MTAHub: a website to share and discuss software packages

Qais Patankar

Table of Contents

# Motivation

MTA: San Andreas (MTA) is an open-source software project that “incorporates an extendable network play element into a proprietary commercial single-player PC game” (Lucas C. [2009](#ref-AddedMarkdownREADME)). This project “provides a minimal sandbox style gameplay that can be extended through the Lua scripting language” (Lucas C. [2009](#ref-AddedMarkdownREADME)), and these extensions are known as “resources”.

In 2006 a community platform was written in PHP for the MTA community, allowing users to share, vote and discuss resources. In 2006 many web applications did not consider security and user experience, and MTA’s community platform was no exception.

This project modernises and replaces the existing platform with an improved system that abides by these fundamentals.

# Contributions

I was able to rewrite the base functionality of the existing platform, allowing users to create new resources and upload additional versions, with following additional features:

* Compared to the previous system, resources are no longer globally scoped and are scoped per user (sec. [17](#sec:squatting)).
* Resource creators now have greater permissions than users invited as collaborators, preventing collaborators from performing destructive actions.
* Authors can now insert rich text content into their resource description by using Markdown. This allows them to market their resources better.
* Website internationalisation has been replaced with a system that is more powerful and more sustainable (sec. [12.1](#sec:libs-frontend)).
* User interface and user experience has been improved, allowing for greater user retention.
* The user interface supports all screen sizes, allowing for the platform to be comfortably used on mobiles.
* The website has been developed with security in mind, keeping it safe from bad actors.

# Structure

* Chapter 2 discusses and analyses the existing platform, highlighting security vulnerabilities and areas where user experience could be improved.
* Chapter 3 explains and discusses initial project setup and the initial decisions that went into building MTA Hub. This chapter also analyses existing websites relevant to MTA Hub, to help influence design decisions.
* Chapter 4 explores the challenges in implementing MTA Hub, discusses further design decisions, and describes how features were implemented.
* Chapter 5 evaluates the platform in terms of speed, whether it satisfies the non-functional requirements, and summarises user feedback.
* Chapter 6 concludes the work completed and outlines potential future extensions.

# Terminology

The terminology used throughout this report has been chosen as to not be ambiguous, but may not apply when referring to other systems.

* resource: a collection of assets (scripts, images, models, audio files) — “partly equivalent to a program running in an operating system” (Chowdhury and MTA Community [2008](#ref-ResourcesMultiTheft))
* A resource can be of the following types (Lyons and MTA Community [2008](#ref-MetaXmlMulti)):
  + gamemode: provides core game mechanics
  + map: a collection of game world objects, allowing for unique twists on the core mechanics provided by gamemodes
  + script: provide extra features that interact with gamemodes
  + misc: utility resources
* package: a particular version of a resource, when differentiating between versions is important
* server: an instance of the game server (which can run many resources)

These terms describe the stakeholders:

* maintainers:
  + moderator: someone that issues bans or reviews content, keeping the platform safe
  + administrator: a moderator that also manages user permissions
  + sysadmin: system administrator, an administrator that deploys and has full access to the platform
  + contributor: someone who contributes code to this project (does not necessarily have special permissions)
* users:
  + scripter: someone that authors *resources*
  + server owner: a non-technical server owner

The existing platform[[1]](#footnote-24) was written for the MTA community in 2006 by Stanislav Bobrov and other contributors.

# Navigation

All pages on the website, in terms of visual design, follow a consistent template:

* country flags to change translation, when clicked, change the interface’s language for the remainder of the active PHP session
* a sidebar
* the top navigation icons
  + when logged out, has links to the “Home”, “Register”, “Login” and “Resources” pages, and an external link to the game’s forum
* three top navigation icons “groups”, “resources” and “servers”; leading to their respective pages on the platform
* a navigation bar
  + when logged out: has links to “Register” and “Login”
  + when logged in, links to:
    - the user’s profile (“My View”)
    - the admin panel, if they are an administrator (“Admin Panel”:)
    - an external download page (“Download MTA:SA”)
    - the logout page (“Logout (foo)”, with “foo” replaced with the user’s username)

The index.php script generates all pages on the website. GET parameters in the query string indicate the active page, as shown in the deconstruction of the following URL.

/index.php?p=stats&show=vehicles&id=254572:

* p=stats: statistics page
* show=vehicle: show vehicle submodule
* id=254572: show statistics for a user with ID 254572

The p parameter describes the top-level module and each module tends to handle submodules in its own way.

The existing platform originally included support for groups, resources and servers; but today we only use the resources feature. This project does not aim to recreate the groups or servers feature of the website.

# Translation

The existing system includes support for many different languages, making the website very accessible to the international MTA community.

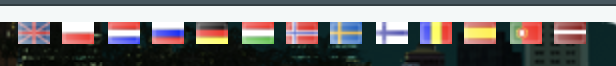


Figure 1: Flags that behave as buttons at the top of each page

Clicking one of the links in fig. [1](#fig:comm1-flags) will switch standard parts the user interface to the language that the user has selected. On the home page, this currently:

* changes:
  + the sidebar links
  + some navigation bar links (excluding “Admin Panel” and “Download MTA:SA”)
* does not change:
  + the top navigation icons
  + the website description and headers

The language selected is saved for the duration of the PHP session, which lasts until the browser is closed or the user’s IP address has changed.

**Implementation and Security**

Each flag is a hyperlink reference to the current page with the set\_lang GET parameter appended. For example, if the user clicks on the German flag when on the resource listing page (?p=resource), they will be directed to the ?p=resources&set\_lang=de page.

On receipt of the set\_lang GET parameter, the backend immediately saves the language in the PHP session, via the $\_SESSION dictionary. This parameter is used directly, without sanitisation, when using include inside PHP scripts. This is an attack vector, as per CWE-22 (MITRE [2020b](#ref-CWE22ImproperLimitation)).

Each language file is a PHP script that adds strings sequentially to an array, as shown below:

<?  
// Main  
$text\_item[1] = 'My View';  
$text\_item[2] = 'Register';  
$text\_item[3] = 'Login';  
$text\_item[4] = 'Logout'; // [...]  
  
// Auth [...]  
$text\_item[32] = 'Username';  
$text\_item[33] = 'Password';  
$text\_item[34] = 'Forgot your password?'; // [...]  
  
// Profile  
$text\_item[51] = 'My view'; // [...]

This is prone to errors and results in a poor user and contributor experience:

* a typographical error can result in the wrong text being shown to users, or even a page failing to load due to script error.
* translation effort can be duplicated, as shown in text item 1 and 51 above
* a contributor building the website will not have a clear idea what text is being shown.
* internationalisation concepts such as pluralisation, number formatting, and date formatting are unsupported.

# Homepage (?p=main)

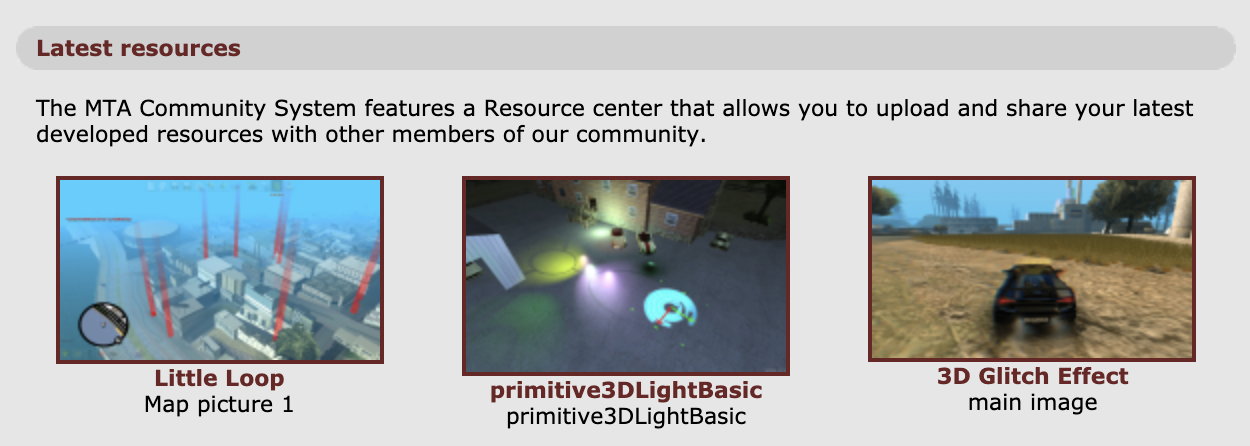


Figure 2: “Latest resources” section on the homepage

As shown in Figure [2](#fig:comm1-resources), the homepage contains a short description of what the system is for and also a link to the resources page titled “Latest resources”. There is also a preview of the 100 most recently uploaded resources underneath. Only resources that have pictures uploaded in the resource gallery are shown.

At the bottom of the page there are also pagination buttons. These pagination buttons simply submit a form that refreshes the page and simultaneously incrementing or decrementing the oset query string parameter by 100.

If the oset query parameter is 0 (or missing, as it is when you initially visit the page), no pagination buttons are shown and only a single “MORE..” button is shown. The oset parameter is not capped, and can go as far as the earliest resource that fits the above criteria.

If the oset parameter goes beyond the number of potentially viewable results, no resources will be shown, and the pagination buttons will still be enabled. This is considered to be poor user experience (UX) as a user may not know that they have reached the end of the list, and may keep clicking “Next” in an attempt to show more items. Currently a user is presented seventy pages, and in practice, a user most likely will not click through all those pages of pictures, so this pagination feature can be considered unnecessary.

# Resources (?p=resources)

This page has several headings:

* Upload resource
* Filter options
* Top Downloads
* Top rated Resources
* Recently uploaded Resources

**Upload new resource**

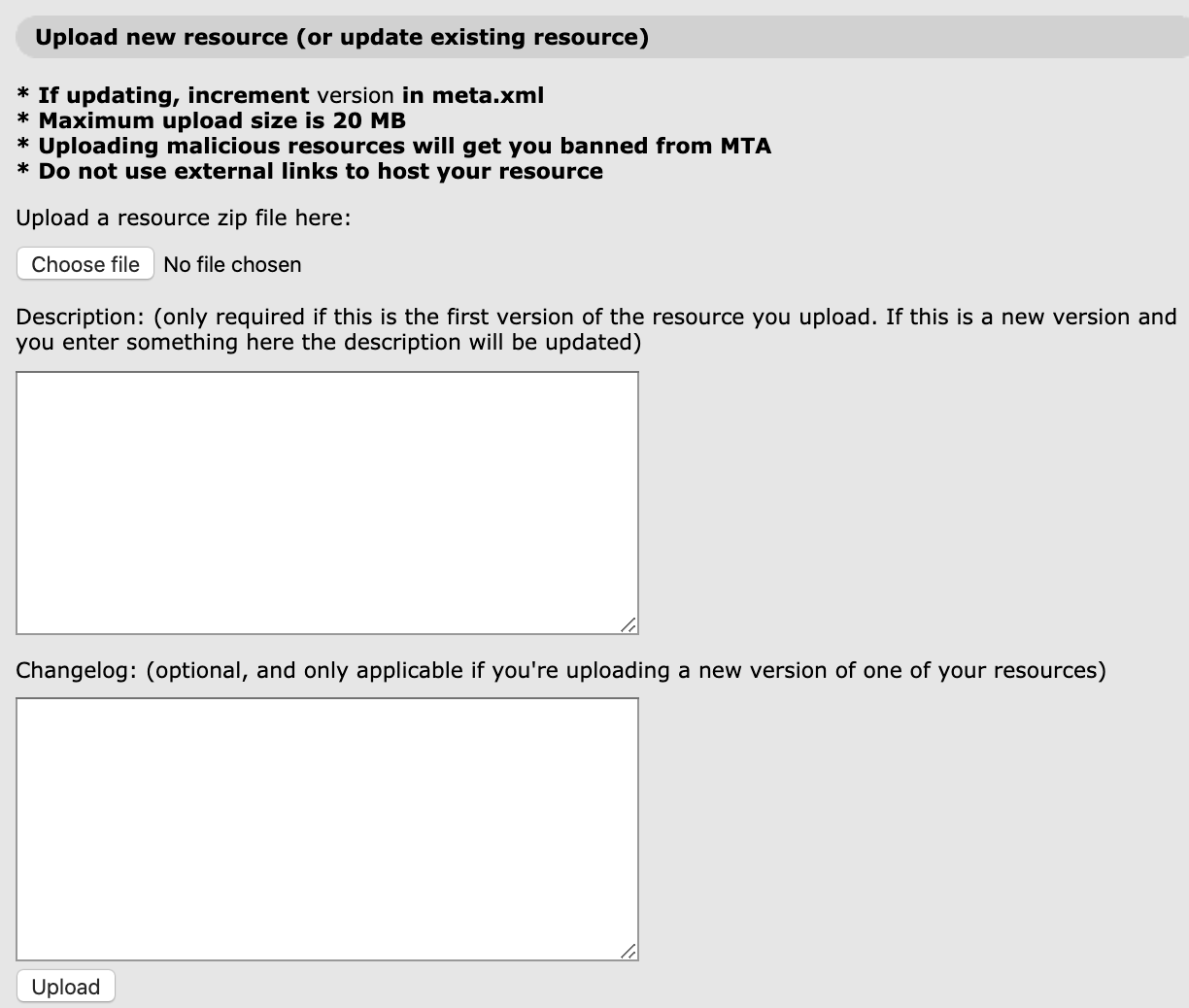


Figure 3: Primary resource upload interface

Underneath the “Upload resource” heading is a singular “Upload!” link. When the user clicks this link, if they are not logged in, they are shown a login panel with a username and password field.

If the user is logged in, they are shown the interface in fig. [3](#fig:comm1-pkg-upload).

This same flow is followed for both uploading brand new resources and updating existing resources. This has several user experience repercussions:

* It is not clear that this upload interface is how you update existing resources. This has been a common question to support staff in the past.
* The existing description should be shown for existing resources, allowing the user to quickly make minor tweaks. Current users find copying-and-pasting the old description too inconvenient and error-prone.
* The changelog field should not be shown if the user is creating a brand new resource.
* Accidental changes to the description can cause distress to the user as previous descriptions are not recoverable.
* In case of an upload error or clash in resource name, the user will lose the text they have submitted.

**Filter options**

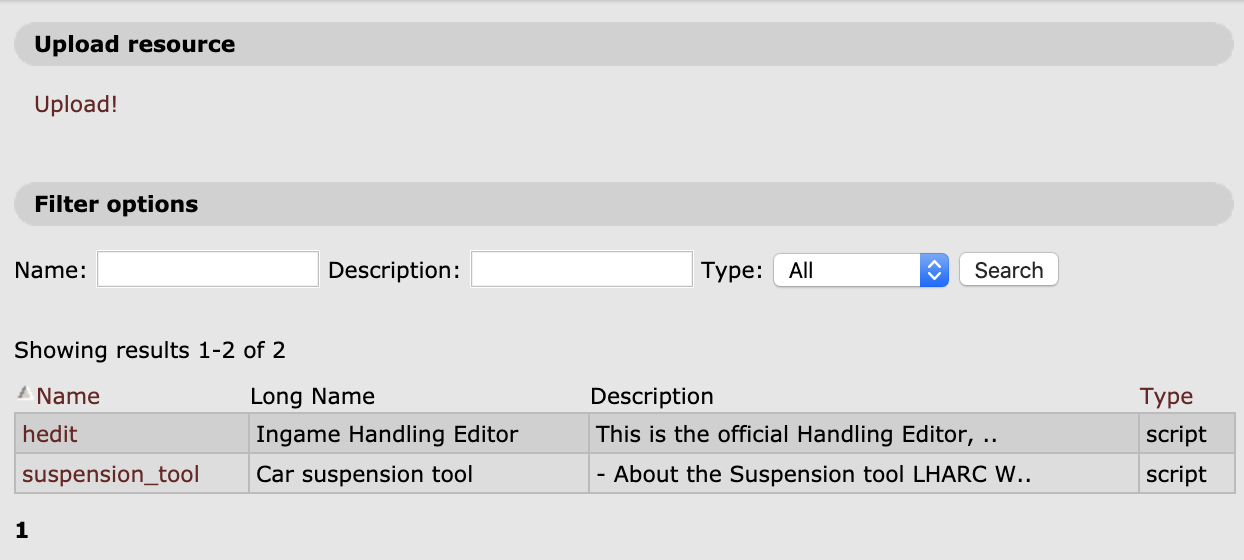


Figure 4: Searching resources for “handling editor” in the description

This section contains three input fields allowing the user to filter by name, description, or type. The type field is a dropdown consisting of the four allowed resources types listed in sec. [4](#sec:terminology).

Pressing the search button will send a POST request to the same page with the search query sent via the submitted form. As shown in fig. [4](#fig:comm1-search), the search results page has:

* the “Upload resource” header
* the “Filter options” header
* a list of search results in a table

This page has several UX problems:

* There is no need to show the Upload resource section - this wastes vital vertical space.
* Input fields are cleared, requiring the user to manually type their query again.
* Because this is a POST request, clicking on a resource and then pressing the back button will result in a browser popup asking the user if they would like to resend the search query request.
  1. There is no need to show the popup as the user will always want to see the search results again.
  2. The user may become confused and choose to cancel the operation.

Additionally, using the POST method in the HTTP request here is unnecessary as no data is being changed. This violates representational state transfer (REST) principles. REST-compliant web services provide “uniform interface semantics – essentially create, retrieve, update and delete – rather than arbitrary or application-specific interfaces” (World Wide Web Consortium [2004](#ref-WebServicesArchitecture)). This is a software architectural style shared by all production-grade APIs.

**“Top Downloads”**

This section shows 15 unsuspended resources, ordered by *download count* in descending order.

This section is quite useful to new users that are server owners - it gives them an idea of the most popular resources since the beginning of the platform. These resources are often quite mature and are easy for server owners to install. However, download count is not an accurate metric for quality, and can simply reflect the community’s historical interest in the resource.

This section is not useful to returning users as the section changes infrequently. This is because long-standing resources are likely to accumulate many downloads over a long period of time - it is unlikely for a new resource to legitimately accumulate more downloads than a resource that was created over ten years ago.

**“Top rated Resources”**

This section shows 15 unsuspended resources with at least ten votes, ordered by *rating* in descending order.

The “Top rated Resources” section attempts to increase effectiveness and prevent abuse by limiting list entries to resources with at least ten votes, but this is still ineffective as the rating system does not take into account the number of votes. The current system will score a resource with six 5-star votes *higher* than a resource with one hundred 5-star votes and two 1-star votes.

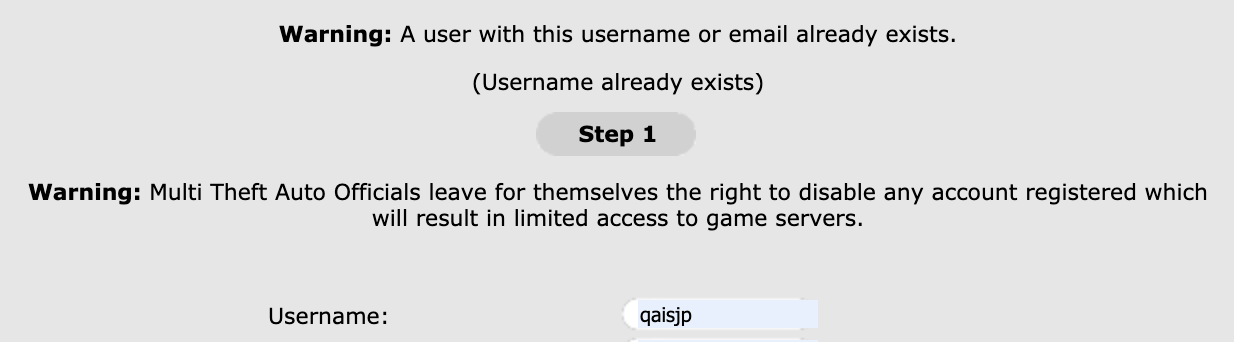
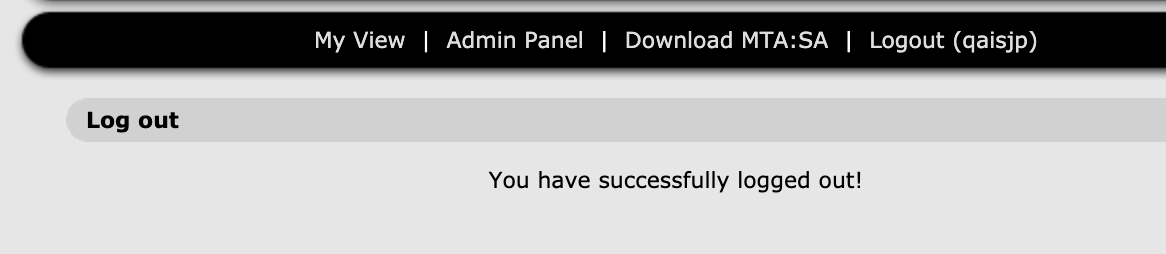
One benefit of the current rating system is that this section is always kept “fresh”, despite being inaccurate.

**“Recently uploaded Resources”**

This section simply lists the resources of the 15 most recent unsuspended packages. This section is not very useful for resource discovery and is susceptible to abuse as a user can upload many packages in succession to flood this list with just their resource.

# Authentication

The authentication process has several UX problems:

* Entering incorrect credentials when logging in will present the user with an error “Invalid user id”, rather than a “Invalid username/password” message.
* Errors during registration shows duplicate “Username already exists” warnings and does not visually stand out, as shown in fig. [5](#fig:comm1-registration-error).
* 
* Figure 5: A warning is shown twice and does not stand out.
* The logout landing page shows links in the navigation bar that should only be shown to logged in users, as shown in fig. [6](#fig:comm1-logout). This is can confuse t he user into thinking that the logout process failed.
* 
* Figure 6: The navigation bar shows incorrect links on the logout page.
* The logout landing page only shows for one second before redirecting to the homepage. This is unexpected behaviour. According to (Wright [2016](#ref-InteractionDesignRedirect)) “if there is text on a redirect screen give the user enough time to engage with the concept of the content, regardless of the time it takes to redirect”.

# Individual Resources (?p=resources&s=details)

Individual resource pages are split into four main sections:

* the header,
* the description,
* calls to action, and
* version history

**Heading**

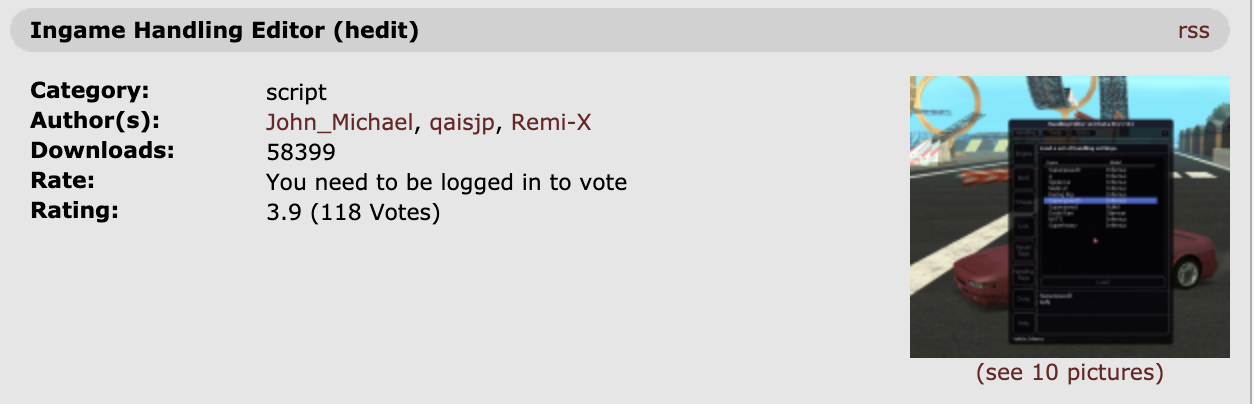


Figure 7: The header of the “hedit” resource

UX issues here include:

* The creator of this resource is the user name Remi-X, but this isn’t clear.
* If the user is logged out, instead of saying “You need to be logged in to vote”, the Rate row should either:
  + be hidden if the user is not logged in
  + be shown, but prompt for a login when a vote button is clicked
* If the user is logged in and they are an author, the Rate row is hidden. This can confuse the author if they would like to tell other users how to vote on their resource. Instead, the vote buttons should be shown, but *disabled* with a tooltip saying “You cannot vote on your own resource”.
* If the user is logged in, not an author, and has already voted - as shown in fig. [8](#fig:comm1-vote-already), there should not be a message reminding the user that they can change their vote. The message is unnecessary as the other vote buttons are still clickable and are not disabled.

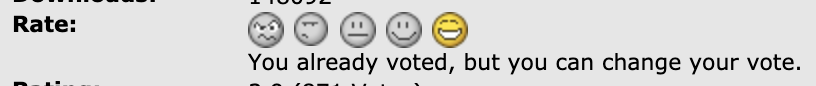


Figure 8: Voting when the user is logged in, not an author, and has already voted.

**Description**

The description field does not support the embedding of rich media or other formatting. To improve user experience, MTA Hub could explore allowing users to insert rich text either through HTML or Markdown.

Links inserted in the description are rendered as invalid links due to poor HTML escaping. This is also susceptible to Cross-Site Scripting (XSS) attacks, which can allow an attacker to “transfer private information, such as cookies that may include session information, from the victim’s machine to the attacker”, as per CWE-79 (MITRE [2020c](#ref-CWE79ImproperNeutralization)).

**Calls to action**

Table [1](#tbl:comm1-rescalls) shows that there are two links available to non-authors (or logged out users) and three available to elevated users (site admins or resource authors).

Table 1: All possible calls to action on a resource

|  |  |
| --- | --- |
| Link name | Elevated? |
| Download latest version | No |
| Upload pictures | Yes |
| Edit resource | Yes |
| Resource log | Yes |
| Report | No |

The edit resource page allows the user to:

* edit the description
* delete previous versions
* add or remove authors
* delete the resource (if admin)
* suspend the resource (if admin)

Resources do not have a concept of “creators” and “collaborators”, only “authors”, and this means that the add or remove authors feature has serious security implications. All authors have equal permissions, meaning that a resource can be hijacked and stolen from its creator by a rogue collaborator. This is a security vulnerability we should resolve in MTA Hub.

Authors should also be able to delete their own resources, especially since they are already permitted to delete all resource packages. One possible reason that this feature was gated to administrators only is to prevent malicious resource authors quickly name-squatting a resource after it has been deleted.

**Version History**

This section is a table with at least four columns, with additional columns shown to administrators: “Version”, “Publish Date”, “Changes” and “Download”.

This list can be very long and most users are only interested in the most recent updates, or updates since the last visit. To improve user experience, we could:

* Show a “see more” button to expand the list of versions shown, if many versions have been uploaded.
* Track what resources the user downloads, and highlight versions that have been published since the user downloaded the resource.

For administrators, an additional “Contents” link is shown. This leads to a page that shows the meta.xml file of the resource and the contents of each script that has been included. If the script is compiled to bytecode, the backend will decompile the script and present the administrator with the decompiled version of the script, for auditing purposes.

To improve user experience, we should present a list of scripts names in the form of hyperlinks, instead of requiring the user to scroll through the entire page. Additionally, the user should be able to collapse individual scripts that have been reviewed.

We should also make this “Contents” feature available to all users, and not just administrators. Only administrators would be able to access the decompilation feature. Since opening large zip files in memory can put unnecessary stress on our backend, we should support client-side downloading and unpacking.

In this chapter, to help influence our design decisions, we first analyse existing websites relevant to MTA Hub. We then explain and discuss the initial decisions that went into building this system.

# Existing software hosting websites

We discuss a number of the following features for each site:

* **API** Does the website have an API?
  + **Architecture.** Is the API and/or website REST-compliant? What is the URL scheme like?
  + **Use of public APIs.** Does it use its own public API?
* **Public access?** Does the API and/or website allow public access?
* **SPA.** Is the website a single page application?
* **Naming.** How does the system handle name clashes?

A single page application (SPA) is “a website whose current page is updated dynamically rather than being entirely downloaded from a server” (Vu [2019](#ref-PWAVsSPA)). These applications typically function by requesting data from a web API that has been built separately to the frontend.

## Npm Registry (npmjs.com)

Npm is the package manager for Node.js, which is a “JavaScript runtime built on [Google] Chrome’s V8 JavaScript engine” (Rogers [2015](#ref-NodeJs)). npmjs.com is a single page application built using React, which is a “JavaScript library for building user interfaces” (React [2018](#ref-ReactJavaScriptLibrary)).

The website does not have a public API, so we navigated through several pages on the website and analysed the network requests that were made.

When the user visits https://www.npmjs.com/package/leftpad the page response simply contains an application bundle - a set of scripts - so that the web application can be rendered in-browser:

“The page is fundamentally empty, but it includes a couple JS scripts. Once the browser downloads and parses those scripts, React will build up a picture of what the page should look like, and inject a bunch of DOM nodes to make it so. This is known as *client-side rendering*, since all the rendering happens on the client (the user’s browser).” (Comeau [2020](#ref-PerilsRehydration))

The web application retrieves JSON data by making a web request to the same url as in window.location (the browser’s current page). When visiting https://www.npmjs.com/package/leftpad, to make the backend return a JSON response instead of a page containing the application bundle, the frontend sets the X-Spiferack header with 1 as a value. As denoted by the X- prefix, this header is non-standard. Additionally, the X-Spiferack header is not used in any open source software, so its meaning is only known to the Npm Registry developers. A semantically correct way to implement this feature would:

* have the frontend set the Accept header of the request to application/json, and
* have the web server read this header to determine whether it should respond with a JSON payload.

Packages on npmjs.com were initially only globally scoped, making packages susceptible to name conflicts. Npm version 2 introduces support for scopes, which “allows you to create a package with the same name as a package created by another user or Org without conflict”. (npm [2014](#ref-ScopesNpmDocumentation))

## Rust Package Registry (crates.io)

The Rust Package Registry website is a single page application that uses the Ember.js web framework[[2]](#footnote-51). All packages are globally scoped.

The website uses its own REST-compliant public API which is available at https://crates.io/api.

Clicking on the “rustfm” package on the search results for “lastfm” triggers several API calls, listed in tbl. [2](#tbl:crates-publend). The user is presented with a “Loading…” indicator blocking the entire page when all this information is being fetched, despite the user not necessarily needing to know all this information.

Table 2: Public endpoints on https://crates.io/api/v1/crates/rustfm

|  |  |
| --- | --- |
| Path | Description |
| / | Crate metadata, statistics, links and versions |
| /versions | Version metadata and stats (the same data is included above) |
| /0.1.2/authors | Author names and email addresses |
| /owner\_user | Profile data for the users that own the crate |
| /owner\_team | Profile data for the teams that own the crate |
| /0.1.2/dependencies | ID and versions of the crate’s dependencies |
| /downloads | Statistics about individual download entries |

This pop-in effect results in poor user experience, and can be resolved in one or more of the following ways:

1. Using a single endpoint to return all the data necessary.
2. Rendering data as soon as it is available, and showing a seamless loading indicator for information that is still being fetched.
3. Lazy loading statistics - only fetching this data when the user scrolls down to the statistics section.
4. Enabling HTTP/2 for the API, so that requests can be multiplexed into a single connection.

Table 3: Private endpoints on https://crates.io/api/private/session

|  |  |  |
| --- | --- | --- |
| Method | Path | Description |
| DELETE | / | Logout the authenticated user |
| GET | /begin | Initiate authentication process using GitHub OAuth |
| GET | /authorize? code=<code>& state=<state> | Complete authentication process using GitHub OAuth, returning session information |

Table [3](#tbl:crates-privend) shows that the only private endpoints are relating to third-party authentication. If we implement third-party authentication into our platform, we should also make it clear that those endpoints are private.

Sobers ([2012](#ref-WhatOAuthDefinition)) says:

“OAuth is an open-standard authorization protocol or framework that provides applications the ability for “secure designated access.” For example, you can tell Facebook that it’s OK for ESPN.com to access your profile or post updates to your timeline without having to give ESPN your Facebook password."

OAuth allows users to log into the Rust Package Registry using their GitHub credentials, without giving the Rust Package Registry their GitHub credentials.

## GitHub (github.com)

GitHub has two main APIs, REST API v3 and GraphQL API v4, and also supports push messages via “webhooks.”[[3]](#footnote-55)

Their documentation describes why GitHub uses GraphQL:

“GitHub chose GraphQL for our API v4 because it offers significantly more flexibility for our integrators. The ability to define precisely the data you want—and only the data you want—is a powerful advantage over the REST API v3 endpoints. GraphQL lets you replace multiple REST requests with a single call to fetch the data you specify.” (GitHub [2016](#ref-GitHubGraphQLAPI))

This has clear development and performance benefits:

* we can use our own API more aggressively (the benefits of using our own API is discussed in sec. [13](#sec:design-api))
* a single request can be called for all the data needed, which results in faster load times for the user
* less *contributor* time needs to be spent creating endpoints to provide bulk data

Despite GraphQL being very powerful, we decided to build a simpler REST-compliant API. This is because GraphQL is quite difficult to set up and prototype with, especially when fleshing out a brand new API.

**Naming**

All repositories on GitHub are scoped to a user account or organisation - this means that two users can create a repository with the same name.

Their URL design follows https://github.com{/owner}{/repo}, replacing {/owner} with the username or the owner, and {/repo} as the repository name. This has the benefit of making it clear to all users, at a glance, who owns the repository.

We have chosen to follow a similar model in MTA Hub, as discussed in sec. [17](#sec:squatting).

# Technology Stack

A technology stack “are the frameworks, languages, and software products”(Nevogt [2019](#ref-WhatTechnologyStack)) that developers use to build an application. In this section we give a high level overview of the technologies we used to build MTA Hub and give an overview of how all of these technologies fit together.

## Web application frameworks

In sec. [11](#sec:bg-api-analysis) we evaluated a number of existing APIs; based on that information we decided to adopt a REST architectural style in designing our API.

We chose to build a single page application as SPAs are typically easy to transform into *progressive web applications*, which “deliver native-like capabilities, reliability, and installability while reaching *anyone, anywhere, on any device* with a single codebase” (Richard and LePage [2020](#ref-WhatMakesGood)).

Of Ryan Donovan’s *Top 10 Frameworks* ([2019](#ref-Top10Frameworks)), React, Angular and Vue were the top frontend frameworks that support single page applications.

We did not have any experience with Vue, but had worked with Angular and React before, so we did not consider Vue for this project. Despite React being more popular than Angular (Romanyuk [2019](#ref-AngularVsReact)) we chose Angular as it is “a full-fledged framework for software development, which usually does not require additional libraries” (Romanyuk [2019](#ref-AngularVsReact)). This allows us to develop rapidly and lets us focus on building the application, compared to React which “is unopinionated and leaves developers to make choices about the best way to develop” (TechMagic [2019](#ref-ReactVsAngular)).

We were particularly drawn to Angular’s in-built ng command line interface, which:

* gives us a standard project structure to use from the start which decreases developer overhead and makes it easy for new contributors to join the project.
* can generate new modules, components, services and directives quickly.
* automatically generates foundational tests, which encourages test-driven development.

Angular also had an inbuilt internationalisation library which we could use to handle pluralisation, string extraction, number formatting, and date formatting. According to Angular’s guide, “*Internationalization* is the process of designing and preparing your app to be usable in different languages” (Google [2018](#ref-AngularInternationalizationI18n)).

### Angular concepts

Angular has the following concepts:

* Components: functional views which have associated TypeScript have HTML templates and CSS styles associated with them, and are included in other HTML templates.
* Directives: extend the templating system, making it possible to add new HTML attributes.
* Services: persistent single classes that are shared between components.
* Modules: which can import other modules, declare components & directives, and export components & directives.

All of these concepts are implemented as TypeScript classes. Components and Directives can only interact with each other through services, and these services are *injected* into the class via the constructor. This process is known as *dependency injection* and it enhances software testing by allowing “dependencies to be mocked or stubbed out” (Thiago [2008](#ref-DesignPatternsWhat)).

### CSS frameworks

We wanted a CSS framework that had the following properties:

* **responsive**: so that we can support mobile and desktop screen sizes
* **utility classes**: makes it easier to rapidly prototype the interface
* **flexbox**: provides “a more efficient way to lay out, align and distribute space among items in a container” (Coyier [2020](#ref-CompleteGuideFlexbox))
* **Angular support**: the framework should work with Angular

We were initially drawn to Bootstrap as it satisfies all the constraints and we had previous experience using the framework. We discovered that Bootstrap 4 uses jQuery, which is a dependency that does not align with the declarative nature of Angular, and can cause bugs if used improperly.

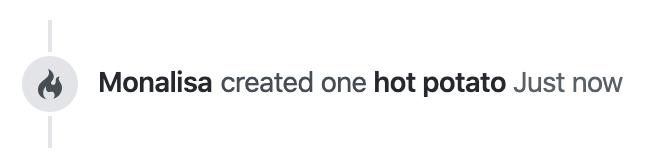


Figure 9: “The TimelineItem component is used to display items on a vertical timeline, connected by TimelineItem-badge elements.” (Rohan [2019](#ref-TimelinePrimerCSS))

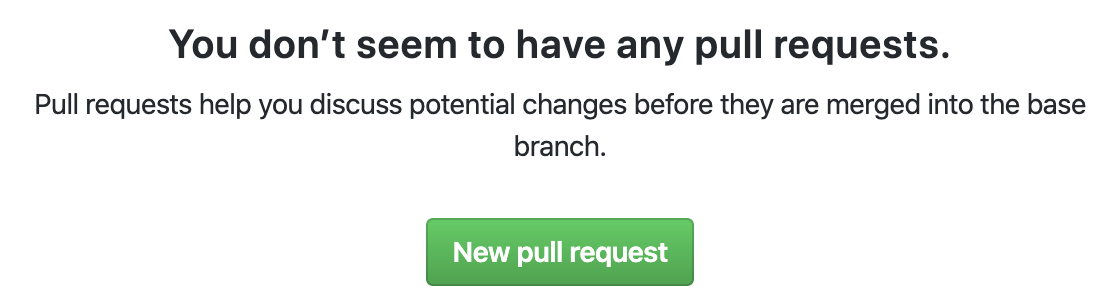


Figure 10: “Blankslates are for when there is a lack of content within a page or section. Use them as placeholders to tell users why something isn’t there.” (Bemis [2019](#ref-BlankslatePrimerCSS))

For this reason, we chose Primer - GitHub’s open-source design system. Since GitHub is similar to MTA Hub in that both platforms allow users to upload software archives, we could also leverage Primer’s additional inbuilt components that are missing in Bootstrap, as shown in fig. [9](#fig:primer-timeline) and fig. [10](#fig:primer-blankslate).

## Backend

We had three main reasons for choosing to build our API in Go:

1. Go compiles down to a single statically linked binary, making it easy to deploy.
2. Go is highly performant compared to the other languages we considered, while also being memory safe, type safe and mostly free of undefined behaviour.
3. Go does not slow the developer down: we can write code quickly, that code builds fast, and its runtime behaviour is predictable.

Go’s webserver is multithreaded by default which improves its performance, but also makes it susceptible to Go’s only undefined behaviour: *race conditions*. We can make use of Go’s inbuilt concurrency primitives - *channels* - to alleviate this problem[[4]](#footnote-65).

We also had the most prior experience writing software in Go, making the language an obvious choice.

Other languages we considered were:

* Python, which lacks many of the above properties. It is interpreted and weakly typed, resulting in a lot of runtime overhead, and therefore poor performance. While it is the fastest to write and does not need compiling, the lack of static typing makes it susceptible to runtime errors which could easily be resolved at compile time.
* Rust can be statically linked and has better performance than Go, but it is slow to write and compile, making it difficult to build and experiment with.
* C++ has the best performance but is not memory safe or free of undefined behaviour. These two properties would make the backend at risk of being vulnerable through memory exploits. A lot of developer time would be spent being extra careful to prevent the introduction of security vulnerabilities. Compile time is also an issue — “Go is significantly faster to compile over C++” (Scully [2020](#ref-GoVsComparison)).

### Database

Storing information in a database allows us to persist data across multiple independent requests. We chose to store our data in a PostgreSQL database as

* Our data is relational, making SQL the better choice over NoSQL. (Wodehouse [2019](#ref-SQLVsNoSQL))
* Compared to MySQL, PostgreSQL provides additional types (such as enumerations), case insensitive pattern matching (the ilike operator), and other extra features. (Hristozov [2019](#ref-MySQLVsPostgreSQL))
* The existing system already uses PostgreSQL so it will be easier to migrate to a new database schema than a completely different type of database.

## Deployment

Deployment is an important part of creating a web platform — merely building an application is insufficient, it should be easy to deploy locally (for development) as well as to deploy in the production environment.

To make it easier for contributors to develop, we use Docker to create a consistent development environment for the backend. Docker allows us to “easily pack, ship, and run any application as a lightweight, portable, self-sufficient container, which can run virtually anywhere” (Vaughan-Nichols [2018](#ref-WhatDockerWhy)). This gives us the following advantages:

* Potential contributors can quickly “spin up” a local instance of the website and can make improvements quickly.
* A seasoned contributor can spend more time developing, and less time helping new contributors get started.
* Our project will automatically be supported on operating systems that are not POSIX-compliant, and we can spend less time supporting those non-compliant operating systems.
* Portable Operating System Interface for Unix (POSIX) is “a set of [IEEE] standards that define how to develop programs for UNIX (and its variants)”. Windows is one example of a non-POSIX compliant operating system.

We intend for this same Docker image to be used in production, keeping the development environment as close to the production environment as possible. This helps keep environment or configuration-related bugs to a minimum.

The Angular command line interface ensures that the tools currently being used match the version defined in the repository’s package.json file, so we do not need to use Docker for the frontend.

TODO: add actual docker config discussion in implementation

In this chapter we explore the challenges in implementing MTA Hub, discuss further design decisions, and describe how features were implemented.

# API

## Versioning

Throughout the lifetime of MTA Hub we may choose to make breaking changes such as deprecating or changing the behaviour of certain API endpoints. We have implemented a versioning strategy that, when we make breaking API changes, “allows clients to continue using the existing REST API and migrate their applications to the newer API when they are ready” (xMatters [2019](#ref-FourRESTAPI)).

All of our endpoints are prefixed with the version number, for example /v1/users to get the list of users. In the future, if we choose to change how our API is designed, we can root those new endpoints under the /v2 path. Note that this is the same approach that GitHub and the Rust Package Registry have implemented, as discussed in sec. [11](#sec:bg-api-analysis).

## Endpoints

TODO: List of endpoints

Full list of endpoints implemented000

|  |  |
| --- | --- |
| Path | Elevated? |
| Download latest version | No |
| Upload pictures | Yes |
| Edit resource | Yes |
| Resource log | Yes |
| Report | No |

## Consuming our own API

For the following reasons, our frontend uses MTA Hub’s own public API wherever possible:

* It will ensure we “spend the time to architect and proof out an API that will stay around for a while” and prevent us from “updating the API frequently” (Evers [2012](#ref-eversShouldWebsiteUse2012)).
* Dante says “Unless the performance overhead of using the web service is an issue, you should definitely use your public API. This will help you get a consistent behavior between your application and the consumers. It will also avoid code duplication […]” (Dante [2013](#ref-danteArchitectureHowMy2013))

Since our API is REST-compliant and the endpoints are organised specific to *entities*, and not to specific webpages on the frontend, some webpages on the frontend will need to make multiple requests to MTA Hub’s API to display all the information necessary.

Making multiple API requests can cause a webpage to take a while to load. For example, to fetch the user’s profile we would need to first request /v1/users/qaisjp to check that the user exists and get their basic information, and then query /v1/users/qaisjp/resources to list their resources.

To avoid making extra requests and to improve performance, we have chosen implement a number of *private* endpoints for internal use only. This will allow us to “make data access more performant by using the database directly instead of doing extra requests” (Virkkunen [2010](#ref-virkkunenApiDesignIt2010)). These internal endpoints live at /private/ instead of the regular /v1/ prefix, discouraging those that reverse engineer our webapp from building software using these endpoints.

## Permissions

We decided that permissions will be granular for site administrators, but kept simple on a resource level. Original creators of a resource retain permanent access rights to a resource and can also designate additional resource administrators. These designated resource admins have all the same permissions as the creator.

TODO: why?s

## Error handling

We initially returned all errors to the user, even internal errors. This is a security weakness, specific CWE-209 “an error message that includes sensitive information about its environment, users, or associated data” (MITRE [2020a](#ref-CWE209GenerationError)). Instead, we should simply log internal errors, and if those errors are fatal, we should have a standard response with:

* HTTP status code 500, which represents “Internal Server Error”, and
* a standard JSON result

Listing lst. [1](#lst:api-somethingWentWrong) shows our simplification of error handling by creating a helper function API.somethingWentWrong(...), and shows how we use that function inside API.getCurrentUserProfile:

1. This function shows how we implemented our helper function.
   1. Declare API.somethingWentWrong, that takes a HTTP context (\*gin.Context) and an error handle (error), and returns a log entry.
   2. Abort the request with status code 500 and return the following JSON:
   * { "message": "Something went wrong" }
   1. Return a log entry with the given error as a context
2. This function shows how we used our helper function.
   1. Access the database to get the user’s followers, which is an operation that can fail.
   2. Check whether the operation failed.
   3. Call API.somethingWentWrong, which mutates the HTTP request and returns a log entry.
   4. Annotate the return log entry with the user’s ID, to help aid debugging.
   5. Print the log entry using the “Error” level to the server log.

Listing 1: Helper function to log fatal internal errors with an example.

// (1a) Declaration  
func (a \*API) somethingWentWrong(  
 ctx \*gin.Context,  
 err error,  
) \*logrus.Entry {  
 ctx.AbortWithStatusJSON(http.StatusInternalServerError, gin.H{  
 "message": "Something went wrong",  
 }) // (1b) Abort Request  
 return a.Log.WithError(err) // (1c) Return Log Entry  
}  
  
func (a \*API) getUserFollowers(ctx \*gin.Context) {  
 user := ctx.MustGet("user").(\*User)  
 rows, err := user.GetFollowers(ctx, a.DB) // (2a) Access Database  
 if err != nil { // (2b) Check Error  
 a.somethingWentWrong(ctx, err). // (2c) Abort Request, Get Log Entry  
 WithField("user\_id", user.ID). // (2d) Annotate Log Entry  
 Errorln("could not get this user's followers") // (2e) Print Error  
 return  
 }  
 ctx.JSON(http.StatusOK, rows) // (2f) Success  
}

## Selecting HTTP status codes

HTTP status codes are a “3-digit integer result code of the attempt to understand and satisfy [a HTTP] request” (World Wide Web Consortium et al. [1999](#ref-HTTPStatusCode)).

**401 Unauthorized vs. 403 Forbidden**

The status code 401 is means “Unauthorized”, despite the code being used for *authentication* and not *authorisation*. This means it’s important to carefully selected status codes for certain scenarios.

* 401: being unauthenticated for a request that requires authentication (Mogul et al. [1997](#ref-HypertextTransferProtocola))
* 403: being authenticated but not authorised to perform an action (Mogul et al. [1997](#ref-HypertextTransferProtocola))

Okta ([2018](#Xe99650e9eb31952c89cddbd2b91359acfb9e29e)) describes authentication as “the act of validating that users are who they claim to be”, and authorisation as “the process of giving the user permission to access a specific resource or function”.

(Irvine [2011](#ref-irvineUnderstanding403Forbidden2011)) says:

“Receiving a 401 response is the server telling you, ‘you aren’t authenticated–either not authenticated at all or authenticated incorrectly–but please re-authenticate and try again.’”

“In summary, a 401 Unauthorized response should be used for missing or bad authentication, and a 403 Forbidden response should be used afterwards, when the user is authenticated but isn’t authorized to perform the requested operation on the given resource.”

**Processing uploads**

When processing uploads we return the following HTTP status codes:

* For any unexpected errors, we return status code 500 Internal Server Error.
* If the HTTP request is encoded in a way that does not support file uploads, we return the status code 415 Unsupported Media Type.
* We return the status code 422 Unprocessable Entity in the following scenarios:
  + The file is missing.
  + The file is not a ZIP file.
  + We check that the file type by ensuring that the Multipurpose Internet Mail Extensions type (MIME type) of the uploaded file is application/zip.
  + We have processed the ZIP file and have determined the resource to contain invalid metadata.
* We use 422 Unprocessable Entity because “the server understands the content type of the request entity (hence a 415 Unsupported Media Type status code is inappropriate), and the syntax of the request entity is correct (thus a 400 Bad Request status code is inappropriate) but was unable to process the contained instructions” (Dusseault [2007](#ref-HTTPExtensionsWeb)).

# Angular

This section describes the high-level design and implementation of our Angular frontend.

## Logging

We created a LogService to handle logging centrally throughout our application. Traditionally logging is handled using the console.log function, but this is not the “best practice for production applications” (Sheriff [2017](#ref-LoggingAngularApplications)).

At the moment our LogService is extremely simple, simply containing functions like the following:

warn(...msg: any) {  
 console.warn(...msg);  
}  
  
debug(...msg: any) {  
 console.debug(...msg);  
}

This logging service can be extended in a production environment to make it easy to share exceptions in real-time with a backend. This is useful as it allows us to “quickly triage and resolve issues [..] by providing cross-stack visibility and deep context about errors” (Sentry [n.d.](#ref-Sentry)).

## Components

Components are functional views written in TypeScript, HTML and CSS. All of our components are declared using the Component decorator. TypeScript decorators, denoted by an identifier with a leading @ symbol, are a “special kind of declaration that can be attached to a class declaration, method, accessor, property, or parameter” (Buckton [2015](#ref-DecoratorsTypeScript)).

Listing 2: Declaration of our ProfileComponent, used to display user profiles.

@Component({  
 templateUrl: './profile.component.html',  
 styleUrls: ['./profile.component.scss'],  
 selector: 'app-profile'  
})  
export class ProfileComponent implements OnInit {  
 // ...  
}

All components we declare only the change the following attributes, an example of which is shown in lst. [2](#lst:ProfileComponent-decl).

* templateUrl is the path to a file containing an Angular template.
* styleUrls is an array of paths pointing to stylesheets that can be used to style this component.
* selector is the custom HTML tag that can be used to display this component.

**Templates**

Angular templates are HTML files that allow users to interpolate strings, manage what HTML tags are rendered, and manage how those tags are rendered.

“Interpolation refers to embedding expressions into marked up text. By default, interpolation uses as its delimiter the double curly braces, {{ and }}.” (Darwin [2017](#ref-AngularTemplateSyntax))

This templating system can be extended by adding new:

* directives, making it possible to add new HTML attributes
* pipes, to make data transformation easier

**Styles**

CSS rules defined during component declaration only apply to that component. This has the advantage of not needing to worry about CSS rules being unique between components. Angular achieves this by using a shadow document object model (shadow DOM), described below:

“An important aspect of web components is encapsulation — being able to keep the markup structure, style, and behavior hidden and separate from other code on the page so that different parts do not clash, and the code can be kept nice and clean.” (Mills [2018](#ref-UsingShadowDOM))

## Dates and Times

Our API converts formats date/time objects as string *timestamps* using the format as defined in RFC 3339. RFC 3339, “Date and Time on the Internet: Timestamps”, is a specification document that “defines a date and time format for use in Internet protocols” (Newman et al. [2002](#ref-DateTimeInternet)).

For example, the time *4 minutes and 5 seconds past 3 PM on the 2nd of January 2006 in the timezone UTC-7* is represented as the string "2006-01-02T15:04:05Z07:00". Optionally, nanoseconds are also supported.

We could have represented dates and times as Unix time - the number of seconds since 00:00:00 UTC on 1 January 1970 - but this format is unable to represent timezones and can lose precision if the number is converted or parsed using the incorrect type.

We installed the moment package, a “lightweight JavaScript date library for parsing, validating, manipulating, and formatting dates” (Wood [n.d.](#ref-MomentNpm)), to parse these timestamps in our frontend. This allows us to generate relative date strings such as “5 minutes ago”.

To streamline our use of moment we also installed the ngx-moment package. This provided us with custom pipes which we could use inside templates, as shown in lst. [3](#lst:moment-example-tgl)

Listing 3: This is used to show the time a resource was created, relative to the user’s current time.

<span class="text-gray-light">{{ res.created\_at | amTimeAgo }}</span>

## Modules

Every module imports the CommonModule module from the Angular standard library.

The modules that we created are:

* AppModule - handles the entire application
  + AppRoutingModule - handles routing for the entire application
  + UserModule - provides a service to interact with all user *information*
  + ProfileModule - provides components to interact with all user *profile pages*
    - ProfileRoutingModule - handles routes relating to a user’s profile
  + ResourceModule - provides services and components that interact with all resources
    - ResourceRoutingModule - handles routes relating to an individual resource
  + AuthModule - provides services and components relating to user *authentication*
  + OcticonModule - provides a directive to insert Primer’s icons (known as “octicons”)

Routing is discussed in more detail in sec. [14.5](#sec:routing).

## Routing

Routing in Angular is provided by the RoutingModule in the Angular standard library. This helped us make MTA Hub a single-page application by:

* determining what components are rendered on-screen,
* updating the URL shown in the browser’s address bar, and
* providing a way to access navigation data

The routing module provides many Angular services, but the two services we used the most is:

* ActivatedRoute, which allows us to access information about the currently activated route
* Router, which allows us to manage routing in generally, so we can navigate to different sections of the website

**Lazy loading**

Lazy loading - lazy load modules https://stackoverflow.com/a/44402953/1517394

**Navigating without reloading**

A user’s profile URL is /u/ followed by their username, for example /u/alice. Our API, however, supports querying a user by just using their ID. This if alice had the ID 2, our users could access alice’s profile by visiting /u/2. This ID is not user friendly and we would like our users to know what page they are on just by looking at the URL. This section describes how implemented a redirector from /u/2 to /u/alice.

Our initial implementation (Simoes [2017](#ref-JavascriptChangeRoute)) is described below:

Listing 4: Initial implementation using window.location.pathname.

1 this.route.params.subscribe(params => {  
2 this.users.getUserProfile(params.username).subscribe(data => {  
3 // Update url from ID to username if necessary  
4 if (data.username !== params.username) {  
5 window.location.pathname = "/u/" + data.username;  
6 }  
7 // ...

This solution, shown in lst. [4](#lst:routing-redirect-1), is jarring and slow because updating window.location.pathname results in a full page reload. Additionally it results in an extra history entry, resulting in poor user experience as the user’s browser Back button will perform in an unexpected way.

As shown in lst. [5](#lst:routing-redirect-2), this was resolved by importing Location from the @angular/common package and using the replaceState() function. This function “changes the browser’s URL to a normalized version of the given URL, and replaces the top item on the platform’s history stack” (Ford and Angular Community [2015](#ref-AngularLocationReplaceState)).

Listing 5: Second implementation, using this.location.replaceState.

5 this.location.replaceState("/u/" + data.username);

Navigating to a new page in this way results in screen flicker and additional HTTP requests. as shown in lst. [6](#lst:routing-redirect-final), this is resolved by using the navigate function of the Router service from @angular/router, with the following options (Savkin and Angular Community [2016](#ref-AngularNavigationExtras)):

Listing 6: Final implementation using this.router.navigate.

1 this.route.params.subscribe(params => {  
 2 this.users.getUserProfile(params.username).subscribe(data => {  
 3 // Update url from ID to username if necessary  
 4 // without causing a page reload  
 5 if (data.username !== params.username) {  
 6 this.router.navigate(['u', data.username], {  
 7 preserveFragment: true,  
 8 queryParamsHandling: 'preserve',  
 9 replaceUrl: true,  
10 });  
11 }  
12 // ...

## RxJS

Angular has comprehensive support for RxJS, which is “a library for reactive programming using Observables, to make it easier to compose asynchronous or callback-based code” (Wortmann and RxJS Authors [2018](#ref-RxJS)).

“Observables provide support for passing messages between parts of your application. They are used frequently in Angular and are the recommended technique for event handling, asynchronous programming, and handling multiple values.” (Adon and Angular Community [2018](#ref-AngularUsingObservables))

**Promises vs. Observables**

Both a Promise and Observable “provide us with abstractions that help us deal with the asynchronous nature of our applications” (Vo [2016](#ref-AngularWhatDifferencea)), but the primary difference is that a “Promise handles a single event when an [asynchronous] operation [successfully] completes or fails” (Zöchbauer [2016](#ref-AngularWhatDifference)).

We chose to use observables as they also provide the following additional features:

* Cancellation. Expressive subscriptions such as HTTP requests can be cancelled when we navigate to another page. This improves our performance.
* A unified API “to handle 0, 1, or multiple events” (Zöchbauer [2016](#ref-AngularWhatDifference)), making it easier to write code.
* Tight integration with the Angular framework.
* RxJS provides a functional API and a function pipe() that allows us to mutate results and react to responses effectively.

See TODO reference UPLOADING RESOURCES for our most complex use of the RxJS library.

## HTTP Requests

Our frontend needs to be able to communicate with our web API using an API that provides HTTP client functionality. There are two standard ways to make HTTP requests in JavaScript:

* The XMLHTTPRequest API, released by Internet Explorer 5 in 1998 (Copes [2018](#ref-FetchAPI)), and standardised by the World Wide Web Consortium (W3C) international standards organisation in 2006 (Kesteren and Jackson [2006](#ref-XMLHttpRequestObject)).
* The Fetch API, released in , which “provides an interface for fetching resources (including across the network)” (Mills and Mozilla [2015](#ref-FetchAPIa)).

We initially chose to use Fetch’s API as, compared to XMLHTTPRequest, it is much simpler and “provides a more powerful and flexible feature set” (Mills and Mozilla [2015](#ref-FetchAPIa)). The Angular framework, however, provides a HttpClientModule that can be imported from the @angular/common/http package, with the following additional features:

* Request interception allowing us to automatically include authentication tokens in all requests.
* Improved testing integration provided by the @angular/common/http/testing package.
* Typed requests and responses, “to make consuming the output easier and more obvious” (Angular Community [n.d.](#ref-AngularCommunicatingBackend)).
* “Streamlined error handling” (Angular Community [n.d.](#ref-AngularCommunicatingBackend)).

## AuthInterceptor

Our AuthInterceptor class, a HTTP interceptor, is defined in . Our interceptor includes the user’s authentication token in all requests sent to MTA Hub’s API unless the request is explicitly set to be unauthenticated.

Our AuthInterceptor class implements the HttpInterceptor class defined by the @angular/common/http package. The intercept method takes two parameters:

1. req, the current HTTP request. In other implementations, this request could have been modified by another interceptor, but since we define exactly one interceptor, this is the original request sent from our services.
2. next, the next handler, which may be another interceptor or, in our case, Angular’s HTTP backend which internally performs the actual web request using the browser’s XMLHttpRequest API.

Our intercept method is shown in lst. [7](#lst:http-interceptor) and described below:

1. If the URL does not begin with MTA Hub’s API base URL, as defined in the environment configuration, do not update the HttpRequest defined by the req variable.

* This is security check prevents the user’s authorization token being shared with unintended servers.

1. Or, if the set of request headers includes the X-Authorization-None key:
   1. Create a new set with the X-Authorization-None header removed.
   2. Stored a cloned request in req using the new headers from (i).

* This is used on MTA Hub’s homepage, which should be consistent between logged in and logged out users, so there’s no need to send an authorization token. (Karp [2017](#ref-AllowPassingMisc))

1. Or, if the authorisation service has an authorisation token saved, stored a cloned request in req with the Authorization header set appropriately.
2. Finally, apply the next handler against the current value of req.

Listing 7: AuthInterceptor from website/src/app/auth/auth.interceptor.ts

@Injectable()  
export class AuthInterceptor implements HttpInterceptor {  
 constructor(private auth: AuthService) { }  
  
 intercept(req: HttpRequest<any>, next: HttpHandler):  
 Observable<HttpEvent<any>> {  
  
 if (!req.url.startsWith(environment.api.baseurl)) {  
 // (1) Do nothing if url does not start with our API endpoint  
 } else if (req.headers.has('X-Authorization-None')) {  
 // (2) Optionally exclude authorization header  
 const headers = req.headers.delete('X-Authorization-None');  
 req = req.clone({ headers });  
 } else if (this.auth.accessToken !== null) {  
 // (3) Include authorization header if accessToken is set  
 req = req.clone({  
 setHeaders: {  
 Authorization: `Bearer ${this.auth.accessToken}`  
 }  
 });  
 }  
  
 return next.handle(req); // (4)  
 }  
}

# Authentication

#### Our frontend

manages authentication through the AuthModule class.

This class:

* declares:
  + LoginComponent: provides a login form
  + LoginPageComponent: a page containing just the LoginComponenet
  + RegisterPageComponent: a page with a registration form
* imports:
  + FormsModule: allows us to build forms declaratively
  + ReactiveFormsModule: provides “a model-driven approach to handling form inputs whose values change over time” (Roberts and Angular Community [2018](#ref-AngularReactiveForms))
  + RouterModule

talk about passwords:

* https://www.chromium.org/developers/design-documents/create-amazing-password-forms
* https://www.chromium.org/developers/design-documents/form-styles-that-chromium-understands

#### Our backend

implements authentication as middleware.

Initially we had two middleware functions:

* authRequired - this is returned by the JWT library:
* authRequired := authMiddleware.MiddlewareFunc()
* Any route that includes this middleware function requires the request to have an authenticated user.
* authMaybeRequired - this is a function we’ve created that, if an auth token is provided, verifies the user (via authRequired), and otherwise sets the user context variable to nil.

This works well when authMaybeRequired isn’t used frequently, but we soon discovered that a lot of our routes included this. Some entities — resources, packages & gallery items — may be in a state that means that they should only be accessible to resource managers and site admins.

We decided to change to three middleware functions:

* authMiddlewareFunc - this is returned by the JWT library via authMiddleware.MiddlewareFunc(), as authRequired above
* authMaybeRequired - this is the same as above except it verifies the user via authMiddlewareFunc
* authRequired - this is a function we’ve created that, if the user context variable is nil, will abort, and send a response containing:
  + the header \*WWW-Authenticate to JWT realm=multitheftauto.com,
  + the status code to 401 Status Unauthorized, and
  + the body {"message": "You must be logged in to perform that operation."}

All authenticated administrators should be able to access the following entities:

* resources - unpublished resources, suspended resources
* packages - draft packages

# Profile Pictures

Profile pictures should be displayed prominently in places where a user’s username is shown. To provide this feature we depend on the *Gravatar* web service. Gravatar stands for “Globally Recognized Avatars” and “is an image that follows you from site to site appearing beside your name when you do things like comment or post on a blog” (Gravatar [2020](#ref-GravatarGloballyRecognized)).

We chose to use Gravatar as it provides an easy way for us to implement this feature, and is consistent with the rest of our web services - which also use Gravatar. We derived the user’s Gravatar using the code shown below:

1 gravatar := fmt.Sprintf(  
2 "https://www.gravatar.com/avatar/%x",  
3 md5.Sum([]byte(  
4 strings.ToLower(  
5 strings.TrimSpace(u.Email),  
6 ),  
7 ),  
8 ))

The lines above are described by the following:

* Line 5: first we trim spaces from the front and back of the user email address.
* Line 4: we convert that string to lowercase.
* Line 3: since our input is a *string*, we use the []byte(..) function to return a copy of the string’s underlying byte array (a []byte). We can then use the Sum function from the md5 library to generate an MD5 checksum from our array of bytes, returning another array of bytes.
* Line 2: the %x *verb* in our *format string* encodes a []byte using “base 16, lower-case, [with] two characters per byte”(Cox and The Go Authors [2011](#ref-FmtGoProgramming)).
* Line 1: we use the Sprintf function from the fmt package to generate a *string* based on a *format string*.

Gravatar does not come without privacy implications, though. One could “generate a list of email addresses and compute the corresponding md5 hash [and then] look for collisions in your list of gravatars”(Bell [2009](#ref-GravatarsWhyPublishing)) – this is known as a **rainbow table** attack.

To combat this problem, before the final public release we plan to introduce a setting that allows the user to disable this feature.

In our initial implementation we used the ngx-gravatar[[5]](#footnote-95) package to provide the ngxGravatar *directive*. As shown below, we could provide the [email] attribute to display a user’s profile picture at a specific size.

<img ngxGravatar [email]="'alice@example.com'" size="30">

We realised that this would reveal email addresses to all users of the website, so instead we generate the Gravatar URLs on the server and display the image using simple img tags, as shown below:

<img [src]="user.gravatar + '?size=150'">

In the future we could also implement our own ngxGravatar directive to add support for the size attribute. This would:

* Allow us to accept either a Gravatar hash or, if the user has disabled Gravatar, a URL.
* Increase modularity by allowing us to handle avatar sizing in a separate module without duplicating this logic everywhere, abiding by the *DRY* software design principle - “Don’t Repeat Yourself”.

# Preventing resource squatting

Resources in the existing system are scoped to a global level. This means that there is no way to upload a resource if there already exists a resource with the same name, making the site susceptible to “name-squatting”.

Name-squatting, a form of *cybersquatting*, is the practice of registering - and not legitimately using - popular names in the hope to mislead others, or block others from using the name for a genuine purpose.

We checked to see if there are any user accounts that appear to “squat” on a large number of resource names. On the existing system’s database, we first we ran the following query to get the list of users that own at least twenty resources.

with owners as (  
 select owner\_id, count(owner\_id) as freq  
 from resource\_owners  
 group by owner\_id  
 having count(owner\_id) > 20  
 order by freq desc  
)

This allowed us to run a number of additional queries to find more detailed information.

**Usernames, IDs and frequency of uploaded resources**

select users.id, owners.freq  
from owners, users  
where users.id = owners.owner\_id;

Table 4: Top ten resource authors with a brief description of the kinds of resource that each user has uploaded. Descriptions have been produced by human-interpreting resource descriptions, and not programmatically.

|  |  |  |
| --- | --- | --- |
| id | freq | description |
| 401291 | 131 | racing maps |
| 399788 | 82 | racing maps |
| 386833 | 82 | racing maps |
| 414347 | 80 | unique misc scripts (all Polish) |
| 416671 | 61 | racing maps |
| 320386 | 53 | unique misc scripts, shaders |
| 386795 | 49 | unique misc scripts (all Turkish) |
| 356346 | 47 | racing maps |
| 347609 | 44 | unique misc scripts |
| 281213 | 43 | unique misc scripts (all Spanish) |

From tbl. [4](#tbl:bg-top10-authors), we can deduce that the top authors are useful contributors to the platform.

When we look further into the leading user account (id 401291) we can see many similar uploads of a similar format, with resource names incrementing sequentially from racemap1 to racemap129. This is considered by the community to be abusive behaviour.

**Summary**

The analysis above suggests that “resource squatting” is a problem that needs to be resolved. To solve this problem we will scope resources to individual user accounts. This also makes it easier for scripters to upload resources as they will not need to choose a globally unique resource name, and they can just use the resource name that they used during development.

# Database

Postgresql timezones - we set the database timezone to UTC, and we don’t use timezone specific timestamps. All dates are generated by the database (dates aren’t inserted anywhere) - so problem is solved!

## Migrations

## More

SQL transaction - commit vs rollback - golang - https://stackoverflow.com/a/23502629/1517394

select r.\* from resources r, resource\_collaborators c where (r.author\_id = 2) or ((r.id = c.resource\_id and c.user\_id = 2) and c.accepted) was not DISTINCT. Because each r.author\_id was being calculated for the cross product of r and c. Ust SELECT DISTINCT for fix. Was 12 rows, now 6.https://stackoverflow.com/questions/18988467/sql-query-returning-duplicate-rows

version history - we initially had a draft field realised that we wanted to sort different based on published at. so we got rid of draft and had a published\_at nullable field. “The SQL standard does not explicitly define a default sort order for Nulls” - https://knowledgebase.progress.com/articles/Article/How-to-control-the-sort-order-for-NULL-values-with-the-DataServer - we discovered!!! we found that we could use ORDER BY DESC NULLS FIRST. and multiple order by clauses on one. so we can float unpublished to the top (and sort those by when they were most recently updated), and sort all the published ones by their publish date descending (rather than the date they were updated). one problem we encountered in the draft refactor was that we had to change our conditional inserts - https://github.com/Masterminds/squirrel/issues/239#issuecomment-603479602 - we had columns(…., “draft”) values(…., input.Draft0. Now we had to find a way to insert NULL if draft was true, and to insert the current time if draft was false. The first half is easy, we just use a pq.NullTime{} object. But for the second bit we don’t want to generate that time client-side or even in the backend API, because the API in some odd scenarios could be in a different timezone than the SQL server. Or the time could have drifted a bit. We use the about squirrel link to set the value based on an expression. sq.Expr(“now()”). Then we use https://www.postgresql.org/docs/9.5/dml-returning.html returning data from modified rows to get that ID and date afterwards

# Upload packages

A user should be able to send multiple POST /v1/resources/:resource\_id/pkg requests, each creating a blank “draft”.

TODO / QUESTION: which bits here are design and which bits are implementation? I suppose in Design we can do a more high level overview and then go into the nitty gritty stuff later in Implementation?

Once a file is uploaded by the client and is memory, we check the MIME type of the uploaded file. If the file is not of the “application/zip” MIME type, we return a “415 Unsupported Media Type” and discard the data.

TODO: then we check the ZIP in memory using a number heuristics. What are these heuristics? Considering first getting somewhat feature parity first (or explaining why certain parts won’t get implemented). And then add extra features in a new chapter after it’s implemented?

Once we’ve verified that the zip is safe to use, we upload to a storage service using the gocloud.dev/blob library (a Go package).

“Blobs are a common abstraction for storing unstructured data on Cloud storage services and accessing them via HTTP. This guide shows how to work with blobs in the Go CDK.” (Light and van Gent [2019](#ref-BlobGoCDK))

This is useful as it is a generic backend for various file storage services, including support for Google Cloud Storage, Amazon S3, Azure Blob Service, and of course Local Storage. This makes the website scalable and resilient as we can rely on one of those services doing backups for us, and also use them to deliver stuff for us. Less bandwidth. But we can still use Local Storage when sysadmins are testing or if they would prefer to use local storage (if they do not want to pay for an external service)

This library gives us safety as it converts filenames to something safe. This means that we don’t have to worry about malicious filenames when storing files locally.

The filename ../test is stored as ..\_\_0x2f\_\_test. This is secure.

However, gocloud allows filenames to contain forward slashes (creating a subfolder). Although we could ensure that our filename has no directory separators, we chose to force filenames to be stored as pkg$ID.zip (such as pkg6.zip).

This additionally means that we don’t need to write code to delete old packages when reuploading a file (during initial package creation, when in draft state). We can just rely on this library replacing the blob with the new file.

We only need to delete the files when packages are deleted!

When implementing this we tried to use io.Copy to copy from the input file to the bucket writer, but we could not do this. So we refactored our initial code and chose to use “io.TeeReader to duplicate the stream” (Klose [2016](#ref-GoHowRead)).

“TeeReader returns a Reader that writes to w what it reads from r. All reads from r performed through it are matched with corresponding writes to w. There is no internal buffering - the write must complete before the read completes. Any error encountered while writing is reported as a read error.” (Chu [2011](#ref-IoTeeReaderGo))

We realised that TeeReader returns an io.Reader which does not implement io.ReaderAt, so in the end we chose to use ioutil.ReadAll to read the entire file into a byte slice ([]bytes). This means that we don’t need to use io.Copy, and can produce a io.ReaderAt for the archive/zip library using bytes.NewReader — this function returns a bytes.Reader which \_does\_\_ implement io.ReaderAt. Note that we must also “have sufficient memory for handling [the] zip file” (Jeevanandam M [2018](#ref-GoGolangUnzip)).

Once an actual package has been uploaded the user can choose to publish it, changing the package from the “draft” state to the “pending\_review” state.

Listing **¿lst:complex-rxjs-result?** shows our most complex use of the RxJS library.

createPackage(blob: Blob): Observable<PackageID> {  
 return this.resource$.pipe(  
 switchMap(r => this.resources.createPackage(r.author\_id, r.id, blob)),  
 map(event => this.getUploadEventMessage(event)),  
 tap(message => console.log(message.ok, message.description, message.value)),  
 first(msg => msg.ok),  
 map(msg => msg.value),  
 catchError((err: HttpErrorResponse) => {  
 let reason = 'Something went wrong';  
 if (err.status !== INTERNAL\_SERVER\_ERROR) {  
 reason = err.error.message;  
 }  
 this.alerts.setAlert(reason);  
 return throwError(reason);  
 })  
 );  
}

: ResourceViewService.createPackage creates a new package for the resource currently being viewed.

# Views

# Resource Frontend

use a specific service for ResourceViewService https://stackoverflow.com/a/41451466/1517394

now only owners have access to serious admin operations, which is an improvement over everyone having equal access. this is much more important now that resources are namespaced under their creators username.

## About

The About page is managed by the

### Rich Text

In the existing system, scripters can only insert plain text into their resource description. This makes it difficult for them to market their resource as scripters are unable to format text, include links, lists or embed images. To solve this issue we chose to render the resource description as Markdown.

Markdown is an “easy-to-read, easy-to-write plain text format” (Oruber [2004](#ref-DaringFireballMarkdown)) supported by most software-hosting websites, including the websites we analysed in sec. [11](#sec:bg-api-analysis): Npm Registry, Rust Package Registry and GitHub.

Resource descriptions are stored in plaintext in database using the text SQL type and rendered on the frontend using the ngx-markdown package[[6]](#footnote-106).

# Resource Version History

## Downloads

listening to upload/download progress events https://angular.io/guide/http#listening-to-progress-events

## Uploads

File uplaods - https://medium.com/(**???**), - https://www.techiediaries.com/angular-formdata/

# User Interface & User Experience

Show profile resources page - xl, lg and super small

UX feature: create resource interface will match name and title when you type into “name”, but once you start typing into Title, it will no longer sync the values.

ux - only show private/public indicator if viewing a page where you can view that sort of stuff (if you are viewing own page. or if you are admin) gravatar

guide to flex box https://css-tricks.com/snippets/css/a-guide-to-flexbox/

breadcrumbs - spaces missing https://stackoverflow.com/a/52535597/1517394

LONG BIO WORD BREAKING - overflow-wrap vs word-break - https://tympanus.net/codrops/css\_reference/word-wrap/ - i chose word-break: break-word because we want it to prefer to break between words if possible (long bio - see jusonex)

Uploading a new version is on the resource page - and not hidden in the same place as uploading a resource itself

Resource descriptions. UX truncate pipe https://stackoverflow.com/a/44669515/1517394 - Ultimately decided to let users control this by just taking the first line of the description.

.form-group-header > label uses display block - this ensures good UX and allows the user click anywhere on the line of the label to select the edit box

responsive - u/2/hedit/manage is responsive

date text should take wide enough space - version history - https://stackoverflow.com/a/40389147/1517394

## Form hints

autcomplete hints https://developer.mozilla.org/en-US/docs/Web/HTML/Attributes/autocomplete

# Additional Challenges

This section describes additional technical challenges encountered during the implementation of MTA Hub.

## Setting variables in templates

Sometimes we want to set a variable inside our template instead of inside our class implementation. This can be achieved by using the \*ngFor dynamic directive to perform a for loop through an array containing exactly one item (Tran [2016](#ref-HtmlHowDeclare)). We sometimes do this to avoid writing tedious code multiple times, as shown in lst. [8](#lst:angular-template-forloop-one).

Listing 8: Performing creator = resource.authors[0].

<h1 \*ngFor="let creator of [resource.authors[0]]">  
 <a [routerLink]="['/u', creator.username]">  
 {{ creator.username }}  
 </a>  
 <span class="path-divider">{{ ' / ' }}</span>  
 <a [routerLink]="['/u', creator.username, resource.name]">  
 {{ resource.name }}  
 </a>  
</h1>

Using \*ngFor in the way described above is also useful when want to use asynchronous data in multiple places without using the async pipe multiple times. The async pipe “subscribes to an Observable [..] and returns the latest value it has emitted” (Schoener [2016](#ref-AngularAsyncPipe)). Unnecessary subscriptions to an Observable can cause unexpected behaviour.

## Converting Go types

We use the encoding/json package in the Go standard library to convert Go types to and from JSON. This process is called marshalling and unmarshalling:

“Marshal returns the JSON encoding of v.”

func Marshal(v interface{}) ([]byte, error)

“Unmarshal parses the JSON-encoded data and stores the result in the value pointed to by v.”

func Unmarshal(data []byte, v interface{}) error

(Cox and The Go Authors [2008](#ref-JsonGoProgramming))

Go supports tagging structs with plaintext strings. These strings have a well-established convention for standard introspection during runtime using Go’s reflection library.

Where the json tag is provided (see AuthenticatedUser.Level in lst. [15](#lst:typa-AuthenticatedUser)), the value corresponds to the field inside the JSON dictionary. Nested fields are not supported.

As shown in UserProfile.UserID (lst. [13](#lst:typa-UserProfile)), a value of "-" means that this field should be ignored during marshalling or unmarshalling. Providing an additional value omitempty (see PublicUserInfo.FollowsYou in lst. [14](#lst:typa-PublicUserInfo)) means that a zero value[[7]](#footnote-116) will result in that field being omitted during JSON marshalling.

Our SQL package, github.com/jmoiron/sqlx, internally uses the database/sql package from the standard library, allowing us to use take advantage of the pre-existing tagging functionality available.

Where the db tag is provided (see User.ID in lst. [12](#lst:typa-User)), the value corresponds to the column name in the database (in User.ID the column is id).

## TypeScript Challenges

#### Global variables

One problem we encountered was the accidental access of global variables. The browser sets a number of default global variables - name, window, and more. We expected using the name variable, in a context where it was not defined, to result in a compile-time error, but instead it resulted in a runtime error! To resolve this we set the no-restricted-globals setting in our TypeScript linting configuration, which provides the following feature:

“Disallow specific global variables. Disallowing usage of specific global variables can be useful if you want to allow a set of global variables by enabling an environment, but still want to disallow some of those.” (Dahiya and Palantir Technologies [2019](#ref-RuleNorestrictedglobals))

#### Composing Types

Listing 9: Fields readable by all users

export interface User {  
 readonly id: number;  
 readonly created\_at: string;  
 readonly username: string;  
 readonly gravatar: string;  
 readonly level: number;  
  
 readonly follows\_you?: boolean;  
}

Listing 10: Fields only available to the currently authenticated user

export interface AuthenticatedUser extends User {  
 readonly updated\_at: string;  
 // in the future, more fields  
}

Listing 11: Interfaces relating to a user’s *profile*

export interface UserProfileData {  
 bio: string;  
 location: string;  
 organisation: string;  
 website: string;  
}  
  
export interface UserProfile extends User, UserProfileData {  
 readonly resources: Resource[];  
 readonly following: User[];  
 readonly followers: User[];  
}

In our code we did not want to explicitly write multiple similar type declarations as this would make it difficult to change the types of our structures later on. To get around this problem we made use of TypeScript’s type composition features.

In lst. [9](#lst:typf-User) we declare a User interface with several *public* fields that the world is allowed to view about every user. Since the authenticated user (lst. [10](#lst:typf-AuthenticatedUser)) would be able to see at *least* these fields, we made sure to use extends so that we only provided the additional fields available to AuthenticatedUser.

Most API responses that return entities relating to users return the User interface (lst. [9](#lst:typf-User)). The API response for the user profile page, however, contains additional user *profile* data, such as the user’s bio – this additional data is represented by the UserProfileData interface lst. [11](#lst:typf-UserProfile).

This interface is used when both modifying the data and also reading the data, so UserProfileData does not extend User directly. For the user profile response we created an additional UserProfile interface lst. [11](#lst:typf-UserProfile) that extends *both* User and UserProfileData. This interface also includes extra information such as the user’s list of resources, and public information about the people that the user *follows* or is *followed by*.

In lst. [23](#lst:typf-ResDerived) we build other types using TypeScript’s type transformation utilities:

* ResourceCreateResponse takes the Resource type lst. [20](#lst:typf-Resource), picks only the id property, and constructs a new type with all the properties of the resultant object as read-only.
* **Declaration**
* export type ResourceCreateResponse = Readonly<Pick<Resource, 'id'>>;
* **Effective type**
* export interface ResourceCreateResponse {  
   readonly id: number;  
  }
* ResourcePatchRequest takes picks a number of properties from Resource, then makes them the properties *optional*. Optional properties are denoted by a question mark:
* **Declaration**
* export type ResourcePatchRequest = Partial<  
  Pick<Resource,  
   'name' | 'title' | 'description' | 'visibility' | 'archived'  
  >>;
* **Effective type**
* export interface Resource {  
   name?: string;  
   title?: string;  
   description?: string;  
   visibility?: ResourceVisibility;  
  }
* ResourceID allows a type to be either the type of the id field or the name field, of the Resource interface.
* **Declaration**
* export type ResourceID = Resource['id'] | Resource['name'];
* **Effective type**
* export type ResourceID = number | string

#### Destructuring Objects

TODO: "This seems to recount how you learned a new bit of Go syntax? I’m not sure why that’s relevant here.

This is TypeScript syntax which I suppose needs to be made clearer. I could probably just get rid of this section, but I wanted to describe how I debugged this syntactical issue in TypeScript.

Declaring extra types that are only being used once is wasteful. In a number of lambda functions we only want a single field of a single parameter.

Take the following example:

data = {  
 pkg: { ... } // a ResourcePackage  
};  
  
l(data); // `l` is some lambda function

Inside our lambda function l, to access pkg, one might write the following code:

l(data): {  
 let pkg: ResourcePackage = data.pkg;  
 // we never touch the `data` object again...  
 // use `pkg` here  
}

We want to write more concise code, so we take advantage of TypeScript’s support for object *destructuring*:

l({ pkg }) {  
 // `data` is not defined, so we can't use it...  
 // use `pkg` here  
}

However, this pkg variable doesn’t actually have the type ResourcePackage, it has the any type, allowing it to be used in place of *any* type. To fix this, we initially included a type annotation for the pkg variable, like so:

l({ pkg: ResourcePackage }) {  
 // error?!  
}

However, this code *doesn’t* destructure an object, pluck pkg, and give pkg the type ResourcePackage. It actually plucks a ResourcePackage attribute, which is in our case non-existent, and names that variable as pkg.

The solution we found, with the assistance of Schulz ([2015](#ref-TypingDestructuredObject)), was to first provide the type of data – (data: { pkg: ResourcePackage }) – and *then* destructure it, like so:

l({ pkg } : { pkg: ResourcePackage }) {  
 // use `pkg` here  
}

#### Manually Creating an Object That Satisfies an Interface

In our ProfileComponent we created a field called form:

form: FormGroup = this.formBuilder.group({  
 bio: '',  
 location: '',  
 organisation: '',  
 website: '',  
} as UserProfileData);

In TypeScript it is not possible to create a new object with default ‘zero’ values of an interface. We must explicitly create an object with all the fields we want, with zero values.

To prevent bugs, it is important for our code to raise a compile-time error if we changed the declaration of UserProfileData. However, we discovered that this code would not cause an error because the as keyword converts an object of type any into the type on the right hand side.

We discovered that in a previous version of TypeScript <UserProfileData> { ... } was a type *assertion* - asserting that a given object was of the type UserProfileData, without actually checking the object. In that same version, using { ... } as UserProfileData would throw an compile-time error as the types do not match.

In later versions, TypeScript had to remove the <T> x syntax as it clashed with a JavaScript extension called “JSX”, which allows programmers to write HTML inside JavaScript code. To provide the same type assertion feature, the TypeScript developers weakened the as keyword, no longer making it possible to write an object, declare *and* check its type at the same time (Klaus [n.d.](#ref-HowCanCreate)).

We discovered a workaround - if we declare a variable “first”, and *then* assign an object to it, TypeScript will raise a compile time error. At the top of our file we wrote the following code:

const zeroUserProfileData: UserProfileData = {  
 bio: '',  
 location: '',  
 organisation: '',  
 website: '',  
};

Note that objects in JavaScript (and TypeScript) are pass-by-reference, so to prevent bugs we performed a shallow copy when using our zeroUserProfileData object:

form: FormGroup = this.formBuilder.group({...zeroUserProfileData});

If we did not do this, future recreations of the component would have stale data as the default, rather than zero values.

# Testing

The technologies used in this project were carefully chosen to support automated testing. The backend supports testing via Go’s inbuilt testing infrastructure (go test). Every component on the frontend has a test file, automatically generated via the ng generate command, ready for manual expansion by contributors.

As mentioned in sec. [12.1.1](#sec:angular-concepts), the frontend makes extensive use of dependency injection, which makes testing much easier.

# Stakeholders

**Administrators and Moderators**

Both these stakeholders currently have equal access across the entire system, and have implicit “owner” status over every resource. This means they can see all resources, public and private.

Further work to be done here includes:

* limiting permanent destructive operations to administrators
* implementing a warning and ban system
* introducing a resource approval process

**System administrator (sysadmin)**

Sysadmins can quickly deploy the website using Docker and database migrations can easily be applied via the command line.

**Contributors**

Contributors can quickly deploy a local version of the website using Docker.

Database migrations can easily be applied but if multiple contributors submit pull requests, a conflict will arise that needs to be handled manually. These migrations are newly created files resulting in a lack of Git merge conflict, so extra care needs to be taken when reviewing pull requests. This problem could be solved via automated repository checks.

The backend follows Go’s conventions including established commenting practices, meaning that the go doc tool can be used to quickly generate documentation from the code. Since the code will be open source, any contributor would be able to access this documentation online via the godoc.org service.

The frontend follows a standard directory structure and supports Angular’s ng command line tool, so it is easy to generate new modules, components, services and directives.

**Scripter**

Scripters have the ability to upload resources, add authors without giving them full access, manage resource settings, and provide rich text media in the resource description.

**Server owner**

Servers owners have the ability to “Follow” authors, but this currently does not have any useful effect.

Servers owners are able to create special accounts for their server and host their private resources on the website, and add their developers as collaborators. However, they would benefit from an additional “resource lists” feature, so that other users’ resources can be included in their personal lists.

This would behave similar to a playlist on YouTube, where lists could be made private, unlisted or public; and private resources in the list would be hidden from those who do not have read access, but public resources in the list *would* still be accessible. In the future, servers could automatically sync resources from these lists.

# User Experience

MTA Hub includes user interface and workflow improvements, making the system more pleasing to use than the existing system.

Every page of MTA Hub is built mobile-first, is responsive, and supports all screen sizes. The existing system, however, does not have any responsive pages and only supports a large desktop screen size.

MTA Hub has improved accessibility and achieves this by using aria-\* attributes wherever necessary. The existing system does not take into account accessibility at all.

# Performance

MTA Hub outperforms the existing system in all aspects.

Due to the use of a more efficient database schema, supported by the proper use of table indexing and other advanced features offered by PostgreSQL, most SQL queries have sped up compared to a local PHP instance of the existing system.

The use of a statically compiled binary in our API, compared to the interpreted nature of PHP, has improved runtime performance.

TODO: actual numbers, graphs

# User Testing

Due to the COVID-19 crisis we were unable to perform valuable in-person user testing.

# Overview

We achieved some of the project goals by creating a secure web application that allows users to share, discover and download resources.

We made the web application more efficient, be built using better infrastructure, and take into account secure practices. It is also much easier for contributors to develop with.

# Extensions

Further work includes implementing the changes suggested in our Evaluation chapter. More ambitious extensions to the system have been outlined below:

## Shared authentication

MTA has a wide variety of services: forum, server listing manager, wiki, staff area, and of course this project - “community”. Each of these services use a separate authentication system, and we could unify these by implementing an OAuth Provider. This would allow other platforms that support OAuth to use MTA Hub’s authentication system as a login mechanism.

This would help improve access control as senior staff can manage all permissions in one place, but reduce security as there is a single point of failure.

## Social features

An additional stakeholder, not mentioned in sec. [4](#sec:terminology), are “players”. A “player” is a user that connects to a server and interacts with these *resources*.

We could provide social features for these players, allowing them to add other users as friends, find what server their friends are on, and track in-game statistics.

## Additional taxonomy

Resources aren’t the only kind of shareable “object” available to the MTA community. We could add additional support for sharing skins and sharing vehicle handlings.

**Skin sharing**

The MTA client is customisable and users can install custom themes that alter the way the user interface looks. We could provide a way for graphic designers to publish these skins, and allow users to preview these skins in the browser. Additionally, we can make use of URI schemes to providing single-click installations of these skins.

**Vehicle configurations**

The way vehicles (cars, planes, trains and boats) behave in the game client are customisable, and users may want to persist these changes across servers, or share these configurations with other users. We could provide a service to share these via a URL, make it possible to create them in the browser, and even include an in-browser simulation of the vehicle configurations.

## Support diagnostics

Users may submit forum topics or join live chat to get help with MTA’s default resources. Sometimes we need to collect diagnostic data and so moderators direct them to install “MTA Diag”[[8]](#footnote-136). MTA Diag generates a text file containing system information, automatically uploads that file to our internal Pastebin service (saved for two weeks), and prompts the user to share the file with us. This file provides vital information to help moderators diagnose the user’s problem.

We could extend MTA Hub to support structured processing of this data, and then present this information in a more user friendly way, such as introducing collapsible sections or the ability to highlight potential problems.

Diagnostic reports would ordinarily be saved for two weeks, but staff would be able to mark the report as *starred*, changing the retention period from two weeks to a much longer time period.

In case of an influx of similar support requests, individual diagnostic reports could be “tagged” with metadata or “linked” to a GitHub issue that discusses the problem. After enough reports are gathered, the system could search for similarities across reports to help find the common problem.

Adon, Jason, and Angular Community. 2018. “Angular - Using Observables to Pass Values.” January 9, 2018. <https://angular.io/guide/observables>.

Angular Community. n.d. “Angular - Communicating with Backend Services Using HTTP.” Accessed April 20, 2020. <https://angular.io/guide/http#error-handling>.

Bell, A. 2009. “Gravatars: Why Publishing Your Email’s Hash Is Not a Good Idea.” December 8, 2009. <http://www.developer.it/post/gravatars-why-publishing-your-email-s-hash-is-not-a-good-idea>.

Bemis, Cole. 2019. “Blankslate | Primer CSS.” August 12, 2019. <https://primer.style/css/components/blankslate>.

Buckton, Ron. 2015. “Decorators · TypeScript.” October 14, 2015. <https://www.typescriptlang.org/docs/handbook/decorators.html>.

Chowdhury, Danial, and MTA Community. 2008. “Resources - Multi Theft Auto: Wiki.” April 27, 2008. <https://wiki.multitheftauto.com/wiki/Resources>.

Chu, Hector. 2011. “Io.TeeReader - the Go Programming Language.” August 26, 2011. <https://golang.org/pkg/io/#TeeReader>.

Comeau, Josh. 2020. “The Perils of Rehydration.” March 2, 2020. <https://www.joshwcomeau.com/react/the-perils-of-rehydration/>.

Copes, Flavio. 2018. “The Fetch API.” February 2, 2018. <https://flaviocopes.com/fetch-api/>.

Cox, Russ, and The Go Authors. 2008. “Json - the Go Programming Language.” December 11, 2008. <https://golang.org/pkg/encoding/json/>.

———. 2011. “Fmt - the Go Programming Language.” January 19, 2011. <https://golang.org/pkg/fmt/>.

Coyier, Chris. 2020. “A Complete Guide to Flexbox.” CSS-Tricks. February 3, 2020. <https://css-tricks.com/snippets/css/a-guide-to-flexbox/>.

Dahiya, Adi, and Palantir Technologies. 2019. “Rule: No-Restricted-Globals.” February 23, 2019. <https://palantir.github.io/tslint/rules/no-restricted-globals/>.

Dante, Dante. 2013. “Architecture - How My Website Should Use Its Own API?” June 4, 2013. <https://softwareengineering.stackexchange.com/a/200569/72872>.

Darwin, Pete. 2017. “Angular - Template Syntax.” February 22, 2017. <https://angular.io/guide/template-syntax>.

Donovan, Ryan. 2019. “The Top 10 Frameworks and What Tech Recruiters Need to Know About Them.” Stack Overflow Blog. December 17, 2019. <https://stackoverflow.blog/2019/12/17/the-top-10-frameworks-and-what-tech-recruiters-need-to-know-about-them/>.

Dusseault, Lisa. 2007. “HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV).” June 2007. <https://tools.ietf.org/html/rfc4918#section-11.2>.

Evers, Steven. 2012. “Should a Website Use Its Own Public API?” May 14, 2012. <https://softwareengineering.stackexchange.com/a/148561/72872>.

Ford, Brian, and Angular Community. 2015. “Angular - Location - replaceState.” December 8, 2015. <https://angular.io/api/common/Location#replaceState>.

GitHub. 2016. “GitHub GraphQL API V4.” GitHub Developer. September 14, 2016. <https://developer.github.com/v4/#why-github-is-using-graphql>.

Google. 2018. “Angular - Internationalization (I18n).” August 29, 2018. <https://angular.io/guide/i18n>.

Gravatar. 2020. “Gravatar - Globally Recognized Avatars.” April 15, 2020. <https://en.gravatar.com/>.

Hristozov, Krasimir. 2019. “MySQL Vs PostgreSQL – Choose the Right Database for Your Project.” Okta Developer. July 19, 2019. <https://developer.okta.com/blog/2019/07/19/mysql-vs-postgres>.

Irvine, Daniel. 2011. “Understanding 403 Forbidden.” July 18, 2011. <https://web.archive.org/web/20190704233547/http://www.dirv.me/blog/2011/07/18/understanding-403-forbidden/index.html>.

Jeevanandam M. 2018. “Go - Golang Unzip Response.Body.” Stack Overflow. May 26, 2018. <https://stackoverflow.com/a/50539327/1517394>.

Karp, Daniel. 2017. “Allow Passing Misc Data to Http Client Options · Issue #18155 · Angular/Angular.” GitHub. November 6, 2017. <https://github.com/angular/angular/issues/18155>.

Kesteren, Anne van, and Dean Jackson. 2006. “The XMLHttpRequest Object.” April 5, 2006. <https://www.w3.org/TR/2006/WD-XMLHttpRequest-20060405/>.

Klaus, Alex. n.d. “How Can I Create an Object Based on an Interface File Definition in TypeScript?” Stack Overflow. Accessed April 15, 2020. <https://stackoverflow.com/questions/13142635/how-can-i-create-an-object-based-on-an-interface-file-definition-in-typescript#comment92142331_13142840>.

Klose, Philipp. 2016. “Go - How to Read Multiple Times from Same io.Reader.” Stack Overflow. September 30, 2016. <https://stackoverflow.com/a/39792097/1517394>.

Light, Ross, and Robert van Gent. 2019. “Blob · Go CDK.” March 22, 2019. <https://gocloud.dev/howto/blob/>.

Lucas C. 2009. “Added Markdown README · Multitheftauto/Mtasa-Blue@eafa075.” GitHub. November 14, 2009. <https://github.com/multitheftauto/mtasa-blue/commit/eafa0751a1d2ee00e4bab200654e311a00936faa>.

Lyons, Edwin, and MTA Community. 2008. “Meta.Xml - Multi Theft Auto: Wiki.” January 13, 2008. <https://wiki.multitheftauto.com/wiki/Meta.xml>.

Mills, Chris. 2018. “Using Shadow DOM.” MDN Web Docs. January 29, 2018. <https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_shadow_DOM>.

Mills, Chris David, and Mozilla. 2015. “Fetch API.” MDN Web Docs. February 26, 2015. <https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API>.

MITRE. 2020a. “CWE-209: Generation of Error Message Containing Sensitive Information (4.0).” February 20, 2020. <https://cwe.mitre.org/data/definitions/209.html>.

———. 2020b. “CWE-22: Improper Limitation of a Pathname to a Restricted Directory (’Path Traversal’) (4.0).” February 20, 2020. <https://cwe.mitre.org/data/definitions/22.html>.

———. 2020c. “CWE-79: Improper Neutralization of Input During Web Page Generation (’Cross-Site Scripting’) (4.0).” February 20, 2020. <https://cwe.mitre.org/data/definitions/79.html>.

Mogul, Jeffrey C., Jim Gettys, Tim Berners-Lee, and Henrik Frystyk. 1997. “Hypertext Transfer Protocol – HTTP/1.1.” January 1997. <https://tools.ietf.org/html/rfc2068>.

Nevogt, Dave. 2019. “What Is a Technology Stack and Why Do I Need One.” Hubstaff Time Tracking Software. March 27, 2019. <https://blog.hubstaff.com/technology-stack/>.

Newman, Chris, Graham Klyne, Clearswift Corporation, and Sun Microsystems. 2002. “Date and Time on the Internet: Timestamps.” July 2002. <https://tools.ietf.org/html/rfc3339>.

npm. 2014. “About Scopes | Npm Documentation.” September 13, 2014. <https://docs.npmjs.com/about-scopes>.

Okta. 2018. “Authentication Vs. Authorization.” Authentication Vs. Authorization. October 15, 2018. <https://www.okta.com/identity-101/authentication-vs-authorization>.

Oruber, John. 2004. “Daring Fireball: Markdown.” December 17, 2004. <https://daringfireball.net/projects/markdown/>.

React. 2018. “React – A JavaScript Library for Building User Interfaces.” April 7, 2018. <https://reactjs.org/>.

Richard, Sam, and Pete LePage. 2020. “What Makes a Good Progressive Web App?” January 6, 2020. <https://web.dev/pwa-checklist/>.

Roberts, Brandon, and Angular Community. 2018. “Angular - Reactive Forms.” June 19, 2018. <https://angular.io/guide/reactive-forms>.

Rogers, Mikeal. 2015. “Node.Js.” August 27, 2015. <https://nodejs.org/en/>.

Rohan, Jon. 2019. “Timeline | Primer CSS.” September 26, 2019. <https://primer.style/css/components/timeline>.

Romanyuk, Oleg. 2019. “Angular Vs React: Which One to Choose for Your App.” freeCodeCamp.org. October 8, 2019. <https://www.freecodecamp.org/news/angular-vs-react-what-to-choose-for-your-app-2/>.

Savkin, Victor, and Angular Community. 2016. “Angular - NavigationExtras.” May 27, 2016. <https://angular.io/api/router/NavigationExtras>.

Schoener, Tom. 2016. “Angular - AsyncPipe.” June 1, 2016. <https://angular.io/api/common/AsyncPipe>.

Schulz, Marius. 2015. “Typing Destructured Object Parameters in TypeScript.” Marius Schulz. November 13, 2015. <https://mariusschulz.com/blog/typing-destructured-object-parameters-in-typescript>.

Scully, Ethan. 2020. “Go Vs C++: A Comparison.” January 11, 2020. <https://careerkarma.com/blog/go-vs-c-plus-plus/>.

Sentry. n.d. “About Sentry.” Sentry | Error Tracking & Exception Handling — JavaScript, Python, PHP. Accessed April 16, 2020. <https://sentry.io/about/>.

Sheriff, Paul D. 2017. “Logging in Angular Applications.” November 2017. <https://www.codemag.com/Article/1711021/Logging-in-Angular-Applications>.

Simoes, Bernardo. 2017. “Javascript - Change Route Params Without Reloading in Angular 2.” Stack Overflow. September 29, 2017. <https://stackoverflow.com/a/46486677/1517394>.

Sobers, Rob. 2012. “What Is OAuth? Definition and How It Works.” Inside Out Security. April 5, 2012. <https://www.varonis.com/blog/what-is-oauth/>.

TechMagic. 2019. “React Vs Angular Vs Vue.Js — What to Choose in 2019? (Updated).” Medium. June 19, 2019. <https://medium.com/@TechMagic/reactjs-vs-angular5-vs-vue-js-what-to-choose-in-2018-b91e028fa91d>.

Thiago, Arrais. 2008. “Design Patterns - What Is Dependency Injection?” Stack Overflow. September 26, 2008. <https://stackoverflow.com/a/140655/1517394>.

Tran, mberly. 2016. “Html - How to Declare a Variable in a Template in Angular.” Stack Overflow. November 22, 2016. <https://stackoverflow.com/a/40751358/1517394>.

Vaughan-Nichols, Steven J. 2018. “What Is Docker and Why Is It so Darn Popular?” ZDNet. March 21, 2018. <https://www.zdnet.com/article/what-is-docker-and-why-is-it-so-darn-popular/>.

Virkkunen, Matti. 2010. “Api Design - Is It Good Practice for a Website to Use Its Own API? - Stack Overflow.” August 28, 2010. <https://stackoverflow.com/a/3590574/1517394>.

Vo, Trung. 2016. “Angular - What Is the Difference Between Promises and Observables? - Stack Overflow.” October 19, 2016. <https://stackoverflow.com/a/40135509/1517394>.

Vu, Luke. 2019. “PWA Vs SPA: Alike but Different - SimiCart.” SimiCart Blog | eCommerce Insights & Magento Tutorials. September 18, 2019. <https://www.simicart.com/blog/pwa-vs-spa/>.

Wodehouse, Carey. 2019. “SQL Vs. NoSQL: What’s the Difference?” Hiring Headquarters. June 20, 2019. <https://www.upwork.com/hiring/data/sql-vs-nosql-databases-whats-the-difference/>.

Wood, Tim. n.d. “Moment - Npm.” Npm. Accessed April 17, 2020. <https://www.npmjs.com/package/moment>.

World Wide Web Consortium. 2004. “Web Services Architecture.” February 11, 2004. <https://www.w3.org/TR/2004/NOTE-ws-arch-20040211/#relwwwrest>.

World Wide Web Consortium, MIT, T Berners-Lee, Microsoft, P Leach, Xerox, L Masinter, et al. 1999. “HTTP/1.1: Status Code Definitions.” June 1999. <https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>.

Wortmann, Jan-Niklas, and RxJS Authors. 2018. “RxJS.” May 31, 2018. <https://rxjs-dev.firebaseapp.com/>.

Wright, Nathan. 2016. “Interaction Design - Redirect Screen and Good UX?” User Experience Stack Exchange. November 24, 2016. <https://ux.stackexchange.com/questions/101785/redirect-screen-and-good-ux>.

xMatters. 2019. “Four REST API Versioning Strategies.” xMatters. August 14, 2019. <https://www.xmatters.com/blog/devops/blog-four-rest-api-versioning-strategies/>.

Zöchbauer, Günter. 2016. “Angular - What Is the Difference Between Promises and Observables?” Stack Overflow. May 21, 2016. <https://stackoverflow.com/a/37365955/1517394>.

# API

Listing 12: User

// User represents a user account  
type User struct {  
 ID uint64 `db:"id"`  
 CreatedAt time.Time `db:"created\_at"`  
 UpdatedAt time.Time `db:"updated\_at"`  
  
 Username string `db:"username"`  
 Password string `db:"password"`  
 Email string `db:"email"`  
 Level int `db:"level"`  
 Activated bool `db:"is\_activated"`  
 Banned bool `db:"is\_banned"`  
}

Listing 13: A user’s public profile information

// UserProfile represents a user's public profile information  
type UserProfile struct {  
 UserID uint64 `db:"user\_id" json:"-"`  
 Location string `db:"location" json:"location"`  
 Organisation string `db:"organisation" json:"organisation"`  
 Website string `db:"website" json:"website"`  
 Bio string `db:"bio" json:"bio"`  
}

Listing 14: Public fields of a user object

// PublicUserInfo represents the public fields of a user object  
type PublicUserInfo struct {  
 ID uint64 `json:"id"`  
 CreatedAt time.Time `json:"created\_at"`  
 UpdatedAt time.Time `json:"updated\_at"`  
  
 Username string `json:"username"`  
 Gravatar string `json:"gravatar"`  
  
 FollowsYou \*bool `json:"follows\_you,omitempty"`  
}

Listing 15: Fields available to an authenticated user

// AuthenticatedUser represents the fields available  
// to an authenticated user  
type AuthenticatedUser struct {  
 PublicUserInfo  
 Level int `json:"level"`  
}

Listing 16: Structure for resources

type Resource struct {  
 ID uint64 `db:"id" json:"id"`  
 CreatedAt time.Time `db:"created\_at" json:"created\_at"`  
 UpdatedAt time.Time `db:"updated\_at" json:"updated\_at"`  
 AuthorID uint64 `db:"author\_id" json:"author\_id"`  
  
 Name string `db:"name" json:"name"`  
 Title string `db:"title" json:"title"`  
 Description string `db:"description" json:"description"`  
 ShortDescription string `db:"-" json:"short\_description"`  
  
 Visibility string `db:"visibility" json:"visibility"`  
 Archived bool `db:"archived" json:"archived"`  
 DownloadCount int `db:"download\_count" json:"download\_count"`  
  
 CanManage bool `db:"-" json:"can\_manage"`  
}

Listing 17: Constants that represent the possible resource visibilities (Go does not support enums)

const (  
 ResourceVisibilityPublic string = "public"  
 ResourceVisibilityPrivate string = "private"  
)

Listing 18: Structure for resource packages

type ResourcePackage struct {  
 ID uint64 `db:"id" json:"id"`  
 CreatedAt time.Time `db:"created\_at" json:"created\_at"`  
 UpdatedAt time.Time `db:"updated\_at" json:"updated\_at"`  
  
 ResourceID uint64 `db:"resource\_id" json:"resource\_id"` // relation  
 AuthorID uint64 `db:"author\_id" json:"author\_id"` // relation  
 Version string `db:"version" json:"version"`  
 Description string `db:"description" json:"description"`  
  
 PublishedAt \*time.Time `db:"published\_at" json:"published\_at"`  
 FileUploaded bool `db:"-" json:"file\_uploaded"`  
 UploadedAt \*time.Time `db:"uploaded\_at" json:"uploaded\_at"`  
}

# Frontend

Listing 19: LoginResponse

// auth.service.ts  
interface LoginResponse {  
 token: string;  
 expire: string;  
}

Listing 20: Resource

// resource.service.ts  
export interface Resource {  
 readonly id: number;  
 readonly created\_at: string;  
 readonly updated\_at: string;  
 author\_id: number;  
 name: string;  
 title: string;  
 description: string;  
 short\_description: string;  
 visibility: ResourceVisibility;  
 archived: boolean;  
 authors: User[];  
 readonly can\_manage: boolean;  
 download\_count: number;  
}

Listing 21: ResourceVisibility

export enum ResourceVisibility {  
 PUBLIC = 'public',  
 PRIVATE = 'private',  
}

Listing 22: ResourcePackage

export interface ResourcePackage {  
 readonly id: number;  
 readonly created\_at: string;  
 readonly updated\_at: string;  
 published\_at?: string;  
 uploaded\_at?: string;  
  
 readonly resource\_id: number;  
 readonly author\_id: number;  
 version: string;  
 description: string;  
 file\_uploaded: boolean;  
}

Listing 23: Derived resource types

export type ResourceCreateResponse = Readonly<Pick<Resource, 'id'>>;  
export type ResourcePatchRequest = Partial<  
 Pick<Resource,  
 'name' | 'title' | 'description' | 'visibility' | 'archived'  
 >>;  
  
// ResourceID can either be the name of the resource, or its numeric ID  
export type ResourceID = Resource['id'] | Resource['name'];

1. https://community.mtasa.com [↑](#footnote-ref-24)
2. https://emberjs.com/ - “Ember.js is a productive, battle-tested JavaScript framework for building modern web applications” (emberjs.com) [↑](#footnote-ref-51)
3. Detailed information on these APIs can be found at https://developer.github.com/v3/, https://developer.github.com/v4/ and https://developer.github.com/webhooks/ [↑](#footnote-ref-55)
4. More information on Go’s concurrency features can be found at: https://github.com/golang/go/wiki/LearnConcurrency [↑](#footnote-ref-65)
5. https://www.npmjs.com/package/ngx-gravatar [↑](#footnote-ref-95)
6. ngx-markdown package: https://www.npmjs.com/package/ngx-markdown [↑](#footnote-ref-106)
7. Zero values are 0 for numeric types, false for booleans, "" for strings, and nil for pointers [↑](#footnote-ref-116)
8. MTADiag is available on GitHub - https://github.com/multitheftauto/mtadiag [↑](#footnote-ref-136)