Comment Anywhere

Pennwest California

CSC 490: Senior Project 1

Design Document

Dr. Chen

12/11/2022

Group Members

Karl Miller	Computer Science	Design
Frank Bedekovich	Computer Science	Specifications & Analysis
Robert Krency	Computer Science	Requirements
Luke Bates	Computer Science	Implementation

Instructor Comments and Evaluation

Table of Contents

Abstract

We specify the complete design of the Comment Anywhere software product, a full stack web application for adding comments to any web page on the internet. We begin with an overview of the system architecture by characterizing the separate modules and their coupling. Next, we describe the purpose of each class in each module and list their data members and methods. We then provide detailed descriptions for each method of the classes, a narrative describing user interaction, and an implementation timeline. Finally, we describe the testing process used in the production of this document.

Description of Document

Purpose of Document

This document, referred to as the Design Document, describes the architecture and design of the software product, Comment Anywhere. It lays out the Client-Server architecture and the modules that comprise each. The modules are then planned out with accompanying methods, data types, and messaging protocols defined in the planned implementation languages. This document is to be used by the team to guide development.

Intended Audience

The Design Document is written with the development team as an intended audience. The team is expected to use this to guide development and limit feature creep or unintended additions. It is intended to provide the developer with an understanding of the product architecture and how feature interaction happens.

Ties To Specification Document

This Design Document synthesizes the information from the Specification Document. It lays out an architecture for the product, categorizes the product's required features into modules, and defines the modules with respect to the intended implementations. Several changes were made to types described in the Specification Document. Those changes are listed below.

Changes from Specifications Document

States

- Four states were added for viewing a user's profile. They are, "LoggedOut: ViewUserProfile", "LoggedIn: Member: ViewUserProfile", "LoggedIn: Moderator: ViewUserProfile", "LoggedIn: Admin: ViewUserProfile". The current state is described by the fields in the Front-End Class *State*.

Client-Server Communication Entities

- The first letter of all fields was capitalized to align with Go syntax, which distinguishes public/exported members by their capitalization.
- "GetComments.sortOrder" was changed to "GetComments.SortAscending" and the type was changed from "int8" to "bool".
- "Feedback.type" was changed to "Feedback.FeedbackType" to prevent name collision with postgres.
- "ViewFeedback.type" was changed to "ViewFeedback.FeedbackType" to prevent name collision with postgres.
- "ViewBans.forDomains" was changed to "ViewBans.ForDomains" and the type was changed from "string" to "string[]" (an array of strings).
- "ViewMods.forDomains" was changed to "ViewBans.ForDomains" and the type was changed from "string" to "string[]" (an array of strings).
- "AssignGlobalModerator" was added to communicate when a new assignment is made by an admin user.
- "AssignDomainModerator" was added.

- “ViewDomainReport” was added.
- “ViewUsersReport” was added.
- “Message” was added to represent text that the server wants the client to see, not necessarily associated with one particular communication.
- “requestValidation” was changed to “RequestVerification”.
- “Validate” was changed to “Verify”.
- Two fields were added to “ViewLogs”; “ViewLogs.StartingAt” and “ViewLogs.EndingAt”, to allow for a range of logs to be queried.

Server-Client Communication Entities

- The first letter of all fields was capitalized to align with Go syntax, which distinguishes public/exported members by their capitalization.
- The field “UserId” was added to “Comment”, because it is simpler for a user to request a comment posters profile if the front end has access to the user ID.
- “CommentVote” was refactored to “CommentVoteDimension” to better describe its data.
- “UserProfile.DomainsModerating” was changed from type “string” to “string[]”.
- “FeedbackRecord.type” was changed to “FeedbackRecord.FeedbackType” to prevent Postgres collision.
- The fields “BannedByUserID”, “BannedByUsername”, “BannedAt”, and “BanReason” were added to “BanRecord” to provide more data to the viewer of a ban record.
- An empty entity called “LogoutResponse” was added.

Caching Entities

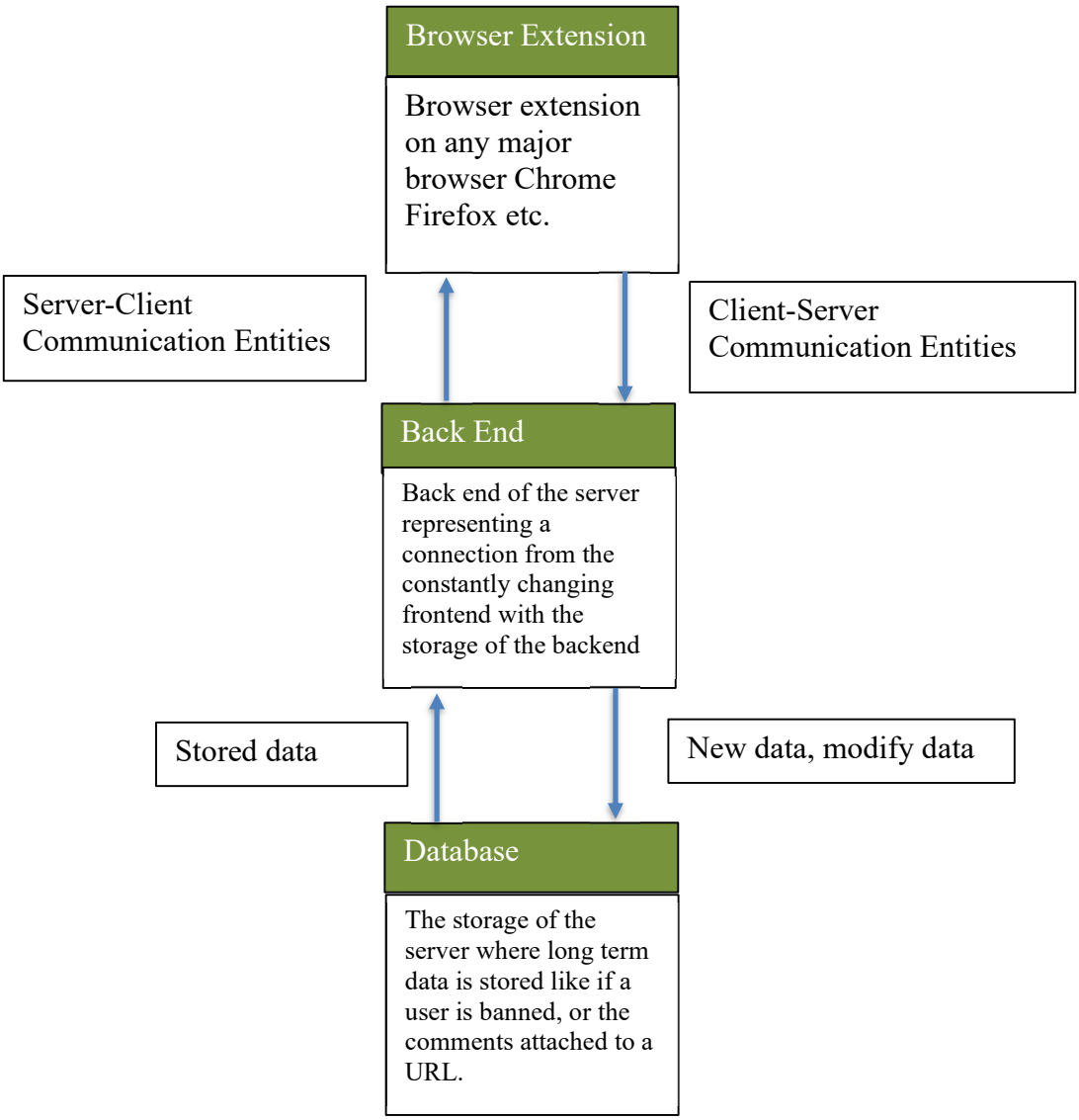
- The fields “id” and “content” were added to “CachedComment”.

- The type of “Page.comments” was changed from type “[]CachedComment” to “Map<int64, CachedComment>”. Storing the comment cache as a hash map based on Comment IDs instead of an array will make retrieval much faster.

Database Schema

- All instances of “user” were changed to “user_id” to prevent collision with the reserved PostgresSQL keyword, “user”.
- The “DomainModerators” table was renamed to “DomainModeratorAssignments” to better reflect the purpose of the table and the fields “ID” and “is_deactivation” were added.
- The “GlobalModerators” table was renamed to “GlobalModeratorAssignments” and the fields “ID” and “is_deactivation” were added.
- The “Admins” table was renamed to “AdminAssignments” and the field “assigned_at” was added.
- The “VoteRecord” table was renamed to “VoteRecords” and the field “commentId” was renamed to “comment_id” to preserve style among tables.
- The field “CommentId” in “CommentModerationActions” was renamed to “comment_id”.
- The “Reports” table was renamed to “CommentReports” and the field “comment”, containing a foreign key linking a “Comment” record, was added.
- The field “at_time” was added to the “Logs” table to track the time a log was made.
- The table “ValidationCodes” was renamed to “VerificationCodes”.

Project Block Diagram



Design Details

System Modules and Responsibilities

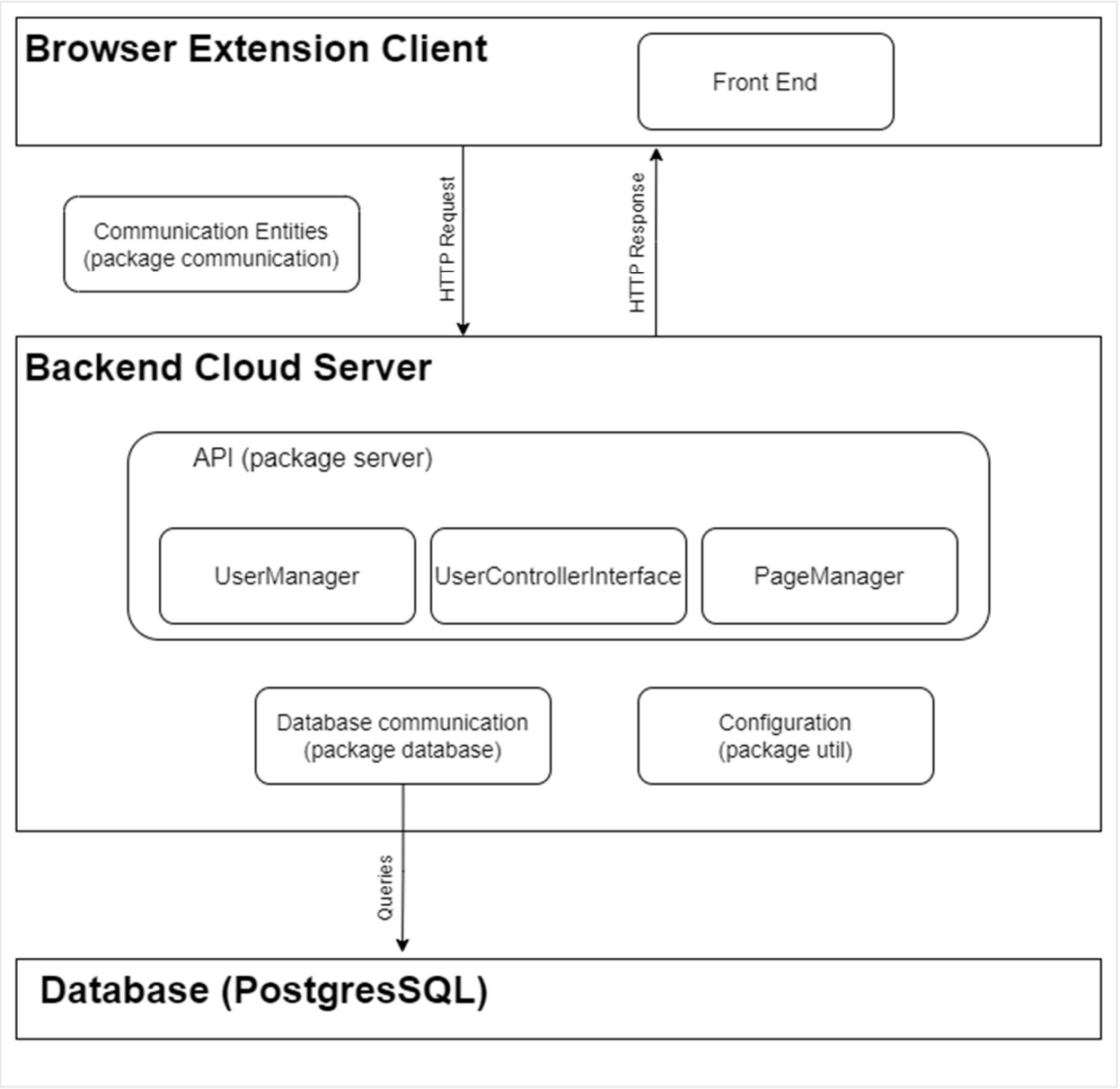


Figure 2: Architectural Diagram

Comment Anywhere is trifurcated into a Client-Server-Database architecture, as depicted in the Architectural Diagram (Figure 2 above). The client is the Browser Extension, a platform specific graphical user interface that runs within a web browser. The Browser Extension Client and Backend Cloud Server communicate through an Application Programmer Interface (API) using HTTP requests and responses to submit and retrieve comments, to register, login, and change user data, and access moderation functionality. The server communicates to the database through the database package.

Within the Back End Cloud Server are three functional packages: *database*, *server*, and *util*. Package *server* handles the processing of HTTP Requests and caching of data. Package *database* handles connections to the PostgresSQL instance running on a separate port and the execution of database queries. Package *util* assists in the initialization of the system from environment variables and validating that connection configurations are valid.

Package *server* consists of four main components: the overall API (class Server), Context Controllers (UserControllerInterface), UserManager, and PageManager. The API Module is responsible for accepting HTTP Requests at each exposed endpoint. It ensures valid incoming requests, associates the request with a Context Controller instance which is unique for each user, then passes control to that Context Controller.

Context Controllers use UserManager to create, edit, and log in to accounts and PageManager to access and update content information. They may also access the database. Queries object directly to realize other functionalities. They are responsible for storing response data to a user and, ultimately, writing the HTTP Response which the Front End receives for each request. They represent a user on the server.

The Front-End receives Server-Client communication entities within HTTP Responses as it dispatches HTTP Requests containing Client-Server communication entities. It updates or initializes a boundary class for each entity which the client needs to view. As the user interacts with the extension in the front end, when an action occurs on a boundary class requiring communication with the server, an event is dispatched, a Client-Server communication entity is constructed to send, and that entity ultimately dispatched to the server.

The communication module is effectively type definitions shared by the Front End and the Server that describe the configuration of communication packets. Each definition is written once for the front-end implementation and once for the back-end implementation.

Module Cohesion

Functional module cohesion is achieved through restricting module functionality to a category of that each module is responsible for. For example, package server is the sole manager of all logic which occurs on the server process, such as directing an HTTP Request to the appropriate end point function. Package database is the sole manager of querying and updating the database. Front End is solely responsible for rendering the graphical user interface.

Module Coupling

The Front End and Back End modules are data coupled around Server-Client and Client-Server communication entities. The Back End and database are data coupled around the structs generated by sqlc from the database schema and the queries. Classes within Server are largely data coupled around Client-Server Communication Entities. The endpoint handler in Server extracts that entity, a simple data structure, from an HTTP Request Body, and passes it to a

specific UserControllerInterface method. If a function from another class, such as PageManager, must be called, it will receive the same Server-Client communication entity. There is also control coupling, as Server also passes a pointer to itself to UserControllerInterface, so that it can act as a data highway to access the Server.DB.Queries object and open access to the database. This control coupling is necessary to implement our design pattern. Indeed, the class is named *Controller* for this passing of Control. Tight Coupling on the Front End is entirely avoided using JavaScript CustomEvents. Instead of calling a tightly-coupled parent class, a boundary class simply emits a custom event on the global “document” object in response actions that could ultimately require server communication.

Design Analysis

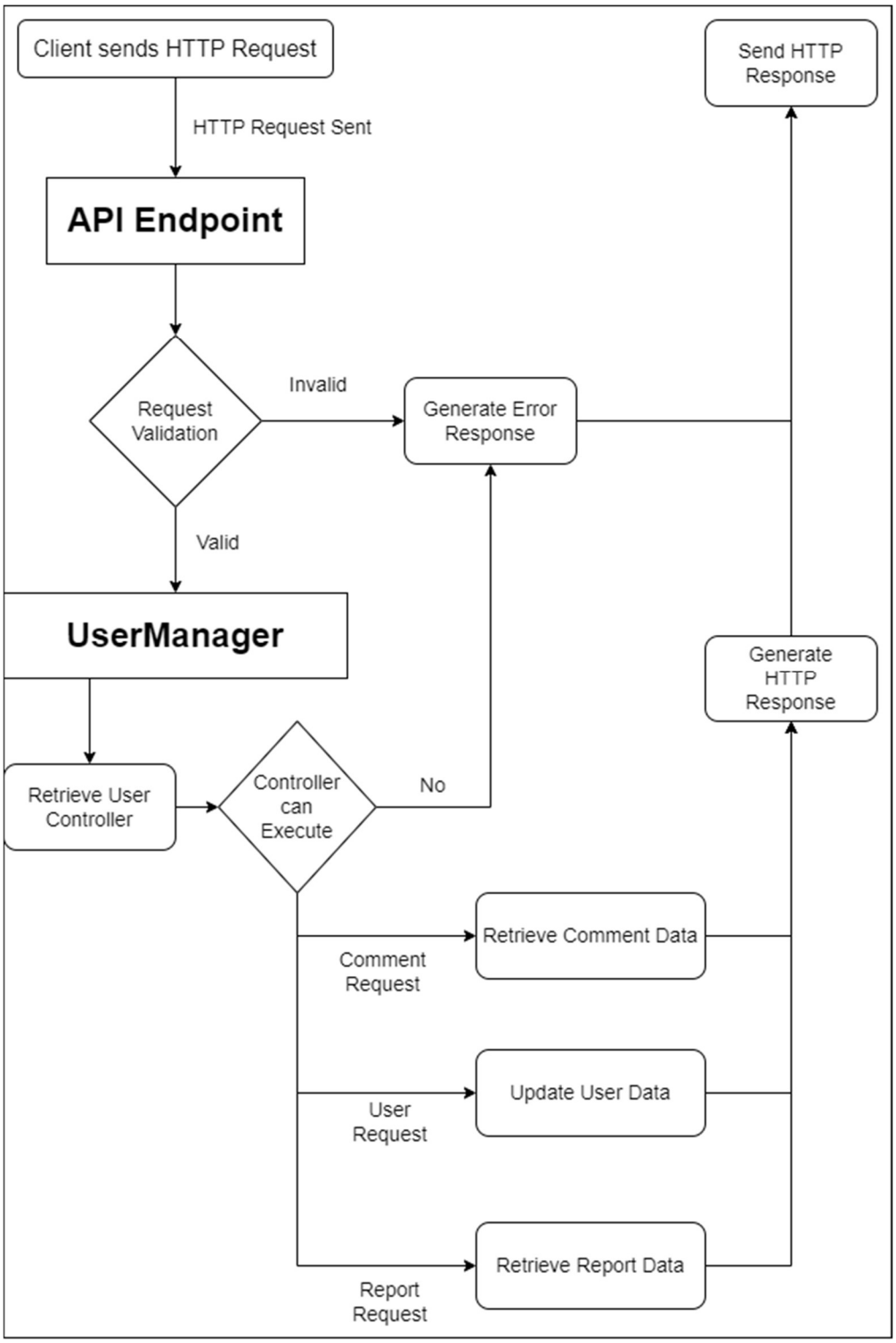


Figure 3: Data Flow Diagram

The Data Flow Diagram (Figure 3 above) shows a high-level view of the lifecycle of an HTTP Request sent from a Client to the Server. An incoming request is validated to be a properly formed request before the appropriate API Endpoint generates a Context Controller for the request. The Context Controller implementation provided depends on the access level of the user. If the Controller is able to execute the user's command, it performs the request action based on the request type and sends a successful payload response back to the client in the form of an HTTP Response. Failed request responses will be delivered with appropriate error messaging.

Design Organization

Description of Classes

Package util

The util package is primarily responsible for reading and validating an .env file to set up the server and database configuration. If future changes to the design require that stand-alone functions be written that are available to multiple parts of the back end, those functions will also be included in the util package.

Note: The capitalization of classes is not arbitrary. Go exports members which are capitalized, while members which are not are only available in that scope.

Package util classes

dbCredentials

Description: dbCredentials holds information the Server needs to connect to the Database, such as the Port, Password, and Database name.

Data Members: string Host, string Port, string User, string Password, string DBName

Class Methods: ConnectString(): string, loadDBEnv()

serverConfig

Description: serverConfig holds information the Server needs to run, such as the port it runs at and the cookie name that will be used.

Data Members: string Port, bool DoesLogAll, string JWTKey, string JWTCookieName

Class Methods: loadServerEnv()

config

Description: config holds both the Server and Database configurations and loads them from the .env file. It is exported as the singleton Config (with a capital C, for Go exports) and accessible around the program.

Data Members: dbCredentials DB, serverConfig Server

Class Methods: Load(string)

Package server

The Server package contains classes for performing the majority of the business logic on the back end, such as authentication and responding appropriately to an HTTP Request at a particular endpoint.

Controller Interface Pattern Diagram

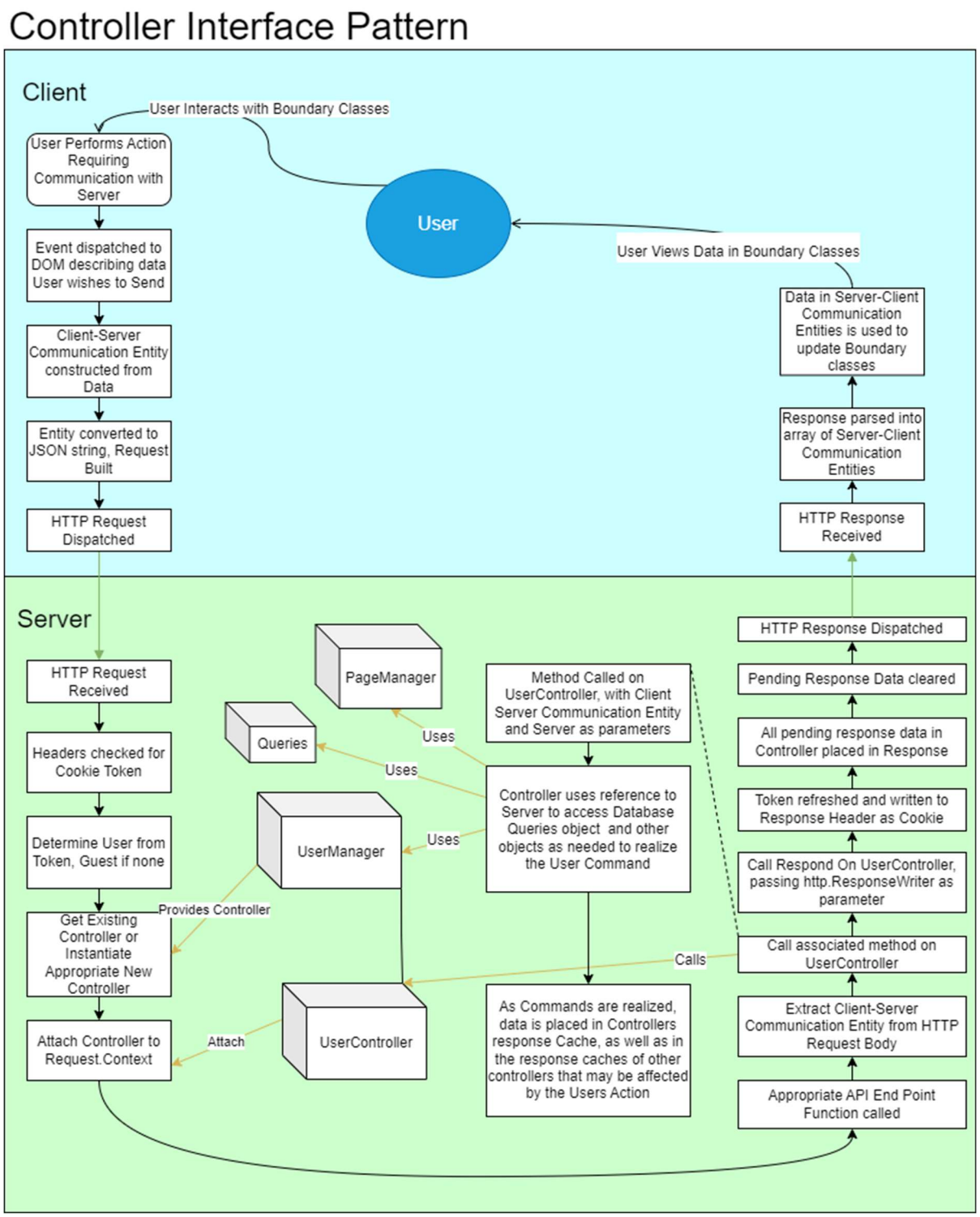


Figure 4: Controller Interface Pattern

The above diagram illustrates how user commands are associated with a particular user and realized. When a User performs some action on the Front End that requires server action, a Client-Server Communication Entity is constructed, and an HTTP Request is dispatched to the appropriate API endpoint. Upon receiving the request, the Server first checks the Request Headers for a cookie containing an encrypted JWT Token and parses that to determine who, if anyone, the user is. If the token is valid and not-expired, UserManager retrieves the already-instantiated ControllerInterface for that User or instantiates one appropriate to the User's access level. Otherwise, it instantiates a GuestController. A reference to this controller is then attached to the HTTP Request Context and routing to the appropriate API endpoint function occurs.

At the API Endpoint function, Server parses the body of the HTTP Request to extract the Client-Server Communication Entity generated by the Front End. It then calls the associated method on the ControllerInterface that was attached to the HTTP Request, passing to that method the extracted communication entity as well as a reference to the Server so that the ControllerInterface may access other objects as needed to realize the command. As the ControllerInterface realizes the User's command, it populates its own response data and may result in the population of response data for other Users. For example, when a User posts a new comment on a page, all ControllerInterfaces located on that page get information about the new comment. After the ControllerInterface method has been called, Server directs ControllerInterface to create a new authentication token and add the Cookie to the header. Finally, Server calls the Respond method of the ControllerInterface which causes it to populate an HTTP Response body with its saved data before clearing its saved data. It then dispatches the Response back to the client.

When the client receives the HTTP Response, it parses the body of the response to extract data sent by the Server. It uses this data to update and add boundary objects and display to the user the data they requested.

Package server Classes

UserControllerInterface

Description: UserControllerInterface provides method signatures which other UserController types implement. Controller references are attached to HTTP Request Contexts in the first middleware that a Request passes through. Those controller references are subsequently used by API endpoints to execute access-appropriate code associated with a particular user or guest. At the API endpoints, the Server is "blind", and will tell whatever controller is attached to the Request to deal with the command extracted from the Request body, which necessitates the interface polymorphism. UserControllerInterface is also used to track which pages are currently being viewed by users, via maps on Pages.

Data Members: None; interface does not have data members.

Class Methods: HandleCommandBan(*communication.Ban, *Server), HandleCommandChangeEmail(*communication.ChangeEmail, *Server), HandeCommandChangeFeedback(*communication.ChangeFeedback), HandleCommandChangePassword(*communication.SetNewPass, *Server), HandleCommandChangeProfileBlurb(*communication.ChangeProfileBlurb, *Server), HandleCommandCommentReply(*communication.CommentReply, *Server), HandleCommandCommentVote(*communication.CommentVote, *Server),

HandleCommandFeedback(*communication.Feedback, *Server),
HandleCommandGetComments(*communication.GetComments, *Server),
HandleCommandGetUserProfile(*communication.GetUserProfile, *Server),
HandleCommandLogin(*Server), HandleCommandLogout(*Server),
HandleCommandModerate(*communication.Moderate, *Server),
HandleCommandPasswordResetCode(*communication.PasswordResetCode, *Server),
HandleCommandPasswordResetRequest(*communication.PasswordResetRequest, *Server),
HandleCommandCommentReport(*communication.PostCommentReport, *Server),
HandleCommandRequestVerification(*communication.RequestVerification, *Server),
HandleCommandVerify(*communication.Verify, *Server),
HandleCommandViewBans(*communication.ViewBans, *Server),
HandleCommandViewCommentReports(*communication.ViewCommentReports, *Server),
HandleCommandViewLogs(*communication.ViewLogs, *Server),
HandleCommandViewModRecords(*communication.ViewModRecords, *Server),
HandleCommandViewMods(*communication.ViewMods, *Server), Respond(r http.Request, w http.ResponseWriter), GetCurrentPage() :*Page, dispatchResponse(r http.Request, w http.ResponseWriter)

UserControllerBase

Class Description: UserControllerBase provides data members for UserControllers. It does not implement UserControllerInterface fully. Other controllers are defined by extending this Base class and implementing the rest of the interface. Controllers also retain an array of messages that need to be sent to the client, which will be dispatched the next time a request from that user is received.

Class Data Members: generated.User User, time.Time lastTokenRefresh, *Page OnPage, [][]byte nextResponse

Class Methods: dispatchResponse(http.Request, http.ResponseWriter), GetCurrentPage() *Page

MemberControllerBase

Class Description: MemberControllerBase provides data members for MemberControllers. It extends UserControllerBase, adding some fields necessary for validation and password reset tracking.

Class Data Members: boolean canResetPassword, *extends UserControllerBase*

Class Methods: *extends UserControllerBase*

GuestController

Class Description: This Controller is attached to an HTTP Request Context when a non-logged in user accesses Comment Anywhere.

Class Data Members: *extends UserControllerBase*

Class Methods: *Implements UserControllerInterface, extends UserControllerBase*

MemberController

Class Description: This Controller is attached to an HTTP Request Context when a regular member accesses Comment Anywhere.

Class Data Members: *extends MemberControllerBase*

Class Methods: *Implements UserControllerInterface, extends MemberControllerBase*

DomainModeratorController

Class Description: This Controller is attached to an HTTP Request Context when a domain moderator accesses Comment Anywhere.

Class Data Members: []string DomainsModerated, *extends MemberControllerBase*

Class Methods: *Implements UserControllerInterface, extends MemberControllerBase*

GlobalModeratorController

Class Description: This Controller is attached to an HTTP Request Context when a global moderator accesses Comment Anywhere.

Class Data Members: *extends MemberControllerBase*

Class Methods: *Implements UserControllerInterface, extends MemberControllerBase*

AdminController

Class Description: This Controller is attached to an HTTP Request Context when an administrator accesses Comment Anywhere.

Class Data Members: *extends MemberControllerBase*

Class Methods: *Implements UserControllerInterface, extends MemberControllerBase*

UserManager

Class Description: UserManager maintains a map of all instantiated controllers for logged-in users and a map of all instantiated controllers for guests. UserManager is responsible for retrieving controllers associated with a userID or temporary guest ID, either by instantiating a new controller, querying the database if necessary, or by supplying an existing controller if one has already been instantiated for that ID.

Class Data Members: map[int64]UserControllerInterface members, map[int64]UserControllerInterface guests

Class Methods: Ban(*communication.Ban, *Server server), Login(*communication.Login, *UserControllerInterface, *Server server): *UserControllerInterface, Logout(*UserControllerInterface, *Server server) : *GuestControllerInterface, Register(*UserControllerInterface, *Server server), GetMemberController(int64 id): *UserControllerInterface, GetGuestController(int64 id): *UserControllerInterface, DispatchPasswordResetEmail(*UserControllerInterface, *Server server)

PageManager

Class Description: PageManager maintains a map of all instantiated Pages that are currently being viewed by some user or guest. It is responsible for ‘placing’ and ‘removing’ users from pages.

Class Data Members: map[string]Page

Class Methods: MoveMemberToPage(*UserControllerInterface user, string pagePath, *Server server), MoveGuestToPage(*UserControllerInterface user, string pagePath, *Server server), UnloadEmptyPages(*Server server), loadPage(string path, *Server server)

Page

Class Description: Page contains cached data for a page, which is a discrete set of comments associated with a particular URL. It also contains a map of all users and guests on the current page.

Class Data Members: string fullPath, map[int64]CachedComment comments, map[int64]UserControllerInterface membersOnPage, map[int64]UserControllerInterface guestsOnPage

Class Methods: GetComments(*UserInterface, string sortedBy, bool ascending): []communication.Comment, addMemberToPage(*UserControllerInterface user), removeMemberFromPage(*UserControllerInterface user), addGuestToPage(*UserControllerInterface user), removeGuestFromPage(*UserControllerInterface user), Moderate(*communication.Moderate, *Server), CreateComment(*communication.CommentReply, *Server server), VoteComment(*communication.CommentVote, *Server server)

CachedComment

Class Description: CachedComment contains data for a single comment which has been loaded from the database.

Class Data Members: int64 id, string content, int64 userID, int64 parent, string username, []CachedVote votes, int64 createdAt, bool hidden, bool removed

Class Methods: Vote(*communication.CommentVote, *Server server), dataForUser(int64 userId) : communication.Comment

CachedVote

Class Description: CachedComment contains data for a single comment vote which has been loaded from the database.

Class Data Members: int64 userId, string username, string, category, int8 value

Server

Class Description: Server holds references to core data structures, such as UserManager, PageManager, database.Store, and Router. It has a method for each API end point. At each end point, it extracts the communication entity the User sent and calls the command handler on the Controller which has been attached the HTTP Request with the extracted data. It generally passes a reference to itself to Controller method calls so that the Controller can access components such as the database and page manager. It is, essentially, the “highway” of the Back End.

Class Data Members: *mux.Router router, database.Store DB, ControllerManager ControllerManager, PageManager PageManager

Class Methods: New(): *Server, setupRouter(), Start(), MiddlewareAttachController(handler http.Handler): http.Handler, postAssignDomainModerator(*http.Request, http.ResponseWriter), postAssignGlobalModerator(*http.Request, http.ResponseWriter), postBan(*http.Request, http.ResponseWriter), postChangeEmail(*http.Request, http.ResponseWriter),

postChangeFeedback(*http.Request, http.ResponseWriter),

postChangeProfileBlurb(*http.Request, http.ResponseWriter),

postCommentReply(*http.Request, http.ResponseWriter), postCommentVote(http.Request,

http.ResponseWriter), postFeedback(*http.Request, http.ResponseWriter),

GetComments(*http.Request, http.ResponseWriter), getUserProfile(*http.Request,

http.ResponseWriter), postLogin(*http.Request, http.ResponseWriter),

postLogout(*http.Request, http.ResponseWriter), postModerate(http.Request,

http.ResponseWriter), postPasswordResetCode(*http.Request, http.ResponseWriter),

postPasswordResetRequest(*http.Request, http.ResponseWriter),

postCommentReport(*http.Request, http.ResponseWriter), postRegister(*http.Request,

http.ResponseWriter), postRequestVerification(*http.Request, http.ResponseWriter),

putSetNewPass(*http.Request, http.ResponseWriter), postVerify(*http.Request,

http.ResponseWriter), getBans(*http.Request, http.ResponseWriter),

getCommentReports(*http.Request, http.ResponseWriter), getDomainReport(*http.Request,

http.ResponseWriter), getUsersReport(*http.Request, http.ResponseWriter),

getFeedback(*http.Request, http.ResponseWriter), getLogs(*http.Request, http.ResponseWriter),

getModRecords(*http.Request, http.ResponseWriter), getMods(*http.Request,

http.ResponseWriter), getLoginStatus(*http.Request, http.ResponseWriter)

Example, Server Code

```
// src/server/postCommentReport.go

package server

import (
    "context"
    "database/sql"
```

```
	"encoding/json"
	"net/http"

	"github.com/comment-anything/prototype1/communication"
	"github.com/comment-anything/prototype1/database/generated"
)

// API Endpoint for https://commentanywhere.net/newReport
func (server *Server) postCommentReport(request *http.Request, writer
http.ResponseWriter) {
	// instantiate a new empty report
	report := communication.PostCommentReport{}
	// attempt to read the body of the request to the report
	err := json.NewDecoder(request.Body).Decode(&report)
	if err != nil {
		writer.WriteHeader(http.StatusBadRequest)
	} else {
		controller := getControllerInterfaceFromContext(request.Context())
		controller.HandleCommandCommentReport(&report, server)
		controller.Respond(request, writer)
	}

}

// What occurs when a Guest attempts to report a comment.
func (c *GuestController) HandleCommandCommentReport(msg
*communication.PostCommentReport, server *Server) {
	// create an error message for transmission to the client
	message := communication.Message{
		Success: false, Text: "You must be logged in to report a comment.",
	}
	// convert that message into a packet for front-end parsing
	bytes, err := communication.CreatePacket(message,
communication.ServerMessage)
	if err != nil {
		// append the message to the responses the client is waiting on
		_ = append(c.nextResponse, bytes)
	}
}

// What occurs when a logged-in user attempts to report a comment; a record is
inserted into the database.
func (c *MemberController) HandleCommandCommentReport(msg
*communication.PostCommentReport, server *Server) {
	// create the comment report in the database
	server.DB.Queries.CreateCommentReport(context.Background(),
generated.CreateCommentReportParams{
		ReportingUser: c.User.ID,
		Comment:       msg.CommentId,
		Reason:        sql.NullString{String: msg.Reason},
	})
```

```
            // create a response message
            message := communication.Message{
                Success: true, Text: "Comment Report submitted.",
            }
            bytes, err := communication.CreatePacket(message,
communication.ServerMessage)
            if err != nil {
                // append the message to the responses the client is waiting on.
                _ = append(c.nextResponse, bytes)
            }
        }
```

Package database and database.generated

Package database contains classes the Server needs for interfacing with the database. It provides an access point between the server and the generated packages. The database.generated package contains code generated by sqlc which is created from the schema of the database and queries for that database.

Database Schema Diagram

Figure 5: Database Schema

Package database Classes

Store

Class Description: Store handles connecting with the database and provides an instance of Queries for utilizing the sqlc generated code for parameterized queries. It is the only class in this section which is not in the database.generated package. Its primary purpose is to wrap the Queries object and connect to the Postgres instance.

Class Data Members: *sql.DB DB, *generated.Queries Queries

Class Methods: New(): Store, Connect(), Disconnect()

AdminAssignment

Class Description: A model of an entry in the AdminAssignments table.

Class Data Members: int64 ID, int64 AssignedTo, int64 AssignedBy, bool isDeactivation

BanAction

Class Description: A model of an entry in the BanActions table.

Class Data Members: int64 ID, int64 TakenBy, int64 TargetUser, string Reason, int64 TakenOn, string Domain, bool SetBannedTo

Comment

Class Description: A model of an entry in the Comments table.

Class Data Members: int64 ID, int64 PathId, string Author, string Content, time.Time CreatedAt, int64 Parent, bool Hidden, bool Removed

CommentModerationAction

Class Description: A model of an entry in the CommentModerationActions table.

Class Data Members: int64 ID, int64 TakenBy, int64 CommentId, string Reason, time.Time TakenOn, bool SetHiddenTo, bool SetRemovedTo, int64 AssociatedReport

CommentReport

Class Description: A model of an entry in the Comments table.

Class Data Members: int64 ID, int64 ReportingUser, int64 Comment, string Reason, bool ActionTaken, time.Time TimeCreated

Domain

Class Description: A model of an entry in the Domains table.

Class Data Members: string ID

DomainBan

Class Description: A model of an entry in the DomainBans table.

Class Data Members: int64 UserID, string BannedFrom, int64 BannedBy, time.Time BannedAt

DomainModeratorAssignment

Class Description: A model of an entry in the DomainModeratorAssignments table.

Class Data Members: int64 ID, string Domain, int64 AssignedTo, int64 AssingedBy, time.Time AssignedAt, boolean isDeactivation

Feedback

Class Description: A model of an entry in the DomainBans table.

Class Data Members: int64 ID, int64 UserID, string Type, time.Time SubmittedAt, string Content, boolean Hidden

GlobalModeratorAssignment

Class Description: A model of an entry in the GlobalModeratorAssignments table.

Class Data Members: int64 ID, int64 AssignedTo, time.Time GrantedAt, int64 AssignedBy, boolean IsDeactivation

Log

Class Description: A model of an entry in the Logs table.

Class Data Members: int64 ID, int64 User, string IP, string URL

PasswordResetCode

Class Description: A model of an entry in the PasswordResetCodes table.

Class Data Members: int64 ID, int64 UserID, string VerifyCode, time.Time CreatedOn

Path

Class Description: A model of an entry in the Paths table.

Class Data Members: int64 ID, string Domain, string Path

Report

Class Description: A model of an entry in the Reports table.

Class Data Members: int64 ID, int64 ReportingUser, int64 Comment, string Reason, bool ActionTaken

User

Class Description: A model of an entry in the Users table.

Class Data Members: int64 ID, string Username, string Password, string Email, time.Time CreatedAt, time.Time LastLogin, string ProfileBlurb, boolean Banned, boolean isVerified

ValidationCode

Class Description: A model of an entry in the ValidationCodes table.

Class Data Members: int64 ID, int64 UserID, string VerifyCode, time.Time CreatedOn

VoteRecord

Class Description: A model of an entry in the VoteRecords table.

Class Data Members: int64 CommentId, string Category, int64 UserId, int8 Value, int64 CommentId, string Category, int64 UserId, int8 Value

Queries

Class Description: A generated struct which provides methods and for every database query used by Comment Anywhere.

Class Methods: CreateDomainModeratorAssignment(context.Context, CreateDomainModeratorAssignmentParams), CreateGlobalModeratorAssignment(context Context, CreateGlobalModeratorAssignmentParams), CreateDomainBanRecord(context Context, CreateDomainBanRecordParams), UpdateUserBanStatus(ctx context.Context, UpdateUserBanStatusParams), UpdateUserEmail(ctx context.Context, UpdateUserEmailParams), UpdateFeedbackHidden(context.Context, UpdateFeedbackHiddenParams), UpdateUserBlurb(context.Context, UpdateUserBlurbParams), CreateComment(context.Context, CreateCommentParams), CreateCommentVote(context.Context, CreateCommentVoteParams), UpdateCommentVote(context.Context, UpdateCommentVoteParams), DeleteCommentVote(context.Context, DeleteCommentVoteParams), CreateFeedback(context.Context, CreateFeedbackParams), GetCommentsAtPath(context.Context, int64 pathID): []GetCommentsAtPathRow, GetCommentVotes(context.Context, int64 commentID): []GetCommentVotesRow, GetUserByUserID(context.Context, int64 id): User, CreateModerationRecord(context.Context, CreateModerationRecordParams), UpdateCommentHidden(context.Context, UpdateCommentHiddenParams), UpdateCommentRemove(context.Context, UpdateCommentRemoveParams), CreateLog(context.Context, CreateLogParams), GetDomainModeratorAssignments(context.Context, int64 id), GetGlobalModeratorAssignments(context.Context, int64 id), GetAdminAssignments(context.Context, int64 id), GetUserByUsername(context.Context, string username) : User, UpdateUserLastLogin(context.Context, int64 id), CreateCommentReport(context.Context, CreateCommentReportParams),

CreateUser(context.Context, CreateUserParams), DeleteUser(context.Context, int64 id), GetUserByUsername(context.Context, string username): User, GetUserByEmail(context.Context, string email): User, UpdateUserPassword(context.Context, UpdateUserPasswordParams), UpdateUserVerification(context.Context, UpdateUserVerificationParams), CreateVerificationRecord(context.Context, CreateVerificationRecordParams) GetVerificationRecord(context.Context, int64 userID): []VerificationCode, DeleteVerificationRecords(context.Context, int64 userID), GetPWResetRecord(context.Context, int64 userID): []PasswordResetCode, CreatePWResetRecord(context.Context, CreatePWResetRecordParams), DeletePWResetRecords(context.Context, int64 userID), GetBanRecords(context.Context): []GetBanRecordsRow, GetCommentReports(context.Context, Boolean actionTaken): []GetCommentReportsRow, GetFeedback(context.Context, Boolean hidden) []GetFeedbackRow, GetLogsForDateRange(context.Context, GetLogsForDateRangeParams): []GetLogsForDateRangeRow, GetModRecordsForModerator(context.Context, int64 id): []GetModRecordsForModeratorRow, GetDomainModerators(context.Context, string domain): []GetDomainModeratorsRow, GetGlobalModerators(context.Context): []GetGlobalModeratorsRow, GetAdmins(context.Context): []GetAdminsRow, GetNewestUser(context.Context): User, GetUserCount(context.Context): int64, GetLogsForIP(context.Context, string IP): []GetLogsForIPRow, GetLogsForUser(context.Context, int64 userID): []GetLogsForUserRow

GetLogsForUserRow

Class Description: The result of executing the GetLogsForUser method of the Queries object.

Class Data Members: int64 ID, string ID, string Url, time.Time AtTime

GetLogsForIP

Class Description: The result of executing the GetLogsForUser method of the Queries object.

Class Data Members: int64 ID, int64 UserID, string Username, string Url, time.Time AtTime

GetAdminsRow

Class Description: The result of executing the GetGlobalModerators method of the Queries object.

Class Data Members: int64 ID, int64 AssignedTo, string AssignedToUsername, time.Time AssignedAt, int64 AssignedBy, string AssignedByUsername

GetGlobalModeratorsRow

Class Description: The result of executing the GetGlobalModerators method of the Queries object.

Class Data Members: int64 ID, int64 AssignedTo, string AssignedToUsername, time.Time AssignedAt, int64 AssignedBy, string AssignedByUsername

GetDomainModeratorsRow

Class Description: The result of executing the GetDomainModerators method of the Queries object.

Class Data Members: int64 ID, int64 AssignedTo, string AssignedToUsername, time.Time AssignedAt, int64 AssignedBy, string AssignedByUsername

GetModRecordsForModeratorRow

Class Description: The result of executing the GetModRecrodsForModerator method of the Queries object.

Class Data Members: int64 ID, int64 TakenBy, string Username, int64 CommentID, string Reason, time.Time TakenOn, Boolean SetHiddenTo, Boolean SetRemovedTo, int64 AssociatedReport

GetLogsForDateRangeRow

Class Description: The result of executing the GetLogsForDateRange method of the Queries object.

Class Data Members: int64 ID, int64 UserID, string Username, string IP, string URL

GetFeedbackRow

Class Description: The result of executing the GetFeedback method of the Queries object.

Class Data Members: int64 ID, int64 UserID, string Username, string Type, time.Time SubmittedAt, string Content, Boolean Hidden

GetCommentReportsRow

Class Description: The result of executing the GetCommentReports method of the Queries object.

Class Data Members: int64 ID, int64 ReportingUser, string Username, int64 Comment, string Reason, bool ActionTaken, time.Time TimeCreated

GetBanRecordsRow

Class Description: The result of executing the GetBanRecords method of the Queries object.

Class Data Members: int64 ID, int64 TakenBy, string TakenByUsername, int64 TargetUser, string TargetUsername, string Reason, time.Time takenOn, string Domain, bool SetBannedTo

CreatePWResetRecordParams

Class Description: Parameters used for executing the CreatePWResetRecord method of the Queries object.

Class Data Members: in64 UserID, string VerifyCode

CreateVerificationRecordParams

Class Description: Parameters used for executing the CreateVerificationRecord method of the Queries object.

Class Data Members: int64 UserID, string VerifyCode

UpdateUserVerificationParams

Class Description: Parameters used for executing the UpdateUserVerification method of the Queries object.

Class Data Members: int64 ID, Boolean IsVerified

UpdateUserPasswordParams

Class Description: Parameters used for executing the UpdateUserPassword method of the Queries object.

Class Data Members: int64 ID, string Password

CreateDomainModeratorAssignmentParams

Class Description: Parameters used for executing the CreateDomainModeratorAssignment method of the Queries object.

Class Data Members: string Domain, int64 AssignedTo, int64 AssignedBy, bool isDeactivation

CreateGlobalModeratorAssignmentParams

Class Description: Parameters used for executing the CreateGlobalModeratorAssignment method of the Queries object.

Class Data Members: int64 AssignedTo, int64 AssignedBy, Boolean isDeactivation

CreateDomainBanRecordParams

Class Description: Parameters used for executing the CreateGlobalModeratorAssignment method of the Queries object.

Class Data Members: int64 TakenBy, int64 TargetUser, string Reason, string Domain, Boolean SetBannedTo

UpdateUserBanStatusParams

Class Description: Parameters used for executing the UpdateUserBanStatus method of the Queries object.

Class Data Members: int64 ID, Boolean Banned

UpdateUserEmailParams

Class Description: Parameters used for executing the UpdateUserEmail method of the Queries object.

Class Data Members: int64 ID, string Email

UpdateFeedbackHiddenParams

Class Description: Parameters used for executing the UpdateFeedbackHidden method of the Queries object.

Class Data Members: int64 ID, bool Hidden

UpdateUserBlurbParams

Class Description: Parameters used for executing the UpdateUserBlurb method of the Queries object.

Class Data Members: in64 ID, string ProfileBlurb

CreateCommentParams

Class Description: Parameters used for executing the CreateComment method of the Queries object.

Class Data Members: int64 PathID, int64 Author, string Content, int64 Parent

CreateCommentVoteParams

Class Description: Parameters used for executing the CreateCommentVote method of the Queries object.

Class Data Members: int64 CommentID, string Category, int64 UserID, int8 Value

UpdateCommentVoteParams

Class Description: Parameters used for executing the UpdateCommentVote method of the Queries object.

Class Data Members: int64 UserID, int64 CommentID, int8 Value

DeleteCommentVoteParams

Class Description: Parameters used for executing the DeleteCommentVote method of the Queries object.

Class Data Members: int64 UserID, int64 CommentID

CreateFeedbackParams

Class Description: Parameters used for executing the CreateFeedback method of the Queries object.

Class Data Members: int64 UserID, string Type, string Content

GetCommentsAtPathRow

Class Description: The result of executing the GetCommentsAtPath method of the Queries object.

Class Data Members: int64 ID, int64 Author, string Content, time.Time CreatedAt, int64 Parent, Boolean Hidden, Boolean Removed, string Username

GetCommentVotesRow

Class Description: The result of executing the GetCommentVotes method of the Queries object.

Class Data Members: int64 UserID, string Category, int8 Value

CreateModerationRecordParams

Class Description: Parameters used for executing the CreateModerationRecord method of the Queries object.

Class Data Members: int64 TakenBy, int64 CommentID, string Reason, Boolean SetHiddenTo, Boolean SetRemovedTo, int64 AssociatedReport

UpdateCommentHiddenParams

Class Description: Parameters used for executing the UpdateCommentHidden method of the Queries object.

Class Data Members: int64 ID, Boolean Hidden

UpdateCommentRemoveParams

Class Description: Parameters used for executing the UpdateCommentRemove method of the Queries object.

Class Data Members: int64 ID, Boolean Removed

CreateLogParams

Class Description: Parameters used for executing the CreateLog method of the Queries object.

Class Data Members: int64 UserID, string IP, string URL

GetUserAssignmentRow

Class Description: The result of executing the GetUserAdminAssignments method of the Queries object.

Class Data Members: int64 ID, time.Time AssignedAt

GetUserDomainModeratorAssignmentsRow

Class Description: The result of executing the GetUserDomainModeratorAssignments method of the Queries object.

Class Data Members: int64 ID, time.Time AssignedAt, string Domain

GetUserGlobalModeratorAssignmentsRow

Class Description: The result of executing the GetUserGlobalModeratorAssignments method of the Queries object.

Class Data Members: int64 ID, time.Time AssignedAt

CreateCommentReportParams

Class Description: Parameters used for executing the CreateCommentReport method of the Queries object.

Class Data Members: int64 ReportingUser, int64 Comment, string Reason

CreateUserParams

Class Description: Parameters used for executing the CreateUser method of the Queries object.

Class Data Members: string Username, string Password, string Email

Example, sqlc-generated Code

```
// Code generated by sqlc. DO NOT EDIT.
// versions:
//   sqlc v1.15.0
// source: commentReports.sql

package generated

import (
    "context"
    "database/sql"
```

```
    )

    const createCommentReport = `-- name:
CreateCommentReport :exec
    INSERT INTO "CommentReports" (
        reporting_user,
        comment,
        reason
    ) VALUES ($1, $2, $3)
    `

    type CreateCommentReportParams struct {
        ReportingUser int64          `json:"reporting_user"`
        Comment       int64          `json:"comment"`
        Reason        sql.NullString `json:"reason"`
    }

    func (q *Queries) CreateCommentReport(ctx context.Context,
arg CreateCommentReportParams) error {
        _, err := q.db.ExecContext(ctx, createCommentReport,
arg.ReportingUser, arg.Comment, arg.Reason)
        return err
    }
```

Package communication

Package communication describes the client-server communication entities and the server-client communication entities for use with the back end. These were previously described in the Specifications document with some minor changes as described in the Ties to the Specifications Document section of this document. The same communication classes described herein are realized with typescript type definitions on the front end so that the front end and back-end share communication paths.

Client-Server Communication Entities

AssignDomainModerator

Description: AssignDomainModerator is dispatched when a global moderator or administrator assigns a new domain moderator.

Data Members: int64 AssignTo, string Domain

AssignGlobalModerator

Description: AssignGlobalModerator is dispatched when an administrator assigns a new global moderator.

Data Members: int64 AssignTo

Ban

Description: Ban is dispatched when a moderator or administrator bans a user.

Data Members: int64 UserId, string Reason, string Domain

ChangeEmail

Description: ChangeEmail is dispatched to the server when a client wants to change their email. They must supply the correct password as well.

Data Members: string NewEmail, string Password

ChangeFeedback

Description: ChangeFeedback is dispatched to the Server when an admin wants to remove or hide a Feedback record from being shown to them again.

Data Members: bool Delete, int64 FeedbackId, bool SetHiddenTo

ChangeProfileBlurb

Description: ChangeProfileBlurb is dispatched to the server when a client updates their profile blurb.

Data Members: string NewBlurb

CommentReply

Description: CommentReply is dispatched to the server when a logged-in user submits a reply to an existing comment or posts a new root-level comment on a page.

Data Members: int64 ReplyingTo, string Reply

CommentVote

Description: CommentVote is dispatched to the server when a logged-in user votes on a comment.

Data Members: int64 VotingOn, string VoteType, int8 Value

Feedback

Description: Feedback is dispatched to the Server when a user submits feedback on Comment

Data Members: string FeedbackType, string Content

GetComments

Description: GetComments is dispatched to the server when a user opens the Browser Extension or when they navigate to a new page with the browser extension over. It is a request for all comments associated with the given url.

Data Members: string Url, string SortedBy, bool SortAscending

GetUserProfile

Description: GetUserProfile is dispatched to the server when the user needs to see a user's profile.

Data Members: int64 UserId

Login

Description: Login is dispatched to the server when the client clicks "Submit" on the login form.

Data Members: string Username, string Password

Logout

Description: Logout is dispatched to the server when the client clicks "Logout". It does not carry any additional data.

Data Members: None

Moderate

Description: Moderate is dispatched to the server when a moderator or admin takes a moderation action on a comment.

Data Members: int64 CommentId, int64 AssociatedReport, bool SetHiddenTo, bool SetRemovedTo, string Reason

PasswordResetCode

Description: PasswordResetCode is dispatched by a user when they enter a password reset code. After a user clicks “Forgot My Password”, users may enter the code emailed to them. When they subsequently click the “submit” button, this request is dispatched to the server.

Data Members: int32 Code

PasswordReset

Description: PasswordReset is dispatched to the server when a password reset is requested. The client supplies the email associated with their account.

Data Members: string Email

PostCommentReport

Description: PostCommentReport is dispatched to the server when the user reports a comment.

Data Members: int64 CommentId, string Reason

Register

Description: Register is dispatched to the server when the client clicks “Submit” on the register form.

Data Members: string Username, string Password, string RetypePassword, string Email, bool AgreedToTerms

RequestVerification

Description: RequestVerification is dispatched to the server when the client wants a new validation code. If a client does not validate their account in a timely fashion, the validation code expires. The client may request a new validation code through their settings tab. When they do so, this entity is created and dispatched to the server.

SetNewPass

Description: SetNewPass is dispatched to the Server when the user changes their password. After submitting a valid password reset code, users are prompted to set a new password. When they subsequently click “submit”, this request is dispatched to the server.

Data Members: string Password, string RetypePassword

Verify

Description: Verify is dispatched to the server when the client inputs a validation code they received in an email to verify their account.

Data Members: Code int32

ViewBans

Description: ViewBans is dispatched to the server when an admin requests records of banned users.

Data Members: []string ForDomains

ViewCommentReports

Description: ViewCommentReports is dispatched to the server when a moderator requests comment reports. It does not have any data. The server will always respond to this with all reports which have not already been moderated. If the client is a DomainModerator, the server will filter appropriately and does not require additional information from the client.

ViewDomainReport

Description: ViewDomainReport is dispatched to the server when an admin requests a report on a domain.

Data Members: string domain

ViewUsersReport

Description: ViewUsersReport is dispatched to the server when an admin requests a report on the overall users of comment anywhere.

ViewFeedback

Description: ViewFeedback is dispatched to the Server when an admin wishes to view feedback submitted by users of Comment Anywhere.

Data Members: int64 From, int64 To, string FeedbackType

ViewLogs

Description: ViewLogs is dispatched to the server when an admin requests access logs.

Data Members: int64 ForUserId, string ForIp, string ForDomain, int64 StartingAt, int64 EndingAt

ViewModRecords

Description: ViewModRecords is dispatched to the server when an admin requests moderation records. It does not have any data. The server will always respond to this with all moderation records, sorted from newest to oldest.

ViewMods

Description: ViewMods is dispatched to the server when an admin requests records of who has been assigned as moderators.

Data Members: []string ForDomains

Server-Client Communication Entities

AdminAccessLog

Description: AdminAccessLog contains data needed by Admins to see an access log.

Data Members: string Ip, int64 LogId, string Url, string UserId, string Username

AdminDomainReport

Description: AdminDomainReport contains data needed by Admins to see information about activity on a particular domain.

Data Members: int32 CommentCount, string Domain

AdminUsersReport

Description: AdminUsersReport is dispatched when an Admin requests the Users report.

Data Members: int32 LoggedInUserCount, int64 NewestUserId, string NewestUsername, int64 UserCount

BanRecord

Description: BanRecord contains data about a banning or unbanning which occurred, which is used by Admins to see information about Moderator actions in certain reports.

Data Members: string BannedFrom, int64 BanRecordId, int64 BannedUserId, string BannedUsername, int64 BannedByUserID, string BannedByUsername, bool SetBannedTo, int64 BannedAt, string Reason

Comment

Description: Comment provides the data the Front End needs to render a comment.

Data Members: string UserId, int64 CommentId, string Content, CommentVoteDimension Factual, CommentVoteDimension Funny, CommentVoteDimension Agree, bool Hidden, int64 Parent, bool Removed, int64 TimePosted, string Username

CommentReport

Description: CommentReport contains data the Front End needs to render a CommentReport, which are reports submitted by users and which Moderators can review and take action on.Data Members: bool ActionTaken, Comment CommentData, string ReasonReported, int64 ReportId, int64 ReportingUserId, string ReportingUsername, int64 TimeReported

CommentVoteDimension

Description: CommentVoteRecord contains data for the number of votes on a comment.

Data Members: int8 AlreadyVoted, int64 Downs, int64 Ups

DomainModeratorRecord

Description: DomainModeratorRecord contains data needed by Admins to see information about DomainModerator assignments.

Data Members: int64 GrantedAt, int64 GrantedBy, string GrantedByUsername, int64 GrantedTo, string GrantedToUsername, int64 RecordId

FeedbackRecord

Description: FeedbackRecord contains data the Front End needs to render a FeedbackRecord, which is a record of a user-submitted feedback, viewed by an Admin, such as a feature request, or bug report.

Data Members: string Content, bool Hide, int64 Id, int64 SubmittedAt, string FeedbackType, int64 Userid, string Username

GlobalModerator

Description: GlobalModerator record contains data needed by Admins to see information about GlobalModerator assignments.

Data Members: int64 GrantedAt, int64 GrantedBy, string GrantedByUsername, int64 GrantedTo, string GrantedToUsername, int64 RecordId

LoginResponse

Description: LoginResponse is sent to the client when they successfully log in.

Data Members: UserProfile LoggedInAs

Message

Description: Message is a general communication entity used to provide feedback to a client that some action has completed (or not completed) on requests where the client has not asked for any particular data

Data Members: bool Success, string Text

ModerationRecord

Description: ModerationRecord contains data the Front End needs to render a ModerationRecord, which is a record of a moderator action, such as hiding or removing a comment.

Data Members: CommentReport AssociatedReport, int64 ModerationRecordId, int64 ModeratorUserId, string ModeratorUsername, string Reason, bool SetHiddenTo, bool SetRemovedTo, int64 TimeModerated

UserProfile

Description: UserProfile contains data needed by the Front End to display a profile for a user.

Data Members: int64 CreatedOn, []string DomainsModerating, bool IsAdmin, bool IsDomainModerator, bool IsGlobalModerator, string ProfileBlurb, int64 UserId, string Username

Example, type definitions

Back end type: client.go

```
    // ChangeEmail is dispatched to the server when a client
wants to change their email. They must supply the correct
password as well.
    type ChangeEmail struct {
        // The new email to associate with the client.
        NewEmail string
        // The user's password.
        Password string
    }
```

Front end type: CLIENT.d.ts

```
    // ChangeEmail is dispatched to the server when a client
wants to change their email. They must supply the correct
password as well.
        type ChangeEmail = {
            // The new email to associate with the client.
            NewEmail: string
            // The user's password.
            Password: string
    }
```

Back end type: server.go

```
    // UserProfile contains data needed by the Front End to
display a profile for a user.
```

```
    type UserProfile struct {

        // The date that the user's account was created on.
        CreatedOn int64

        // The server will generate a comma separated list of
all domains that the user is responsible for moderating, if
applicable. Otherwise, this will be an empty string.
        DomainsModerating []string

        //  If the user is Admin.
        IsAdmin bool

        //  If the user is DomainModerator.
        IsDomainModerator bool

        // If the user is GlobalModerator.
        IsGlobalModerator bool

        // The profile of the user.
        ProfileBlurb string

        // The ID of the user.
        UserId int64

        // The name of the user.
        Username string
    }
```

Front end type: SERVER.d.ts

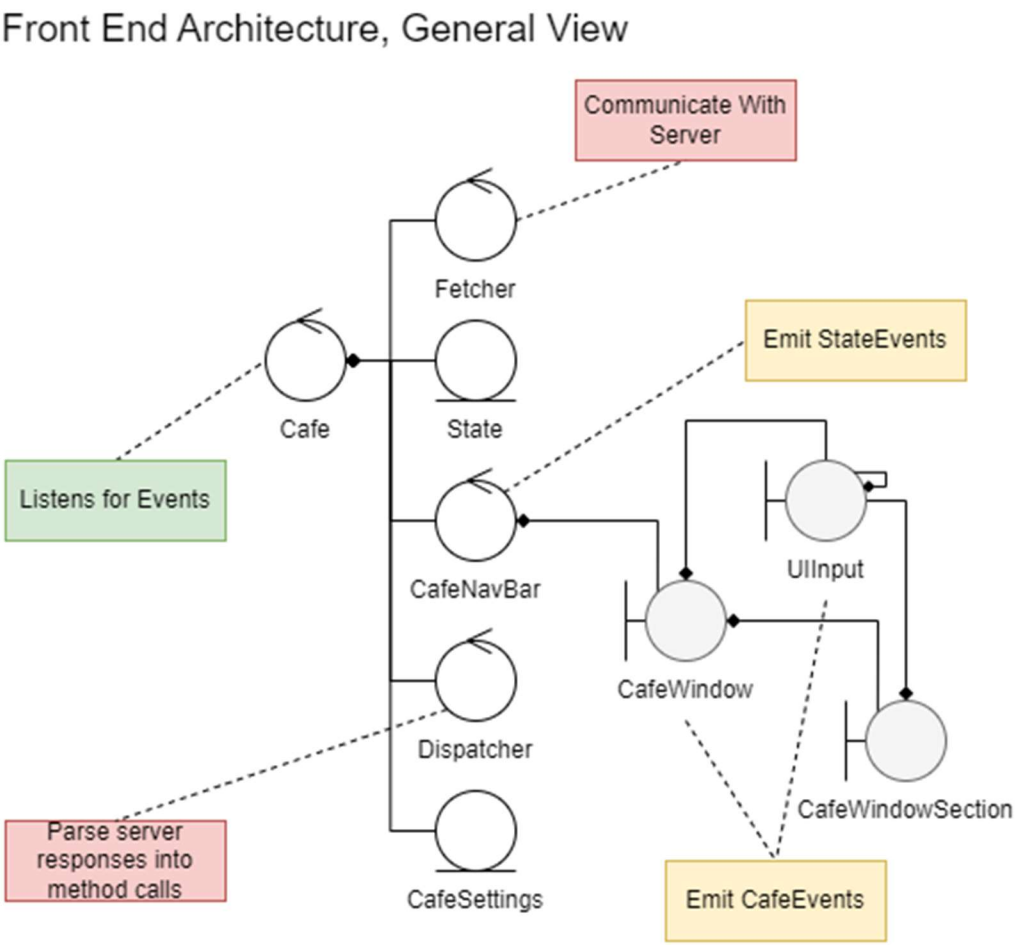
```
    // UserProfile contains data needed by the Front End to
display a profile for a user.
    type UserProfile = {

        // The date that the user's account was created on.
        CreatedOn : number

        // The server will generate a comma separated list of
all domains that the user is responsible for moderating, if
applicable. Otherwise, this will be an empty string.
        DomainsModerating : string[]

        //  If the user is Admin.
        IsAdmin : boolean
```

```
        //  If the user is DomainModerator.
        IsDomainModerator : boolean

        // If the user is GlobalModerator.
        IsGlobalModerator : boolean

        // The profile of the user.
        ProfileBlurb : string

        // The ID of the user.
        UserId : number

        // The name of the user.
        Username : string
    }
```

Front End

The Comment Anywhere front end consists of classes written in TypeScript/JavaScript. It is responsible for displaying the data to the user, capturing user interaction, converting those actions into Client-Server communication entities, and dispatching them to the server. It runs in the browser engine.

Front End Architecture Diagram



The above diagram is a general overview of the Front End Classes and how they interact. The subclasses that inherit from CafeWindow, UIInput, and CafeWindowSection are compressed into their base classes in the diagram, and the possible relationships between the three are shown. That is, a CafeWindow may be composed of some number of UIInputs and some number of CafeWindowSections while a CafeWindowSection may be composed of some number of UIInputs. UIInputs may be composed of additional UIInputs. The actual composition of a given CafeWindow or CafeWindowSection varies.

UIInputs display a Server-Client Communication Entity to the User, such as a Comment. They may provide controls in the form of HTMLElements for the user to interact with that entity. If that interaction needs to be sent to the Server, the UIInput builds the associated Client-Server communication entity from the User inputs and emits it in an event on the global "document" object of the Document-Object-Model (DOM).

Cafe, the top-level instance of the program, listens for these custom events on the "document" object. When it detects one, it passes it along to Fetcher, along with a callback function, and fetcher dispatches that data to the server. When the server responds, Fetcher passes the response to the callback provided by Cafe. In this callback, Cafe provides the server response to Dispatcher along with a reference to itself. Dispatcher calls a distinct method for each type of server response. These methods route the data provided by the Server to where it belongs in the front-end.

Another type of event that can be emitted is a StateEvent, which is emitted by CafeNavBar when the client clicks one of the navigation buttons. Rather than directly update the UI, CafeNavBar instead emits a StateEvent so the primary State object can be updated with the change. Cafe listens for this event too, updates the state accordingly, and may make method calls depending on the state the user wants to transition to, such as the call on CafeNavBar to display the new state.

Description of Front End Classes

Cafe

Class Description: Cafe stands for "Comment Anywhere Front End". Cafe is the base class that is composed of other major classes used in the Front end. It is responsible for updating the State and listening for user input events on the DOM, and transmitting them to the fetcher when appropriate.

Listens For: StateEvent, CafeEvent

Class Data Members: Fetcher fetcher, State state, CafeNavbar navbar, Settings settings, Dispatcher dispatcher, string currentUrl

Class Methods: userChange(Server.UserProfile | undefined), pageChange(string), changeState(string), checkForResponses(), stateChangeEventReceived(), clientEventReceived(), settingEventReceived(Settings)

Fetcher

Class Description: Fetcher is responsible for dispatching requests to the server at the appropriate API endpoints and populating its responses object with the server responses.

Class Data Members: responses ServerResponse[]

Class Methods: fetch(string, client-server Communication Entity), getAndClearResponses() : ServerResponse[]

State

Class Description: State holds the current state of the front end, including who is logged in and what window is being viewed. Cafe passes State to NavBar to realize a state change.

Class Data Members: string viewing, UserProfile ownProfile

Class Methods: loadProfile(UserProfile | undefined), setViewingTo(string)

CafeNavbar

Class Description: CafeNavBar displays navigation buttons for the user to move between states, holds the active window, and holds a general message display object.

Events: Fires StateEvent on nav button click.

Class Data Members: CafeCommentsWindow commentsWindow, CafeSettingsWindow settingsWindow, CafeModerationWindow modWindow, CafeAdminWindow adminWindow, CafeLoginWindow loginWindow, CafeRegisterWindow registerWindow, CafePWResetCodeWindow pwResetWindow, CafeNewPasswordWindow newPwWindow, CafeMessageDisplay globalMessage, CafeUserDisplay userHoverDisplay, CafeWindow currentlyViewing, HTMLButtonElement commentsButton, HTMLButtonElement settingsButton, HTMLButtonElement modButton, HTMLButtonElement adminButton, HTMLButtonElement loginButton, HTMLButtonElement registerButton, HTMLButtonElement logoutButton

Class Methods: commentsButtonClicked(), settingsButtonClicked(), modButtonClicked(), adminButtonClicked(), loginButtonClicked(), registerButtonClicked(), logoutButtonClicked(), setFromState(state:State), displayMessage(data:Server.Message)

CafeSettings

Class Description: CafeSettings holds comment sorting settings configured by the user.

Class Data Members: Boolean viewHidden, string sortBy, boolean sortAscending

Dispatcher

Class Description: Dispatcher is responsible for parsing an array of server responses and dispatching them to the appropriate objects around the front end for rendering to the user.

Class Methods: dispatch(serverResponse[], Cafe), dispatchMessage(Message, CafeNavbar), dispatchUserUpdate(UserProfile, Cafe), dispatchFeedbackUpdate(FeedbackRecord, FeedbackReportSection), dispatchCommentUpdate(Comment, CafeCommentsWindow) , dispatchComments(Comment[], CafeCommentsWindow), dispatchUserProfile(UserProfile, CafeUserDisplay), dispatchLoginResponse(LoginResponse, Cafe), dispatchLogoutResponse(LogoutResponse, Cafe), dispatchPwResetCodeResponse(any, Cafe), dispatchPwResetReqResponse(any, Cafe), dispatchReqVerificationResponse(Message, CafeMessageDisplay), dispatchNewPassResponse(Message, CafeMessageDisplay), dispatchVerifyResponse(Message, CafeMessageDisplay), dispatchBanRecords(BanRecord[], BanRecordsSection), dispatchCommentReports(CommentReport[], CommentReportsSection), dispatchCommentReportUpdate(CommentReport, CommentReportsSection), dispatchFeedbacks(FeedbackRecord[], FeedbackReportSection), dispatchLogs(AdminAccessLog[], LogsSection), dispatchModRecords(ModerationRecord[], ModActionsReportSection), dispatchGlobalMods(GlobalModeratorRecord[], ModeratorsReportSection), dispatchDomainMods(DomainModeratorRecord[], ModeratorsReportSection), dispatchDomainReport(AdminDomainReport, CafeAdminWindow), dispatchUsersReport(AdminUsersReport, CafeAdminWindow)

CafeWindow

Class Description: CafeWindow is the base class for all CafeWindows. CafeWindows ultimately hold all the viewable content of CommentAnywhere, except for the navbar buttons and message. They correspond to the front end states. Only one CafeWindow is visible at any given time and is displayed and hidden by the CafeNavBar instance.

Class Data Members: HTMLDivElement el

Class Methods: show(), hide()

CafeCommentsWindow

Extends: CafeWindow

Class Description: CafeCommentsWindow displays all comments for the current page. It is responsible for repopulating with new comments when comments for a new page are retrieved and updating comments as new ones are added and voted on.

Class Data Members: Comment[] data, lastSettings?: CafeSettings, Map<number, CafeComment> displayedComments, CafeCommentSortDisplay commentSortSettings

Class Methods:populateNewComments(Comment[]), updateComment(Comment), resortComments(), updateFromSettings(CafeSettings)

CafeSettingsWindow

Extends: CafeWindow

Class Description: CafeSettingsWindow displays user settings and provides options for the user to reset their password or verify their email.

Class Data Members: CafeCommentSortDisplay commentSortSettings, HTMLButtonElement passwordResetButton, HTMLButtonElement requestEmailValidationButton, HTMLButtonElement verifyEmailButton, HTMLInputElement verifyCodeInput, HTMLInputElement verifyCodeSubmit

Class Methods: updateFromSettings(CafeSettings), verifyEmailClicked(), verifyCodeSubmitClicked(), passwordResetClicked()

CafeModerationWindow

Extends: CafeWindow

Class Description: CafeModerationWindow allows moderator interactions with Comment Anywhere. It is shown when a user clicks the moderation button on the Navbar, transitioning them to that state.

Class Data Members: ModeratorsReportSection moderators, ModActionsReportSection modActions, BanRecordsSection banRecords, CommentReportsSection reports

CafeAdminWindow

Extends: CafeWindow

Class Description: CafeModerationWindow allows admin interactions with Comment Anywhere. It is shown when a user clicks the admin button on the Navbar, transitioning them to that state.

Class Data Members: FeedbackReportSection feedbackReport, LogsSection logs, domainReports CafeDomainReportDisplay[]

Class Methods: CafeUsersReportDisplay userReport, HTMLInputElement domainInput, HTMLButtonElement domainRequestButton, HTMLButtonElement usersReportButton,domainRequestButtonClicked(), showDomainReport(AdminDomainReport), usersReportRequestButtonClicked(), showUsersReport(AdminUsersReport),

CafeLoginWindow

Extends: CafeWindow

Class Description: CafeLoginWindow is displayed when the user clicks the "Login" button in the navbar and transitions to the login state.

Class Data Members: HTMLInputElement username, HTMLInputElement password, HTMLButtonElement submitLoginButton, HTMLButtonElement forgotPasswordButton,

Class Methods: submitLoginButtonClicked(), forgotPasswordButtonClicked()

CafeRegisterWindow

Extends: CafeWindow

Class Description: CafeRegisterWindow is displayed when the user clicks the "Register" button in the navbar and transitions to the register state.

Class Data Members: HTMLInputElement username, HTMLInputElement password, HTMLInputElement retypePassword, HTMLInputElement email, HTMLInputElement agreedToTerms, HTMLButtonElement submitRegister,

Class Methods: submitRegisterClicked()

CafePwResetRequestWindow

Extends: CafeWindow

Class Description: CafePWResetRequestWindow is displayed when the user transitions to the request password reset form state.

Class Data Members: HTMLInputElement email, HTMLButtonElement submitPWResetRequestButton,

Class Methods: submitButtonClicked()

CafePWResetCodeWindow

Extends: CafeWindow

Class Description: CafePWResetCodeWindow is displayed when the user transitions to the input password reset code state.

Class Data Members: HTMLInputElement code, HTMLButtonElement submitCodeButton,

Class Methods:submitButtonClicked()

CafeNewPasswordWindow

Extends: CafeWindow

Class Description: CafeNewPasswordWindow is displayed when the user transitions to the input new password state.

Class Data Members: HTMLInputElement password, HTMLInputElement newPassword, HTMLButtonElement submitButton,

Class Methods: submitButtonClicked()

CafeWindowSection

Class Description: CafeWindowSection represents an area of a window that performs some particular task or shows some collection of data. CafeWindows may be composed of several CafeWindowSections.

Class Data Members: HTMLElement el,

FeedbackReportSection

Extends: CafeWindowSection

Class Description: FeedbackReportSection is part of the CafeAdminWindow and displays information about user-submitted feedback.

Class Data Members: Map<number, CafeFeedbackDisplay> feedbacks,

Class Methods: HTMLInputElement from, HTMLInputElement to, HTMLSelectElement feedbackType, HTMLButtonElement

requestFeedbackButton,updateFeedback(FeedbackRecord), requestFeedbackClicked(), populateFeedback(FeedbackRecord[]), clearFeedback(),

ModeratorsReportSection

Extends: CafeWindowSection

Class Description: ModeratorsReportSection is part of the CafeModerationWindow and allows viewing of moderator assignment records.

Class Data Members: HTMLInputElement forDomainInput, HTMLButtonElement requestDomainModeratorsButton, HTMLButtonElement requestGlobalModeratorsButton, Map<number, CafeDomainModDisplay> domainModRecords, Map<number, CafeGlobalModDisplay> globalModRecords

Class Methods: HTMLButtonElement clearModRecordsButton,requestDomainModsClicked(), requestGlobalModsClicked(), populateDomainModerators(DomainModeratorRecord[]), populateGlobalModerators(GlobalModeratorRecord[]), clearModsClicked(),

ModActionsReportSection

Extends: CafeWindowSection

Class Description: ModActionsReportSection is part of CafeModerationWindow and provides functionality for viewing moderation actions taken on comments.

Class Data Members: Map<number, CafeModRecordDisplay> modRecords, HTMLButtonElement requestModRecordsButton, HTMLButtonElement clearModActionsButton

Class Methods: requestModRecordsButtonClicked(), populateModActions(ModerationRecord[]), updateModAction(ModerationRecord), clearModActionsClicked()

BanRecordsSection

Extends: CafeWindowSection

Class Description: BanRecordsSection is part of CafeModerationWindow. It is used to display records of bannings to other moderators.

Class Data **Members**: banRecords: Map<number, CafeBanRecord>, HTMLButtonElement requestGlobalBansButton, HTMLInputElement forDomain, HTMLButtonElement requestDomainBansButton

Class Methods: populateBanRecords(BanRecord[]), requestGlobalBansClicked(), requestDomainBansClicked(), clearBanRecords()

CommentReportsSection

Extends: CafeWindowSection

Class Description: CommentReportsSection is part of CafeModerationWindow. It is responsible for displaying comment reports to moderators.

Class Data Members: Map<number, CafeCommentReportDisplay> reports, HTMLButtonElement viewReportsButton

Class Methods: populateCommentReports(CommentReport[]), viewReportsClicked(), updateCommentReport(CommentReport), clearCommentReports()

LogsSection

Extends: CafeWindowSection

Class Description: LogsSection is part of CafeAdminWindow. It is responsible for populating log entries.

Class Data Members: HTMLInputElement forUser, HTMLInputElement forIP, HTMLInputElement forDomain, HTMLInputElement startingAt, HTMLInputElement endingAt, HTMLButtonElement submitLogsRequestButton, Map<number, CafeLogDisplay> logs

Class Methods: submitLogsRequestClicked(), populateLogs(AdminAccessLog[]), clearLogs()

UIInput<Type>

Generic Description: Type represents the underlying data that UIInput displays.

Class Description: UIInput is a base class for all boundary classes representing a server-client communication entity. It provides methods for gracefully deleting its instance, binding listeners, disabling, and re-enabling the UIInput.

Class Data Members: Type data, HTMLElement el, HTMLElement blocker, [HTMLElement, Function, string][] listeners

Class Methods: clickListen(HTMLElement, Function[], Boolean), disable(), enable(), destroy()

CafeComment

Class Description: CafeComment is responsible for rendering a Comment communication entity.

Extends: UIInput<Comment>

Class Data Members: HTMLDivElement Username, HTMLDivElement Content, CafeCommentVoter funny, CafeCommentVoter factual, CafeCommentVoter agree, HTMLTextArea replyText, HTMLTextArea reportText, HTMLButtonElement submitReplyButton, HTMLButtonElement submitReportButton

Class Methods: replyClicked(), submitReplyClicked(), reportClicked(), submitReportClicked()

CafeCommentVote

Class Description: CafeCommentVote is responsible for rendering a CommentVoteDimension communication entity.

Extends: UIInput<CommentVoteDimension>

Class Data Members: number commentId, string voteType, HTMLDivElement voteLabel, HTMLButtonElement up, HTMLButtonElement down, HTMLDivElement total

Class Methods: upVoteClicked(), downVoteClicked()

CafeMessageDisplay

Class Description: CafeMessageDisplay is responsible for rendering a Message communication entity.

Extends: UIInput<Message>

Class Data Members: HTMLDivElement displayedText

Class Methods: updateMessage(Message), clearMessage()

CafeUserDisplay

Class Description: CafeUserDisplay displays UserProfile communication entity data with HTML Elements.

Extends: UIInput<UserProfile>

Class Data Members: HTMLDivElement Username, HTMLDivElement CreatedOn,HTMLDivElement DomainsModerating, HTMLDivElement IsAdmin, HTMLDivElement IsDomainModerator, HTMLDivElement IsGlobalModerator, HTMLDivElement ProfileBlurb

Class Methods: changeProfile(UserProfile), hide(), show()

CafeFeedbackDisplay

Class Description: CafeFeedbackDisplay displays a feedback record for an admin to view.

Extends: UIInput<FeedbackRecord>

Class Data Members: HTMLDivElement Content, HTMLDivElement Hide, HTMLDivElement SubmittedAt, HTMLDivElement FeedbackType, HTMLDivElement Username, HTMLButtonElement ToggleHidden

Class Methods: hideClicked()

CafeOwnProfileDisplay

Class Description: CafeOwnProfileDisplay is used to display a User's own profile to them.

Extends: UIInput<UserProfile>

Class Data Members: HTMLDivElement Username, HTMLDivElement CreatedOn,HTMLDivElement DomainsModerating, HTMLDivElement IsAdmin, HTMLDivElement IsDomainModerator, HTMLDivElement IsGlobalModerator, HTMLDivElement ProfileBlurb, HTMLButtonElement editProfileBlurbButton, HTMLTextAreaElement editProfileTextarea, HTMLButtonElement editProfileSubmitButton, HTMLButtonElement changePasswordButton

Class Methods: updateProfile(UserProfile), editProfileBlurbClicked(), editProfileSubmitClicked()

CafeBanRecordDisplay

Class Description: CafeBanRecordDisplay displays an individual ban or unban record.

Extends: UIInput<BanRecord>

Class Data Members: HTMLDivElement BannedFrom, HTMLDivElement BannedUsername, HTMLDivElement BannedByUsername, HTMLDivElement SetBannedTo, HTMLDivElement BannedAt, HTMLDivElement Reason

Class Methods: bannedUserClicked(MouseEvent), bannedByClicked(MouseEvent)

CafeCommentReportDisplay

Class Description: CafeCommentReportDisplay displays data for a single CommentReport.

Extends: UIInput<CommentReport>

Class Data Members: HTMLDivElement ActionTaken, CafeComment CommentData, HTMLDivElement ReasonReported, HTMLDivElement ReportingUsername, HTMLDivElement TimeReported, HTMLButtonElement moderateButton, HTMLInputElement setHiddenTo, HTMLInputElement setRemovedTo, HTMLTextAreaElement reason, HTMLButtonElement submitModerationButton

Class Methods: reportingUserClicked(MouseEvent), moderateButtonClicked(), submitModerationButtonClicked()

CafeLogDisplay

Class Description: CafeLogDisplay displays data from a single AdminAccessLog

Extends: UIInput<AdminAccessLog>

Class Data Members: HTMLDivElement Ip, HTMLDivElement Url, HTMLDivElement Username

Class Methods: usernameClicked(MouseEvent)

CafeModRecordDisplay

Class Description: CafeModRecordDisplay displays data from a ModerationRecord.

Extends: UIInput<ModerationRecord>

Class Data Members: CafeCommentReportDisplay AssociatedReport?, HTMLDivElement ModeratorUsername, HTMLDivElement Reason, HTMLDivElement SetHiddenTo, HTMLDivElement SetRemovedTo, HTMLDivElement TimeModerated

Class Methods: moderatorUsernameClicked(MouseEvent)

CafeGlobalModDisplay

Class Description: CafeGlobalModDisplay renders data from a GlobalModeratorRecord.

Extends: UIInput<GlobalModeratorRecord>

Class Data Members: HTMLDivElement GrantedAt, HTMLDivElement GrantedByUsername, HTMLDivElement GrantedToUsername

Class Methods: grantedByUsernameClicked(MouseEvent), grantedToUsernameClicked(MouseEvent)

CafeDomainModDisplay

Class Description: CafeDomainModDisplay displays data for a DomainModeratorRecord.

Extends: UIInput<DomainModeratorRecord>

Class Data Members: HTMLDivElement Domain, HTMLDivElement GrantedAt, HTMLDivElement GrantedByUsername, HTMLDivElement GrantedToUsername

Class Methods: grantedByUsernameClicked(MouseEvent), grantedToUsernameClicked(MouseEvent)

CafeDomainReportDisplay

Class Description: CafeDomainReportDisplay displays data from an AdminDomainReport object.

Extends: UIInput<AdminDomainReport>

Class Data Members: HTMLDivElement Domain, HTMLDivElement CommentCount, HTMLButtonElement xButton

Class Methods: xButtonClicked()

CafeUsersReportDisplay

Class Description: CafeUsersReportsDisplay displays data from an AdminUsersReport object.

Extends: UIInput<AdminUsersReport>

Class Data Members: HTMLDivElement LoggedInUserCount, HTMLDivElement NewestUsername, HTMLDivElement UserCount

Class Methods: newestUserClicked(MouseEvent), update(AdminUsersReport)

CafeCommentSortDisplay

Class Description: CafeCommentSortDisplay shows the users comment viewing settings and allows the user to change them.

Extends: UIInput<CafeSettings>

Class Data Members: HTMLSelectElement sortBy,HTMLInputElement viewHidden,HTMLInputElement sortAscending

Class Methods: settingChange(), updateFromSettings(CafeSettings)

Functional Descriptions

Package util

dbCredentials

ConnectString() : string

Output: ConnectString forms and returns a Postgres connection string from its members, in the form “host=%s, port=%s, user=%s, password=%s, dbname=%s, sslmode=disable”.

Returns: A string which can be used by the SQL package to connect to a Postgres instance.

loadDBEnv(),

Output: loadDbEnv makes calls to os.GetEnv for each .env variable needed to populate dbCredential’s members. It validates that each exists and loads their values into dbCredential’s members.

Error: If loadDBEnv fails to find any env variables it needs, an error message is printed to the console and the process shuts down. The server cannot connect to the database, and therefore cannot run, without being able to connect to the Database.

config

Load(string)

Input: A path to the .env file.

Files Accessed: A .env text file containing key value pairs for configurations.

Output: The imported godotenv package function Load() is called with the path as a parameter. This makes the environment variables available. Then dbConfig.loadDBEnv() and serverConfig.loadServerEnv() are called to validate the configuration.

Error: If gotdotenv fails to find an env file an error message is printed to the console and the process shuts down. The server and database cannot run without a configuration.

serverConfig

loadServerEnv()

Output: loadServerEnv makes calls to os.GetEnv for each .env variable needed to populate serverConfig's members. It validates that each exists and loads their values into serverConfig's members.

Error: If loadServerEnv fails to find an env variable, an error message is printed to the console and the process shuts down. The server cannot run without a proper configuration.

Package server

UserControllerInterface

Note: While UserControllerInterface is abstract, the implementing Controllers realize these functions in similar manners. For the sake of brevity, only the interface methods are described and the different realizations are hinted at in the Output description.

HandleCommandBan(*communication.Ban, *Server)

Input: A pointer to communication.Ban and a pointer to Server.

Output: If the controller is an Admin or Moderator Controller, Server.UserManager.Ban is called.

HandleCommandChangeEmail(*communication.ChangeEmail, *Server)

Input: A pointer to communication.ChangeEmail and a pointer to Server.

Output: If the controller is a member Controller, the database record for the User is updated with the new email and the is_verified field is set to false until the new email is verified.

HandleCommandChangeFeedback(*communication.ChangeFeedback)

Input: A pointer to communication.ChangeFeedback and a pointer to Server.

Output: If the controller is an Admin controller, the database record for the Feedback is updated to set hidden to true or false, indicating that the feedback has been reviewed.

HandleCommandChangePassword(*communication.SetNewPass, *Server)

Input: A pointer to communication.SetNewPass and a pointer to Server.

Output: If the controller is a Member controller, the user's password is updated in the database.

HandleCommandChangeProfileBlurb(*communication.ChangeProfileBlurb, *Server)

Input: A pointer to communication.ChangeProfileBlur and a pointer to Server.

Output: If the controller is a Member controller, the database record for the User's profile blurb is changed and the profile blurb is updated in the cache memory associated with the User.

HandleCommandCommentReply(*communication.CommentReply, *Server)

Input: A pointer to communication.CommentReply and a pointer to Server.

Output: If the controller is a Member controller, Page.CreateComment is called for the page the controller is on.

HandleCommandCommentVote(*communication.CommentVote, *Server)

Input: A pointer to communication.CommentVote and a pointer to Server.

Output: If the controller is a Member controller, Page.VoteComment is called for the page the controller is on.

HandleCommandFeedback(*communication.Feedback, *Server)

Input: A pointer to communication.Feedback and a pointer to Server.

Output: If the controller is a Member controller, a new Feedback entry is inserted into the Feedbacks table.

HandleCommandGetComments(*communication.GetComments, *Server)

Input: A pointer to communication.GetComments and a pointer to Server.

Output: A new Page is instantiated if one does not already exist for the page that the user wants comments for. GetComments is called for that page, and the returned data is added to the nextResponse field for the controller.

HandleCommandGetUserProfile(*communication.GetUserProfile, *Server)

Input: A pointer to communication.GetUserProfile and a pointer to Server.

Output: The database is queried and Server-Client communication Entity, UserProfile, is instantiated and added to the nextResponse field for the controller.

HandleCommandLogin(*communication.Login, *Server),

Input: A pointer to communication. Login and a pointer to Server.

Output: If the controller is a GuestController, it calls UserManager.Login.

HandleCommandLogout(*Server)

Input: A pointer to the server.

Output: If the controller is a Member Controller, it calls UserManager.Logout.

HandleCommandModerate(*communication.Moderate, *Server)

Input: A pointer to communication.Moderate and a pointer to Server.

Output: If the controller is at least a Moderator controller, and they have permission to moderate that domain, PageManager.Moderate is called.

HandleCommandPasswordResetCode(*communication.PasswordResetCode, *Server)

Input: A pointer to communication.PasswordResetCode, and a pointer to Server.

Output: If the client inputs a correct request, the database field for that password reset is updated to reflect that a valid code has been submitted and the client may enter a new password.

HandleCommandPasswordResetRequest(*communication.PasswordResetRequest, *Server)

Input: A pointer to communication.PasswordResetRequest containing the clients email, and a pointer to Server.

Output: If an email associated with a user is submitted, a new record is inserted into the PasswordResetCodes table and an email is dispatched containing the code.

HandleCommandCommentReport(*communication.PostCommentReport, *Server)

Input: A pointer to communication.PostCommentReport and a pointer to Server.

Output: If the controller has appropriate access, a new record is inserted into the CommentReports table.

HandleCommandRequestVerification(*communication.RequestVerification, *Server)

Input: A pointer to communication.RequestVerification and a pointer to Server.

Output: A new record is inserted into the VerificationCodes table and an email is dispatched containing the verification code.

HandleCommandVerify(*communication.Verify, *Server)

Input: A pointer to communication.Verify and a pointer to Server.

Output: If the code matches the data in the VerificationCodes table, the is_verified field of the User record is changed to true, indicating that the client has verified their email.

HandleCommandViewBans(*communication.ViewBans, *Server)

Input: A pointer to communication.ViewBans and a pointer to Server.

Output: If the controller is one with appropriate access, an array of the server-client communication entity BanRecords is created from data in the BanRecords table, converted to a packet, and added to the nextResponse field of the Controller.

HandleCommandViewCommentReports(*communication.ViewCommentReports, *Server)

Input: A pointer to communication.ViewCommentReports and a pointer to Server.

Output: If the controller is one with appropriate access, an array of the server-client communication entity CommentReport is created from data in the CommentReports table, converted to a packet, and added to the nextResponse field of the Controller.

HandleCommandViewLogs(*communication.ViewLogs, *Server)

Input: A pointer to communication.ViewLogs and a pointer to Server.

Output: If the controller is one with appropriate access, an array of the server-client communication entity AdminAccessLog is created from the data in the Logs table, converted to a packet, and added to the nextResponse field of the Controller.

HandleCommandViewModRecords(*communication.ViewModRecords, *Server)

Input: A pointer to communication.ViewModRecords and a pointer to Server.

Output: If the controller is one with appropriate access, an array of the server-client communication entity ModerationRecord is created from data in the ModerationRecords table,

HandleCommandViewMods(*communication.ViewMods, *Server)

Input: A pointer to communication.ViewMods, and a pointer to Server.

Output: If the controller is one with appropriate access, an array of the server-client communication entities DomainModeratorRecords or an array of GlobalModeratorRecords, depending on the client request, is converted to a packet and added to the nextResponse field of the controller.

Respond(r http.Request, w http.ResponseWriter), GetCurrentPage() :*Page, dispatchResponse(r http.Request, w http.ResponseWriter)

Input: A pointer to the http.Request and an http.ResponseWriter

Output: The controller responds with all packets saved in its nextResponse field by writing them to the body of the http.ResponseWriter

UserManager

Ban(*communication.Ban, *Server server)

Input: A pointer to a communication.Ban entity and a pointer to Server.

Output: If the User is bannable, a new record is created in the database and the target user's banned field may be changed. If a controller instance is active for that user, it is deleted.

Login(*UserControllerInterface, *communication.Login, *Server server): *UserControllerInterface

Input: A pointer to a communication.Login and a pointer to Server.

Output: If the username and password are valid, A new UserControllerInterface is instantiated and added to the UserManager.members map or retrieved from the instantiated controllers. The current guest controller is deleted and removed from the guests map. The new controller is added to the page the guest was on, and the guest controller is removed from that map as well. A communication.LoginResponse entity is created and added to the nextResponse field of the controller.

Returns: A pointer to the newly instantiated UserControllerInterface.

Logout(*UserControllerInterface, *Server server): *GuestController

Input: None

Output: The controller is removed from the UserManager.members map and the map at the page the controller was on. A new GuestController is instantiated and placed in the UserManager.guests map and the guests map of the page the user was on. A

communication.LogoutResponse entity is created and added to the nextResponse field of the guest controller.

Returns: A pointer to the newly instantiated GuestControllerInterface

Register(*UserControllerInterface, *communication.Register, *Server server)

Input: A pointer to the GuestController previously associated with the user, a pointer to the communication.Register entity, and a pointer to the Server.

Output: If the user doesn't already exist and has a valid name and password, a new user is created in the database and communication.Login entity is constructed and UserManger.Login is called.

GetMemberController(int64 id): *UserControllerInterface

Input: A 64-bit integer representing a userID.

Output: If the controller does not exist in the UserManager.members map, a new controller is instantiated.

Returns: A pointer to the associated controller.

GetGuestController(int64 id): *UserControllerInterface

Input: A 64-bit integer representing a temporary guest userID.

Output: If the controller does not exist in the UserManager.guests map, a new GuestController is instantiated.

Returns: A pointer to the associated controller.

DispatchPasswordResetEmail(*UserControllerInterface, *Server server)

Input: A pointer to a user controller interface and a pointer to Server.

Output: A new entry is created in the PasswordResetCodes table and an email with that code is dispatched to the email associated with the UserControllerInterface.

PageManager

MoveMemberToPage(*UserControllerInterface user, string pagePath, *Server server),

Input: A pointer to a UserControllerInterface, a string representing the path, and a pointer to the server.

Output: A new page is instantiated if necessary. The UserControllerInterface is removed from the members map on its current Page and added to the new Page.

MoveGuestToPage(*UserControllerInterface user, string pagePath, *Server server),

Input: A pointer to a UserControllerInterface, a string representing the path, and a pointer to the server.

Output: A new Page is instantiated if necessary. The UserControllerInterface is removed from the guest map on its current Page and added to the new Page.

UnloadEmptyPages(*Server server)

Input: A pointer to Server.

Output: Every instantiated Page in PageManger.pages is iterated through. References to pages which have no controllers in their users or guests map are removed from PageManager.pages to allow the garbage collector to delete them.

loadPage(string path, *Server server)

Input: A string representing a URL path.

Output: The database is queried for necessary data to instantiate a Page for that path. The Page is added to the pages map of PageManager.

Page

GetComments(string sortedBy, bool ascending): communication.Comment

Input: A string representing the field to sort by and a Boolean representing whether to sort ascending or descending.

Returns: The cachedComments on the Page are iterated through and sorted in the appropriate order. They are returned as an array.

addMemberToPage(*UserControllerInterface user)

Input: A pointer to a UserControllerInterface

Output: The UserControllerInterface is added to the members map on the page.

removeMemberFromPage(*UserControllerInterface user)

Input: A pointer to a UserControllerInterface

Output: The UserControllerInterface is removed from the members map on the page.

addGuestToPage(*UserControllerInterface user)

Input: A pointer to a UserControllerInterface

Output: The UserControllerInterface is added to the guests map on the page.

removeGuestFromPage(*UserControllerInterface user)

Input: A pointer to a UserControllerInterface

Output: The UserControllerInterface is removed from the guests map on the page.

Moderate(*communication.Moderate, *Server)

Input: A pointer to a communication.Moderate entity and a pointer to Server.

Output: A new record is inserted into the ModerationActions table. The CachedComment on the page is updated. Comment changes are pushed to all users viewing that Page. The underlying comment data is updated in the database.

CreateComment(*communication.CommentReply, *Server server)

Input: A pointer to a communication.CommentReply entity and a pointer to Server.

Output: A new record is inserted into the Comments table. A new CachedComment is instantiated from that data and added to the comments map of the page. Comment changes are pushed to the nextResponse field of all users viewing that page.

VoteComment(*communication.CommentVote, *Server server)

Input: A pointer to a communication.CommentVote entity and a pointer to Server.

Output: Vote is called on the CachedComment associated with the communication.CommentVote entity.

CachedComment

Vote(*communication.CommentVote, *Server server)

Input: A pointer to a communication.CommentVote entity and a pointer to Server.

Output: A new entry is inserted into the CommentVotes table and a new CachedCommentVote is instantiated and added to the votes map of the CachedComment.

getDataForGuest(): communication.Comment

Input: None

Output: The cached votes data is aggregated and converted into a communication.Comment.

Returns: A communication.Comment

getDataForUser(int64 userId) : communication.Comment

Input: A number representing a user ID

Output: The cached votes data is aggregated into communication.CommentVotes. The userID allows the population of the “alreadyVoted” field.

Returns: A communication.Comment

Server

New(): *Server

Input: None

Output: A new Server is instantiated.

setupRouter()

Input: None

Output: Routing is set up and middleware is attached.

Start()

Input: None

Output: The server begins listening on the port configured in the .env file.

MiddlewareAttachController(handler http.Handler): http.Handler

Input: A handler function, which is one that takes a pointer to an http.Request and an http.ResponseWriter as parameters.

Output: Middleware to extract the token, instantiate a controller, and attach it to the http.Request.Context wraps the parameter function. The new wrapping function is returned.

<Server API endpoint functions>(*http.Request, http.ResponseWriter)

Note: The many endpoint functions are compressed here for the sake of brevity.

Input: A pointer to an http.Request and an http.ResponseWriter.

Output: The Server retrieves the controller from the http.Request. It extracts the relevant communication entity from the http.Request.body. It passes that entity and a reference to itself to the associated handler method of the extracted controller.

Package database

Store

New(): Store

Input: None

Output: Instantiates a new Store and returns it.

Connect()

Input: None

Output: Uses environment variables configured in a secret .env file to connect to the Postgres server on another port.

Disconnect()

Input: None

Output: Disconnects from the Postgres instance.

Queries

Note: *The methods of Queries are generated by sqlc. They take a context.Context as a parameter but it is not used.*

Files Accessed: Queries methods ultimately change files created internally by postgres which it needs to realize the database.

CreateDomainModeratorAssignment(context.Context, CreateDomainModeratorAssignmentParams)

Input: A CreateDomainModeratorAssignmentParams object.

Output: Inserts a new record into the DomainModeratorAssignments table.

CreateGlobalModeratorAssignment(context Context, CreateGlobalModeratorAssignmentParams)

Input: A CreateGlobalModeratorAssignmentParams object.

Output: Inserts a new record into the GlobalModeratorAssignments table.

CreateDomainBanRecord(context Context, CreateDomainBanRecordParams)

Input: A CreateDomainBanRecordParams object.

Output: Inserts a new record into the BanRecords table.

UpdateUserBanStatus(ctx context.Context, UpdateUserBanStatusParams)

Input: A UpdateUserBanStatusParams object.

Output: Updates the users banned status in the Users table.

UpdateUserEmail(ctx context.Context, UpdateUserEmailParams)

Input: A UpdateUserEmailParams object.

Output: Updates the users email in the Users table.

UpdateFeedbackHidden(context.Context, UpdateFeedbackHiddenParams),

Input: An UpdateFeedbackHiddenParams object.

Output: Updates an entry in the Feedbacks table to set the field “hidden” to true or false.

UpdateUserBlurb(context.Context, UpdateUserBlurbParams),

Input: An UpdateUserBlurbParams object.

Output: Updates and set’s the user’s profile blurb to a specified value by the user.

CreateComment(context.Context, CreateCommentParams),

Input: A CreateCommentParams object.

Output: Inserts a comment into the Comments table.

CreateCommentVote(context.Context, CreateCommentVoteParams),

Input: A CreateCommentVoteParams object.

Output: Inserts a vote into the VoteRecords table.

UpdateCommentVote(context.Context, UpdateCommentVoteParams)

Input: An UpdateCommentVoteParams object.

Output: Updates a record in the VoteRecords table.

DeleteCommentVote(context.Context, DeleteCommentVoteParams)

Input: A DeleteCommentVoteParams object.

Output: Removes a record from the VoteRecords table.

CreateFeedback(context.Context, CreateFeedbackParams)

Input: A CreateFeedbackParams object

Output: Inserts a new record into the Feedbacks table.

GetCommentsAtPath(context.Context, int64 pathID): []GetCommentsAtPathRow

Input: An ID associated with an entry in the Paths table.

Output: Returns records from the Comment table.

Returns: An array of GetCommentsAtPathRow objects.

GetCommentVotes(context.Context, int64 commentID): []GetCommentVotesRow,

Input: An ID associated with an entry in the Comments table.

Output: Returns records from the VoteRecords table.

Returns: An array of GetCommentVotesRow objects.

GetUserByUserID(context.Context, int64 id): User

Input: An ID associated with an entry in the Users table.

Output: Returns records from the Users table.

Returns: An Entry of GetCommentVotesRow objects.

CreateModerationRecord(context.Context, CreateModerationRecordParams)

Input: An CreateModerationRecordParams object.

Output: Creates a record in the BanActions table.

UpdateCommentHidden(context.Context, UpdateCommentHiddenParams)

Input: An UpdateCommentHiddenParams object.

Output: Updates a record in the Comments table.

UpdateCommentRemove(context.Context, UpdateCommentRemoveParams)

Input: An UpdateCommentRemoveParams object.

Output: Updates a record in the Comments table.

CreateLog(context.Context, CreateLogParams)

Input: A CreateLogParams object.

Output: Creates a record in the Logs table.

GetDomainModeratorAssignments(context.Context, int64 id)

Input: An ID associated with an entry in the DomainModeratorAssignments table.

Output: Returns records from the DomainModeratorAssignments table.

GetGlobalModeratorAssignments(context.Context, int64 id)

Input: An ID associated with an entry in the GlobalModeratorAssignments table.

Output: Returns records from the GlobalModeratorAssignments table.

GetAdminAssignments(context.Context, int64 id)

Input: An ID associated with an entry in the AdminAssignments table.

Output: Returns records from the AdminAssignments table.

GetUserByUsername(context.Context, string username): User

Input: A username associated with an entry in the Users table.

Output: Returns records from the Users table.

UpdateUserLastLogin(context.Context, int64 id)

Input: An ID associated with an entry in the Users table.

Output: Updates a record in the Users table.

CreateCommentReport(context.Context, CreateCommentReportParams)

Input: A CreateCommentReportParams object.

Output: Creates a record in the CommentReports table.

CreateUser(context.Context, CreateUserParams)

Input: A CreateUserParams object.

Output: Creates a record in the Users table.

DeleteUser(context.Context, int64 id)

Input: An ID associated with an entry in the Users table.

Output: Remove a record in the Users table.

GetUserByUsername(context.Context, string username): User

Input: A username associated with an entry in the Users table.

Output: Returns a record in the Users table.

GetUserByEmail(context.Context, string email): User

An Email associated with an entry in the Users table.

Output: Returns a record in the Users table.

UpdateUserPassword(context.Context, UpdateUserPasswordParams)

Input: A CreateUserParams object.

Output: Updates a record in the Users table.

UpdateUserVerification(context.Context, UpdateUserVerificationParams)

Input: An UpdateUserVerificationParams object.

Output: Updates a record in the Users table.

CreateVerificationRecord(context.Context, CreateVerificationRecordParams)

Input: A CreateVerificationRecordParams object.

Output: Creates a record in the VerificationCodes table.

GetVerificationRecord(context.Context, int64 userID): []VerificationCode

Input: A userID associated with an entry in the VerificationCodes table.

Output: Returns a record in the VerificationCodes table.

Returns: An Entry of VerificationCode objects.

DeleteVerificationRecords(context.Context, int64 userID)

Input: A userID associated with an entry in the VerificationCodes table.

Output: Removes a record in the VerificationCodes table.

GetPWResetRecord(context.Context, int64 userID): []PasswordResetCode

Input: A userID associated with an entry in the PasswordResetCodes table.

Output: Returns a record in the PasswordResetCodes table.

Returns: An Entry of PasswordResetCode objects.

CreatePWResetRecord(context.Context, CreatePWResetRecordParams)

Input: A CreatePWResetRecordParams object.

Output: Creates a record in the PasswordResetCodes table.

DeletePWResetRecords(context.Context, int64 userID)

Input: A userID associated with an entry in the PasswordResetCodes table.

Output: Creates a record in the VerificationCodes table.

GetBanRecords(context.Context): []GetBanRecordsRow

Output: Returns a record in the Users table.

GetCommentReports(context.Context, Boolean actionTaken): []GetCommentReportsRow

Input: An actionTaken associated with an entry in the CommentReports table.

Output: Returns a record in the CommentReports table.

Returns: An array of GetCommentReportsRow objects.

GetFeedback(context.Context, Boolean hidden) []GetFeedbackRow

Input: An actionTaken associated with an entry in the Feedbacks table.

Output: Returns a record in the Feedbacks table.

Returns: An array of GetFeedbackRow objects.

GetLogsForDateRange(context.Context, GetLogsForDateRangeParams):
[]GetLogsForDateRangeRow

Input: A GetLogsForDateRangeParams object.

Output: Returns a record in the Logs table.

Returns: An array of Logs objects.

GetModRecordsForModerator(context.Context, int64 id):

[]GetModRecordsForModeratorRow

Input: An ID associated with an entry in the CommentModeratorActions table.

Output: Returns a record in the CommentModeratorActions table.

GetDomainModerators(context.Context, string domain): []GetDomainModeratorsRow

Input: An domain associated with an entry in the DomainModeratorAssignments table.

Output: Returns a record of the DomainModeratorAssignments table.

Returns: An array of GetDomainModeratorsRow objects.

GetGlobalModerators(context.Context): []GetGlobalModeratorsRow

Input: An domain associated with an entry in the GlobalModeratorAssignments table.

Output: Returns a record of the GlobalModeratorAssignments table.

Returns: An array of GetGlobalModeratorsRow objects.

GetAdmins(context.Context): []GetAdminsRow

Input: An domain associated with an entry in the AdminAssignments table.

Output: Returns a record of the AdminAssignments table.

Returns: An array of GetAdminsRow objects.

GetNewestUser(context.Context): User

Output: Returns a record of the Users table.

Returns: An Entry of a Users object.

GetUserCount(context.Context): int64

Output: Returns a record of the Users table.

Returns: An Entry of a Users object.

GetLogsForIP(context.Context, string IP): []GetLogsForIPRow

Input: An IP associated with an entry in the Logs table.

Output: Returns a record of the Logs table.

Returns: An Entry of a Logs object.

GetLogsForUser(context.Context, int64 userID): []GetLogsForUserRow

Input: An IP associated with an entry in the Logs table.

Output: Returns a record of the Logs table.

Returns: An Entry of a Logs object.

Front End

Cafe

userChange(UserProfile | undefined)

Input: userChange takes a new profile for the user using comment anywhere, such as resulting from a log-in. It can take no value, or undefined, to effectively log the user out.

Output: userChange is called on a log in, a log out, and at all other times when user data changes (such as updating their profile blurb). It calls loadProfile on the State object and passes that state to other objects that may need to change what is visible if the users access level has changed.

pageChange(string)

Input: URL string of new page.

Output: pageChange is called when a user navigates to a new page. It tells the fetcher to fetch comments.

changeState(string)

Input: A string corresponding to the new state key.

Output: changeState updates the viewing field in the state object and passes the state to the navbar so that it may render the new state.

checkForResponses()

Output: checkForResponses is called as a callback after every fetch. The server responses array is retrieved from the fetcher and passed to the dispatcher, along with a reference to cafe so the dispatcher can call the correct methods to realize the information retrieved from the server.

stateChangeEventReceived()

Output: stateChangeEventReceived listener is called when a user performs an action that causes a state change. It parses the event and calls changeState to realize the change.

clientEventReceived()

Output: clientEventReceived is called when an event is picked up that reflects an action performed by the user that requires server communication. It calls fetcher.fetch with the data from that event.

settingEventReceived(Settings)

Input: A new Settings object.

Output: settingsEventReceived is called when a user changes their settings, such as when they change whether comments are sorted by ascending or descending. It updates the settings object with the new data and tells the appropriate objects to update their displayed data based on the new settings.

Fetcher

fetch(string, communication entity, Function)

Input: A string corresponding the correct API endpoint and a client-server communication entity associated with that endpoint.

Output: Fetch dispatches the HTTP Request to the endpoint, stores responses in the responses member of fetcher, and calls the callback function when complete.

getAndClearResponses()

Output: Clears the responses member of fetcher.

Returns: The data in the responses member of fetcher (an array of server responses).

State

loadProfile(UserProfile | undefined)

Input: A UserProfile to load or nothing, to clear the user.

Output: Changes the ownProfile member of State.

setViewingTo(string)

Input: A string representing a state key.

Output: Changes the viewing member of State to the parameter value.

CafeNavbar

commentsButtonClicked()

Output: commentsButtonClicked is called when commentsButton is clicked. It will cause the state to transition.

settingsButtonClicked()

Output: settingsButtonClicked is called when settingsButton is clicked. It will cause the state to transition.

modButtonClicked()

Output: modButtonClicked is called when modButton is clicked. It will cause the state to transition.

adminButtonClicked()

Output: adminButtonClicked is called when adminButton is clicked. It will cause the state to transition.

loginButtonClicked()

Output: loginButtonClicked is called when loginButton is clicked. It will cause the state to transition.

registerButtonClicked()

Output: registerButtonClicked is called when registerButton is clicked. It will cause the state to transition.

logoutButtonClicked()

Output: logoutButtonClicked is called when logoutButton is clicked. It will cause the state to transition.

setFromState(State)

Input: A State Object

Output: setFromState causes buttons to show and hide depending on whether the user is logged in, what their access level is, and what state they are in. It also changes the currentlyViewing member to the appropriate CafeWindow and hides all other windows.

displayMessage(Message)

Input: A Message object.

Output: displayMessage shows the data in the parameter Message in the globalMessage member of CafeNavBar; this member is used to display a variety of general messages the server wants the client to see.

Dispatcher

dispatch(serverResponse[], Cafe)

Input: An array of server responses and an instance of Cafe.

Output: Dispatch uses the "t" field of the ServerResponse to determine what kind of information is contained in the "d" field. The "t" field determines which Dispatcher method is next called, and the object which will render the data is retrieved using the parameter reference to Cafe.

dispatchMessage(Message, CafeNavbar)

Input: A Message and a CafeNavBar.

Output: dispatchMessage calls displayMessage on the CafeNavbar object.

dispatchUserUpdate(UserProfile, Cafe)

Input: UserProfile data and the Café root object.

Output: dispatchUserUpdate calls userChange on the Cafe root object to change state reflecting any changes that may have happened to the User and to change what is visible on their profile.

dispatchFeedbackUpdate(FeedbackRecord, FeedbackReportSection) *Input*: A FeedbackRecord and a FeedbackReportSection.

Output: dispatchFeedbackUpdate calls updateFeedback on a FeedbackReportSection.

dispatchCommentUpdate(Comment, CafeCommentsWindow) *Input*: A Comment with data and a CafeCommentsWindow that will be instantiating and displaying the CafeComment.

Output: dispatchCommentUpdate calls updateComment on a CafeCommentsWindow

dispatchComments(Comment[], CafeCommentsWindow) *Input*: A list of comments, and the window to render the comments to.

Output: dispatchComments calls populateNewComments on a CafeCommentsWindow

dispatchUserProfile(UserProfile, CafeUserDisplay)

Input: The user profile data, and the user display window to render it to.

Output: dispatchUserProfile calls changeProfile on a CafeUserDisplay to allow the user to view profile information for some other user.

dispatchLoginResponse(LoginResponse, Cafe)

Input: The server's response regarding the user's login attempt, and the Cafe instance to target.

Output: dispatchLoginResponse calls userChange on the Cafe root object to change state reflecting any changes that may have happened to the User and to change what is visible on their profile.

dispatchLogoutResponse(LogoutResponse, Cafe)

Input: The server's response to the user logging out, and the Cafe instance to target.

Output: userChange is invoked on the Cafe instance to change state reflecting the logging-out of the user profile.

dispatchPwResetCodeResponse(any, Cafe)

Input: An object that is typically empty, but is reserved for any additional information the server may respond with. There is also the target Cafe object to inflict the new "password change" state on.

Output: The target Cafe object state is set to input a new password.

dispatchPwResetReqResponse(any, Cafe)

Input: An object that is typically empty but is reserved for any additional information the server may respond with. There is also the target Cafe object to inflict the new "password change" state on.

Output: The target Cafe object state is set to input the verification code.

dispatchReqVerificationResponse(Message, CafeMessageDisplay)

Input: The Message contents and the target CafeMessageDisplay to update with that content.

Output: The CafeMessageDisplay's text content indicates a verification email has been dispatched.

dispatchNewPassResponse(Message, CafeMessageDisplay)

Input: The Message contents and the target CafeMessageDisplay to update with that content.

Output: The CafeMessageDisplay's text content indicates the user's password has been updated.

dispatchVerifyResponse(Message, CafeMessageDisplay)

Input: The Message contents and the target CafeMessageDisplay to update with that content.

Output: The CafeMessageDisplay's text content indicates the user's email has been verified.

dispatchBanRecords(BanRecord[], BanRecordsSection)

Input: The list of BanRecords received from the server, and the target BanRecordsSection to update.

Output: Populates the Cafe BanRecordsSection with the list of BanRecords received from the server.

dispatchCommentReports(CommentReport[], CommentReportsSection)

Input: The list of CommentReports received from the server, and the target CommentReportsSection to update.

Output: Populates the Cafe CommentReportsSection with the list of CommentReports received from the server.

dispatchCommentReportUpdate(CommentReport, CommentReportsSection)

Input: A CommentReport that has been updated server-side, and the CommentReportsSection to update with this new data.

Output: The specified Cafe CommentReportsSection is updated to contain the new CommentReport.

dispatchFeedbacks(FeedbackRecord[], FeedbackReportSection)

Input: The list of FeedbackRecords received from the server and the target FeedbackReportSection to be populated with this list.

Output: The FeedbackReportSection entity is populated with the FeedbackRecords content. This information is displayed to the user.

dispatchLogs(AdminAccessLog[], LogsSection)

Input: The list of AdminAccessLogs received from the server and the target LogsSection to be populated with this list.

Output: The LogsSection entity is populated with the AdminAccessLogs content. This information is displayed to the user.

dispatchModRecords(ModerationRecord[], ModActionsReportSection)

Input: The list of ModerationRecords received from the server and the target ModActionsReportSection to be populated with this list.

Output: The ModActionsReportSection entity is populated with the ModerationRecords content. This information is displayed to the user.

dispatchGlobalMods(GlobalModeratorRecord[], ModeratorsReportSection)

Input: The list of GlobalModeratorRecords received from the server and the target ModeratorsReportSection to be populated with this list.

Output: The ModeratorsReportSection entity is populated with the GlobalModeratorRecords content. This information is displayed to the user.

dispatchDomainMods(DomainModeratorRecord[], ModeratorsReportSection)

Input: The list of DomainModeratorRecords received from the server and the target ModeratorsReportSection Section to be populated with this list.

Output: The ModeratorsReportSection entity is populated with the DomainModeratorRecords content. This information is displayed to the user.

dispatchDomainReport(AdminDomainReport, CafeAdminWindow)

Input: The list of AdminDomainReports received from the server and the target CafeAdminWindow Section to be populated with this list.

Output: The CafeAdminWindow entity is populated with the AdminDomainReports content. This information is displayed to the user.

dispatchUsersReport(AdminUsersReport, CafeAdminWindow)

Input: The list of AdminUsersReports received from the server and the target CafeAdminWindow Section to be populated with this list.

Output: The CafeAdminWindow entity is populated with the AdminUsersReports content. This information is displayed to the user.

CafeWindow

show()

Output: Shows a CafeWindow instance.

hide()

Output: Hides a CafeWindow instance.

CafeCommentsWindow

populateNewComments(Comment[])

Input: The list of new comment data.

Output: All instances of CafeComment is cleared, and new instances are created for every Comment in the parameter array.

updateComment(Comment)

Input: The comment to be updated.

Output: Updated data for the target CafeComment. A new CafeComment is created if the target doesn't exist.

resortComments()

Output: Removes and resorts all CafeComments on the page according to the user's settings.

updateFromSettings(CafeSettings)

Input: CafeSettings instance specifying comment viewing preferences.

Output: Saves previous settings and calls resortComments.

CafeSettingsWindow

updateFromSettings(CafeSettings)

Input: CafeSettings instance containing modified user settings.

Output: Calls updateFromSettings for the comment sort settings.

verifyEmailClicked()

Output: Calls show() to show the validation code input window.

verifyCodeSubmitClicked()

Output: Dispatch a Verify object to the server to verify the input code.

passwordResetClicked()

Output: When the user clicks the password reset button, a password reset request will be dispatched to the server, the user will be logged out, and the password change state transition process will begin.

CafeAdminWindow

domainRequestButtonClicked()

Output: It dispatches a ViewDomainReport entity to the server with a value retrieved from the domainInput element.

showDomainReport(AdminDomainReport)

Input: The AdminDomainReport response from the server.

Output: Instntiates a new CafeDomainReportDisplay object.

usersReportRequestButtonClicked()

Output: When the usersReportButton is clicked, it will dispatch a ViewUsersReport entity to the server.

showUsersReport(AdminUsersReport)

Input: An AdminUsersReport response from the server.

Output: Updates userReport with the data from the AdminUsersReport object.

CafeLoginWindow

submitLoginButtonClicked()

Input: The user clicks the “submit” button on the login page.

Output: Will dispatch a Login entity to the server containing username and password login information.

forgotPasswordButtonClicked()

Input: The user clicks on the “forgot my password” button on the login page.

Output: It causes a PasswordResetRequest to be dispatched to the server and the state to transition to the request password reset form.

CafeRegisterWindow

submitRegisterClicked()

Input: The user clicks the “submit” button on the register page.

Output: A Register entity will be dispatched to the server containing the registration data.

CafePwResetRequestWindow

submitButtonClicked()

Input: The user clicks the “submit” button after entering their email.

Output: A PasswordResetRequest entity will be dispatched to the server.

CafePWResetCodeWindow

submitButtonClicked()

Input: The user clicks “submit” after entering the password reset code.

Output: A PasswordResetCode entity will be dispatched to the server.

CafeNewPasswordWindow

submitButtonClicked()

Input: The user clicks “submit” after entering their new password.

Output: Will dispatch a SetNewPass entity to the server.

FeedbackReportSection

updateFeedback(FeedbackRecord)

Input: Updated feedback data from the server.

Output: The feedback is updated on the client-side.

requestFeedbackClicked()

Input: The user clicks the requestFeedbackButton.

Output: Will dispatch a ViewFeedback entity to the server containing data from the input controls.

populateFeedback(FeedbackRecord[])

Input: Cafe receives a Feedback entity from the server, which contains data for feedback to display.

Output: It causes the section to instantiate CafeFeedback objects for each feedback item after clearing the extant ones. This will call the clearFeedback() function.

clearFeedback()

Output: Destroys all CafeFeedbackDisplay entities in the window.

ModeratorsReportSection

requestDomainModsClicked()

Input: The user clicks the requestDomainModerators button.

Output: Will dispatch a ViewMods entity to the server.

requestGlobalModsClicked()

Input: The user clicks on the requestGlobalModerators button.

Output: Will dispatch a ViewMods object to the server.

populateDomainModerators(DomainModeratorRecord[]) *Input*: Cafe receives a DomainModeratorRecord from the server.

Output: It instantiates CafeDomainModDisplays for each entry.

populateGlobalModerators(GlobalModeratorRecord[]) *Input*: Cafe receives a GlobalModeratorRecord array from the server.

Output: A CafeGlobalModDisplay is initialized for each entry.

clearModsClicked()

Input: A user clicks the button to clear all moderator records.

Output: Destroys all CafeDomainModDisplays and CafeGlobalModDisplays.

ModActionsReportSection

requestModRecordsButtonClicked()

Input: A user clicks the button to request moderator records.

Output: A CafeModRecordDisplay is initialized for each ModerationRecord.

populateModActions(ModerationRecord[])

Input: A list of ModerationRecords received from the server.

Output: A CafeModRecordDisplay is initialized for each ModerationRecord.

updateModAction(ModerationRecord)

Input: An updated ModerationRecord.

Output: The record's data will be updated.

clearModActionsClicked()

Input: A user clicks the button to clear moderator actions.

Output: Destroys all CafeModRecordDisplays currently visible.

BanRecordsSection

populateBanRecords(BanRecord[])

Input: A list of BanRecords is received from the server.

Output: Clears the currently displayed ban records and instantiates a new CafeBanRecord for each BanRecord.

requestGlobalBansClicked()

Input: A user clicks the button to request global bans.

Output: A ViewBans request is dispatched to the server.

requestDomainBansClicked()

Input: A user clicks the button to request domain bans.

Output: A ViewBans request is dispatched to the server containing the specific dimain field.

clearBanRecords()

Output: Destroys all instances of CafeBanRecord.

CommentReportsSection

populateCommentReports(CommentReport[])

Input: A list of CommentReports is received from the server.

Output: Creates an instance of CafeCommentReportDisplay for each CommentReport after destroying the ones currently visible.

viewReportsClicked()

Input: A user clicks the button to view reports.

Output: A ViewCommentReports object is dispatched to the server.

updateCommentReport(CommentReport)

Input: The most recent comment report data.

Output: CafeCommentReportDisplay is updated with the new data.

clearCommentReports()

Output: All instances of CafeCommentReportDisplay is destroyed.

LogsSection

submitLogsRequestClicked()

Input: A user clicks the button to submit a logs request.

Output: A ViewLogs instance is dispatched to the server.

populateLogs(AdminAccessLog[])

Input: Cafe receives a list of AdminAccessLogs.

Output: Current logs are cleared and a new CafeLogDisplay is initialized for each AdminAccessLog.

clearLogs()

Output: Destroys all CafeLogDisplay instances currently visible.

UIInput<Type>

clickListen(HTMLElement | HTMLElement[], Function | Function[], Boolean)

Input: One or more HTML elements to associate a click listener with. There will also be one or more functions to invoke upon clicking one of these HTML elements. There is a boolean option to bind each function to each element.

Output: If the option to bind the functions is true, the function will iterate through each function and bind it to the current instance of UIInput. If this option is false, this process will be skipped. Next, for each HTML element and function, add a new event listener to the element, and record the listener as a triplet, containing the element, the function, and the event, specified as a mouse click.

disable()

Output: Overlay an element on top of the button to temporarily block mouse clicks.

enable()

Output: Hide the overlaying element to once again allow access to the button.

destroy()

Output: Iterate through the recorded listeners and remove each listener given the anonymous function and event name fields.

CafeComment

replyClicked()

Input: A user clicks the reply button for a specific comment.

Output: The reply container will be shown if previously hidden, and vice versa.

submitReplyClicked()

Input: A user clicks the button to submit a reply.

Output: A CommentReply client-server communication entity is constructed from data in HTMLInput elements and the underlying comment data. It is emitted on the global document object which will subsequently be picked up by Cafe and sent to the server.

reportClicked()

Input: A user clicks the report button for a specific comment.

Output: The report container will be shown if previously hidden, and vice versa.

submitReportClicked()

Input: A user clicks the button to submit a report.

Output: A PostCommentReport client-server communication entity is constructed from data in HTMLInput elements and the underlying comment data. It is emitted on the global document object which will subsequently be picked up by Cafe and sent to the server.

CafeCommentVote

upVoteClicked()

Input: A user clicks the button to upvote a specific comment.

Output: A CommentVote client-server communication entity is constructed from data in the desired rating type and the underlying comment data. It is emitted on the global document object which will subsequently be picked up by Cafe and sent to the server.

downVoteClicked()

Input: A user clicks the button to downvote a specific comment.

Output: A CommentVote client-server communication entity is constructed from data in the desired rating type and the underlying comment data. It is emitted on the global document object which will subsequently be picked up by Cafe and sent to the server.

CafeMessageDisplay

updateMessage(Message)

Input: The Message object to update the CafeMessageDisplay with.

Output: The CafeMessageDisplay is updated with the new Message.

clearMessage()

Output: The previous CafeMessageDisplay contents is cleared.

CafeUserDisplay

changeProfile(UserProfile)

Input: Updated UserProfile information.

Output: The CafeUserDisplay is updated to contain the new UserProfile information. This information is shown to a member viewing someone else's profile page.

hide()

Output: Hides the CafeUserDisplay object.

show()

Output: Shows the CafeUserDisplay object.

CafeFeedbackDisplay

hideClicked()

Input: The user clicks the button to toggle a feedback's hidden status.

Output: A ChangeFeedback instance is sent to the server to show or hide a specific feedback instance.

CafeOwnProfileDisplay

updateProfile(UserProfile)

Input: Updated UserProfile information.

Output: The CafeUserDisplay is updated to contain the new UserProfile information.

editProfileBlurbClicked()

Input: The user clicks the button to edit their profile blurb information.

Output: A text input field is shown. This will allow the user to update their profile blurb information.

editProfileSubmitClicked()

Input: The user clicks the button to submit their updated blurb information.

Output: A ChangeProfileBlurb event is sent to the server.

CafeBanRecordDisplay

bannedUserClicked(MouseEvent)

Input: The mouse position to set the UserProfileDisplay position to.

Output: The server will retrieve UserProfile data for the banned user, the Cafe UserProfileDisplay will be populated with this information, and the UserProfileDisplay will be moved to the mouse pointer location.

bannedByClicked(MouseEvent)

Input: The mouse position to set the UserProfileDisplay position to.

Output: The server will retrieve UserProfile data for the “banned by” user, the Cafe UserProfileDisplay will be populated with this information, and the UserProfileDisplay will be moved to the mouse pointer location.

CafeCommentReportDisplay

reportingUserClicked(MouseEvent)

Input: The mouse position to set the UpdateProfileDisplay position to.

Output: The server will retrieve UserProfile data for the "reporting user", the Cafe UserProfileDisplay will be populated with this information, and the UserProfileDisplay will be moved to the mouse pointer location.

moderateButtonClicked()

Input: The user clicks on the button to moderate.

Output: The moderation controls become visible.

submitModerationButtonClicked()

Input: The user clicks the submit button to apply all changes made regarding moderation.

Output: A Moderate object is dispatched to the server.

CafeLogDisplay

usernameClicked(MouseEvent)

Input: The mouse position to set the UpdateProfileDisplay position to.

Output: The server will retrieve UserProfile data for the target user, the Cafe UserProfileDisplay will be populated with this information, and the UserProfileDisplay will be moved to the mouse pointer location.

CafeModRecordDisplay

moderatorUsernameClicked(MouseEvent)

Input: The mouse position to set the UpdateProfileDisplay position to.

Output: The server will retrieve UserProfile data for the target moderator, the Cafe UserProfileDisplay will be populated with this information, and the UserProfileDisplay will be moved to the mouse pointer location.

CafeGlobalModDisplay

grantedByUsernameClicked(MouseEvent)

Input: The mouse position to set the UpdateProfileDisplay position to.

Output: The server will retrieve UserProfile data for the target user, the Cafe UserProfileDisplay will be populated with this information, and the UserProfileDisplay will be moved to the mouse pointer location.

grantedToUsernameClicked(MouseEvent)

Input: The mouse position to set the UpdateProfileDisplay position to.

Output: The server will retrieve UserProfile data for the target user, the Cafe UserProfileDisplay will be populated with this information, and the UserProfileDisplay will be moved to the mouse pointer location.

CafeDomainModDisplay

grantedByUsernameClicked(MouseEvent)

Input: The mouse position to set the UpdateProfileDisplay position to.

Output: The server will retrieve UserProfile data for the target user, the Cafe UserProfileDisplay will be populated with this information, and the UserProfileDisplay will be moved to the mouse pointer location.

grantedToUsernameClicked(MouseEvent)

Input: The mouse position to set the UpdateProfileDisplay position to.

Output: The server will retrieve UserProfile data for the target user, the Cafe UserProfileDisplay will be populated with this information, and the UserProfileDisplay will be moved to the mouse pointer location.

CafeDomainReportDisplay

xButtonClicked()

Input: The user clicks the CafeDomainReportDisplay close button.

Output: The destroy function is invoked and the CafeDomainReportDisplay element is removed.

CafeUsersReportDisplay

newestUserClicked(MouseEvent)

Input: The mouse position to set the UpdateProfileDisplay position to.

Output: The server will retrieve UserProfile data for the target user, the Cafe UserProfileDisplay will be populated with this information, and the UserProfileDisplay will be moved to the mouse pointer location.

update(AdminUsersReport)

Input: The updated AdminUsersReport received from the server.

Output: The CafeUsersReport is updated to contain the AdminUsersReport content.

CafeCommentSortDisplay

settingChange()

Input: This is called whenever some HTMLElement representing a setting is changed.

Output: A Settings object is constructed and emitted so that Cafe can update the global settings object and percolate changes down anywhere CafeSortDisplay is used.

updateFromSettings(CafeSettings)

Input: A CafeSettings object, representing changes made to the global settings elsewhere in the program.

Output: All HTMLElement fields are updated to reflect the new settings, and the underlying data is updated.

Messages

Messages in Comment Anywhere are formed around the communication entities described in package communication. Client-Server communication entities are sent from the Front End to the Server and Server-Client communication entities are sent from the Server to the Front End. Each Client-Server communication entity is associated with an API end-point. Each API end-point is associated with a Server handler. Each handler is associated with a method of

ControllerInterface. These methods cause one or more Server-Client communication entities to be added to 'nextResponse' field of the ControllerInterface that will be sent to the Front End after the processing of the user command has been completed. These message pipelines are recorded in the following table.

Back End Messages Pipeline

API Endpoint	**Client-Server Communication Entity**	**Server method called**	**UserInterface method called**	**Server-Client communication entities sent in response**
/assignDomainMod	AssignDomainModerator	postAssignDomainModerator	HandleAssignDomainModerator	Message
/assignGlobalMod	AssignGlobalModerator	postAssignGlobalModerator	HandleAssignGlobalModerator	Message
/ban	Ban	postBan	HandleCommandBan	Message
/changeEmail	ChangeEmail	postChangeEmail	HandleCommandChangeEmail	LoginResponse
/changeFeedback	ChangeFeedback	postChangeFeedback	HandleCommandChangeFeedback	Message
/changeProfile	ChangeProfileBlurb	postChangeProfileBlurb	HandleCommandChangeProfileBlurb	LoginResponse
/newComment	CommentReply	postCommentReply	HandleCommandCommentReply	Message, CommentChange
/voteComment	CommentVote	postCommentVote	HandleCommandCommentVote	Message, CommentChange
/newFeedback	Feedback	postFeedback	HandleCommandFeedback	Message

/comments	GetComments	getComments	HandleCommandGetComments	Comment[]
/user	GetUserProfile	getUserProfile	HandleCommandGetUserProfile	UserProfile
/login	Login	postLogin	HandleCommandLogin	LoginResponse
/logout	Logout	postLogout	HandleCommandLogout	LogoutResponse
/moderate	Moderate	postModerate	HandleCommandModerate	Message, CommentChange
/pwResetCode	PasswordResetCode	postPasswordResetCode	HandleCommandPasswordResetCode	Message
/pwResetReq	PasswordResetRequest	postPasswordResetRequest	HandleCommandPasswordResetRequest	Message
/newReport	PostCommentReport	postCommentReport	HandleCommandCommentReport	Message
/register	Register	postRegister	HandleRegister	LoginResponse
/reqVerification	RequestVerification	postRequestVerification	HandleCommandRequestVerification	Message
/newPassword	SetNewPass	postSetNewPass	HandleCommandChangePassword	Message
/verify	Verify	postVerify	HandleCommandVerify	Message
/viewBans	ViewBans	getBans	HandleCommandViewBans	BanRecord[]
/viewCommentReports	ViewCommentReports	getCommentReports	HandleCommandViewCommentReports	CommentReport[]
/viewFeedback	ViewFeedback	getFeedback	HandleCommandViewFeedback	FeedbackRecord[]

/viewLogs	ViewLogs	getLogs	HandleCommandViewLogs	AdminAccessLog[]
/viewModRecords	ViewModRecords	getModRecords	HandleCommandViewModRecords	ModerationRecord[]
/viewMods	ViewMods	getMods	HandleCommandViewMods	DomainModeratorRecord[], GlobalModeratorRecord[]
/viewDomainReport	ViewDomainReport	getDomainReport	HandleCommandViewDomainReport	AdminDomainReport
/amILoggedIn	*none*	getLoggedInStatus	HandleCommandAmILoggedIn	LoginResponse
/viewUsersReport	ViewUsersReport	getUsersReport	HandleCommandViewUsersReport	AdminUsersReport

Narrative/PDL

A. Start screen

   a. Log in

   b. Create an account

B. Log in

   i. Enter username

   ii. Enter password

   iii. Submit. Are the username and password correct? Is the user a moderator account? Is the user an administrator account?

      1. Yes. Go to main screen.

      2. No. Start Log in again.

      3. Yes. Go to Moderator main screen.

4. No. Go to next step.
5. Yes. Go to Administrator main screen.
6. No. Go to main screen.

b. Reset password

C. Reset password

i. Enter new username

ii. Enter new password

iii. Submit. Does the user confirm the reset is them?

1. Yes. Go to Log in.

2. No. Go to reset password.

D. Create an account

i. Enter an email.

ii. Enter a username.

iii. Enter a password.

iv. Submit. Is there an email and a password? Is the email valid?

1. Yes. check if the email is valid.

2. No. go back to create an account and prompt for email username and password.

3. Yes. Create the account and go to the main screen.

4. No. Prompt for a new email and go to create an account.

E. Main screen

a. Rate a comment

b. Make a comment

c. Report a comment

d. Log out

F. Rate a comment

i. Rate the comment. as funny? As factual? As informative?

1. Yes. Add 1 to the total of funny ratings and go to next step

2. No. Go to the next step.

3. Yes. Add 1 to the total of factual ratings and go to next step

4. No. Go to the next step.

5. Yes. Add 1 to the total of informative ratings and go to next step

6. No. Do nothing go to the Main screen

ii. Main screen. Is the user a moderator? Is the user an Administrator?

1. Yes. Go to Moderator main screen
2. No. Go to the next step
3. Yes. Go to the Administrator main screen.
4. No. Go to the Main screen.

G. Make a comment

i. Type in the comment.

ii. Submit the comment. Does the comment contain disallowed or spam content?

1. Yes. Do not submit the comment and go to Make a comment.
2. No. Submit the comment to the database.

iii. Main screen. Is the user a moderator? Is the user an Administrator?

1. Yes. Go to the Moderator main screen
2. No. Go to next step
3. Yes. Go to the Administrator main screen.
4. No. Go to the Main screen.

H. Report a comment

i. Type why the user is reporting the comment.

ii. Report the comment to the moderators and administrators

iii. Main screen. Is the user a moderator? Is the user an Administrator?

1. Yes. Go to Moderator main screen
2. No. Go to next step
3. Yes. Go to the Administrator main screen.
4. No. Go to the Main screen.

I. Log out

a. Start screen

J. Moderator main screen

a. Rate a comment
b. Make a comment
c. Report a comment
d. Ban a comment
e. Ban a user
f. Log out

K. Administrator main screen

a. Rate a comment
b. Make a comment
c. Report a comment

d. Report a user

e. Ban a comment

f. Ban a user

g. Log out

L. Ban a comment

i. Reason for removing the comment.

ii. Removing the comment. Is the user a moderator?

1. Yes. Go to the Moderator main screen.

2. No. Go to the Administrator main screen.

M. Ban a user

i. Reason for banning the user. Is the user being banned by an administrator by a moderator?

1. Yes. Do not ban the administrator and go back to the Moderator main screen.

2. No. Ban the user go to the Moderator main screen.

Decision: Programming language/Reuse/Portability

The Back End of Comment Anywhere will be written in Go. Go provides lightweight threads ideal for processing HTTP requests while reducing cloud resource consumption, which is

key for our shoestring budget. It compiles into a single binary not requiring any additional linking and allowing easy deployment. It has a static type system which reduces user error. It has great IDE and debugging support. [1] The Back End will be placed in a Docker Container to allow straightforward deployment on a variety of hosting platforms.

The Front End will be written in Typescript, compiled to JavaScript, and packed into bundles by WebPack. Typescript is a superset of JavaScript which compiles to JavaScript. It provides type-checking that is critical for reducing user error and managing a large code base. WebPack provides the tools necessary to package multiple JavaScript files into a single bundle compatible on a wider variety of browsers.

The application will be highly portable, as it runs in free web browsers, which are available on many platforms. The Web Extension API will allow us to create extensions from the same code base for multiple browsers, including, at least, Chrome and Firefox. However, because extensions are not currently supported on mobile browsers, Comment Anywhere will not be available on mobile.

Reuse is not a major focus of Comment Anywhere, because it is a standalone web application. If we see the opportunity to split off some functionality into a module or package that we can make open source and distribute, we may do so. Internal reuse is enabled somewhat by using certain base classes, especially on the Front End, that remove a degree of code repetition.

Implementation Timeline

The steps for creating Comment Anywhere from this design document are as follows.

1. Set up the Git repository, readme, and directory structure.

2. Create the necessary compilation files, such as the docker file, makefile, package.json, tsconfig.json, manifest.json, and webpack.config.js.

3. Stub some basic functions for the back end to ensure basic running of Server and Database and cross-module communication.

4. Implement one full message chain (see Messages section) to ensure that the Front End can indeed communicate with the back end.

5. Stub all the Go functions for the back end. No actual code is written, only the signatures across the board. It's refined until no errors are appearing. Copy the information from the design document into comments on the code.

6. Stub all functions for the front end in the same way. (Can be simultaneous to back end)

7. Begin writing the methods, simultaneously writing a parallel testMethod. Unit test each message and integration test each method chain until it is working properly.

    a. In general, back-end systems should be implemented one row of the Messages table at a time. With Go, like other C-oriented languages, the functions requiring definition can all be located in the same file. In this way the somewhat more tightly coupled methods of the Server classes which operate on the same data can be logically grouped and tested with the operation they intend to perform. It is sensible to write the Front End for that category simultaneously so they can be integration tested immediately.

b. The database.generated.Queries class will need to be completed by writing sqlc code and unit tests before most other back end classes and methods can be properly tested.

c. Account creation and management will need to precede the implementation of some other functionalities.

8. Repeat for every method until the code is done.

9. Deploy the back end on the cloud, acceptance test, and release the extension as a downloadable from a statically hosted github page.

10. Submit the extension to the Mozilla and Chrome marketplaces for review.

Activity	**Jan**	**Feb**	**March**	**April**	**May**	**May+**
Setup Git Repository						
Compilation, Configuration Files						
Stub basic back end functions						
Implement one full message chain						
Stub all back end functions						
Stub all front end functions						
Write methods and unit tests						
Write unit tests						
Write integration tests						
Release as downloadable extension						
Improve documentation						
Release to extension marketplaces						
Identify new features						

Testing

The team utilized Trello, a task tracking tool. The construction of the document was divided up into smaller, individual tasks. Each member of the team was assigned specific tasks to work on independently, moving each item into the appropriate category on the board. When an item was moved into the “Review” category, each other team member would review the work and offer feedback. This review process constituted Design testing. After implementing any feedback, final approval is required by the team. This process was used to ensure all tasks were completed in a timely manner and reviewed by the entire team.

Appendix

Appendix A: Technical Glossary

Admin / Administrator

An admin in the context of Comment Anywhere is a user who has been granted to special privileges which include assigning Global Moderators and viewing special reports.

Application Programming Interface (API)

An interface that allows your product or service communicate with other products and services without having to know how they're implemented. [2]

API Endpoint

The point of entry in a communication channel when two systems are interacting. It refers to touchpoints of the communication between an API and a server and corresponds to the path after the domain in an HTTP request. [3]

Architecture

"An architecture defines the structure of the software system and how its organized. It also describes the relationships between components, levels of abstraction, and other aspects of the software system." [4]

Back-End

A Back End is any part of a website or software program the users do not see. It contrasts with the Front End, which refers to a program or website's user interface.

Base Class

"A class, in an object-oriented language, from which other classes are derived. It facilitates the creation of other classes that can reuse the code implicitly inherited from the base class." [5]

Browser Extension

"An extension adds features and functions to a browser. It is created using familiar web-based technologies – HTML, CSS, and JavaScript. It can take advantage of the same web APIs as JavaScript on a web page, but also has access to its own set of APIs." [6]

Cache

In the context of Comment Anywhere, the term "Cache" prefixes classes which encapsulate information retrieved from the database and which have been temporarily instantiated in expectation of repeated access.

Class

Classes are a template for creating objects.

Context (Go)

Context is a built-in package for Go that makes it easy to pass HTTP request-scoped values to all routines involved in handling a request. [7]

Controller (Comment Anywhere)

A Controller in the context of Comment Anywhere is shorthand for any struct implementing UserControllerInterface. It represents a user's abilities and position in the Back End.

Cookie

"A small piece of data that a server sends to a web browser. The browser may store the cookie and send it back to the same server with later requests. Typically, an HTTP cookie is used to tell if two requests come from the same browser. It remembers stateful information for the stateless HTTP protocol." [8]

Coupling

"Coupling in Software Engineering is… used to define the factors of dependency and independence of each module of the software with other modules. It is used as an indicator of interdependency amongst the modules." [9]

Container

"A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package that includes everything needed to run an application." [10]

Database

"A database is an organized collection of structured information, or data, typically stored electronically in a computer system… [it] is typically modeled in rows and columns in a series of tables to make processing and data querying efficient. The data can then be easily accessed, managed, modified, updated, controlled, and organized. Most databases use structured query language (SQL) for writing and querying data." ***[11]***

Docker

"Docker is an open platform for developing, shipping, and running applications [and it] provides the ability to package and run an application in a loosely isolated environment called a container." [12]

Dockerfile

"A Dockerfile is a text document that contains all the commands [necessary to] assemble a [container]." [13]

Document-Object-Model (DOM)

"The Document Object Model (DOM) connects web pages to scripts or programming languages by representing the structure of a document – such as the HTML representing a web page – in memory…. The DOM represents a document with a logical tree. Each branch of the tree ends in a node, and each node contains objects. DOM methods allow programmatic access to the three. With them, you can change the document's structure, style, or content. Nodes can also have event handlers attached to them. Once an event is triggered, the event handlers get executed." [14]

Domain

A domain name is a unique address for a website.

Domain Moderator

A Domain Moderator is a user that has been promoted to moderate a specific URL page they have all the features of a user but are able to ban or hide user comments.

Extends

Extends is synonymous with “inherits from”. An extending class has all members of the class which it is extending, plus additional members.

Field

A field is a member of a class which holds data. Compare to method.

Front-end

The Front End is everything that a user interacts with on their end.

Full-Stack

“Full stack refers to the entire set of software solutions and technologies applied to build a platform, website, or application. All application or website development projects have two basic parts – front end (client-side) and back end (server-side)- which are combined in full-stack development.” [15]

GlobalModerator

A Global Moderator is a domain moderator in all power and responsibility but they can moderate any URL and can ban domain moderators.

Go

“Go is an expressive, concise, clean, and efficient programming language designed to optimized concurrency, which compiles quickly to machine code yet has the convenience of garbage collection. It is fast, statically typed, and compiled.” [16]

godotenv

“A Go port of the Ruby dotenv project which loads env vars from a .env file.” [17] [17]

Guest

A guest is a person that uses comment anywhere that does not have a account guest's have a limited set of features compared to users or any other rank.

HTML

"Hypertext Markup Language is the most basic building block of the Web. It defines the meaning and structure of web content… 'Hypertext' refers to links that connect web pages to another… HTML uses 'markup' to annotate text, images, and other content for display in a Web browser." [18]

HTMLElement

An interface which represents any HTML Element. It operates with the Document Object Model to render content in a particular position on a webpage. [19]

HTTP

Hypertext Transfer Protocol is an application-layer protocol for transmitting documents such as HTML, designed for communication between web browsers and web servers. [20]

HTTP Request

An object encapsulating a request from a browser to a server in the HTTP protocol. [20]

HTTP Response

An object encapsulating a response from a server to the browser in the HTTP protocol. [20]

JavaScript

"JavaScript (JS) is a lightweight, interpreted, or just-in-time compiled programming language with first-class functions." [21] It allows the implementation of complex features on

web pages.

JWT Token

JWT stands for JSON web tokens. It is a “compact, URL-safe means of representing claims to be transferred between two parties. The claims in a JWT are encoded as a JSON object that is used as the payload of a Web Signature or as the plaintext of a Web Encryption structure, enabling the claims to be digitally signed or integrity protected and/or encrypted.” [22]

JSON

“JSON is a syntax for serializing objects, arrays, numbers, strings, booleans, and null. It is based upon JavaScript syntax, but it is distinct from JavaScript.” [23]

makefile

Makefile is a type of file originally designed to work with the GNU program make. It allows the execution of complex building and linking commands by running simple make statements that can be reused every time a project needs to be rebuilt. [24]

manifest.json

A file included with every browser extension which specifies “basic metadata about [the] extension such as the name and version, and [where one] can also specify aspects of [the] extension’s functionality (such as background scripts, content scripts, and browser actions). It is a JSON-formatted file with one exception: it is allowed to contain ‘//’-style comments.” [25]

Member

A field or a method within a class.

Method

A method is a member of a class that is a function, which can access private members of that class.

Model (database)

A model is a class which closely represents a record retrieved from a database table.

Module

A module is a collection of data.

Module Cohesion

Module Cohesion is when a module is restricted to a specific category of data.

mux

Mux is a third-party Go library which “implements a request router and dispatcher for mathcing incoming requests to their respective handler. The name mux stands for ‘HTTP request multiplexer’.” [26] It is used by Server to route requests to their respective handlers.

NodeJS

NodeJS is “an asynchronous event-driven JavaScript runtime.” [27] It is used in Comment Anywhere to access the node package manager (npm)in order to implement TypeScript, WebPack, and other dependencies the Front End needs.

npm

“npm is the world’s largest software registry.” [28] It is used in the Node JavaScript development environment to retrieve package dependencies. It stands for Node Package Manager.

package.json

A file which “holds various metadata relevant to project. [It] is used to give information to npm that allows it to… handle the project’s dependencies.” [29]

Path

In the context of Comment Anywhere, Path refers to the portion of a URL which comes after the domain. It is used to associate a set of comments with a particular location on the web.

Pointer

A pointer holds the memory address of a value. It used frequently in the Back End whenever something needs to be passed by reference rather than by value.

Polymorphism

...

Port

“A port is a virtual point where network connections start and end. Ports are software-based and managed by a computer’s operating system. Each port is associated with a specific process or service. Ports allow computer to easily differentiate between different kinds of traffic: emails go to a different port than webpages, for instance, even though both reach a computer over the same Internet connection.” [30]

PostgresSQL ("postgres")

“PostgreSQL is a powerful, open source object-relational database system with over 35 years of active development that has earned it a strong reputation for reliability, feature robustness, and performance.” [31]

Schema

“A database schema is considered the “blueprint” of a database which describes how the data may relate to other tables or other data models. However, the schema does not actually contain data.” [32]

SQL

“Structured Query Language is a descriptive computer language designed for updating, retrieving, and calculating data in table-based databases.” [33]

sqlc

“sqlc generates fully type-safe idiomatic Go code” from SQL queries. [34]

tsconfig.json

“The presence of a tsconfig.json file in a directory indicates that the directory is the root of a TypeScript project. The tsconfig.json file specifies the root files and the compiler options required to compile the project [to JavaScript].” [35]

TypeScript

Typescript adds additional syntax to JavaScript, most importantly, a static type checker. It is a Typed Superset of JavaScript which compiles to JavaScript. It reduces errors by eliminating ambiguity. [36]

User

A user is a person using comment anywhere that has an account a user is a parent of domain moderator, global moderator and administrator and contains the basic information about that user’s account and features

Web browser

An application which can access and render webpages on the internet.

Webpack

Webpack is a "static module bundler for modern JavaScript applications [which] combines every module your project needs into one or more *bundles*, which are static assets to serve your content from." [37]

Website

"A group of World Wide Web pages usually containing hyperlinks to each other and made available online by an individual, company, educational institution, government, or organization." [38]

webpack.config.js

"A Webpack config is a JavaScript object that configures one [or more] of Webpack's options." [39]

.env file

A file used to set up the server and database configuration can contain key value pairs for configurations.

Appendix B: References

[1] Go Contributors, "Go for Web Development - The Go Programming Language," [Online]. Available: https://go.dev/solutions/webdev. [Accessed 4 December 2022].

[2] RedHat Contributors, "What is an API?," RedHat, 2 June 2022. [Online]. Available: https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces.

[3] Cloudflare, "What is an API endpoint? | Cloudflare," [Online]. Available: https://www.cloudflare.com/learning/security/api/what-is-api-endpoint/. [Accessed 4 December 2022].

[4] InterviewBit, "Systems Architecture - Detailed Explanation - InterviewBit," 17 June 2022. [Online]. Available: https://www.interviewbit.com/blog/system-architecture/.

[5] techopedia, "What is Base Class? - Definition from Techopedia," 29 August 2011. [Online]. Available: https://www.techopedia.com/definition/26896/base-class.

[6] MDN Contributors, "What are extensions? - Mozilla | MDN," 15 November 2022. [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/What_are_WebExtensions.

[7] S. Ajmani, "Go Concurrency Patterns: Context," 29 July 2014. [Online]. Available: https://go.dev/blog/context.

[8] MDN Contributors, "MDN," 2 December 2022. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies.

[9] P. Pedamkar, "Coupling in Software Engineering | 6 Different Types of Coupling," [Online]. Available: https://www.educba.com/coupling-in-software-engineering/. [Accessed 4 December 2022].

[10] Docker, "What is a Container? - Docker," [Online]. Available: https://www.docker.com/resources/what-container/. [Accessed 4 December 2022].

[11] Oracle, "What is a Database | Oracle," [Online]. Available: https://www.oracle.com/database/what-is-database/. [Accessed 4 December 2022].

[12] Docker, "Docker overview | Docker Documentation," [Online]. Available: https://docs.docker.com/get-started/overview/. [Accessed 4 December 2022].

[13] Docker, "Dockerfile reference," [Online]. Available: https://docs.docker.com/engine/reference/builder/.

[14] MDN Contributors, "Document Object Model (DOM) - Web APIs | MDN," 14 November 2022. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model.

[15] A. M. A. Mamun, "What Does Full Stack Mean ? | Webopedia," 9 March 2022. [Online]. Available: https://www.webopedia.com/definitions/full-stack/.

[16] Go Contributors, "The Go Programming Language," [Online]. Available: https://go.dev/. [Accessed 4 December 2022].

[17] joho, "Github.com," 12 September 2022. [Online]. Available: https://github.com/joho/godotenv/blob/main/README.md.

[18] MDN Contributors, "HTML: HyperText Markup Language | MDN," 13 September 2022. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTML.

[19] MDN Contributors, "MDN | HTML Element | Web APIs," 11 November 2022. [Online].

[20] MDN, "HTTP," 13 September 2022. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP.

[21] MDN Contributors, "JavaScript | MDN," 29 November 2022. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/javascript.

[22] Internet Engineering Task Force (IETF), "RFC 7519 - JSON Web Token (JWT)," [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7519.

[23] MDN Contributors, "JSON - JavaScript | MDN," 6 November 2022. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON.

[24] Free Software Foundation, Inc., "GNU make," 31 October 2022. [Online]. Available: https://www.gnu.org/software/make/manual/make.html.

[25] MDN Contributors, "manifest.json - Mozilla | MDN," 9 September 2022. [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/manifest.json.

[26] gorilla, "Github.com," 17 August 2022. [Online]. Available: https://github.com/gorilla/mux.

[27] NodeJS Contributors, "About | Node.js," [Online]. Available: https://nodejs.org/en/about/. [Accessed 4 December 2022].

[28] npm Contributors, "About npm | npm Docs," 27 October 2022. [Online]. Available: https://docs.npmjs.com/about-npm.

[29] NodeJs, "What is the file 'package.json' | NodeJs," 26 August 2011. [Online]. Available: https://nodejs.org/en/knowledge/getting-started/npm/what-is-the-file-package-json/.

[30] Cloudflare, "What is a computer port? | Ports in networking | Cloudflare," [Online]. Available: https://www.cloudflare.com/learning/network-layer/what-is-a-computer-port/. [Accessed 4 December 2022].

[31] Postgres Contributors, "PostgresSQL: The world's most advanced open source database," [Online]. Available: https://www.postgresql.org/. [Accessed 4 December 2022].

[32] IBM, "What is a Database Schema?," [Online]. Available: https://www.ibm.com/cloud/learn/database-schema. [Accessed 4 December 2022].

[33] MDN Contributors, "SQL - MDN Web Docs Glossary: Definitions of Web-related terms | MDN," 20 September 2022. [Online]. Available: https://developer.mozilla.org/en-US/docs/Glossary/SQL.

[34] K. Conroy, "sqlc.dev | Compile SQL to type-safe Go," [Online]. Available: https://sqlc.dev/. [Accessed 4 December 2022].

[35] Microsoft, "Typescript: Documentation - What is a tsconfig.json," [Online]. Available: https://www.typescriptlang.org/docs/handbook/tsconfig-json.html. [Accessed 4 December 2022].

[36] Microsoft, "Typescript: Documentation - TypeScript for the New Programmer," [Online]. Available: https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html. [Accessed 4 December 2022].

[37] Webpack Contributors, "Concepts | Webpack," [Online]. Available: https://webpack.js.org/concepts/. [Accessed 4 December 2022].

[38] Merriam-Webster, "Website Definition & Meaning - Merriam-Webster," [Online]. Available: https://www.merriam-webster.com/dictionary/website. [Accessed 4 December 2022].

[39] Mastering JS, "An Introduction to Webpack Configs - Mastering JS," 2 March 2020. [Online]. Available: https://masteringjs.io/tutorials/webpack/config.

Appendix C: Team Details

I, Karl Miller, attest that I executed the functions listed within the team details section of the document. Also, I agree with all information stated within the Design document.

Karl Miller	Karl L. Miller	12/11/2022
Printed Name	Signature	Date

I, Frank Bedekovich, attest that I executed the functions listed within the team details section of the document. Also, I agree with all information stated within the Design document.

Frank Bedekovich	Frank J Bedekovich	12/11/2022
Printed Name	Signature	Date

I, Robert Krency, attest that I executed the functions listed within the team details section of the document. Also, I agree with all information stated within the Design document.

Robert Krency	Robert Krency	12/11/2022
Printed Name	Signature	Date

I, Luke Bates, attest that I executed the functions listed within the team details section of the document. Also, I agree with all information stated within the Design document.

Luke Bates	Luke Bates	12/11/2022
Printed Name	Signature	Date

Appendix D: Workflow Authentication

Karl Miller directed this phase. He maintained a record of all changes to the Specifications document and identified needed changes. He identified and described all classes, fields, and methods for classes and functions in package server, database, util, and communication and added associated code snippets, schema diagrams, and other diagrams. He identified and described the Front-End Control and boundary classes. He wrote the Implementation Timeline, Language Decision, Messages section, and numerous entries in the technical glossary. He edited and reviewed other sections.

Frank Bedekovich created the Narrative / PDL section, he also filled out the input and output fields in the package database. He also wrote up the description of various terms in the glossary and created the project block diagram and filled out its description.

Luke Bates developed the functional descriptions for the front-end based on Karl's prototype designs. He added and wrote numerous terms in the glossary. He reviewed and edited many other sections. He worked with Karl in the design of the Front-End control and boundary classes.

Robert Krency wrote the initial abstract, description of document, module cohesion, and module coupling sections. He created an architectural diagram and dataflow diagram based on the identified modules and scheduled a writing center meeting. He reviewed and edited numerous other sections.

Appendix E: Writing Center Report

Client: Comment Anywhere Team

Staff or Resource: Caedon Vogel

Date: 11/28/2022 12:00 pm - 1:10 pm

Did the student request that the instructor receive a visit report? **Yes.**

What course was serviced by this visit? **Senior Project 1**

What goals were established for this tutoring session? **Review the document and offer advice.**

How did the process of this consulting session address the established goals? **I reviewed the document, adding comments where necessary, and providing criticism where necessary**

Please provide any additional comments relevant to this session?

- Watch out for spaces and commas. Those were your highest source of mistakes in this document with lots of stray spaces and commas. Overall, they aren't a HUGE deal but they definitely knock off a level of professionalism. I changed the ones I found, but make sure to go back through the document and iron out those little mistakes.
- I found the phrase ‘An user’ a LOT. Although it doesn’t follow the rule of “Use ‘An’ before a vowel,” the convention is primarily used because of the way it sounds. It is subjective, like “An umbrella” versus “A username.” Weird, right? I changed a good bit of the ‘An user’s but make sure they are all gone.
- The only other mistakes I found were a few misspellings, a couple instances of repeated indefinite articles (an an blah), and a few other spacing things which were fixed.