrocketed from just over 20 million, to over 400 million users.

This, of course, was in part due to the constant rollout of new features to the platform, which converted it from just a simple and minimalistic photo sharing social network, to a very complex set of features that intertwined between themselves in a way such that the platform had transformed into a kind of all-in-one social network, much like Facebook, but more oriented to younger audiences and portable devices.

Also in 2015, Instagram received its first redesign and overhaul of the web interface, that until that time had not received nearly as much attention as the iOS and Android applications, since the vast majority of the traffic Instagram received came from mobile devices, mainly iOS and Android devices, using their own *apps*. The web interface moved from a more bloated and round style, to a more flat and minimalistic aesthetic, better matching the overall brand scheme that Facebook had been pursuing for a few years already.

In 2016, Instagram once again received a visual revamp. This time, the application had amassed a Monthly Active User count of more than 500 Million users, and the mobile applications were redesigned to the visual black and white flat UI that we're used to nowadays. This once again was a move to a less skeuomorphic and a more flat, colorful and modern UI.

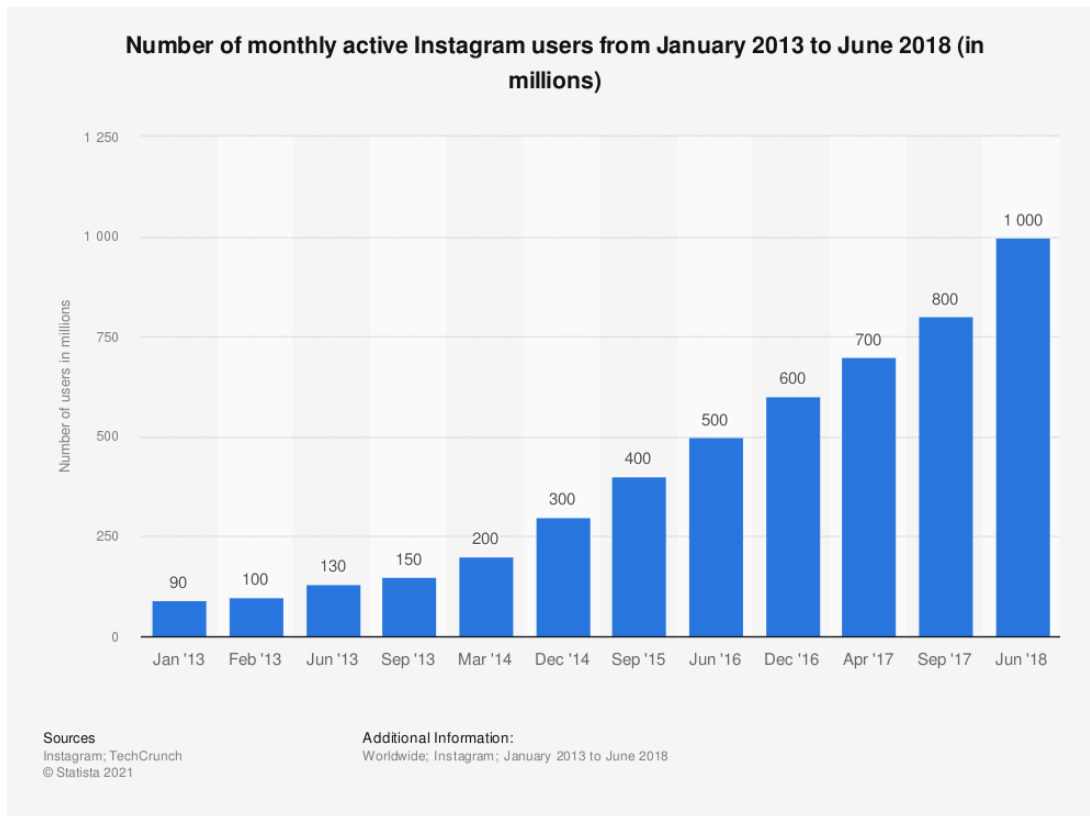


Figure 2.1: 2013 - 2018 Instagram Monthly Active Users

Besides from all the visual and aesthetic design changes that the Instagram team introduced since Facebook's acquisition of the company, the Photo Sharing platform also introduced feature changes and new functions that steadily changed the application from the original idea of a Photo Sharing application, to something more complex and featureful. Some examples of these new feature additions are:

- In 2016, the platform introduced a feature to allow users to like comments. However, authors of a post could deactivate that feature in a post by post manner.
- In that same year, they also released an official Windows 10 Desktop Application, which was a historic demand from Microsoft.
- The release of the *Direct* feature, to allow sending photos and messages to individual users. This was mostly released to compete with Snapchat, which was exponentially growing worldwide at that time, and was the main competitor to Instagram.
- Launch of *Stories*, a product that mimics most of the features that Snapchat was foundationally based on since its creation.

This last addition to the Instagram feature list was quite controversial, since it was obviously clear that this was a direct attack to Snapchat, since the latter had refused several offers from the tech giant to be bought out. This *Stories* feature blatantly copied most of Snapchat's core features. Users were allowed to post ephemeral 24 hour photos or short videos that disappeared completely after that time.

Still, everyone in our interview room knew there was no avoiding the Snapchat question, so I just put it bluntly. "Let's talk about the big thing. Snapchat pioneered a lot of this format. Whole parts of the concept, the implementation, down to the details..."

"Totally," Systrom interrupted me. "They deserve all the credit."

I was flabbergasted.

Figure 2.2: Instagram's CEO Admits to copying Snapchat features to an employee

Shortly after the release of this copied feature, Instagram's CEO Kevin Systrom appeared in a news article on *TechCrunch* where he admitted to having outright copied the feature from Snapchat. It wasn't that the feature was somehow similar to that of Snapchats', it was exactly the same, even the User Interface was exactly the same.

The length of the videos the user could upload was at first also 14 seconds, and the options given to users when editing the Stories prior to uploading them was exactly the same: some drawing, stickers, and light filters.

This of course was not the first time Facebook or one of its companies had been accused of copying other smaller *startups* or bullying smaller companies into being bought out, however it was the first time that the person responsible for doing so had openly admitted to doing it. Previously, Facebook had been accused of copying products like *Slingshot* or copying whole startups like TimeHop that delivered the functionality that nowadays is known as *On This Day* in Facebook Feeds. It also started using *Hashtags*, which were first used by Twitter, on both Instagram and Facebook.

However, Facebook has also been copied by other social networks with concepts that we are very used to these days. The best example of this is the *Feed*. It was first used by Facebook as a way to deliver content to users on their platform in a way that was not chronologically determined, but rather controlled by an underlying algorithm that would try to predict and increase the user's engagement with the content being delivered. The purpose of this was no other than driving up engagement with the platform, and increasing the time users spent browsing this content.

This *Feed* concept was later embraced by other online social networks that to this day still use it, like Twitter and LinkedIn and has become a defacto standard for content delivery in social networks.

In the coming years, Instagram would continue to add new features to the platform, and to embrace almost every kind of social network type in existence. From the original idea of a photo sharing network, to short videos and *stories*, to longer videos in the main Feed.

Shortly after, in November 2016, Instagram launched a feature allowing users to create a *story* which was actually a live video.

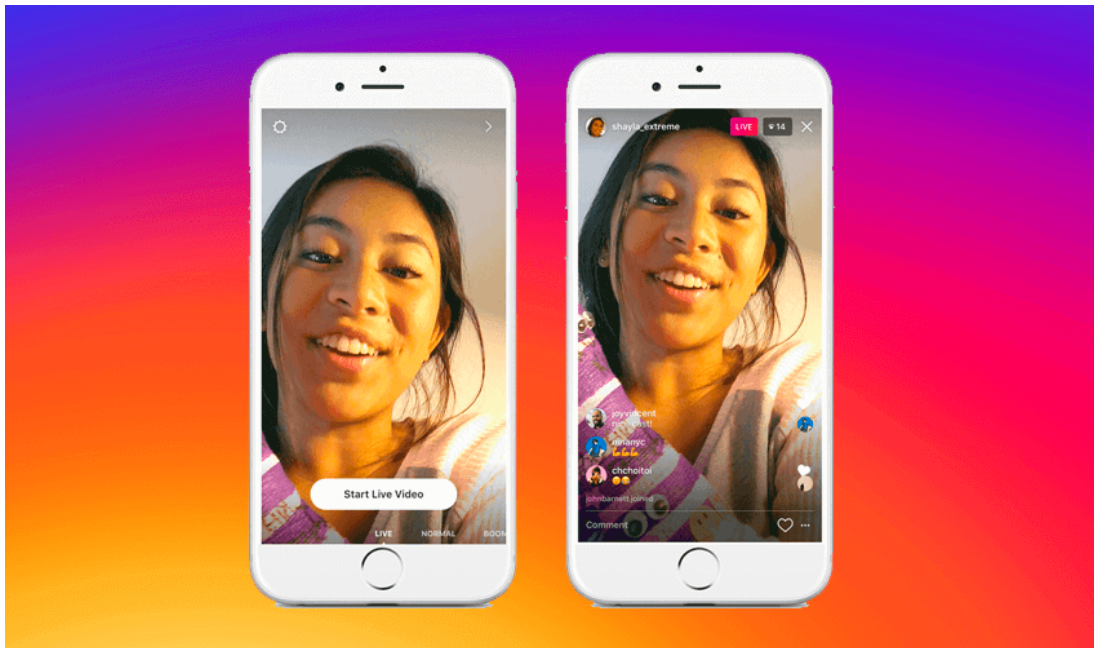


Figure 2.3: Instagram debuts its new video streaming feature.

This was Instagram's answer to video streaming platforms, such as *Twitch.tv*, where people could record themselves doing mundane things such as taking a walk, talking with friends or cooking, all while their followers could send chat messages that were displayed in the livestream's chat window. In this way, interaction between the user and their followers, or between a follower of a given user, and followers of that same user, was way easier than before, and it was all done live.

Not long after implementing the live stories feature intended to rival livestreaming services, Instagram reached a total Monthly Active Userbase of more than 600 Million users, and launched a feature to allow sharing a collection of photographs, that could be displayed in a user's profile and the content feed as a collection of scrollable photos, allowing users to post multiple photos or videos in a single post, allowing for a slideshow display of pictures in a single feed entry.

More small features were soon added, such as allowing users to respond to other user's comments in a more forum-like manner, instead of the actual unstructured way which consisted in mentioning the user's *@handle* to respond to a comment. Liking comments was also

introduced, allowing users and the original poster himself to like a comment on his post.

In September of 2017, a short videos social network called *TikTok* was introduced. TikTok's main zone of influence was Mainland China, however by allowing users to download posted videos with a TikTok watermark, and send them over other platforms, TikTok reached an incredible userbase in no time, and was soon the Most Downloaded App both in the Apple Appstore, and on Google Play. This evidently was a threat to Instagram's business model, which was also based almost in its entirety on mobile devices.

But before launching a TikTok competitor, Instagram launched IGTV (*Instagram TV*). This new feature inside the Instagram ecosystem was developed with the single purpose of rivaling YouTube. Previously, users on Instagram could only share short videos of at most 30 seconds, which didn't allow for a large array of content that needed more time to be uploaded to the platform. With IGTV, Instagram users could now upload videos of up to 10 minutes, with bigger users being able to upload videos of up to 1 hour of length.

A dedicated button in the main Instagram UI, routed users to the IGTV part of their app, where they could access a feed of long videos posted by the profiles they previously followed.

On 2018, Instagram added again a new feature. This time, the purpose of this feature was allowing users to offer products to sell. Users, for example, could tag clothes or accessories in their pictures with a price, and people could tap on them and access an interface where Instagram let them add their payment details and buy products directly from inside the application. This was one of Instagram's most controversial features, since up to that day, the platform had no way of letting users directly monetize their content in any way.

Businesses started promoting their products on Instagram and paying other high ranking users (more commonly referred to as *Influencers*) for posting pictures with their products and tagging them to let other users directly buy them from within the app.

The last of the most notable features launched by Instagram to rival other products or platforms from other companies, was Instagram Reels. This service was launched in 2020 in the midst of TikTok's exponential growth in Europe and America. Reels was, again, just a copy of the rivaling service. Reels allowed Instagram users to post short videos, which were always in vertical perspective, and could include a variety of effects, from music to Artificial Intelligence driven face filters.

2.1 The problem

As of right now, and as a summary, Instagram has the following functionalities:

- Allows users to post pictures and short videos up to 50 seconds.
- Allows users to share temporary 24 hour photos and short videos. (Developed to rival Snapchat)
- Allows users to post long videos, up to 1 hour in length in some cases (Developed to rival Youtube)
- Allows users to tag products in their pictures, videos and stories, in order to sell them through the platform. (Developed in response to the rapid growth of online shopping)
- Allows users to post short music videos called Reels. (Developed to rival TikTok)
- Allows a user, or a group of users to stream live video to their audience, and interact with them by chat. (Developed to rival Twitch.tv)

This set of functionalities, are likely to keep growing every single year, as Instagram adds more features to rival smaller social networks that grow rapidly.

The problem itself is very simple. Instagram doesn't allow its users to post links in any of the aforementioned places.

A normal user **can't**:

- Paste a link as a photo or video description.
- Paste a link in the description of a Reels video.
- Paste a link anywhere in their story's text or tags.
- Paste a link anywhere in their profile info, except in the website field.
- Paste a link anywhere in a livestream.
- Send a link to a group of people using Direct Messages.
- Post a link in any kind of chat, including livestreams.

This should start making clear what the problem here is, however, we will see that there are some exceptions to this rule, mainly:

- **A user can only post a single link, in the website field of his profile**, however:
 - This link cannot be a link to another social network or chat application (except Facebook).
 - This link has to be moderated by Instagram, and can be removed from a user's profile at any given moment.
- Businesses and big brands may post 24 hour lasting links in their Instagram Stories, and they may link to their website for the payment of products instead of using Instagram Shopping for payment processing.

One example of this tight grip on content linking to other platforms outside of the Facebook ecosystem:

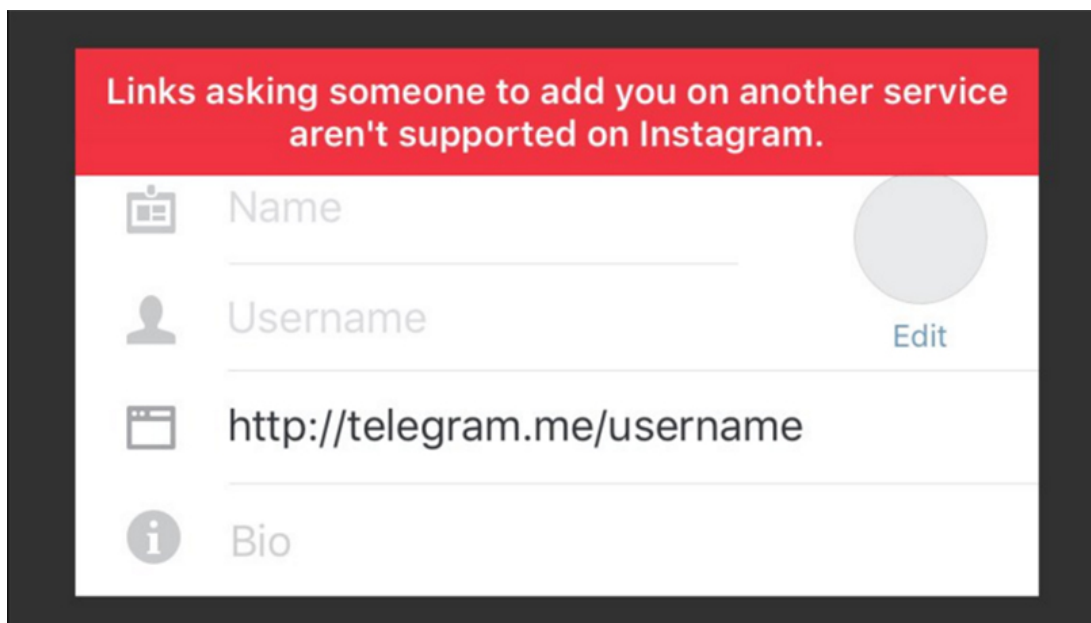


Figure 2.5: Linking to a Telegram profile from the website field of a user profile, is automatically detected and the action is rejected.

Out of all the social networks available and that are widely used, **Instagram is clearly the best example of this practice** of completely censoring the usage of links throughout the application, and even filtering which links can be used in the only place in the application where you can actually paste a link.

However, this practice of making it hard for the user to leave the application by clicking on an external link is not exclusive of Instagram.

Another example of this behaviour, even if not as excessive as Instagram, is *Twitch.tv*, the video streaming service that was bought out by Amazon in 2015 for almost a million USD.

In this case, this behaviour is a bit more tame. By default, any streamer in the platform that has not changed his user settings, will not be able to have visitors of his stream post any kind of links in the chat. This however, can be disabled to allow users to post links to that chat so other users, and the streamer himself, can click:



Figure 2.6: Setting that, if explicitly disabled by the streamer, allows other users to post links in that chat.

But why don't these social networks allow the free usage of links? This question has simple answers:

- These platforms want you to stay inside the platform for as long as possible.
- They want you to interact with other users inside the platform.
- If you click on an external link and leave the platform, this just results in lost ad revenue for the platform. If you're not on the platform, they can't show you ads.
- The more time you stay engaged inside the platform, the more time the platform has to show you ads, thus increasing the company's revenue.
- If people could post links everywhere, engagement metrics would plummet due to users being able to interact with content in ways that would end up with that given user leaving the platform due to an interaction with an external link.

From a business oriented point of view, it makes a lot of sense for social networks to make it harder and harder for users to leave the platform through interactions with external links. If they prohibit the use of these external links, this is no longer a problem, and user engagement is highly increased.

2.2 Example

To better illustrate the problem caused by these link-censoring guidelines in some social networks, we will follow the example of a famous singer throughout this document.

In this example, we will study the needs of a singer or music artist as a user of these social networks with regards to the usage of hyperlinks. The singer might need to link to a big number of places outside that given platform for a variety of different reasons, let's see some examples:

- A Merchandise website where the singer sells T-Shirts and other merchandise.
- A website for the concerts and appearances calendar. In this case, this could be the artist's own website.
- To allow users to listen to his music, the artist will need to link to a variety of music streaming services. Some examples of which are:
 - Apple Music (Mostly used by iOS users)
 - Spotify
 - Youtube Music
 - Amazon Music
 - SoundCloud

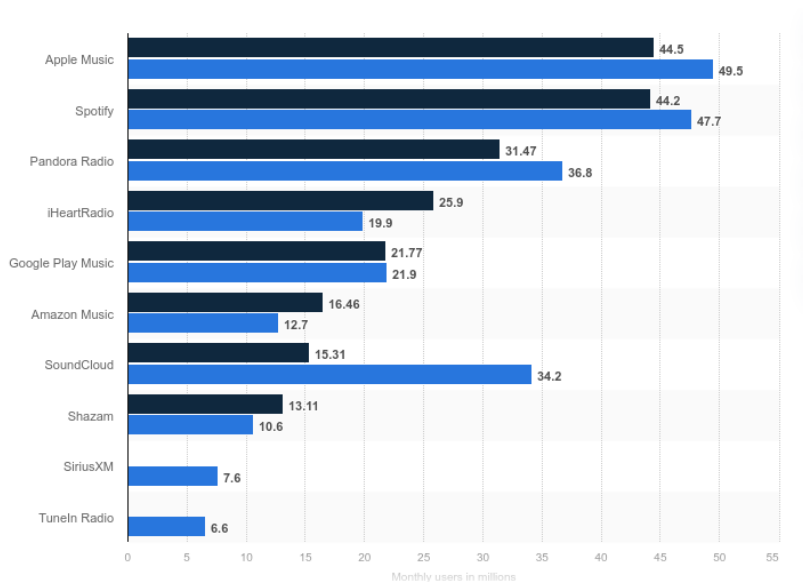


Figure 2.7: Music Services by Millions of Users. 2018-2019, © Statista.

Furthermore, a singer will also want to link to his other social media accounts, to increase his engagement on other platforms. This adds even more links to what the user would like to share on his Instagram account, but the problem arises again. He can only use a single link, on his profile information, on the website field.

This project focuses precisely on providing a solution to this problem. The main objective is to create a Web Application that let its users create an account, and store every single link they need to link to, in such a way that when a visitor visits a profile, he will be able to see basic information of the profile he's visiting, and a list of links that that user has stored in his collection.

A visitor is then able to click in any of these links, opening them in a new tab in their browser.

At the same time, our platform will collect certain metadata from each of the visitor's clicks on links, so as to provide specific insights and analytics for the owner of that profile (in our example, the music artist) on how his visitors are using his links, and other demographic data of his userbase.

The Web Application must be fast, responsive and scale correctly on mobile devices, since the vast majority of traffic Instagram receives, comes from mobile devices, thus the majority of visitors accessing a user profile on our platform, would be using mobile web browsers.

Tools and Methodology

AGILE methodologies are a more result-focused approach to software engineering, that aims to combat the ever and rapidly changing requirements in the field of software development. The basis of these types of methodologies center around planning the development process in such a way that it is adaptive, self organized, and that work is divided into small pieces so as to have short delivery times on single features.

These kinds of methodologies also aim for continuous improvement in quality. Some examples of these methodologies are:

- Scrum
- eXtreme Programming
- Kanban
- Feature Driven Development (FDD)

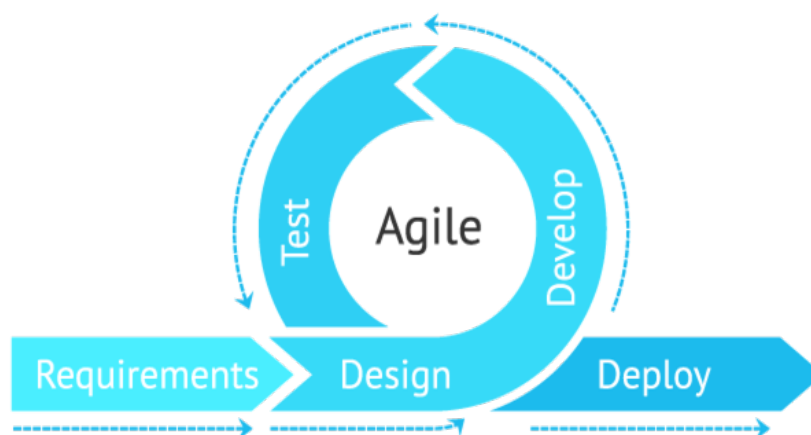


Figure 3.1: Graph detailing the generic agile approach to Software Engineering.

Agile methodologies in Software Engineering have skyrocketed in adoption. This is in part due to the old *Waterfall Model* and *Iterative Model* being too static to allow for the fast changes in requirements and features that the field is used to nowadays. In an environment where the featureset of a Software project, and the functional and non-functional requirements change a big number of times during the engineering process, a flexible methodology that adapts well to these conditions must be used.

Agile methodologies are sometimes criticised because they base themselves on walking a very fine line between having just about enough planning on a Software Engineering project, and not having enough.

These methodologies however provide less risk in some aspects, as they do not force the engineering team to abide by an extremely strict plan that they invested months into analyzing and developing. This is achieved in part by involving the development team in almost every single aspect of the Software Engineering process, where as in a more traditional methodology, the tasks that each team member would be working on would be very limited in comparison.

The best and most widely adopted type of Agile methodology inside the Software Engineering space, is Scrum. However, for Scrum to work correctly, there are several roles that would have to be taken on by some of the team members, as well as having daily meetings.

This is why for this project, the Kanban methodology was chosen for the development process.

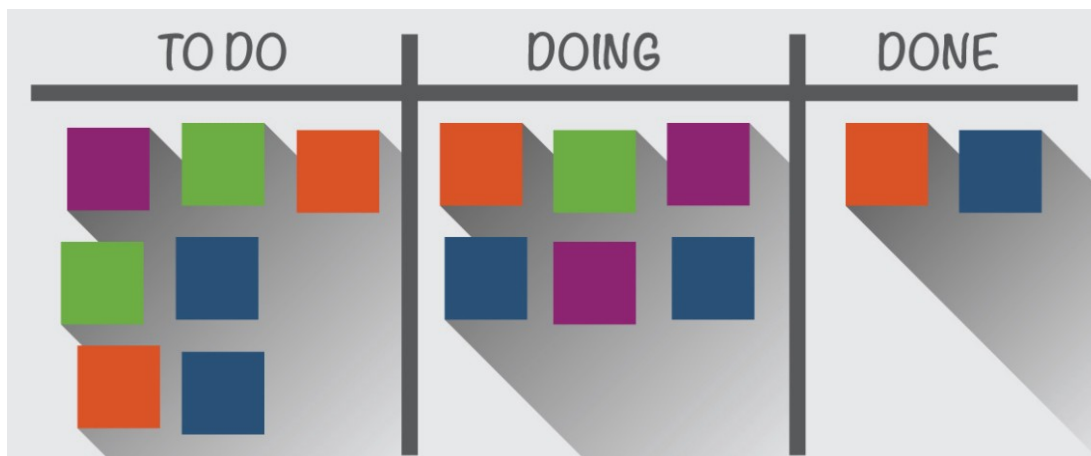


Figure 3.2: Schematic showing a generic Kanban process.

3.1 Methodology

The Kanban methodology allows development teams to implement agile software development.

Kanban requires all team members to maintain a constant stream of communication, and calls for full transparency on what work and what progress is being made by each of the team members.

Each of the tasks of a Kanban project is represented as a card or note in a Kanban board. This allows every member of the team to choose on what task to work on, at any given moment, and to update the progress on that task in a way that is fully transparent to the rest of the development team.

Nowadays, the Kanban methodology is very prominent among software engineering teams that want to have a certain degree of flexibility over a project. However, this methodology was born more than 60 years ago. It dates back to when Toyota, in the 1940s, tried to optimize its engineering process so as to match the demand that their cars had, to the amount of cars they were actually producing, so as to not waste money in excess parts during the construction of new car models.

In order for the factory workers to communicate the amount of a given car part they needed, workers would pass around a card, a *Kanban*, to the supplier or the warehouse, when a given part or item fell under a certain threshold of supply. Thanks to this, the supplier or warehouse worker would know exactly how many parts of that item were needed, so as to not send parts in excess, and reduce the amount of parts that were manufactured without actually being needed.

These kind of principles are known in the software world as JIT or *Just In Time*. The basis is meeting the demand of something, exactly when it is needed, and exactly in the amount needed, but not exceeding that demand and supplying more than what is needed. This same concept applies to software engineering.

The Kanban board helps members of the team visualize the actual state of the project. Usually, Kanban boards in software projects include, at least, the following categories:

- **To Do:** This part of the board stores all the tasks that are yet to be taken on by any of the team members and that are yet to be started.
- **In Progress:** This category includes tasks which are already being worked on by one or more of the team members.
- **Review:** This is one of the examples of categories that were explicitly added when adapting the Kanban model for use in the Software Engineering industry. This category

stores the tasks that have been already completed by one of the team members, but are still pending a peer review from one of the other team members.

- **Done:** This category stores tasks which have already completed, and have successfully passed the review process.

Each of these categories will contain Kanban Cards. These cards contain information on the task at hand. For example, a card will contain the profile of the team member that is working on it, a code to precisely identify that task, and the actual requirements of the task to be completed. Cards on a Kanban board will a lot of times also include a variety of technical details such as screenshots, schematics, diagrams or any other technical information necessary to correctly finish that task.

One of the main goals of tracking the development of a project using a visual board, is helping the team members visually understand how a certain part of the work is progressing.

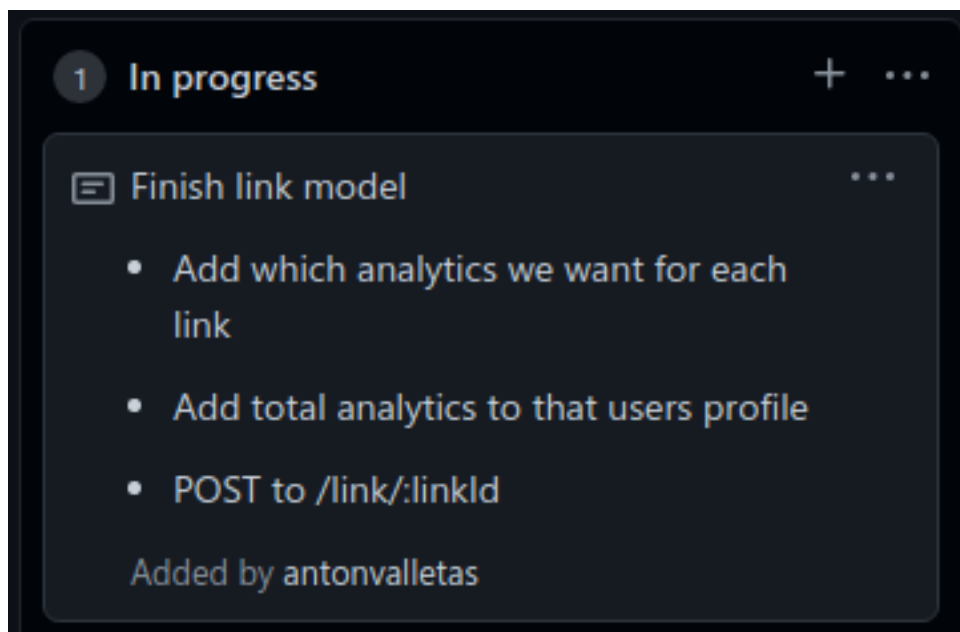


Figure 3.3: An example Kanban Card in the In Progress section of our project.

In order to follow the Kanban methodology in this project, Github Project Boards was used. This is an integrated solution provided by the Configuration Management Solution provider, which in this case is Github, so as to unify configuration management and source control, with a way to track the progress of a project.

Unifying the source control of the project, with the progress tracking has a series of advantages that facilitate working with the project in several ways.

For example, the developer can automate the creation and management of Kanban cards

based on actions performed in the source control system. One of these actions, are pull requests. These can be modeled as *Pending review* Kanban cards, and stored in that same column. Merging these pull requests would automatically move that card to the *Done* column.

Another automation that Github Boards allows us to perform, is the automatic creation of Kanban cards in the *To Do* section of our Kanban board when an issue is opened inside the repository related to that project. Following up on said issue, for example committing a change that says *Fixes #56*, automatically moves that card related to that issue, from *To Do* to *Done* in the Kanban board.

These kind of automations thanks to the unifications between source control and project management, make both tasks easier to manage and to keep consistent between themselves.

Throughout the development of this project, an automated Kanban board has been used to monitor progress, track pull requests, create new tasks and keep up to date with the project status.

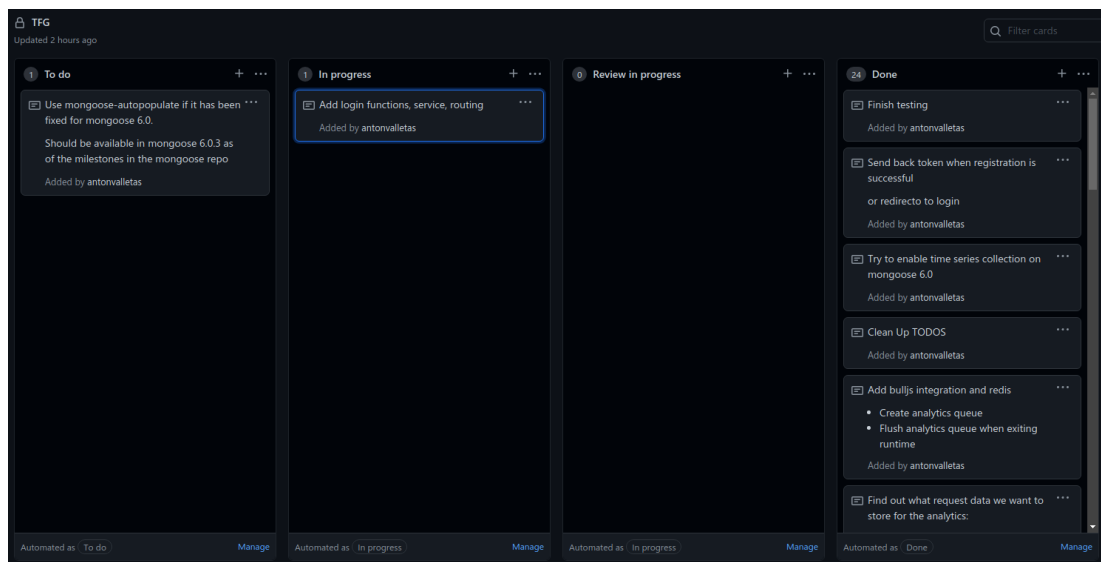


Figure 3.4: The Automated Kanban Board inside our Github Repository.

3.2 Tools and development environment

During the software engineering process with relation to this project, a wide set of tools has been used for a variety of purposes. In this section, we will go over which tools were used, and what purpose they serve in the development process.

- **Integrated Development Environment:** One of the most important tools in a software engineering project, is the IDE or *Integrated Development Environment*. We can consider an IDE as a text editor integrated with a software suite than consolidates all the tools and extensions needed for the development process.

In this project, **Visual Studio Code** was used as the IDE. VSCode is an open-source powered integrated development environment which is community developed but controlled by Microsoft. This IDE includes basic features such as syntax highlighting in a variety of different languages, as well as debugging and source control tools out of the box.

However in VSCode there are extensions with which you can increment the functionality of the IDE to match almost any feature you would require from it. Some examples of VSCode extensions used in this project include:

- Auto Close and Auto Rename HTML tags. This extensions allows us to write HTML or JSX in a more effective and efficient way by automatically renaming and closing HTML or JSX tags.
 - Jest Test runner, which allows us to manipulate, run and debug Jest test suites, which were used throughout the project for unit and integration testing.
 - ESLint, which allows us to have modern ECMAScript formatting and bug detection built in to the IDE.
 - Express, React snippets. This extension allows us to write common pieces of repetitive React or ExpressJS code in a much more efficient way by mapping certain template code into a set of easy to use keyboard shortcuts.
 - Import cost, which tells us the amount of traffic that would be needed to load certain Javascript libraries over an internet connection, so as to better judge which Javascript libraries to include in our frontend code, taking into account the potential latency the user would suffer from loading that library.
 - Live Server, which allows us to have a live web server, which automatically detects changes in the frontend code and reloads the webserver.
- **Javascript ES6+** was chosen as the development language, for both the backend part of the project and the frontend part of the project. Recent ECMAScript standards, which

are the standards that regulate the evolution of the Javascript language, make it easier to program in a more elegant manner with this language, as well as providing structures to implement the functional programming paradigm in Javascript.

The Javascript language has always had some functional programming components, but until recent standards such as ECMAScript 2015, the language didn't have all the structures that a truly functional language needed.

- **MongoDB 5.0 and MongoDB Compass** are used for object persistency and data storage. MongoDB is a document oriented (also known as NoSQL) database, intended to accelerate development due to its higher flexibility during data modeling.

In this case, MongoDB 5.0.X is needed, as in the 5.0 release, Time Series Collections were introduced into MongoDB. These new kind of collections allow storing Time Series data and metadata in a faster manner, and also speeds up the access to these documents even if these collections grow into extremely large datasets.

In our usecase, we will be using a Time Series collection to store the metadata related to visitor clicks on links, in order to extract the relevant insights and analytics from them.

MongoDB Compass is MongoDB's own database management GUI. It's very useful for making on the fly changes to collections or connections, creating indexes, modifying aggregation pipelines or even backing up entire databases.

MongoDB Compass also includes a built in mongo-cli in case there is some operation that can't be performed by means of the GUI.

- **ExpressJS** as of version 4.17.1 is used as the main backend framework. ExpressJS is an unopinionated and minimalistic backend framework which by default only provides routing, connection handling and middleware. Everything else will be done either through plugins or middleware. It's a very fast and easy to develop framework which makes creating REST APIs very straight forward.
- **NodeJS 16** was chosen as the runtime on which to base the project, as it's the latest stable release which supports the latest features from the ECMAScript standards that we needed. This Javascript runtime is based on the V8 Javascript engine that was developed for the Chromium browser. NodeJS wraps the V8 engine with the needed functions and interfaces so as to run Javascript outside of the browser.

NodeJS is based on an eventloop, that means that most of the work is done in a single thread, while some auxiliary threads do exist for performing some functions such as DNS requests or I/O Operations without stalling the main event loop.

- **React v17** was used during the frontend development part of the project in order to create a responsive and dynamic Web Application. It's JSX syntax allows for conditional redering of the DOM and on the fly manipulation of the virtualDOM based on the current application contexts or states.
- **Bootstrap 5** was used as the frontend CSS library in order to achieve a dynamic and well laid out look which scales correctly both in Desktop and on Mobile devices.
- **Redis** is an in-memory data structure store, which in this project we're using as a way to store the pending tasks and job that are yet to be processed by the task queue.
- **Jest** is used as the testing suite and test managing software. It allows writing automated unit and integration tests using a set of operators that are very similar to natural language, thus making testing more straight forward than with other testing frameworks.
- **Postman** was used throughout the development of the whole project so as to test the API endpoints, the authentication mechanisms, metadata collection and other REST API related features.
- **Node Package Manager (NPM)** was used as the default package manager for the backend part of the project, as it's the recommended and default package manager for NodeJS environments. NPM allows us to install, update, modify and delete a project's dependencies, solve dependency cycles and control project scripts and commands.
- **BullMQ** was used as the task queue in charge of processing the jobs that were dispatched by the backend to Redis. This was mainly used in the project as the way to process the analytics and insights that we wanted to extract from the Time Series collection that stored the clicks and visits metadata.
- **Yarn** was used as the package manager for the development of the frontend part of the project, since the main framework used, React, recommends using Yarn instead of NPM.

Other tools and libraries were used during the development of this project either in the backend or frontend phases. We will go into more detail in those specific chapters of the document.

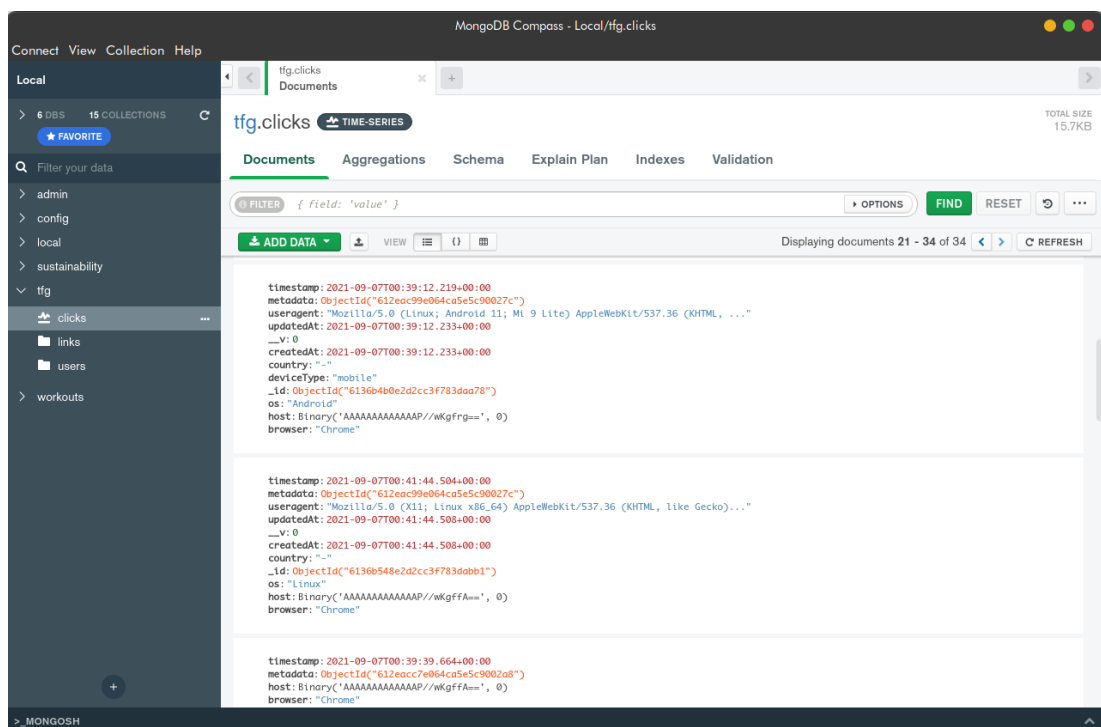


Figure 3.5: Graphical User Interface of MongoDB Compass

Backend and Data Storage

IN this chapter we will be talking about the core backend of the project. We will be talking about the design phases both data related and system architecture related, as well as going through the problems and solutions that were found throughout the development of the implementation.

During this chapter we will also analyze the chosen system architecture, and reference it throughout the subsections, explaining why these decisions were taken and what problems were encountered:

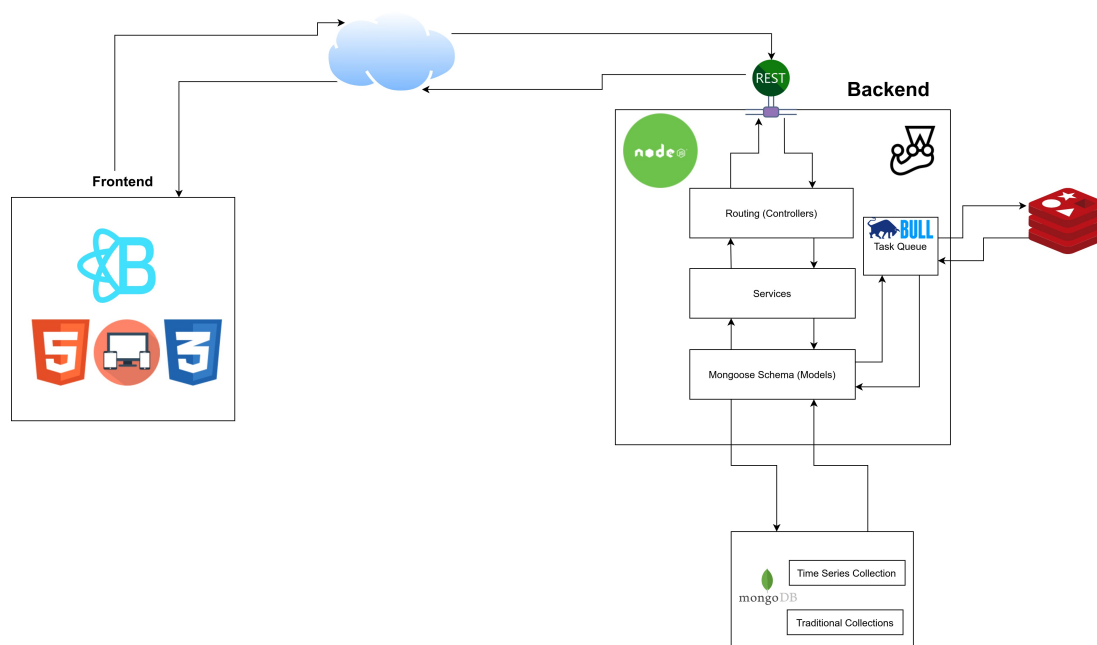


Figure 4.1: General Diagram of the project's system architecture

A more specific view, only of the backend part of the system architecture so as to better analyze in this chapter is also included:

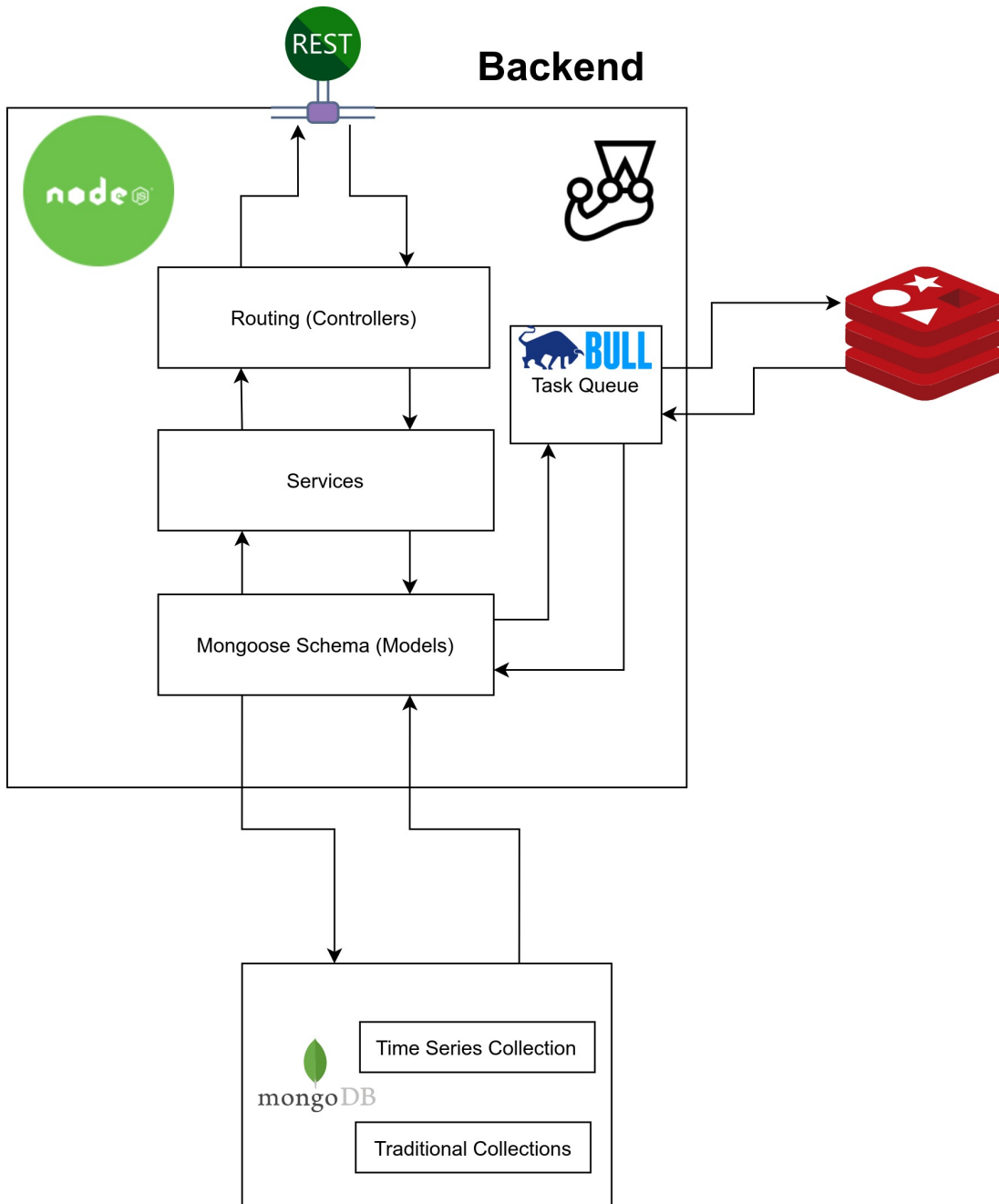


Figure 4.2: Backend Diagram of the project's system architecture

4.1 Analysis

In this section we will analyze the functional and non-functional requirements that were detailed during the analysis phase of the project. We will also go into the decision making process on what solutions were chosen for the functional and non-functional requirements.

4.1.1 Data Storage and Persistence

With regards to Data Storage and Persistence, the chosen solution for the project was MongoDB.

MongoDB is a document oriented database system. These types of databases are also known as NoSQL databases, as they do not use the Structured Query Language as their means to query and retrieve information from the database. Instead, MongoDB uses query filters, which is a notation similar to *JavaScript Object Notation* (also known as JSON) to filter the results and only retrieve the data that is needed.

This query language also has a list of very powerful operators that let us define even more complex queries with conditions that otherwise couldn't be defined using JSON notation. These operators start with the \$ symbol, and also allow evaluating logic conditions over the queried data.

To illustrate this, we will perform the same type of query both in SQL:

```
1 SELECT * FROM inventory WHERE status in ("A", "D")
```

And in MongoDB:

```
1 db.inventory.find( { status: { $in: [ "A", "D" ] } } )
```

As per the requirements of the project, we structured the data that we needed to persist, into the following main three entities:

- **User:** Entity in charge of persisting the data related to a user's profile. This includes account info, user statistics, authentication info, profile picture, biography...
- **Link:** This entity is in charge of storing the data related to a single Link. A clickable Link in a user's profile has to include information such as the URL it points to, the title the user wants to give to that link, timestamps of creation etc...
- **Clicks:** This entity is kind of different from the previous two. It would need to store metadata from a visitors click on a link, such as the user's IP, from which we would later retrieve the country code, timestamps, User Agent strings and parameters... However, in a realistic scenario, this collection would be storing massive amounts of Time Series data. Each click would have it's own timestamp, and the metadata related to that click.

During the design phase we will see in more depth how this MongoDB collection was finally modeled.

4.1.2 Backend Architecture

As for the software architecture that the backend had to follow, we decided to use a traditional 3 layer software architecture as described in the general system architecture diagram.

A *3 layer* or *3 level* architecture is a backend software architecture pattern that is basically comprised of 3 levels of logical computing. These kinds of architectures are very common in client-server applications such as the one we're developing in this project.

Following this kind of design architecture, gives the developers more flexibility allowing them to modify a given part of the stack without touching the other two layers. This also allows bigger development teams to work simultaneously on the backend part without overlapping their work. The main 3 layers of this architecture are:

- **Controller:** This layer is the top layer of the architecture. It is responsible for responding to requests, routing the necessary parameters to the appropriate services, and taking care of sending the required responses.
- **Service:** This layer is responsible for the business logic. It takes care of implementing any logic needed so as to provide the needed functions from the application.
- **Model:** This is the bottom layer of the architecture, and is responsible for interacting with the data persistence layer. In its responsibilities reside actions such as querying the database, modifying data, and storing it in the DB.

However, the generic 3 layer system architecture for backends is a bit different from the one we have implemented in the project. We will talk more in depth about the actual implementation further on.

4.1.3 Mongoose and Models

During the analysis phase of the project, the available options for interacting with the MongoDB database from the backend's Model layer were considered. There are basically two options:

- **Using the MongoDB NodeJS driver:** This option was the first one considered. It is the official driver developed by the MongoDB team so as to use MongoDB from Nodejs. This driver allows for using almost all the available MongoDB features.

The problem with using this driver for developing a backend, is that you have no way of actually designing models to later store in the database. You can store BSON documents

in the database, and retrieve them, but just that. These documents are never translated into any kind of model that facilitates working with the previously mentioned entities.

It's just that. A driver.

- **Mongoose:** This alternative consist of a wrapper over the official MongoDB wrapper. This library allows for building models based on MongoDB documents. These models are called Mongoose Schemas. They allow for converting normal BSON MongoDB documents into actual entities to work with.

It also facilitates a big plethora of operations, such as having pre-save middleware before saving an entity, having middleware that validates certain properties of fields before doing any operation, or allowing the usage of plugins that extend even more the functionality of the library.

4.1.4 Service layer

The service layer in ExpressJS applications is in charge of receiving the needed parameters from the *Routing layer* (this is how controllers are known in ExpressJS), doing the required operation by that query, calling the model to save or query any data from database, and return some kind of result to the router.

Normally in an ExpressJS application, the service layer would be made out of services *classes*. However in our project we want to have the option to scale horizontally in the future, in case there is a very high demand for the application.

To allow for this option, everything related to the backend, but moreover the service layer, is programmed in such a way that is purely functional and highly parallelizable using any process management solution such as *Docker* or *PM2*.

4.1.5 Routers

In an ExpressJS application, the top controllers layer is actually called the routing layer.

The logic is exactly the same as in a generic 3 layer backend architecture. Its purpose is that of listening to requests, calling the services, and sending the response.

However in typical ExpressJS vocabulary, this layer is called the Routing layer because it uses ExpressJS entities called *Routers*. Routers allow for filtering REST API calls based on parameters, path and HTTP Method used. For example, let's see the Router for handling registration requests.


```

/* POST request to register a user */
router.post( path: '/register', handlers: async (req : Request<P, ResBody, ReqBody, ReqQuery, Locals> , res : Response<ResBody, Locals> , next : NextFunction ) => {
  try {
    const {
      _id,
      username
    } = await AuthService.register(req.body)
    res.json( body: {
      _id,
      username
    })
  } catch (err) {
    if (err.name.includes('Validation') || err.name.includes('Validator')) {
      const error = new Error(err)
      error.status = 400
      next(error)
    } else {
      const error = new Error(err)
      error.status = 500
      next(error)
    }
  }
})
}

```

Figure 4.3: Example router receiving a POST request to register a new user.

4.1.6 Analytics and Task Queue

As for the processing of the analytics, the first option would be to just process the data contained inside the *Clicks* Time Series collection of our MongoDB database.

This seems at first as the best and easiest solution, however, it hides a very big caveat.

This solution would scale extremely poorly in an environment where the user growth and visitor traffic starts to grow. When this happens, the *Clicks* collection would grow exponentially and have millions of documents to analyze in order to retrieve a single analytic. For example, when trying to analyze the countries from which a user's profile is visited, we would need to process **on demand** the whole *Clicks* collection.

This obviously is a very bad idea if that collection contains very big amounts of data.

So, in order to achieve these analytics and insights by analyzing the click metadata from a visitors actions, we decided on storing that metadata in the *Clicks* collection, however, the most scalable way to achieve these insights and analytics are by processing them in the background utilizing a job queue, and storing them in a precomputed manner so as to not need to compute them on demand on each request.

This solution lets us achieve two things:

- **Scalability.** The metadata collection will not be saturated by constant on-demand analysis of all the documents.
- **Latency.** When analytics are requested by a logged in user of the platform, the precomputed and stored data for these analytics is instead returned. This has the same latency as any other extremely simple query to the database, as no intermediate computation is performed.

It has however one small caveat. The returned data from the analytics, is as stale as the schedule on which the analytics job runs on the queue.

4.2 Design and Implementation

In this section, we will go into greater depth and details on some of the design choices and implementation of the backend part of the project. To put the actual implementation of the project into context, the structure of the project is also included:

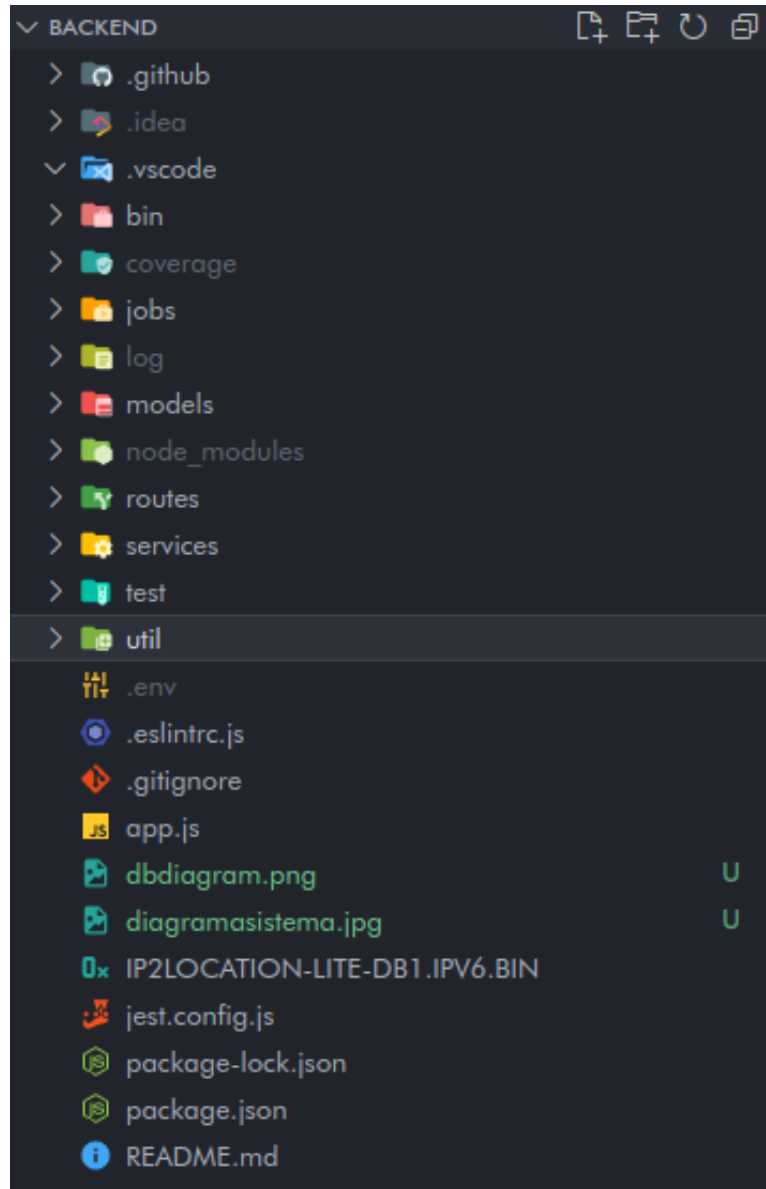


Figure 4.4: Main Structure of the project

Many of these folders' contents are pretty self explanatory, however, we will explain the purpose of some of these folders which we will analyze more in depth in the following sections:

- **.github** directory stores configuration files for the Github services that the project uses. More specifically it stores configuration files for Github's Dependabot and Github Actions CI engine.
- **bin** stores the first executable which NodeJS calls. It contains ExpressJS initialization parameters, starts listening on the given port, and schedules some callbacks in case any error happens.
- **jobs** contains the setup and tasks defined for the task queue.
- **node_modules** is a product of NPM installing all the project dependencies. It's a very heavy folder, as dependencies of dependencies are also installed.
- Some files are also worth mentioning:
 - **.env** is an environment file which contains a JWT Secret, port numbers, and other variables needed when deploying the project. It is read onto the NodeJS runtime at startup which makes all environment variables accessible from the *process.env* global variable.
 - **.eslintrc** includes rules and setup for ESLint, which is the default Javascript formatter and linter.
 - **app.js** is the file where the ExpressJS initialization occurs. This file also includes a lot more configuration operations, such as using all the routers in the project, registering queue workers, registering all the ExpressJS middleware needed for parsing bodies, cookies, URL parameters and many more commands that we will see in the last section of the backend implementation.
 - **IP2LOCATION-LITE-DB1.IPV6.BIN** contains a binary database with the relation of IPs and which country they belong to. This database is provided by ip2location.com and is updated every month.
 - **jest.config.js** contains configuration and environment information needed for running the test suites.
 - **package.json** is the file responsible for keeping track of all dependencies and devDependencies of the project. It also stores commands to run, test and deploy the project.

4.2.1 Data Storage and Persistency

With regards to Data Storage in the MongoDB database, we will design and implement the previously mentioned three entities (*User*, *Link* and *Click*) using Mongoose to design the Schemas that define these models.

For the *User* and *Link* entities, a normal MongoDB collection was chosen. However, for the *Clicks* entity, a special Time Series Collection was used.

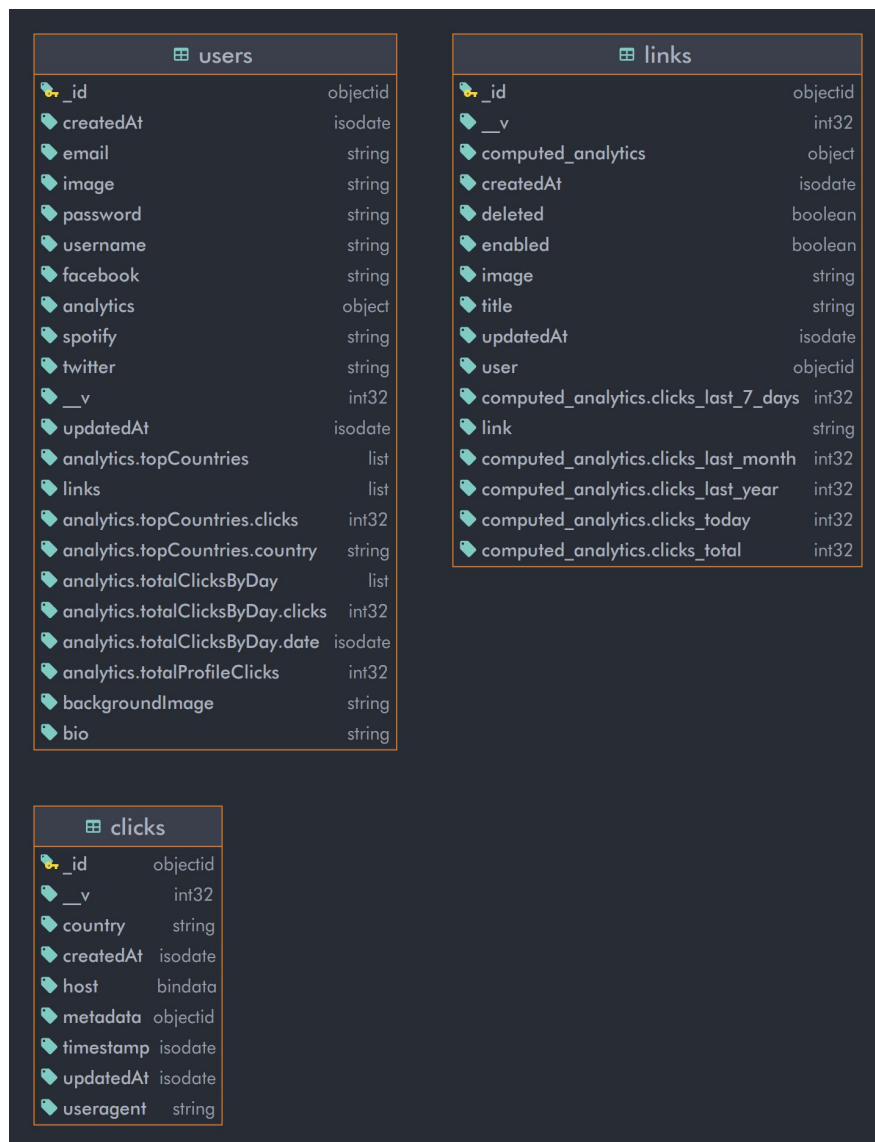
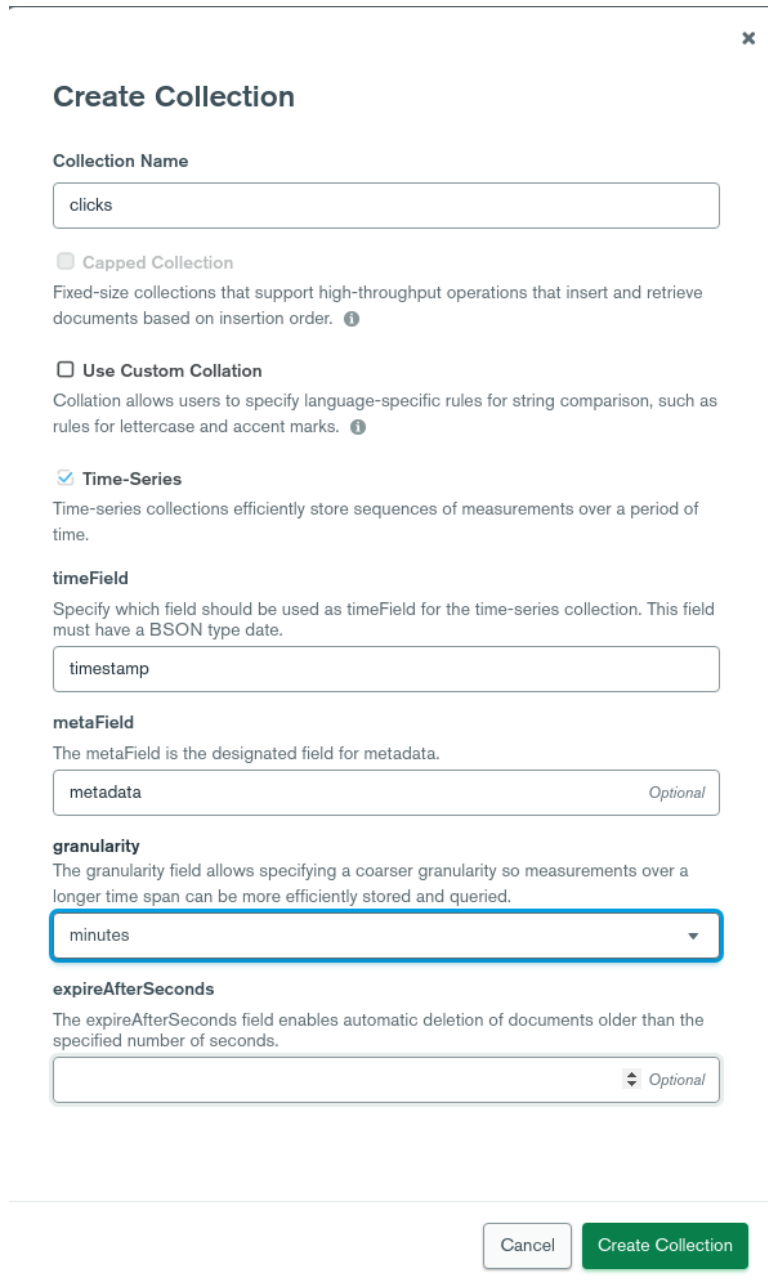


Figure 4.5: Diagram of the resulting MongoDB collections.

These are the collections created and their attributes, some of which are created by MongoDB to keep consistency, such as the `__v` attribute, which is the version tag.

Mongoose takes care of creating these collections automatically when exporting the models in the next section if they don't already exist. However, as of the latest version of Mongoose, support for Time Series Collection creation is yet to be implemented. Because of this, the *Clicks* collection must be created manually. We can create it either in MongoDB Compass or MongoDB CLI. We will create it using MongoDB Compass:



The screenshot shows the 'Create Collection' dialog in MongoDB Compass. The 'Collection Name' field contains 'clicks'. The 'Capped Collection' option is unchecked. The 'Use Custom Collation' option is also unchecked. The 'Time-Series' option is checked. The 'timeField' is set to 'timestamp'. The 'metaField' is set to 'metadata'. The 'granularity' is set to 'minutes'. The 'expireAfterSeconds' field is empty. At the bottom, there are 'Cancel' and 'Create Collection' buttons.

Create Collection

Collection Name
clicks

Capped Collection
Fixed-size collections that support high-throughput operations that insert and retrieve documents based on insertion order. ⓘ

Use Custom Collation
Collation allows users to specify language-specific rules for string comparison, such as rules for lettercase and accent marks. ⓘ

Time-Series
Time-series collections efficiently store sequences of measurements over a period of time.

timeField
Specify which field should be used as timeField for the time-series collection. This field must have a BSON type date.
timestamp

metaField
The metaField is the designated field for metadata.
metadata *Optional*

granularity
The granularity field allows specifying a coarser granularity so measurements over a longer time span can be more efficiently stored and queried.
minutes

expireAfterSeconds
The expireAfterSeconds field enables automatic deletion of documents older than the specified number of seconds.
Optional

Cancel Create Collection

Figure 4.6: Manual creation of the Clicks Time Series Collection using MongoDB Compass

As we can observe, for Time Series Collections, we must set a few predefined fields:

- **timeField** defines which attribute will contain the Timestamp data of each of the documents. This is mandatory as all Time Series documents must have a timestamp.
- **metaField** defines a field which uniquely identifies the source for that information. In our case, this field points to the *ObjectID* of the Link which that *Click* document refers to.
- **granularity** provides MongoDB with a way of knowing what timeframe to optimize for.

Once the *Clicks* collection has been manually initialized, and the backend project has been executed at least once, all three collections will exist inside the database, and Time Series Collections will be differentiated from normal collections:

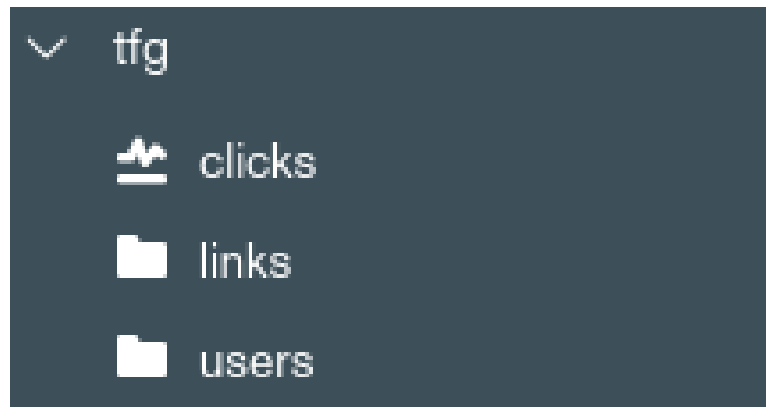


Figure 4.7: All three collections once initialized.

4.2.2 Model Layer

Mongoose allows use to create *Schemas* which are later exported and compiled as models. These models have a set of attributes that are set and stored

in the database, but they also have methods and virtual properties which are defined in the backend.

Mongoose also allows us to register middleware on certain database events, such as *onSave* or *onUpdate*. We can use these methods to, for example, hash and salt a user's password before registering.

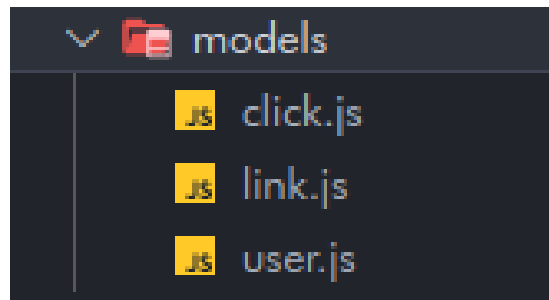


Figure 4.8: Contents of the model folder

Each of these models uses a Mongoose Schema to create an entity model, some methods and middleware functions. We will see the *user.js* example in-depth:

```

user.js > ...
You, seconds ago | 1 author (You)
const mongoose = require(id: 'mongoose')
const bcrypt = require(id: 'bcrypt')
const uniquevalidator = require(id: 'mongoose-unique-validator') 7.3K (gzipped: 2.9K)
const UserSchema = new mongoose.Schema(definition: { You, a month ago • Added mongoose user model methods to authenticat...
  username: {
    type: String,
    required: [true, 'Can\'t be blank'],
    match: /^[a-zA-Z0-9]+$/, 'Is invalid',
    index: { unique: true },
    unique: true,
    minlength: 1,
    maxlength: 40
  },
  email: {
    type: String,
    required: [true, 'Can\'t be blank'],
    match: [/^S+@S+\.S+/, 'Is invalid'],
    index: true
  },
  password: {
    type: String,
    required: [true, 'Can\'t be blank']
  }, // Hashed password
  bio: String,
  image: String, // Base64 encoded image
  backgroundImage: String, // Base64 encoded background image
  facebook: String,
  twitter: String,
  instagram: String,
  tiktok: String,
  spotify: String,
  youtube: String,
  applemusic: String,
  links: [{ type: mongoose.Schema.Types.ObjectId, ref: 'Link' }], // Array of populated link documents from the link collection
  analytics: {
    totalClicksByDay: [{
      _id: false,
      date: mongoose.Schema.Types.Date,
      clicks: { type: Number, default: 0 }
    }],
    totalProfileClicks: Number,
    topCountries: [{
      _id: false,
      country: String,
      clicks: { type: Number, default: 0 }
    }]
  }
}, options: { timestamps: true }) // Keeps creation and update timestamps

```

Figure 4.9: Mongoose Schema used to model the User entity

As we can see, in the schema we can define each of the user's attributes, as well as performing validation in several ways. For example, in the *username* attribute, we're performing validation so as to only have unique usernames. The username can't be blank in any entity, and it must always match the provided Regular Expression as well as complying with the length constraints specified.

The *links* attribute of the *User* entity, is an array of MongoDB ObjectIds. This lets us store an array of ObjectIds, where each and every one of them points to a single document in the *Links* collection. This allows us to use a feature of Mongoose called *Population* where the library replaces these ObjectIds in the links array, with the documents that these Ids point to, allowing us to relate a specific user with its links.

As stated previously, models also contain middleware functions that execute based on certain events, such as saving an entity, and virtual methods that provide behaviour to entities.

```

UserSchema.pre('save', fn: async function save (next: (err?: NativeError) => void): Promise<void> {
  if (!this.isModified('password')) return next()
  try {
    if (this.password.length < 8 || this.password.length > 65) next(err: this.invalidate('password', 'Invalid password'))
    const salt: string = await bcrypt.genSalt(rounds: parseInt(string: process.env.SALT_WORK_FACTOR))
    this.password = await bcrypt.hash(data: this.password, saltOrRounds: salt)
    return next()
  } catch (err) {
    return next(err)
  }
})

0 references
UserSchema.methods.isValidPassword = async function (candidatePassword: any): Promise<Promise<boolean> & ... {
  return await bcrypt.compare(data: candidatePassword, encrypted: this.password)
}

UserSchema.plugin(fn: uniquevalidator)
| You, a month ago · Add more validation for usernames, passwords. Use...
module.exports = mongoose.model(name: 'User', schema: UserSchema)

```

Figure 4.10: Pre-save middleware function for validating, salting and hashing passwords. A method for validating a passwords is also created.

4.2.3 Service Layer

The service layer is structured in exactly the same way as the model layer. There are three main services:

- **AuthService** is responsible for providing the mechanisms related to Authorization and Authentication of users. For example, it provides *changePassword*, *register...* and also includes two PassportJS strategies to perform logins and user verification.
- **UserService** is responsible for providing methods to allow users to change and update their profile information, as well as providing methods for visitors to query a users profile. These operations include getting the public user data, the private user data (this includes analytics and statistics), modifying a users profile...

- **LinkService** provides services that include all kinds of operations over the Links entity, as well as registering clicks on a link. For example, there are methods to create and delete links, as well as methods to store metadata related to a click on a given link.

```

const UserService : { getPublicUserData: (usern... = {}

/**
 * @description Gets public user data
 * @param {String} username username of the queried user
 * @returns {Object} Result profile      You, 3 weeks ago • Add post method to allow for updating user info
 * @throws {Error} When user not found
 */
getPublicUserData: async (username) => {
  const profile : LeanDocument<Document<any, ... = await UserModel.findOne( filter: { username: username }, populate( path: 'links').lean()
  if (profile) {
    delete profile.password
    delete profile.email
    delete profile._id
    delete profile.createdAt
    delete profile.updatedAt
    delete profile.analytics
    delete profile.__v
    | reference
    profile.links = profile.links.filter( link : any => link.deleted !== true)
    for (const link of profile.links) {
      delete link.enabled
      delete link.deleted
      delete link.computed_analytics
      delete link.createdAt
      delete link.updatedAt
      delete link.__v
      delete link.user
    }
    return profile
  } else {
    throw new Error( message: 'User not found')
  }
},

/**
 * @description Gets private user data including analytics
 * @param {String} username username of the queried user
 * @returns {Object} Result profile
 */
getAllUserData: async (userid) => {
  const profile : LeanDocument<Document<any, ... = await UserModel.findOne( filter: { _id: userid }, populate( path: 'links').lean()
  | reference
  profile.links = profile.links.filter( link : any => link.deleted !== true)
  delete profile.password
  delete profile.createdAt
  delete profile.updatedAt
  delete profile.__v
  return profile
},

```

Figure 4.11: Example of user service methods *getPublicUserData* and *getAllUserData*, with their respective JSDocs.

With regards to Authentication and Authorization, a library called PassportJS is used. This library provides strategies to create middleware in order to inject into routers to allow for authentication and validation of users.

The *auth.js* service has two strategies implemented. One for logging in a user, and another one for validating that a user is correctly logged in and has permission to perform an action. For this project, we went with an authentication strategy based on *Javascript Web Tokens* or JWTs.

Authentication and Authorization based on JWTs implemented in this project, follows the following diagram:

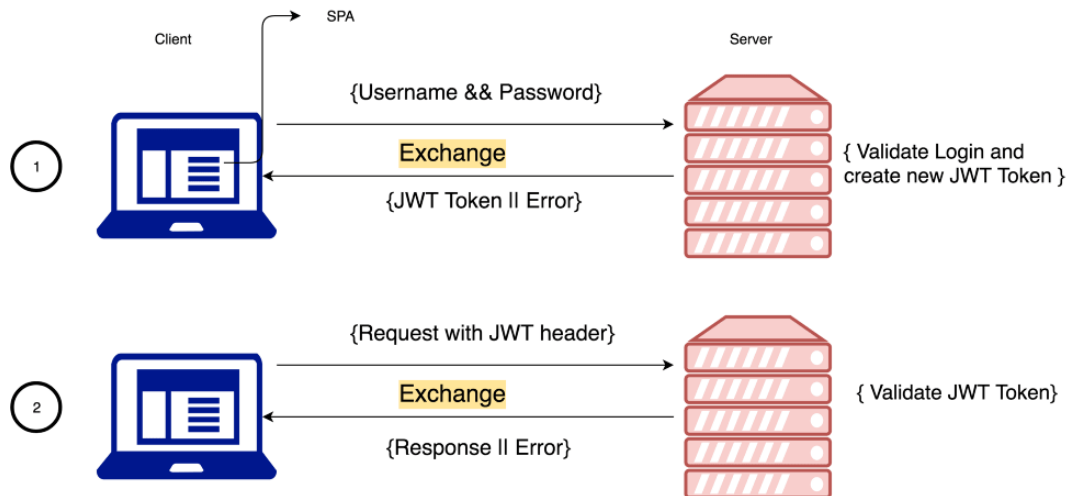


Figure 4.12: Diagram showing the Authentication and Authorization process using JWTs.

```

/**
 * @description Strategy checking jwt tokens in order to secure endpoints
 */
passport.use(
  strategy: new JWTstrategy(
    opt: {
      secretOrKey: process.env.SECRET,
      jwtFromRequest: ExtractJWT.fromAuthHeaderAsBearerToken()
    },
    // 0 references
    verify: async (token : any, done : VerifiedCallback) : Promise<void> => {
      try {
        // Validates the token and calls the next handler in the chain
        return done(error: null, token.user)
      } catch (error) {
        // Returns unauthorized 401 and doesnt call the next route handlers
        done(error)
      }
    }
  )
)

```

Figure 4.13: Implementation of the PassportJS strategy to check a user provided JWT.

4.2.4 Routing layer

As previously mentioned, in ExpressJS applications, the traditionally called *Controllers* are called *Routers*. These routers allow the ExpressJS application to be divided into several controller groups.

In our project, each Router matches a service. For example, there exists a *user.js* router,

which receives all requests that are meant to be processed by the *user.js* service. Routers can accept requests depending on the HTTP Method provided. For example, this is the router taking care of receiving the metadata when a user clicks on a link of a profile:

```

/* Endpoint to register a click on a link */
router.post(path: '/linkId', ...handlers: async (req: Request<{ linkId: string; }... , res: Response<any, Record<string... , next: NextFunction) : Promise<void> => {
  res.status(200).send()
  try {
    await linkService.registerClick(req.params.linkId, userAgent: req.headers['user-agent'], ipAddress: req.headers['x-forwarded-for'] || req.ip)
  } catch (err) {
    logger.error(message: err)
  }
})

```

Figure 4.14: Link router accepting POST request to store metadata related to a click on a link

These routers are all later exported, and they are handled by the main ExpressJS application. There, we set them up to use the main API route, followed by a specific handler for each router:

```

/* Imported routers are set up here, indicating their root route */
app.use(path: '/api/v1/auth', ...handlers: authRouter)
app.use(path: '/api/v1/users', ...handlers: userRouter)
app.use(path: '/api/v1/link', ...handlers: linkRouter)

```

Figure 4.15: All the application's routers being setup in the main *app.js* file.

4.2.5 Task Queue and Analytics

As previously mentioned in the analysis phase, a Task Queue is the best approach, if maybe not the easiest, to processing metadata in order to obtain insights and analytics.

A Task Queue allows scheduling certain tasks to be run as background jobs. Scheduling can happen every certain amount of time, on certain days, at certain times... Some jobs can also be scheduled to run continuously on a limited amount of cores.

Analytics processing in our application is performed every 5 minutes. This is a good middleground between having relatively recent analytics precomputed, and not saturating the database with traffic.

The task queue library used in our project is called BullMQ, which is the Typescript evolution of BullJS. It allows safer developing of jobs and tasks, even if using Javascript. When using BullMQ to manage tasks, Redis needs to also be running in the background, as it is used to store pending tasks, scheduling new tasks and temporarily storing results of the operations.

```

// Starts the analytics processing queue.
analyticsQueue.add(name: 'ComputeAnalytics', data: {}, opts: { repeat: { every: 300000 } })

// Flushes Redis and queues in case we shutdown the runtime
const handleExit : (_: any) => void = _ : any => {
  analyticsQueue.drain(delayed: true)
}
| You, a week ago • Handle the flushing of Redis when exiting runtime
process.on(event: 'exit', listener: handleExit)
process.on(event: 'SIGTERM', listener: handleExit)
process.on(event: 'SIGINT', listener: handleExit)

```

Figure 4.16: Analytics processing task being scheduled every 5 minutes on the main BullMQ Worker. This is done from the main ExpressJS *app.js* file.

Inside the *jobs/* folder, we can see that there is a *analytics.js* file. This file contains the Analytics queue worker, and Task Scheduler.

Inside that worker resides the main logic behind the computation of the analytics. The logic is pretty extense, but we will be looking at some snippets. However, the main process consists of looping through users, checking their links, and for every user and link we compute the appropriate analytics, then store them in the model they were modeled in.

For most of the operations inside the queue, we use Mongoose Aggregation Pipelines. These Aggregation pipelines allow us to perform a complex set of operations in order to obtain and manipulate the data we need in a set of predefined stages. These Aggregation Pipelines, even if executed from the Mongoose library, actually run inside the MongoDB server, and not in the NodeJS environment, so they are considerably faster than if we were to perform these operations inside the analytics task itself.

Here are a few examples of Aggregation Pipelines used throughout the analytics processing task:

```

const totalClicksByDay : any[] = await ClickModel.aggregate(pipeline: [
  { $match: { metadata: { $in: linkIds } } },
  {
    $group: {
      _id: {
        day: { $dayOfMonth: '$timestamp' },
        month: { $month: '$timestamp' },
        year: { $year: '$timestamp' }
      },
      clicks: { $sum: 1 }
    }
  },
  { $sort: { _id: 1 } },
  { $project: { _id: 0, date: '$_id', clicks: 1 } }])

```

Figure 4.17: Analytics pipeline for computing the total clicks by date for a given user.

```
const topCountries : any[] = await ClickModel.aggregate(pipeline: [{ $match: { metadata: { $in: linkIds } } },  
  { $group: { _id: '$country', clicks: { $sum: 1 } } },  
  { $sort: { clicks: -1 } },  
  { $limit: 10 },  
  { $project: { _id: 0, country: '$_id', clicks: 1 } }  
])
```

Figure 4.18: Analytics pipeline for computing the top countries generating clicks on a user's profile.

These analytics pipelines work somewhat intuitively as to how a generic pipeline structure works. In these snippets, each of the pairs of yellow brackets is a single stage in that pipeline. In every stage, a single operation is performed, and that result is then passed down to the next stage, this keeps happening until the last stage of the pipeline is executed.

4.2.6 Other Libraries and Tools

During the development of the project, a whole set of other libraries was used to obtain very needed features that any software backend should have. Some of these libraries and features include:

- **Request and Error Logging using Morgan and Winston.** These two libraries were combined to provide an extensive set of logging functions. For every request received, the metadata from the request is logged, along with the requested resource, the response time our backend achieved, the HTTP Status Code included in the response, and a lot more information.

If however the response to the client is an error code, our backend will log all information related to that error, such as full stack traces and error object descriptions.

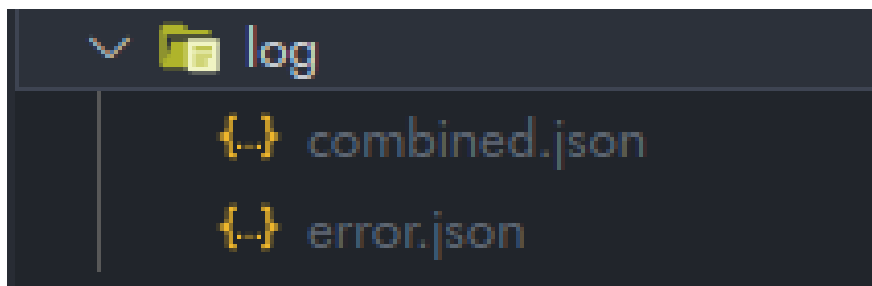


Figure 4.19: Combined and error logs stored in JSON format inside the project.

```

You, 2 weeks ago | 1 author (You)
const winston = require('winston')
const logger = winston.createLogger({
  level: 'info',
  format: winston.format.combine( ... formats: winston.format.timestamp(opts: { format: 'DD-MM-YYYY HH:mm:ss' }), winston.format.json()),
  transports: [
    //
    // - Write all logs with level `error` and below to `error.log`
    // - Write all logs with level `info` and below to `combined.log`
    //
    new winston.transports.File({
      dirname: 'log',
      filename: 'error.json',
      level: 'error'
    }),
    new winston.transports.File({
      dirname: 'log',
      filename: 'combined.json'
    })
  ]
})
// You, 3 weeks ago · Add correct logging using winston. Add stream fun...
module.exports = logger

```

Figure 4.20: Setup of the Winston logging files and their individual configuration.

- **Nodemon.** This tool greatly facilitated the development of NodeJS + ExpressJS backend by automatically detecting any changes in the project structure or project files, and if so, reloading the runtime so as to always be running the latest changes.
- **Dependabot.** This is one of the features that was integrated into the project by using it on the project's Github Repo. Once configured, Dependabot continuously monitors the source code and commit history of the project's Github Repository. If an outdated dependency is found in the *package.json* file, Dependabot automatically makes a new branch with the new dependency, runs needed CI checks and if successfully passed, creates a new pull request in order for the maintainer to be easily able to keep all dependencies up to date.

Dependabot is also able to monitor vulnerability repositories such as CVE in order to warn the repository's maintainer of possible vulnerabilities because of the projects dependency setup. For example:

GitHub Advisory Database / CVE-2021-3757

Prototype Pollution in immer

high severity Published 4 days ago

Vulnerability details Dependabot alerts

Package	Affected versions	Patched versions	CVE ID
immer (npm)	< 9.0.6	9.0.6	CVE-2021-3757

Description

immer is vulnerable to Improperly Controlled Modification of Object Prototype Attributes (Prototype Pollution)

References

- <https://nvd.nist.gov/vuln/detail/CVE-2021-3757>
- [immerjs/immer@fab713e](https://github.com/immerjs/immer/pull/713)
- <https://huntr.dev/bounties/23d38099-71cd-42ed-a77a-71e68094adfa>

CWEs

- CWE-1321

CVSS Score

7.5 High
CVSS3.0/AV:N/AC:L/PR:N/UI:N/S:U/CN:1/I:N/A/H

Figure 4.21: Private Dependabot security alert on one of our projects nested dependencies.

The screenshot shows a GitHub Pull Request interface. At the top, the title is "Bump bullmq from 1.46.2 to 1.46.4 #18". Below the title, there is a green "Open" button and a status bar indicating "dependabot wants to merge 1 commit into master from dependabot/npm_and_yarn/bullmq-1.46.4".

The main content area shows a comment from the dependabot bot, stating "dependabot (bot) commented on behalf of github 2 days ago". The comment text includes: "Bumps bullmq from 1.46.2 to 1.46.4.", "Release notes", "Commits", a "compatibility unknown" label, and instructions: "Dependabot will resolve any conflicts with this PR as long as you don't alter it yourself. You can also trigger a rebase manually by commenting @dependabot rebase.". There is also a section for "Dependabot commands and options".

Below the comment, there is a commit summary: "Bump bullmq from 1.46.2 to 1.46.4" with a "Verified" badge and a commit hash "2264c8f".

Further down, there are two bot actions: "dependabot (bot) added the dependencies label 2 days ago" and "dependabot (bot) mentioned this pull request 2 days ago".

At the bottom of the PR, there is a "Closed" button and a message: "Add more commits by pushing to the dependabot/npm_and_yarn/bullmq-1.46.4 branch on antonvalletas/tfg-backend."

The bottom section shows a green box with the status "All checks have passed" (1 successful check). It lists two checks: "Node.js CI / build (16.x, 5.0) (pull_request) Successful in 1m" and "This branch has no conflicts with the base branch" (Merging can be performed automatically.). A "Merge pull request" button is visible at the bottom of this section.

Figure 4.22: Pull Request from Dependabot to update to the newest version of BullMQ.

Frontend

THE frontend part of the project consists of creating a progressive Web Application that correctly integrates and interacts with the Backend that we have already developed. The backend part of the project exposes a REST API that is to be accessed by the frontend Web Application in order to perform whatever actions are necessary.

Thanks to the analysis and research performed before the implementation of the backend part of the project, we know that one of the most important requirements that this frontend must have, is scaling correctly and being usable in mobile devices. This is due to the fact that the vast majority of Instagram users access the platform from mobile devices usually by using the official iOS or Android Apps.

As well as providing a simple and accessible way for visitors to check a user's profile and browse through his links, the application must also provide a way for the actual registered user, to modify his profile information, create and delete links in his profile, check his analytics and modify his password.

5.1 Analysis

As previously stated, the frontend application should provide:

- A view for a visitor to check out a users profile. This includes:
 - Seeing the users profile picture
 - The user's @ handle
 - The user's biography
 - Scrolling through the list of links that user may have
- A visitor must also be able to register an account if he wants to make a profile for himself.

- A user that is not logged in, must be able to log in.
- When a user is logged in, he must be able to:
 - Check his profile page. In this case, as he is the owner of that profile, private analytics data will be displayed.
 - He will also be able to delete existing links present in his profile page
 - He must be able to change his password
 - He must be able to change his biography
 - He should be able to upload a new profile picture
 - He must be able to create new links to new URLs
 - He must be able to log out.

All these features and functions must be performed in a consistent and user friendly manner, both in mobile and desktop devices. The User Experience must be similar independently of the platform used, besides from the obvious adaptations that will have to be made to the look and feel to adapt the application to each of these device types.

5.2 Design and Mockups

In order to achieve this mobile first, but adaptive design, mockups were created for the mobile application, that were to be adapted later to achieve integration also in desktop devices. This would mainly consist of adapting the width of elements to desktop screen orientations, and making a dynamic navigation bar, that in mobile devices would actually be displayed as a dropdown menu.

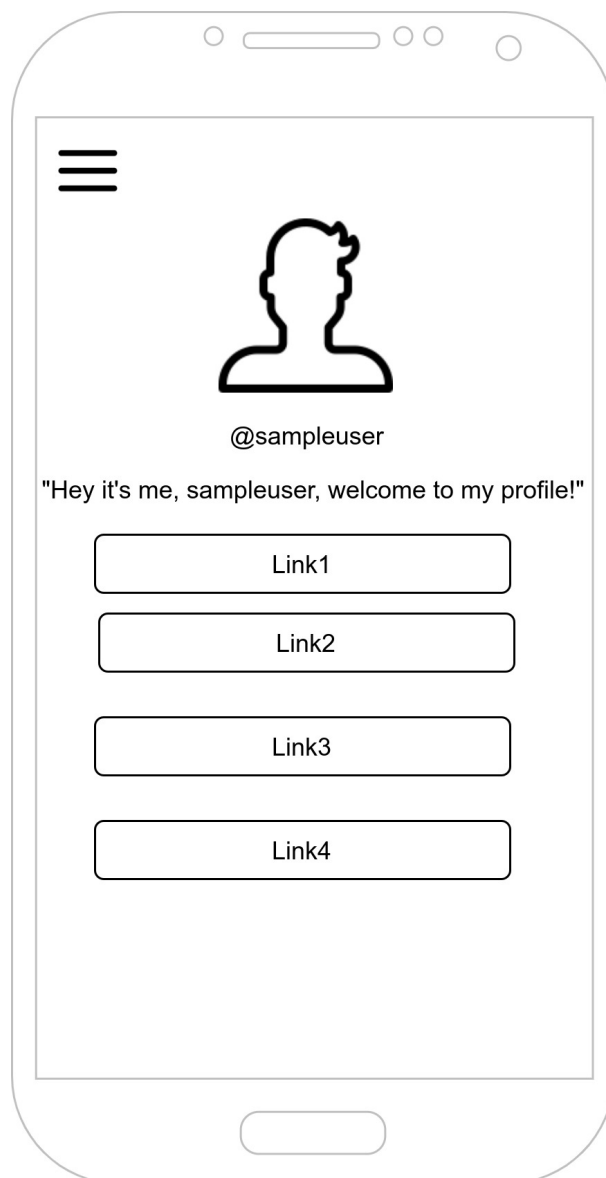


Figure 5.1: Mockup of the Mobile version of a user's profile from a visitor's perspective.

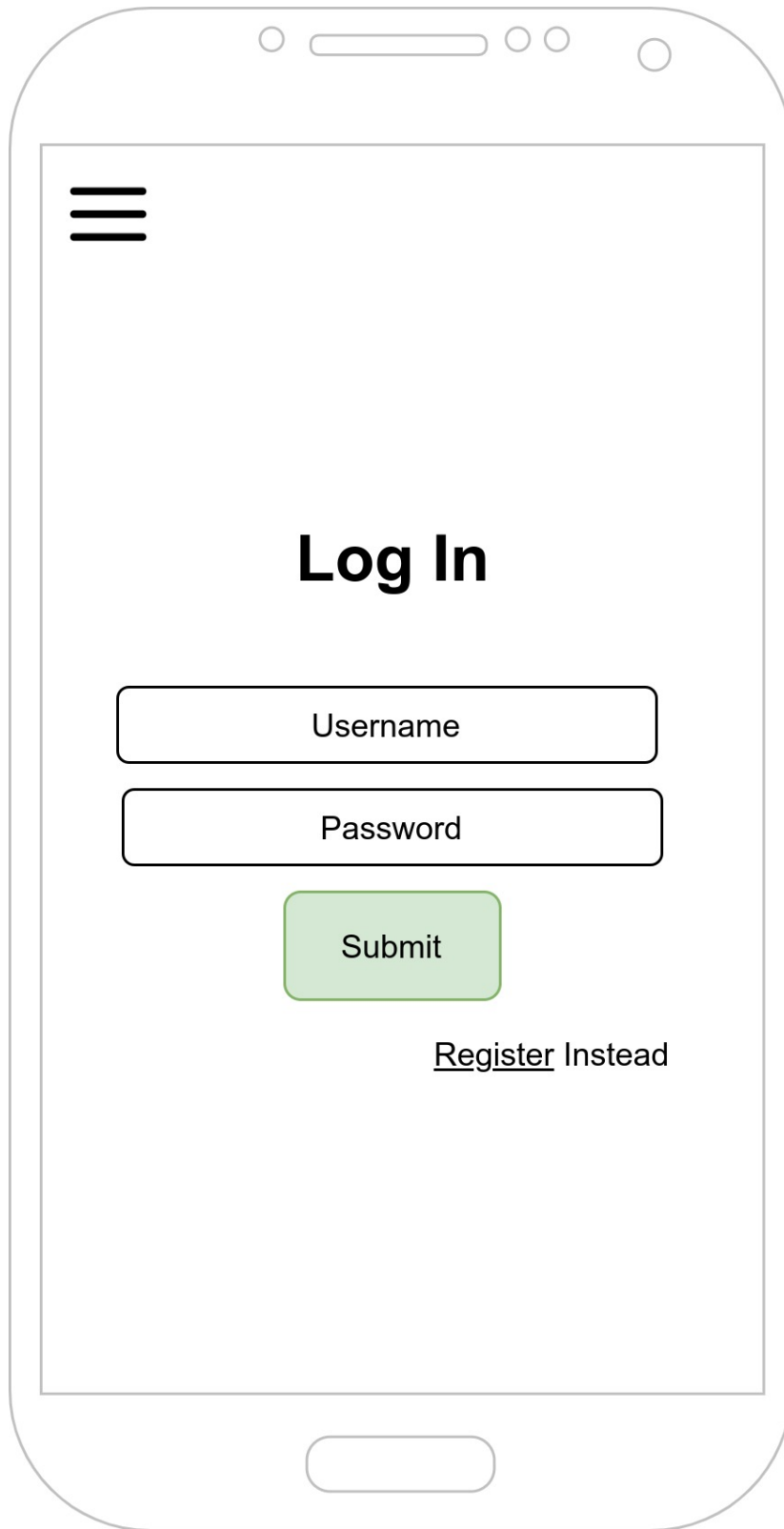


Figure 5.2: Mockup of the Mobile version of the login page.

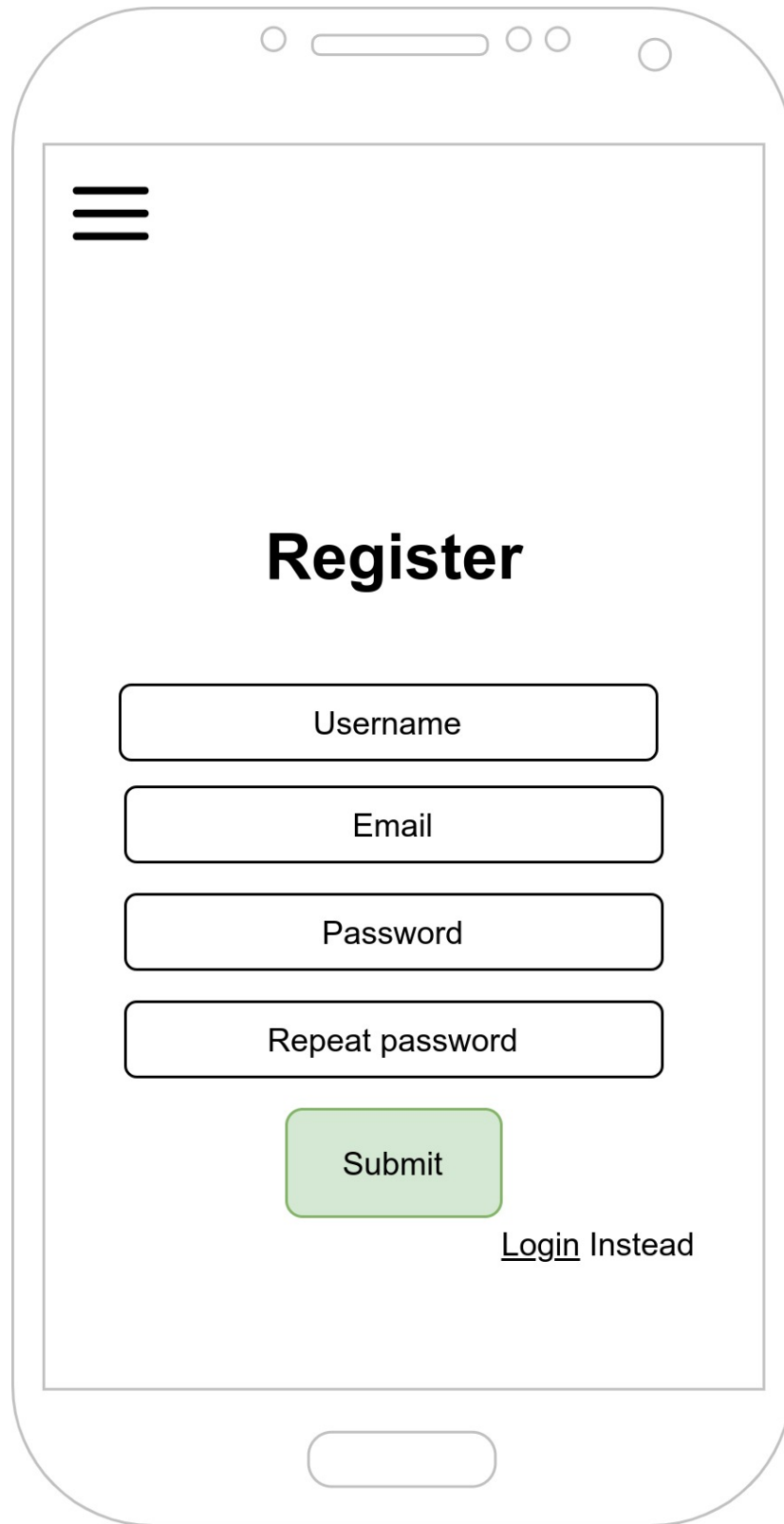


Figure 5.3: Mockup of the Mobile version of the registration page.

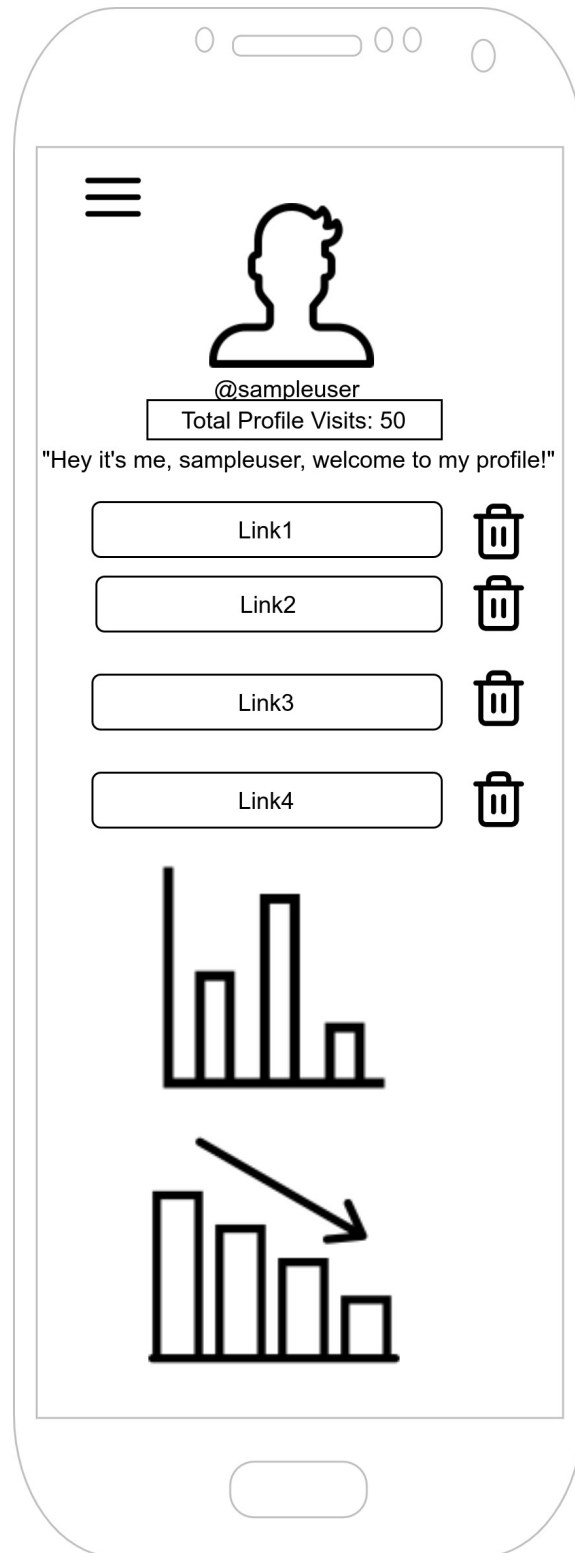


Figure 5.4: Mockup of the Mobile version of a user profile, from that user's perspective.

5.2.1 Libraries and Tools

To achieve this responsive Web Application design, we settled for the following suite of tools:

- **React.** This will be the main framework used to develop the User Interface. React allows us to create responsive and dynamic user interfaces which store context and state, and adapt to user actions on the fly.
- **Bootstrap.** Bootstrap CSS framework gives us a great starting point for building responsive interfaces that adapt well on all types of devices.
- **Recharts.** A React based data visualization and graphing library that we will use to plot the graphs we need to display in order to communicate the needed analytics.
- **Axios.** This is the HTTP Client that will be used to interact with our platforms backend. Axios provides a modern, fast, and promise based API for making web requests. This allows us to use the Javascript *async/await* keywords to better structure our asynchronous code without accidentally using antipatterns such as *Callback Hell*.
- **HTML5 and CSS3** will be used wherever needed throughout the project.

5.3 Implementation

The Web Application is developed around React framework v17. In this case, we have chosen a functional programming style for the development of the project. This allows us to make stateful React components that dynamically adapt to changes in both contexts and states.

The application also has a dynamic navbar (in desktop devices) which is rendered as a dynamic dropdown menu in mobile devices.

The application also uses React DOM Routers throughout every component, enabling us to dynamically redirect the user to the view or component that we need him to see at every point in time.

There are two main contexts in the application. One maintains the User Profile context, and another one maintains the authentication status.

```

Function App() : Element {
  const [isAuthenticated, userHasAuthenticated] = useState(initialState: false)
  const [userProfile, setUserProfile] = useState(initialState: null)

  return (
    <Router>
      <AuthenticatedContext.Provider value={{ isAuthenticated, userHasAuthenticated }}>
        <UserProfileContext.Provider value={{ userProfile, setUserProfile }}>
          <div className="App"> You, 4 days ago + small fix
            <Navigation />
            {!userProfile && <Route exact path="/" component={Login} />}
            {userProfile && <Route exact path="/" component={UserProfile} />}
            {userProfile && <Route exact path="/link/new" component={CreateNewLink} />}
            {userProfile && <Route exact path="/user/modify" component={ModifyProfile} />}
            {userProfile && <Route exact path="/user/changepassword" component={ChangePassword} />}
            <Route exact path="/user/login" component={Login} />
            <Route exact path="/user/register" component={Register} />
            <Route exact path="/:username" component={UserProfile} />
          </div>
        </UserProfileContext.Provider>
      </AuthenticatedContext.Provider>
    </Router>
  )
}

D references
export default App

```

Figure 5.5: Main starting point for the React application. Contexts are initialized and Routes are specified. States are also used.

```

const UserProfile : () => Element = () : Element => {
  const { userProfile } = useUserProfileContext()
  const { username } = useParams()
  let redirectToLogin : boolean = false

  const [requestedUser, setRequestedUser] = useState(initialState: null)
  const [links, setLinks] = useState(initialState: [])
  const [analytics, setAnalytics] = useState(initialState: null)

  D references
  useEffect(effect: Async () : Promisevoid => {
    if (username && username !== userProfile?.username) {
      setRequestedUser(value: (await axios.get(url: API_URL + '/users/' + username)).data)
    } else if (userProfile) {
      setRequestedUser(value: userProfile)
    } else {
      redirectToLogin = true
    }
  }, deps: [])

  D references
  useEffect(effect: () : void => {
    if (requestedUser !== null) {
      D references
      let linksgenerated : any = requestedUser.links.map((link : any, index : any) : Element => {
        return <LinkElement key={index} url={link.link} linkId={link._id} title={link.title} isPrivateData={{'analytics' in requestedUser}} ></LinkElement>
      })
      setLinks(value: linksgenerated)
    }
  }, deps: [requestedUser])
}

```

Figure 5.6: React useState and useEffect hooks being used to dynamically query the REST API or generate components.

5.4 Results

In this section we will see and compare several of the application's user interfaces between desktop and mobile devices. We will also be showing some of the backend features, such as input validation and file uploading to the backend.

During this section, we will use a mock profile to simulate a famous user.

5.4.1 User profile from a visitor's perspective

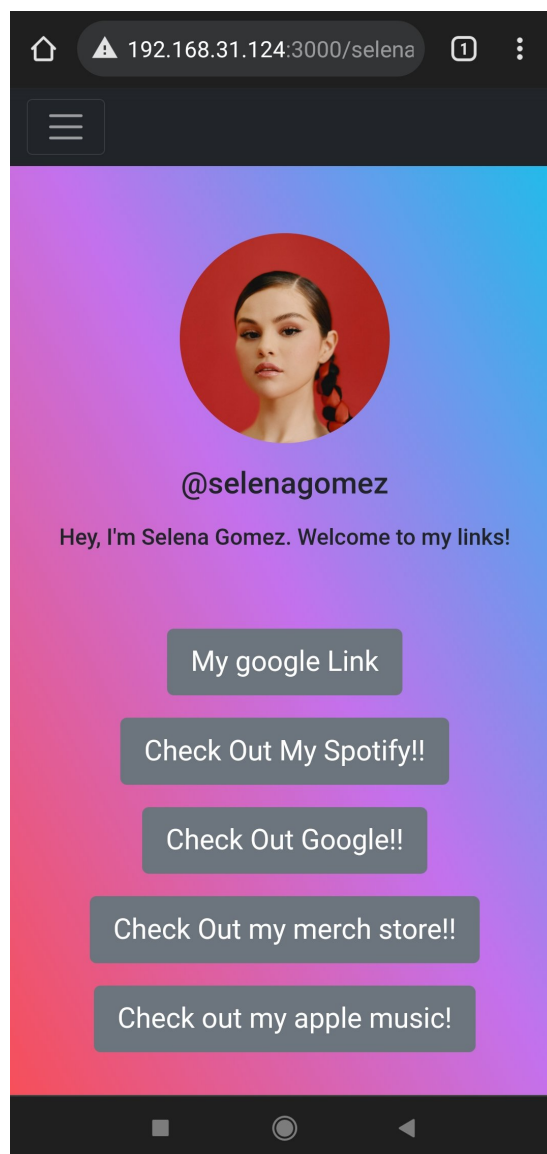


Figure 5.7: Profile page from a visitor's perspective, on a mobile device.

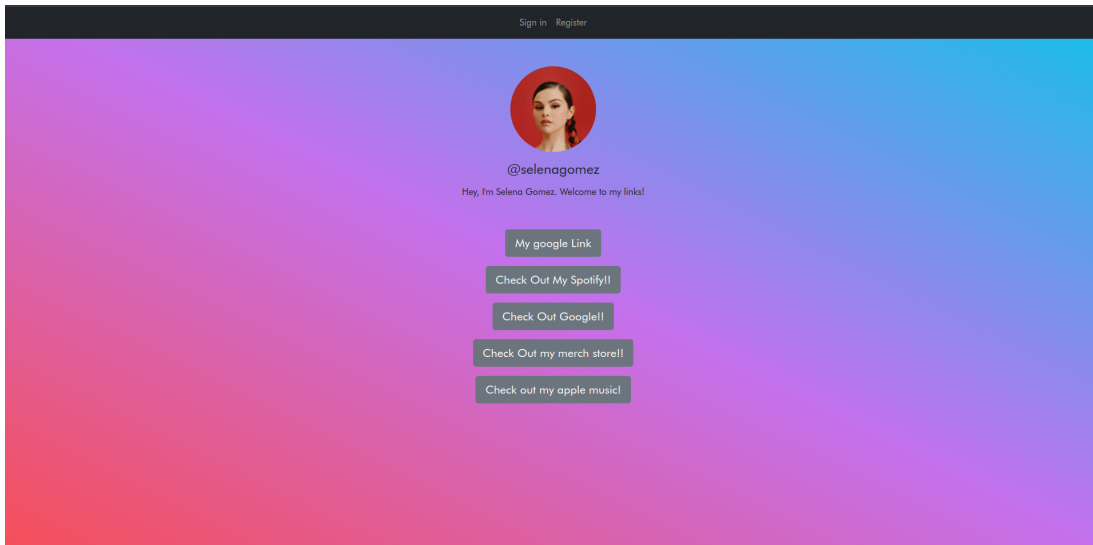


Figure 5.8: Profile page from a visitor's perspective, on a desktop computer.

5.4.2 New user registration

A screenshot of a mobile registration screen. The screen is displayed on a mobile device, with a browser address bar at the top showing '192.168.31.124:3000/user/r'. The background is a gradient of purple and blue. The main heading is 'Sign Up!'. Below the heading are four input fields: 'Email' with the placeholder 'Enter your email', 'Username' with the placeholder 'Enter your username', 'Password' with the placeholder 'Password', and 'Confirm your password' with the placeholder 'Repeat password'. At the bottom, there is a blue 'Sign Up' button and a link that says 'Log in instead'.

Figure 5.9: Registration screen on a mobile device.

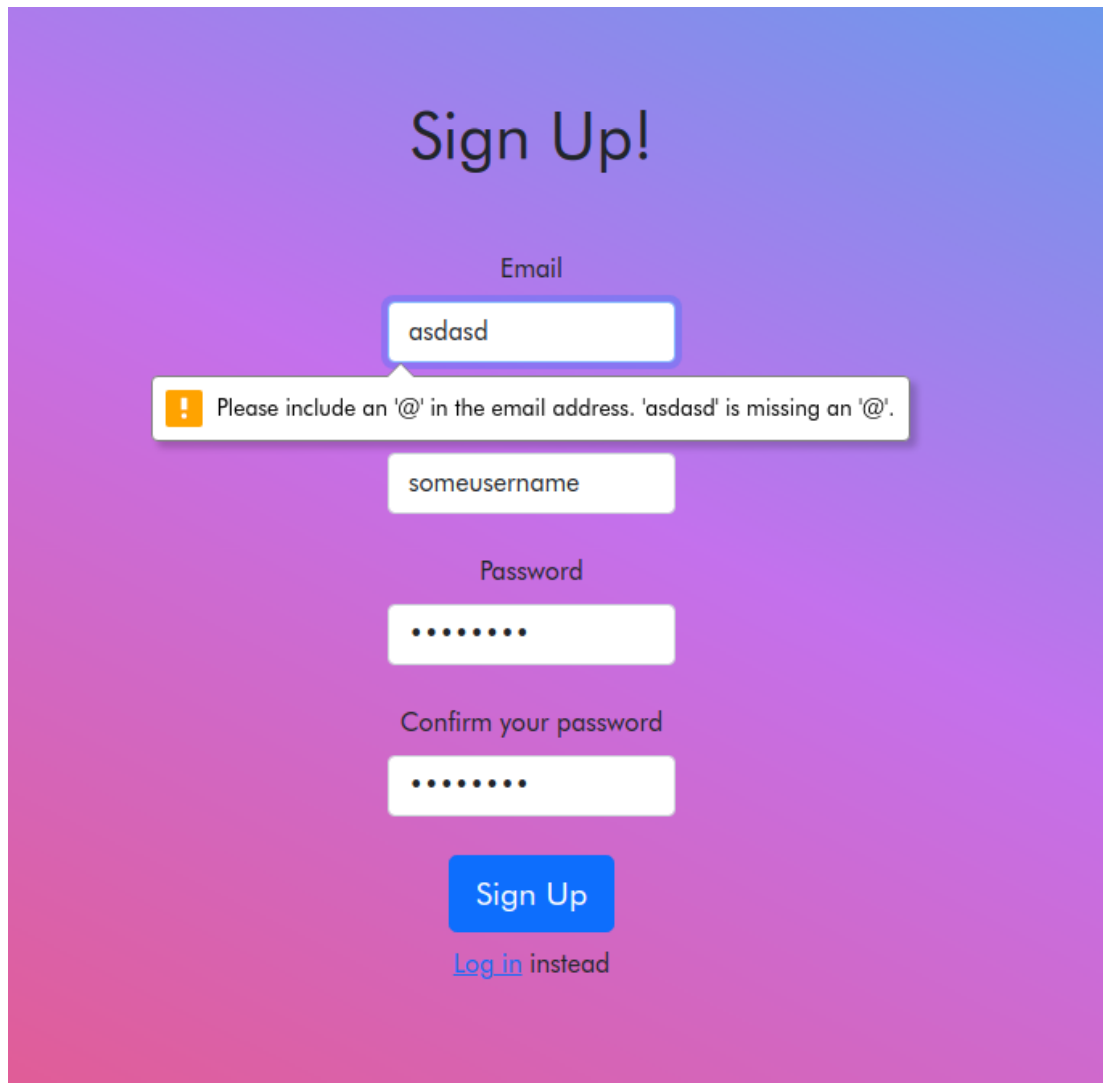


Figure 5.10: Registration screen, on a desktop computer. Email validation failed because it didn't match the email Regular Expression

5.4.3 User's view of his own profile and analytics

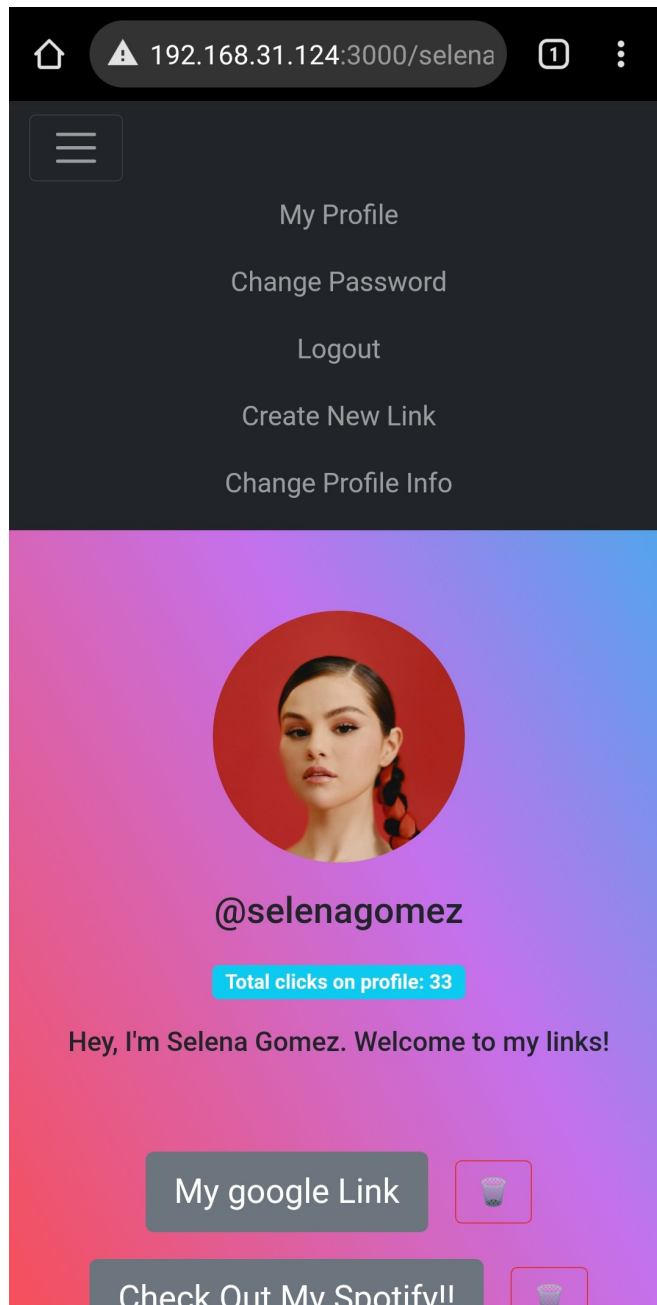


Figure 5.11: Private view of a user profile. Navigation dropdown menu opened.

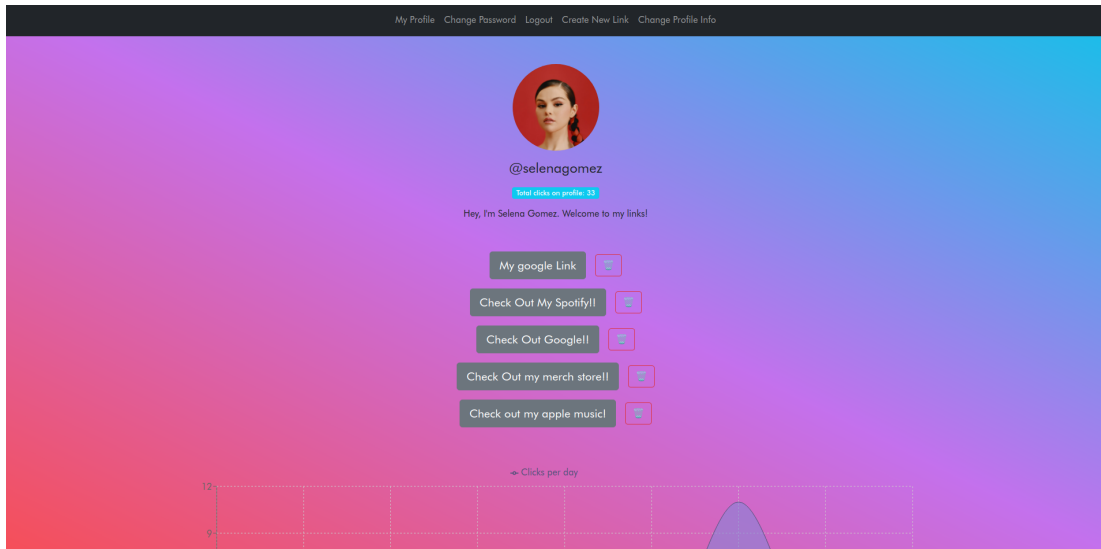


Figure 5.12: Private view of a user profile on a desktop device. All private menu options visible on the navigation bar. First analytics graph visible.

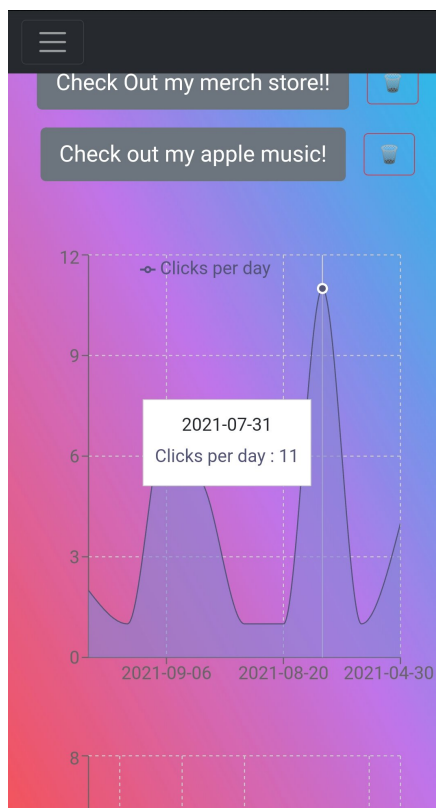


Figure 5.13: Private view of a user profile. Daily clicks interactive graph visible.

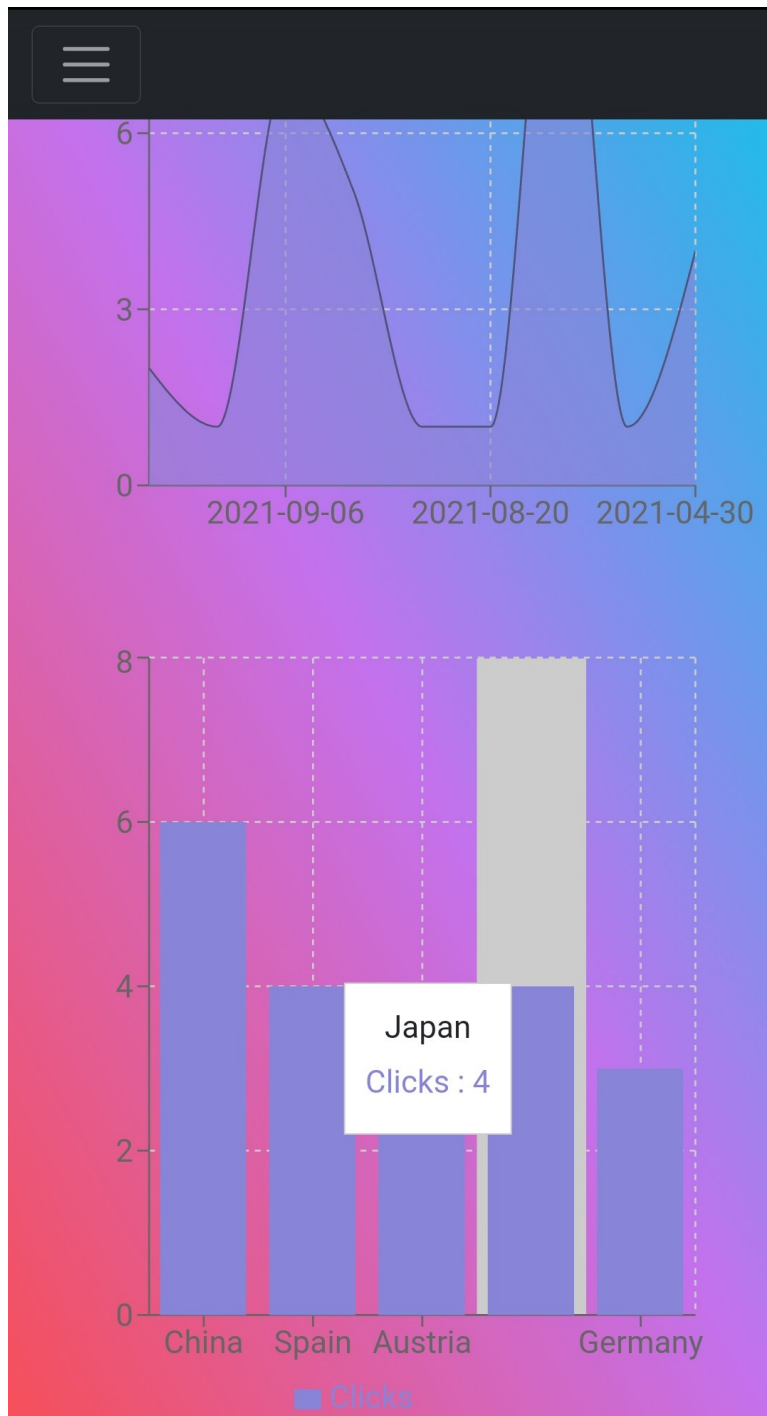


Figure 5.14: Private view of a user profile. Interactive graph showing the top countries that have visited that profile is being tapped on.

5.4.4 Modifying profile information and uploading a new profile picture

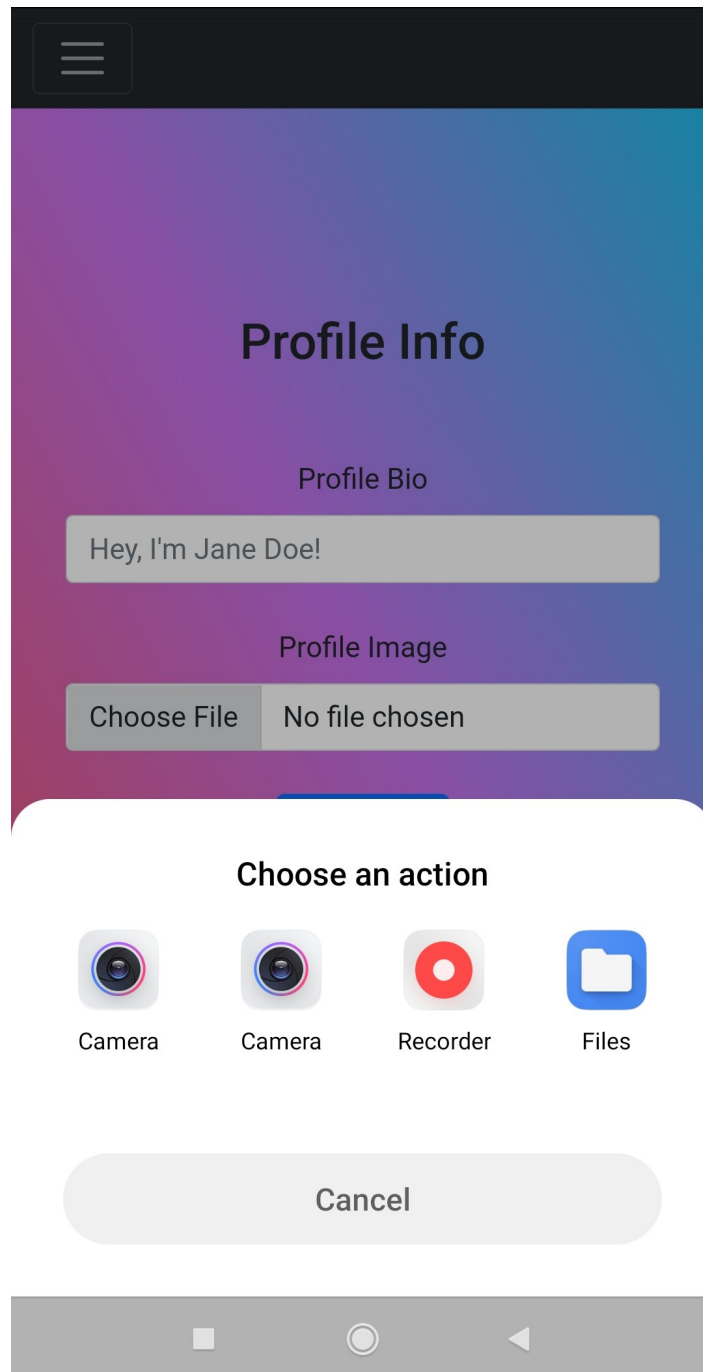
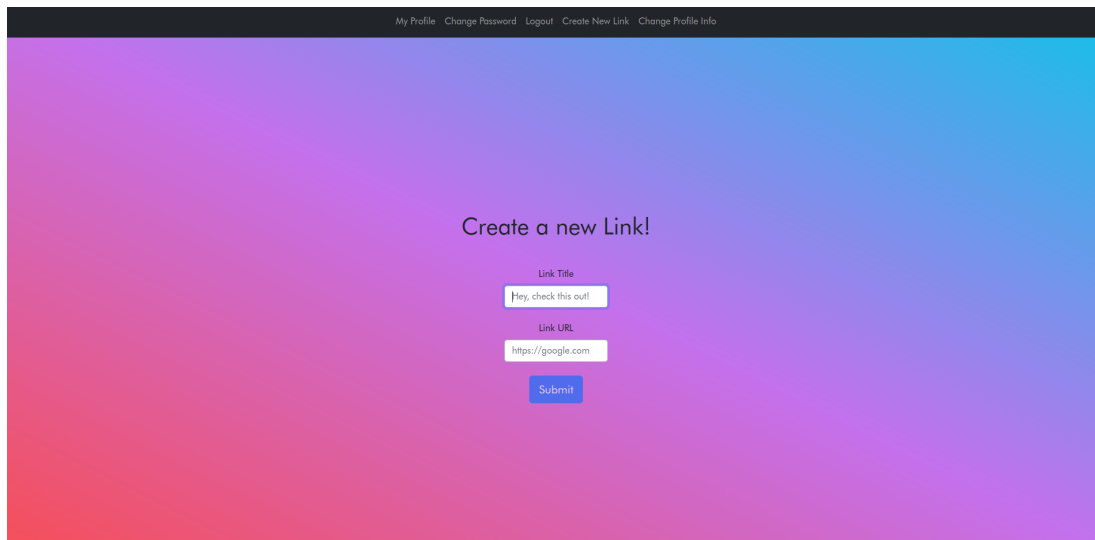


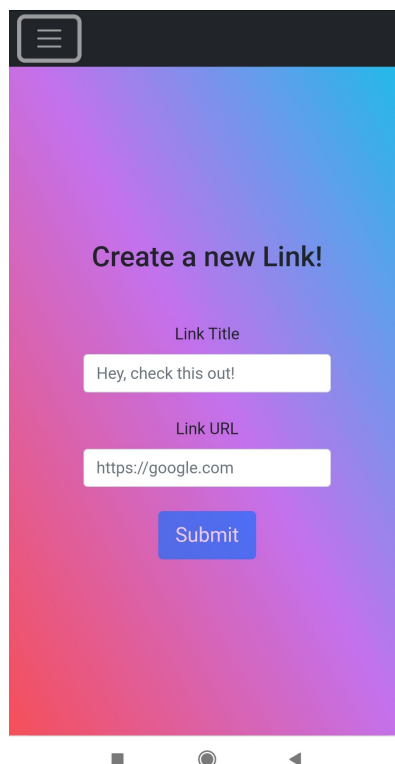
Figure 5.15: File picker popup when selecting the option to upload a new profile picture for a given user.

5.4.5 Creating a new link



The desktop interface for creating a new link features a dark navigation bar at the top with links for "My Profile", "Change Password", "Logout", "Create New Link", and "Change Profile Info". The main content area has a vibrant purple-to-blue gradient background. It displays the heading "Create a new Link!" followed by two input fields: "Link Title" with the text "Hey, check this out!" and "Link URL" with the text "https://google.com". A blue "Submit" button is positioned below the input fields.

Figure 5.16: Desktop interface for creating a new link on your user profile.



The mobile interface for creating a new link is shown on a smartphone screen. It features a dark navigation bar at the top with a hamburger menu icon. The main content area has a vibrant purple-to-blue gradient background. It displays the heading "Create a new Link!" followed by two input fields: "Link Title" with the text "Hey, check this out!" and "Link URL" with the text "https://google.com". A blue "Submit" button is positioned below the input fields. The bottom of the screen shows the standard Android navigation bar with a square, a circle, and a triangle icon.

Figure 5.17: Mobile interface for creating a new link on your user profile.

As shown by these snapshots of the frontend part of the project, running on both Desktop and Mobile devices, we can confirm that the requirements we described earlier in this document were successfully implemented. But not only that. In both mobile and desktop devices, the analytics provided in the charts have smooth animations and allow hovering over the data to provide more information.

User input validation in all 5 forms of the application is done in such a way that is smooth and friendly for the user. Feedback is provided as to which parameter failed the validation, and a smooth popover tells the user what went wrong.

The navigation bar expands and collapses smoothly in mobile devices, while adapting correctly to wider desktop computer screens.

Integration with file browsers for image picking works both in mobile devices as well as desktop devices, allowing the user to choose any type of image to upload to the backend to serve as his new profile picture.

Testing and Continuous Integration

DURING the software engineering process, one of the most essential parts is automated testing. It provides a way to verify that everything is working as intended, in a completely hands free experience to the developer once the tests have been implemented. It is a method that every software engineer should use, to check if a product is defect free, and if it operates in the way it was originally intended in earlier phases of development.

Manual testing of all the REST API Endpoints was also conducted using the Postman tool.

6.1 Testing Tools

Primarily two testing tools were used during testing phases:

- **Jest** was used to implement and run every kind of automated test. From unit tests to full integration test suites. Jest is a testing framework that allows testing Javascript Software, both frontend and backend, in such a way that it behaves almost like natural language.

When using Jest, you first describe what a test *should* do, and then you provide a function to test if it actually performs that action as was expected. Testing is made even easier thanks to the plethora of operators that Jest provides in order to check any conditions, such as errors thrown, truthness of values, assertions...

Jest in its latest version also provides automatic code coverage reports, using the integrated coverage tool from the V8 Javascript Engine called C8. These coverage reports are delivered both in the CLI runs as well as generating a separate coverage folder with HTML reports.

- **Postman** is a great tool that allows software developers to efficiently test and validate the behaviour of REST APIs. It lets developers send requests to a backend, and all the

information associated to those requests and responses. It allows deep customization of the requests, like including explicit HTTP headers, Authentication tokens etc.

6.2 Tests and Coverage

Jest tests suites can be executed by calling `npm jest` or `yarn jest`.

```
> tfg-backend@0.0.0 test
> jest
```

```
PASS test/util/util.test.js
PASS test/models/link.test.js
PASS test/models/user.test.js
PASS test/models/click.test.js
PASS test/services/user.test.js
PASS test/services/auth.test.js
PASS test/services/link.test.js
```

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	100	100	100	100	
models	100	100	100	100	
click.js	100	100	100	100	
link.js	100	100	100	100	
user.js	100	100	100	100	
services	100	100	100	100	
auth.js	100	100	100	100	
link.js	100	100	100	100	
user.js	100	100	100	100	
util	100	100	100	100	
util.js	100	100	100	100	

```
Test Suites: 7 passed, 7 total
Tests: 38 passed, 38 total
Snapshots: 0 total
Time: 2.995 s, estimated 5 s
Ran all test suites.
```

Figure 6.1: Jest test run output via console.

By having the Jest Runner extension installed in our Integrated Development Environment, we can also run Jest tests on demand one by one, or debug them on the fly. We also get helpful hints next to each test, telling us if it passed or failed, and if so, where it failed and why.

```

26 it(name: 'Should create a link', fn: async () : Promise<void> => {
27   const result : any[] = await AuthService.register(user: { username: 'testtest', password: 'testtest', email: 'email@gmail.com' })
28   const link : Document<any, any, any> & { ... } = await LinkService.createLink(link: 'facebook.com', text: 'TitleLink', image: 'asdasdasd', userid: result._id)
29
30   expect(actual: link).toBeInstanceOf(expected: mongoose.Document)
31 }

```

Figure 6.2: Jest testing syntax, as well as integration with testing extension Jest Runner.

As previously mentioned, after every run, Jest generates more advanced coverage reports in the `/coverage/` directory. Inside, we can open `index.html` in order to view the report inside our web browser. In this report, we can see more information such as how many times each line of code was covered.

```

> 24 1x * WARNING: Does not execute on update
53 1x * Also validates passwords before hashing them */
54 1x UserSchema.pre('save', async function save (next) {
55 35x   if (!this.isModified('password')) return next()
56 27x   try {
57 35x     if (this.password.length < 8 || this.password.length > 65) next(this.invalidate('password', 'Invalid password'))
58 27x     const salt = await bcrypt.genSalt(parseInt(process.env.SALT_WORK_FACTOR))
59 28x     this.password = await bcrypt.hash(this.password, salt)
60 25x     return next()
61 1x     /* c8 ignore next 3 */
62 1x   } catch (err) {
63 1x     return next(err)
64 1x   }
65 35x })
66 1x

```

Figure 6.3: Jest coverage report output showing the number of times each line was covered.

6.3 Continuous Integration

Continuous Integration is the process of automating the integration of code changes into a repository where automatic builds and tests occur periodically or when changes are detected.

Continuous Integration reduces the workload of engineers, by making the repository itself be the one to execute tests, as in big software projects tests can take hours to run. This is also helpful in such projects when analyzing pull requests, where the CI process has automatically run and tested the newly proposed code changes.

In our project, we use Continuous Integration by using a Github Actions pipeline configured to run on our repositories. This configuration file is located in the `.github/workflows/` directory of the project, and looks like the following:

```
name: Node.js CI

on:
  push:
    branches: [ master ]
  pull_request:
    branches: [ master ]

jobs:
  build:

    runs-on: ubuntu-latest

    strategy:
      matrix:
        node-version: ['16.x']
        mongodb-version: ['5.0']

    steps:
      - name: Start MongoDB
        uses: supercharge/mongodb-github-action@1.6.0
        with:
          mongodb-version: ${{ matrix.mongodb-version }}

      - uses: actions/checkout@v2
      - name: Use Node.js ${{ matrix.node-version }}
        uses: actions/setup-node@v2
        with:
          node-version: ${{ matrix.node-version }}
          cache: 'npm'
      - run: npm install
      - run: npm test
```

Figure 6.4: node.js.yml Workflow configuration file to use the Github Actions pipeline.

Here, we specify the process to run the testing suites and other checks if necessary.

We configure which branch should be monitored as the main development branch, in this case as we're the only developers, we're directly developing on master. We also tell the Github Actions API which system to run on, in this case the latest version of Ubuntu.

After that, the runtime and database versions are specified, and the steps to enable and run them are executed, which in this case also requires us to install all the NPM dependencies on each run, this of course makes the runs take a bit longer than if they were run on bare metal hardware.

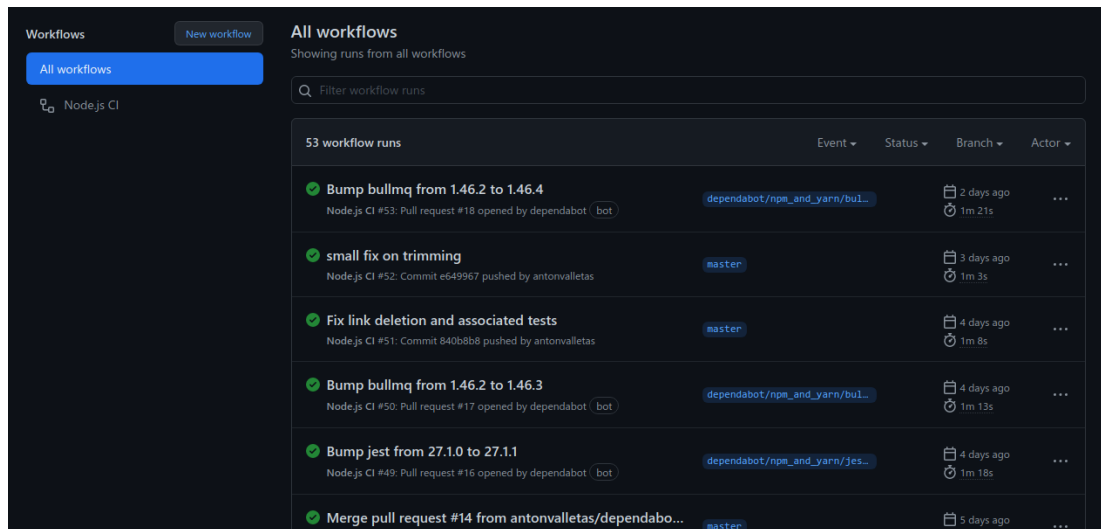


Figure 6.5: In the Workflows tab we can see the state and result of all finished runs of our Node.js workflow. We can also see how much time they took to run, and if they were successful or failed.

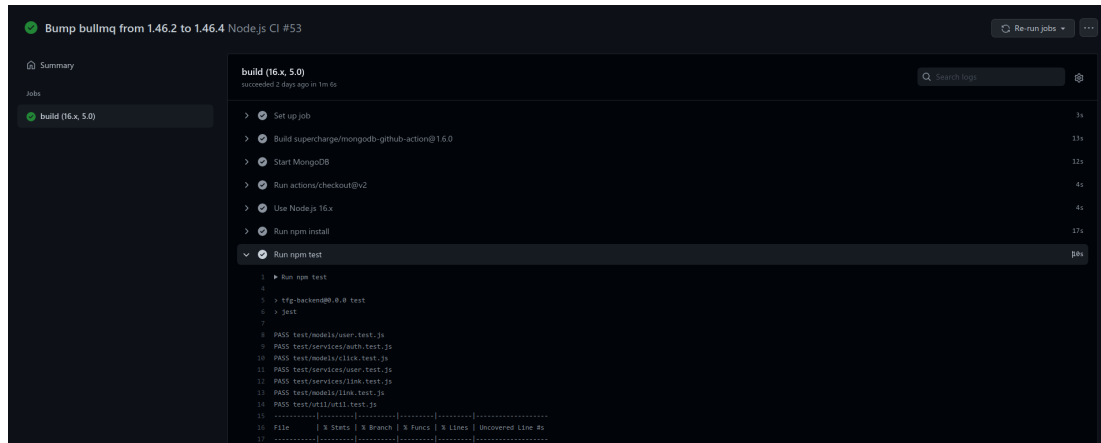


Figure 6.6: For every run, we can see the result of each of the phases of the pipeline, so as to analyze where the pipeline failed if it did.

As previously stated, if there are pull requests that target a branch where there are Workflows running, the source control software will automatically run those workflows against the new proposed changes of that pull request. If changes do not break the workflows configured, merging them is as easy as clicking the *Merge pull request* button.

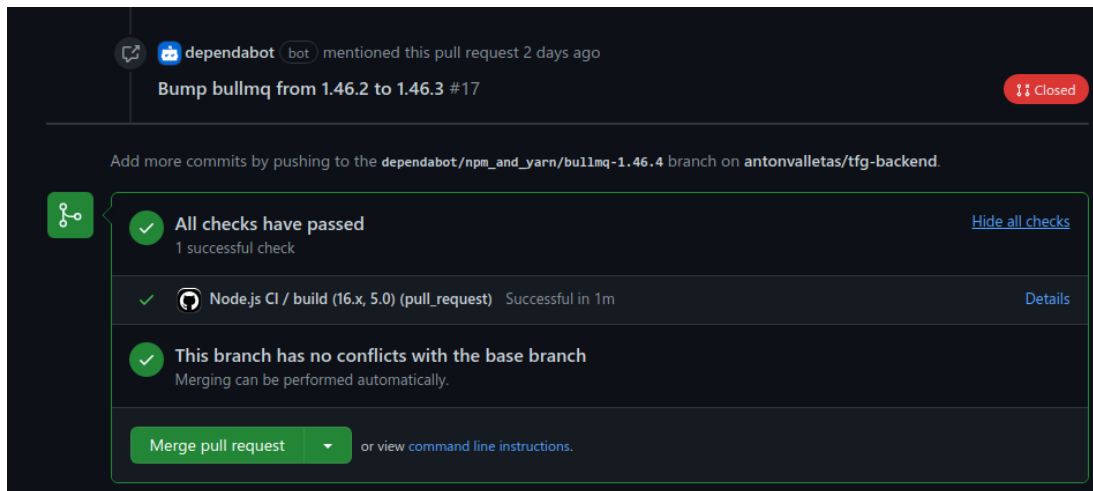


Figure 6.7: Workflows from master branch being run, on a pull request that targets master as destination branch.

Conclusions

IN this last chapter I would like to take the time to reflect upon the things that I have learned from this project. I would also like to do a small retrospective on what objectives were successfully achieved, and what future results could come from this idea.

The basis and main objective of this project as described in its introduction and contextualization is that of providing an easy to use, mobile first approach, to a web application for link management in order for users of link-censoring social networks such as Instagram to use.

As a secondary objective, the project aimed to provide certain insights and analytics that the owner of a user profile in our application could take a look at, to know more about the demographical aspects of his audience, and have a better understanding and knowledge about the growth of his user profile.

With respect to these two, well described objectives, I think both were successfully achieved. Of course, with regards to analytics, one could develop overly complex solutions and insights based on more metadata, but that was not part of the objective. The objective was to provide the most important and simple demographic and growth metrics to the owner of a profile.

I have had the opportunity to work on a full stack software engineering project utilizing a modern stack of technologies. From Javascript, to MongoDB and React. This project was proposed as a means to learn these technologies as well, and I am glad to say that I have greatly increased my knowledge in all these technologies with respect to what I knew about them before starting the project.

With regards to Agile methodologies, I have worked with an automated Kanban board, well integrated into the development process. This has helped me to better understand how a good Kanban-driven software development process should work, and how the methodology should be used by encompassing source control tools.

As a lookback on what the purpose of the project was and why it was needed, I think it was pretty clear that some solution to this problem would be needed sooner or later. As people

usually have more and more social media accounts, linking to them from a place such as Instagram is a growing need, specially from users whose economy is in some way or another based around this social network.

Looking forward in relation to this project, a solution for production deployment using Varnish Cache Server as a caching solution for the backend of the project would greatly increase the amount of traffic the project could handle, as most of the traffic would be just visitors asking for public information of user profiles that could be cached in this way. Moreover, using Varnish Cache Server allows us to explicitly expire a cache copy, for example when a user updates his profile or adds another link.

On a personal note, I don't think this behaviour of censoring hyperlinks in big social networks is over. I think it's just getting started. Every day more social networks are putting up measures for users to stay engaged inside their platform, and these link-banning measures are more commonplace every day. Some social networks will not outright ban them as Instagram has done, but they will start by not making them actionable, or not parsing them so as to not produce big engaging thumbnails for users to follow, as Facebook has recently started doing.

Thanks for having read this far.

List of Acronyms

CSS Cascading Style Sheets.

ERLANG/OTP Erlang Open Telecom Platform.

IGTV Instagram TV.

IPO Initial Public Offering.

MERN MongoDB ExpressJS React NodeJS.

UI User Interface.

URL Unified Resource Locator.

UX User Experience.

Glossary

Backend Usually refers to software that is run server-side, and not client-side..

ECMAScript Standard set by the ECMA Association by which the Javascript language is standardized..

Feed Concept developed by Facebook to deliver content to users in an algorithmically controlled manner..

Frontend Software that is run client-side..

Github A Social Network and Source Control tool based around Git..

Influencer A user of a Social Network who has a big amount of followers..

Jest Testing library for Javascript software developed by Facebook..

Kanban Agile methodology developed by Toyota in the late 1950s..

Nodejs Javascript runtime based on the V8 Javascript engine..

Skeumorphism Term most often used in graphical user interface design to describe interface objects that mimic their real-world counterparts.

Time Series Data Data that clearly follows a chronological order..

V8 Javascript interpreter developed by Google for use in the Chrome Browser..

Visual Studio Code Integrated Development Environment maintained by Microsoft..

Bibliography

- [1] Facebook's initial public offering. [Online]. Available: https://en.wikipedia.org/wiki/Initial_public_offering_of_Facebook
- [2] Instagram timeline. [Online]. Available: https://en.wikipedia.org/wiki/Timeline_of_Instagram
- [3] Kevin systrom admits to copying snapchat. [Online]. Available: <https://techcrunch.com/2016/08/02/silicon-copy/>
- [4] Instagram sells to facebook for 1 billion dollars. [Online]. Available: <https://www.wsj.com/articles/DJFVW00020120409e849k0ssd>
- [5] Monthly instagram users, statista. [Online]. Available: <https://www.statista.com/statistics/253577/number-of-monthly-active-instagram-users/>
- [6] Biggest music streaming platforms, statista. [Online]. Available: <https://www.statista.com/statistics/653926/music-streaming-service-subscriber-share/>
- [7] Atlassian agile methodologies. [Online]. Available: <https://www.atlassian.com/agile>
- [8] Kanban methodology. [Online]. Available: <https://www.atlassian.com/agile/kanban/boards>
- [9] Github project boards. [Online]. Available: <https://www.atlassian.com/agile/kanban/boards>
- [10] Mongodb documentation. [Online]. Available: <https://docs.mongodb.com/>
- [11] Mongodb 5.0 release notes. [Online]. Available: <https://docs.mongodb.com/manual/release-notes/5.0/>
- [12] EcmaScript 2021. [Online]. Available: <https://www.google.com/search?q=ecmascript&oq=ecmascript&aqs=chrome..69i57j0i512l5j69i65j69i60.1577j0j7&sourceid=chrome&ie=UTF-8>

- [13] Reactjs docs. [Online]. Available: <https://reactjs.org/docs/getting-started.html>
- [14] Bootstrap documentation. [Online]. Available: <https://getbootstrap.com/docs/4.1/getting-started/introduction/>
- [15] Redis documentation. [Online]. Available: <https://redis.io/documentation>
- [16] Bullmq task queue documentation. [Online]. Available: <https://docs.bullmq.io/>
- [17] Jest testing framework documentation. [Online]. Available: <https://jestjs.io/docs/getting-started>
- [18] Javascript web tokens website. [Online]. Available: <https://jwt.io/>
- [19] Postman learning center. [Online]. Available: <https://learning.postman.com/docs/getting-started/introduction/>
- [20] Mongoose guide. [Online]. Available: <https://mongoosejs.com/docs/guide.html>