Project Document, User manual

Comment Anywhere

# Project Overview

Comment Anywhere is a web service that allows users to submit to and view a unique comment section for on any webpage on the Internet. Comments are viewed, posted, and interacted with via the Front End, a downloadable browser extension for Mozilla Firefox. The Back End stores comment data and other relevant data, provides that data to users, and validates that user permissions. Comment Anywhere also provides a set of moderation tools for removing objectionable content.

# Motivation

Internet denizens have long found ways to have vibrant communications about a wide variety of content. In the past, more websites supported these conversations through comment sections, but many have shuttered their comment sections in recent years. [Medium-Why websites closing comments…] Instead, the avenues of discourse have become social media sites such as Twitter, Facebook, Reddit, and bulletin board style forums, decoupling the conversation from the content itself.

While these sites may have a higher quantity of overall content than a given news site, they lend themselves to agendas and one-size-fits-all moderation policies while, at the same time, fragmenting the conversation and diluting information available to viewers of the core content. The conversation about a given point of interest on the internet becomes dispersed amongst any number of social media posts, thereby burying much valuable information and well formulated alternate views. While it is usually trivial to move from a conversation on social media to the content discussed, it is much more difficult to move from the content to the decoupled conversation.

There is a usefulness and market desire to see comment sections closely coupled with the content, that will follow the content wherever it goes. Comment Anywhere achieves this by permanently tying a comment section with a given URL and making those comments available immediately to users as they browse the web through a browser extension. A unique comment section is available to every unique domain and path combination on the internet, allowing users to offer feedback and insights on under-commented areas of the internet, especially news sites, but also government and business websites.

## Other Differences from Existing Comment Providers

There are a few other features, besides permanent, content-tied comment sections that make Comment Anywhere unique. Posts have three dimensions of ratings; “funny”, “factual”, and “agree”, allowing users to sort on several dimensions that they are interested in. Other comment sections have only one dimension of rating (e.g. Reddit, Twitter) or limit sorting options. (Facebook) Comment Anywhere also has the notion of Domain Moderators, users with privileges to moderate comments, but only for a specific domain. This allows us to create and enforce tailored moderation policies with respect to comments posted on a particular domain, when necessary, without having to apply those moderation policies universally across the website.

## Social Implications

At their best, comments are a forum for lively debate of the issues reported on or expressed in a piece, as well as a source for personal experiences and further information related to the content. The idea that information should be freely accessible runs deep within the ideology of the internet, and the notion that speech should be free is central to the values of this nation. Comment Anywhere provides voice to alternate and silenced opinions that challenge entrenched agendas across the world, facilitating the speaking of truth to power and enabling societal change. While we must use moderation tools to limit libel, and harmful comments about private citizens, varied opinions about public figures and institutions should be accessible to all, and facilitate the democratic process.

# System Block Diagram

# Implementation Details

## Differences from the Design Document

In the 165-page design document, we tried to anticipate every module, class, method, and message type we would need for Comment Anywhere, but we still had to make a number of changes to achieve the functionality we desired.

### Token Transmission

Users prove their identity using encrypted JSON Web Tokens (JWT) [https://jwt.io/]. We originally planned to transmit these tokens as cookies, which is a common practice. While implementing the register, login, and logout system, we found that we could not use cookies as we had planned. This was due to Cross Origin Resource Sharing (CORS) security policies within the browser. [ <https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>] Our browser extension front-end runs locally in a user’s browser, and its *fetch* requests to the server are fundamentally cross-origin. Mozilla and other browsers prohibits CORS requests from accessing cookies. Instead, we authorized a custom HTTP Header in our preflight request, and had the Front End write its token there.

### Schema Changes

Our schema changes were minor, such as adding “NOT NULL” and “DEFAULT” to various columns, such as timestamps indicating a row’s creation. The primary key of the VoteRecords table had to be altered from being a composite primary key consisting of the comment ID and category to also including the user ID of the voter. In the PasswordResetCodes table, the code itself was made the primary key rather than a separate column, for simplicity. Some columns were renamed for consistency. For example, DomainBans.user was renamed to DomainBans.user\_id and Comments.pathid was renamed to Comments.path\_id.

### Communication Entity Changes

Many of our changes related to communication entities, which are shared types between the front and back end that describe the structure of the JSON strings that the front end and back end pass to each other.

The “Ban” structure sent to the server indicating a user to ban was changed from requiring a user ID to a username, to simplify the transformation of the front end form to the communication entity and to reduce database queries. It was also given a Boolean “ban” field, which can be false if the action is an unban.

The fields Communication.Moderate and Communication.ReportID, both of which reference an associated comment report by its 64-bit id, were changed from type int64 to a pointer to an int64. This allowed the use of “nil” values, in case the action was taken without an associated report.

We got rid of the PasswordResetCode client-server communication entity and combined its field into the SetNewPass entity, so that we wouldn’t need to maintain a state indicating user’s progression through the set new password process. Instead, they submit a new password at the same time they submit their code. This was done to simplify that pipeline.

We added a ProfileUpdateResponse entity that is dispatched to a user whenever their profile is changed. We were able to use this one entity whenever they change their own profile, such as by changing their profile blurb, but also dispatch it to them when their information changes in some other way, such as if they have been granted moderator permissions.

We added a FullPage entity for responding to a user’s first request for comments for a page. It transmits all comments for that page as an array, as well as the Domain and Path of the page. This was always needed, and omitting it from the design document was an oversight.

We renamed ViewModRecords to ViewModActions and added several fields to ViewModActions to allow filtering on several dimensions. ForDomain filters for actions related to a particular domain, ByUser filters for actions performed by a user, and From and To allow the passing of a date range. These filters are only applied if they are not empty, so some or none of them can be applied. This was to make the viewing of mod records more useful, not just providing the moderator with every action ever taken every time.

### Module Database Utilization

…

## Challenges During Implementation

### Dockerizing the HTTP Server

When we first deployed to the cloud, we attempted to install Go on our VPS and run the HTTP Server as a regular system process. This worked to an extent, but we soon realized that unless we ran it headlessly, the process would terminate as soon as the SSH terminal attached to it was closed. Running headlessly was not entirely straightforward, so we decided to implement the HTTP Server as a docker container on our cloud server, since we were already using docker to run our database. This required creating a Dockerfile to build an image for our server and using docker network to connect our server to our database. We also had to implement some flags that the server binary could parse when it was run that would indicate whether it was running in docker or not, causing it to use the environment variables differently. Those flags allowed us to continue to test the server outside of docker as we iterated and were not running it within a container.

### Getting the URL; browser.runtime.Port

The browser extension popup does not directly have access to the URL the user is currently viewing. To access the URL of the user’s current page, we needed to create two additional scripts, a background script [https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Background\_scripts] and a content script [https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Content\_scripts]. The content script runs inside the page the user is viewing and is able to access the URL through *window.location.href*. It cannot communicate to the popup part of the extension directly, however. Instead, it sends that URL to a runtime.Port [https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/Runtime/Port] that is connected to the background script. The background script is able to see the content script and the popup script and is able to pass the URL on to the popup script through a second runtime.Port. The popup script receives that URL and executes the fetch request to get the comment data from the server for the current page and display it. We had to configure Vite [https://vitejs.dev/], our bundling tool, to produce these three scripts instead of one bundle, we had to alter our manifest.json [https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/manifest.json] to tell Mozilla how to use them, and we had to implement checks for the front end to see if it is in fact running in a browser, so that we wouldn’t disable running the extension as a webpage, which is generally a faster iteration when testing new features.

### Debugging the Extension

Extensions do not log messages to the browser console like regular webpages do. That renders all print statements (*console.log*) effectively useless and makes debugging more difficult. To get around this, we created a debug class that places text directly on the DOM as a fixed-position overlay. We set an environment variable that indicates whether the extension is in debug mode or not. If it is, the global *console.log* is rerouted to our *debug.log*, and all our printed messages show on a clearable transparent overlay atop the extension popup content. This was particularly important when implementing the content and background scripts.

### Testing

We wanted to thoroughly unit test every component, but with a huge list of features, a tight deadline, and a small team, we were not able to write the number of unit tests and benchmarks we had hoped for. Building tests is involved for both servers and for webpages.

Thoroughly testing servers require simulating HTTP Requests and HTTP Responses. While Go does provide an httptest framework that allows the use of faked response writers, there are two primary issues that made it too involved to implement prior to the alpha release. First, the effect of a request is dependent on a user’s access level, such as whether they are logged in, so we would have had to log in a user prior to most tests and save the token. Secondly, most of the effects of a request result in changes to the database, so we would have had to implement the rebuilding of a test database that mirrored our regular database. While we did implement the creation of a test database, we did not have the time to write unit tests for each of the API endpoints. We were able to write tests for other components of the server, such as the ExtractPathParts function that uses a Regular Expression [https://pkg.go.dev/regexp] to extract the domain and path from an URL.

On the Front End, tests require simulating a DOM environment. While this is possible through Vitest [https://vitest.dev/], which integrates nicely with our bundler Vite, it was ultimately a new and involved paradigm. With only two coders and many features to push, and with our front end having few or no problems, we were not able to make the time investments to learn Vitest paradigms.

# Software Engineering Principles

## Waterfall Methodology

We followed the Waterfall software development method [REF] as called for by the curriculum. During our first semester, we produced a Proposal, Requirements, Analysis, and Design document. We strove to embrace the Waterfall paradigm and were thorough in our documentation. During that first semester, we also produced two prototypes, one for the back and the front end, to help us anticipate the needs of our documentation. By doing so, we were able to produce a 168-page Design document describing almost every class and method we needed, which we followed closely throughout implementation. In particular, we defined a standardized means of communicating between the front-end and the back-end, and we were able schedule implementation based around these communication pipelines.

## Modularity, Encapsulation

We broke our systems into smaller, more manageable components. The front end was divided into classes, most of which inherited from UIComponent, CafeSection, or CafeWindow. We kept UI classes dedicated to a single representation, while Sections and Windows were effectively simple containers. Indeed, the base UIComponent class has a generic type parameter, as its purpose would always be to represent some Server-Client communication entity received by the front end. On the back end, we used structs (Go does not have classes) to separate out UserControllers, Server, Store, UserManager, Page, and PageManager, as well as a submodule for querying the database, for reading the .env variable, for general utility, and for maintaining communication entity types.

## Abstraction, Don’t-Repeat-Yourself

We found several excellent abstractions throughout the course of implementation that quickened our progress. On the back end, we made heavy use of the *Store* struct, which was responsible for querying the database through its member *Queries*, which was an object procedurally generated from SQL queries using a library called sqlc. [ref] We added methods to *Store* for performing several sequential queries, often returning a communication entity. In this way, we were able to keep the *Server* and *Controller* components responsible for routing requests while delegating the database work to *Store*. These methods saw a lot of re-use. For example, *Store.GetCommUser*, which generates a *communication.UserProfile* type after making a few queries, populating fields such as “DomainsBannedFrom”, is called in at least seven places.

On the front-end, we did a lot with abstracting Document Object Model (DOM) functions. Our first core abstraction was the *Dom* namespace, which provided functions such as *div, button*, *el* and so forth, accepting optional element types, text content, and CSS class names as parameters. We later added a second abstractor, called *CafeDom*, which called *Dom*, but supplied some predefined CSS classes that we were reusing a lot. For example, *CafeDom.formSubmitButtonSmall* returns a button with the CSS class “form-submit-button-small”. Each of these abstractions was only saving us 3-4 lines of code per call, but, considering we created hundreds of HTML Elements, it added up.

## Separation of Concerns

We had a challenge early on when considering how to separate our concerns on the front end. When a user clicked a button on a deeply nested element, there might have to be a request dispatched to the server. We didn’t want every nested button making requests itself and abstracted all server communication to a class called *Fetcher*. However, we needed to figure out how to access *Fetcher* from a deeply nested object. At first, we thought we might pass down a reference to the top level *Cafe* to every object, so they could all access everything, but this meant tightly coupling the entire code base and providing the opportunity to introduce very confusing logic by having deeply nested UI endpoint classes calling top level functions.

We solved this by structuring the entire front end around custom events. Every HTML element has access to a reference to the global *document* object, which can register and signal event listeners. While typically used for events such as “onload”, *document* can also register custom events and listeners. We decided to create types for our own custom events, whose names would always be names of client-Server communication entities and whose data would always be the content of a client-Server communication entity. For example, when a submit button is clicked on the Login form, an object of type *Client.Login* is created and populated with the form data. A custom event named “Login” is dispatched to the *document* object, with that *Client.Login* data as the payload. We delegated the listening of events to our top level *CafeObject*, which would simply pass the data along to *Fetcher*.

Use of software engineering principles

User’s manual; usage, etc.

References  
<https://medium.com/global-editors-network/why-news-websites-are-closing-their-comments-sections-ea31139c469d>

Index/ Glossary

Appendix: Team Details and Individual Contributions

Appendix: Workflow Authentication