

TECHNICAL DOCUMENTATION

“POLLING SYSTEM”

PRODUCED BY:
KEVIN SHËMILI

NUMBER OF MATRICOLA
7169034

Contents

| | |
|--|-----------|
| 1. Project Description | 3 |
| 2. Architecture | 4 |
| 2.1. Domain Layer | 4 |
| 2.2. Application Layer | 6 |
| 2.2.1. Repository | 6 |
| 2.2.2. Use Cases | 9 |
| 2.2.2.1. Authentication Use Cases | 9 |
| 2.2.2.2. Poll Use Cases | 13 |
| 2.2.3. Utility | 20 |
| 2.3. Infrastructure Layer | 22 |
| 2.3.1. Database | 22 |
| 2.3.2. Mail | 23 |
| 2.3.3. WebSocket | 23 |
| 2.4. API Layer | 25 |
| 2.4.1. Dependency Injection | 25 |
| 2.4.2. Middleware | 26 |
| 2.4.3. Controllers | 26 |
| 2.4.4. Requests | 29 |
| 2.4.5. Routes | 30 |
| 3. Tests | 31 |
| 3.1. Unit Tests | 31 |
| 3.2. Integration Tests | 32 |
| 4. Migrations | 32 |
| 5. Dependencies (Framework / Tools / Libraries) | 33 |
| 6. Instructions | 36 |

1. Project Description

The proposed project is a Polling System designed to allow users to create polls, vote on them, and view results of their votes. The system is a backend only service that supports the following functionalities:

- Login & Registration
- Creation of a poll
- Update of a poll
- Automatic expiration of a poll
- Expiry on demand of a poll
- Deletion of a poll
- Voting on a poll
- Real time updates on poll votes
- Real time updates on newly created polls

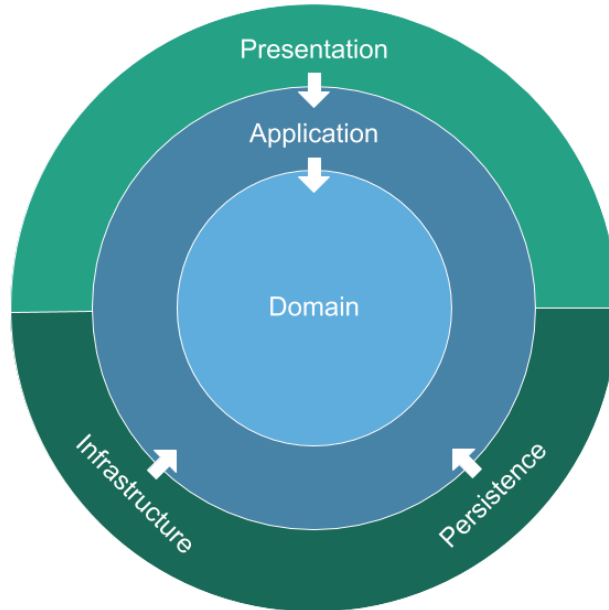
The proposed polling system makes use of various concepts from the course, starting with the foundational principles of Go programming. It utilizes the language's modular structure to organize code into packages. Concurrency is applied to manage WebSocket connections for real-time updates without requiring a page refresh, the sending of email etc.

The backend design incorporates concepts from web programming in Go, particularly in handling and processing of requests. A middleware is used to manage user authentication, providing a critical security layer, while GORM is employed as the ORM library for managing interactions with a PostgreSQL database. Furthermore, the system follows the REST architectural style to structure and expose API endpoints.

The application features dependency injection to maintain loose coupling, enhancing modularity and testability of the components. Swagger is used to document the API endpoints.

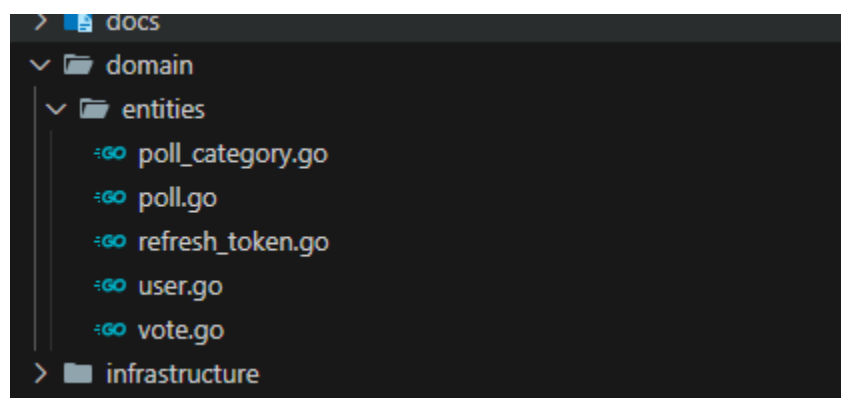
2. Architecture

The application has been developed using the Clean Architecture principles, a widely adopted software design whose main priority is the separation of concerns, scalability, and testability.



In Clean Architecture we organize the code into distinct layers, which communicate with each other through interfaces, thus ensuring loose coupling and isolation.

2.1. Domain Layer



The Domain layer is the heart of the project. It holds domain specific logic that make the application unique. In our application that would be the User, RefreshToken, Poll, PollCategory, & Vote entities. The only dependency of this layer is on GORM for defining the relations between the entities. The actual relations on the other hand are the following:

POLL

POLL M : 1 USER (FOREIGN KEY: CREATORID)

A Poll is created by one User, and a User may have many Polls.

POLL 1 : M POLLCATEGORIES (FOREIGN KEY: POLLID)

A Poll may have many PollCategories, and a PollCategory belongs to one Poll.

POLLCATEGORY

POLLCATEGORY M : 1 POLL (FOREIGN KEY: POLLID)

A PollCategory belongs to one Poll, and a Poll may have many PollCategories

POLLCATEGORY 1 : M VOTES (FOREIGN KEY: POLLCATEGORYID)

A PollCategory can have many Votes, and a Vote belongs to one PollCategory.

REFRESH TOKEN

REFRESHTOKEN M : 1 USER (FOREIGN KEY: USERID)

A RefreshToken belongs to one User, and a User may have many RefreshTokens.

USER

USER 1 : M POLLS (FOREIGN KEY: CREATORID)

A User may create many Polls, and each Poll belongs to one User.

USER 1 : M REFRESHTOKENS (FOREIGN KEY: USERID)

A User may have many RefreshTokens, and each RefreshToken belongs to one User.

VOTE

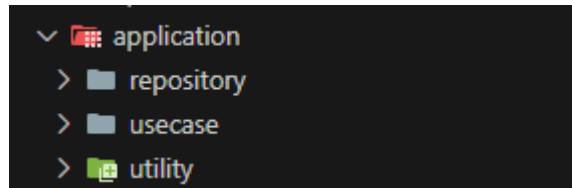
VOTE M : 1 USER (FOREIGN KEY: USERID)

A Vote belongs to one User, and a User may cast many Votes.

VOTE M : 1 POLLCATEGORY (FOREIGN KEY: POLLCATEGORYID)

A Vote belongs to one PollCategory, and a PollCategory may have many Votes.

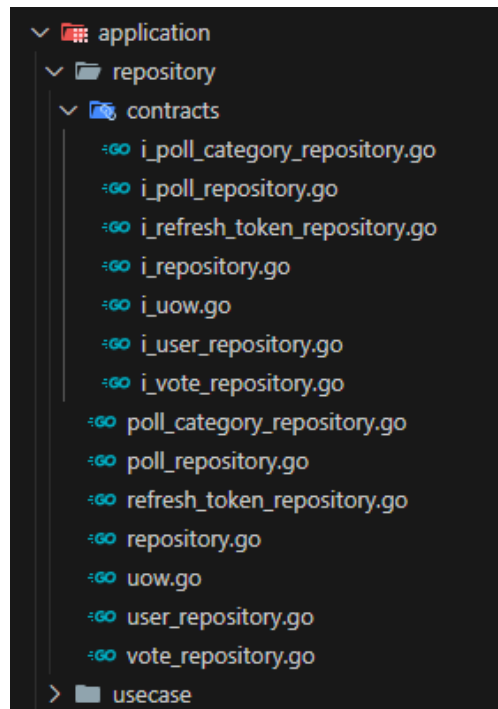
2.2. Application Layer



In the application layer we take the entities defined in the Domain layer and uses them to create the logic (use cases) that the users want. This is the layer that coordinates how things happen in the application, for example a new register will be handled by the appropriate use-case. In our application we include in the application layer the following logic:

- Repositories: Contains the repositories.
- Use Cases: Contains the application logic as use-cases.
- Utilities: Contains functions for stuff like token generation, pagination, etc.

2.2.1. Repository



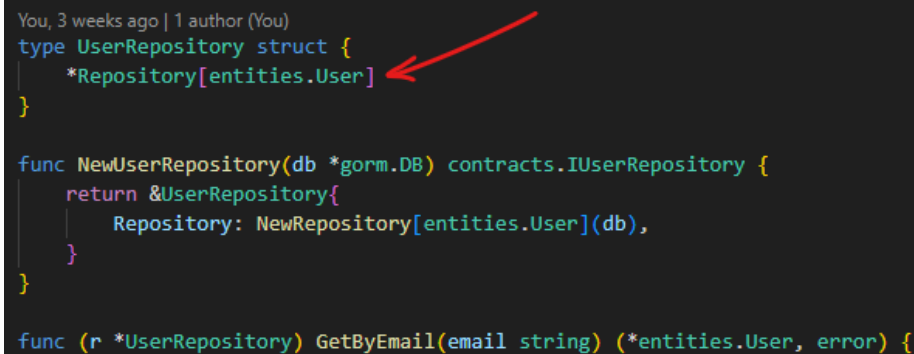
The repository folder is where we implement the Repository Pattern and the Unit of Work Pattern. These patterns are widely used for abstracting database access logic and also ensuring consistent database state.

The Repository Pattern itself, is a design pattern that serves as an intermediary between the application and the database. It abstracts the data access logic, and provides an interface for querying or updating / creating entities. Of particular interest here is the repository.go file. The repository.go file defines shared functionality that will be used in other concrete repository implementations. This base logic can that can be reused by specific repositories like PollRepository, UserRepository, and others.

```
You, 3 weeks ago | 1 author (You)
type UserRepository struct {
    *Repository[entities.User]
}

func NewUserRepository(db *gorm.DB) contracts.IUserRepository {
    return &UserRepository{
        Repository: NewRepository[entities.User](db),
    }
}

func (r *UserRepository) GetByEmail(email string) (*entities.User, error) {
```



For example, by using composition UserRepository will be able to use all of the methods declared in the Repository. This reduces code duplication significantly and promotes clean code.

Another file that is of interest, is the uow.go file. This is the file, where we implement the unit of work pattern. This pattern ensures that multiple repository operations are executed as part of a single transaction, thus preventing inconsistent database states. UoW contains methods for beginning a transaction, committing it, and also rollback in case of failure. Apart from this UoW is also responsible for the creating and managing of repository instances.

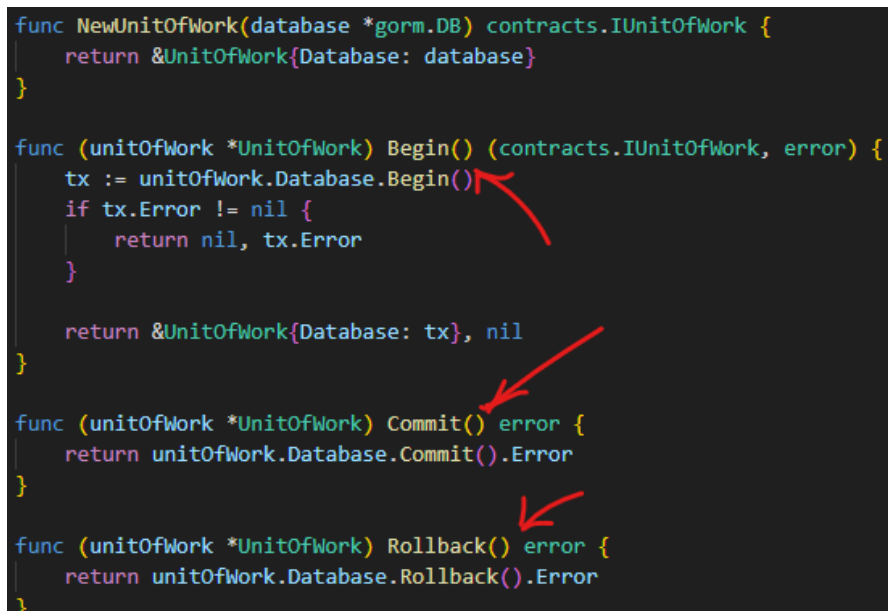
```
func NewUnitOfWork(database *gorm.DB) contracts.IUnitOfWork {
    return &UnitOfWork{Database: database}
}

func (unitOfWork *UnitOfWork) Begin() (contracts.IUnitOfWork, error) {
    tx := unitOfWork.Database.Begin()
    if tx.Error != nil {
        return nil, tx.Error
    }

    return &UnitOfWork{Database: tx}, nil
}

func (unitOfWork *UnitOfWork) Commit() error {
    return unitOfWork.Database.Commit().Error
}

func (unitOfWork *UnitOfWork) Rollback() error {
    return unitOfWork.Database.Rollback().Error
}
```



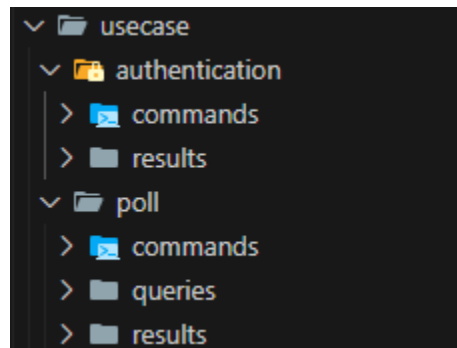
Something that should also be mentioned is that in the UoW, we dynamically initialize repositories. The *gorm.DB instance is immutable, and when a transaction starts, we must make sure that the repositories are using the new *gorm.DB instance produced by this operation. In the code shown in the upper photo that would be tx.

```
func (unitOfWork *UnitOfWork) IRefreshTokenRepository() contracts.IRefreshTokenRepository {  
    if unitOfWork.RefreshTokenRepository == nil {  
        unitOfWork.RefreshTokenRepository = NewRefreshTokenRepository(unitOfWork.Database)  
    }  
    return unitOfWork.RefreshTokenRepository  
}  
  
func (unitOfWork *UnitOfWork) IVoteRepository() contracts.IVoteRepository {  
    if unitOfWork.VoteRepository == nil {  
        unitOfWork.VoteRepository = NewVoteRepository(unitOfWork.Database)  
    }  
    return unitOfWork.VoteRepository  
}
```

By doing this, we ensure that whatever further operations are performed, are performed using the newly created transactional context, rather than the non-transactional base context. If repositories were injected initially using the base context, they would operate outside the transaction, breaking the transaction integrity.

Lastly, we may see that every file has its own interface. We will see this approach everywhere. Interfaces make sure that we decouple the concrete implementations of, in this case, repositories and the UoW from the rest of the application. This promotes modularity, reusability, and also simplifies testing.

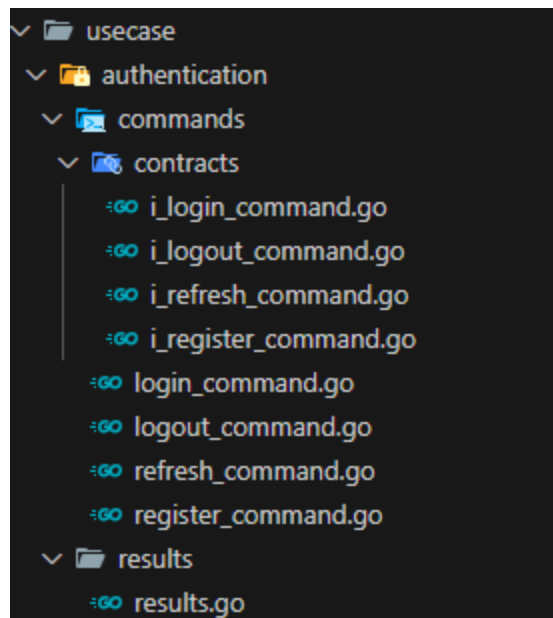
2.2.2. Use Cases



Here is where we implement the actual business logic of our application. The CQRS (Command Query Responsibility Segregation) pattern has been implemented in this section. The pattern separates the handling of commands (operations that cause changes in db) & queries (operations that don't cause changes in db). By doing so, it enforces a distinction between write and read operations, ensuring that each use case has high readability, and high maintainability.

The result of using this pattern are three folders. The commands folder, the queries folder, and also the results folder where we store the structs that our commands & queries will return.

2.2.2.1. Authentication Use Cases



Here we implement the use cases that are concerned with user registration & login. Concretely we have four use cases.

LOGIN USE CASE

The `ILoginCommand` & `LoginCommand` are listed as commands because the use case causes changes in the database. The use case is responsible for user login. The `UnitOfWork` and a validator (coming from a library) are injected in initialization. The `UnitOfWork` provides access to repositories needed for database operations, while the validator ensures that the incoming login request adheres to the required structure.

```
type ILoginCommand interface {
    Login(request *requests.LoginRequest) (*results.LoginResult, *utility.ErrorCode)
}

type LoginCommand struct {
    UnitOfWork repo.IUnitOfWork
    Validator *validator.Validate
}
```

In the method, we validate the request and start a transaction through the UoW. We check if the provided email exists, verify the password using the `bcrypt` library, and finally, generate a JWT & Refresh token for the user. We also remove any existing refresh tokens of the user. Once all is completed, the transaction is committed, and the method returns the JWT and refresh token to the client.

JWTs (JSON Web Tokens) are used in the platform for authentication. A JWT is a token that securely transmits information between parties. It consists of three parts: a header, a payload, and a signature. The tokens are signed using a secret key, making them tamper-proof.

Algorithm HS256

Encoded

PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE3MzU1MDUzMDgsInN1YiI6MX0.SREs1Sx0-dvw0c0AlyIVHe_JzuhNi93AftsA75Mq2xs
```

Decoded

EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

PAYLOAD: DATA

```
{
  "exp": 1735505308,
  "sub": 1
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secretOKI
)
```

☐ secret base64 encoded

✔ Signature Verified

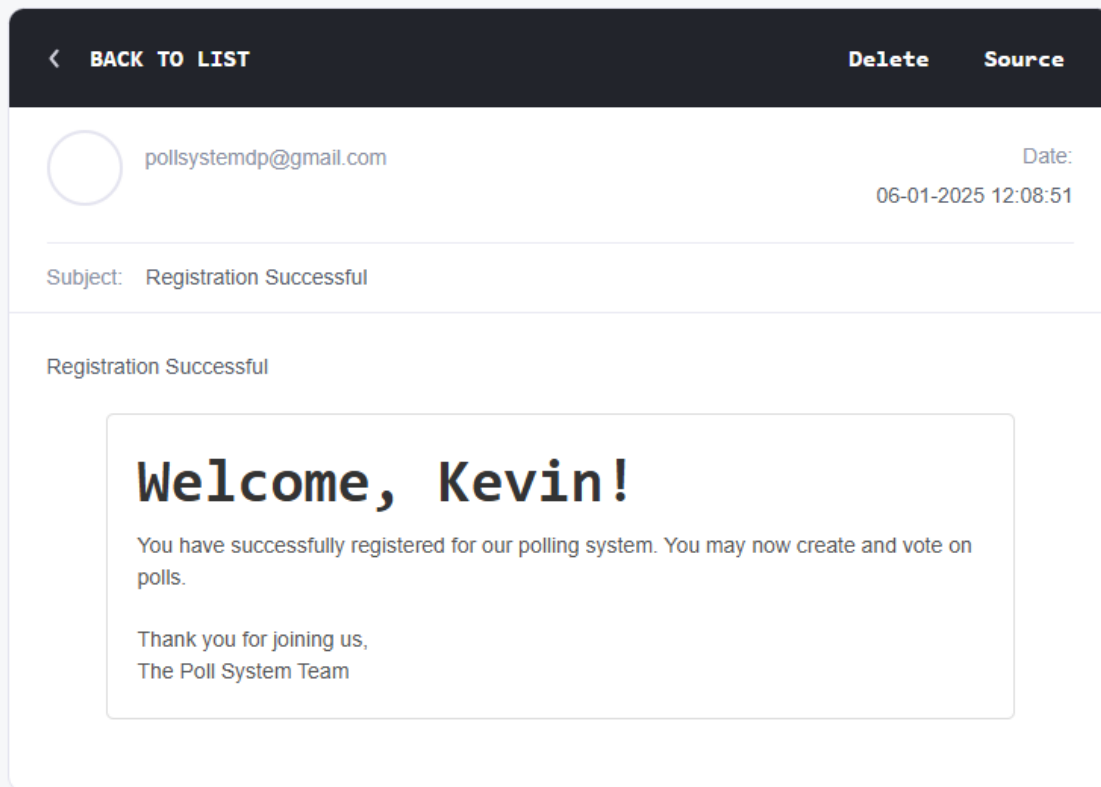
SHARE JWT

REGISTER USE CASE

The IRegisterCommand and RegisterCommand are listed as commands because the use case causes changes in the database. The use case is responsible for user registration. A UnitOfWork and a validator are injected during initialization.

```
type IRegisterCommand interface {  
    Register(request *requests.RegisterRequest) (bool, *utility.ErrorCode)  
}
```

In the method, we validate the request using the validator, check if the email already exists to prevent duplicates, and ensure the email and password formats are correct. The password is securely hashed using the bcrypt library before creating the user in the database. The bcrypt library dealt with the salt of the hash automatically so no action is required from our side. We deal only with the hash and only store it in the database. Finally, after a successful registration, an email is sent to the user.



REFRESH USE CASE

The IRefreshCommand and RefreshCommand are listed as commands because the use case causes changes in the database. The use case is responsible for handling the refresh process for user authentication tokens.

```
type IRefreshCommand interface {  
    Refresh(request *requests.TokensRequest) (*results.RefreshResult, *utility.ErrorCode)  
}
```

In the method, we validate the request, decode the refresh token, and parse claims from the provided JWT. We verify that the user exists, the refresh token matches the database record, and that neither the refresh token nor the JWT is expired. If all checks pass, a new JWT and refresh token are generated. The old refresh token is deleted, and the new one is stored in the database. Finally, the transaction is committed, and the new tokens are returned to the client.

A refresh token is a token with a bigger lifespan than a JWT one. We use it to obtain a new JWT without requiring the user to re login. It provides a mechanism for maintaining user sessions while minimizing the exposure of sensitive credentials like passwords.

LOGOUT USE CASE

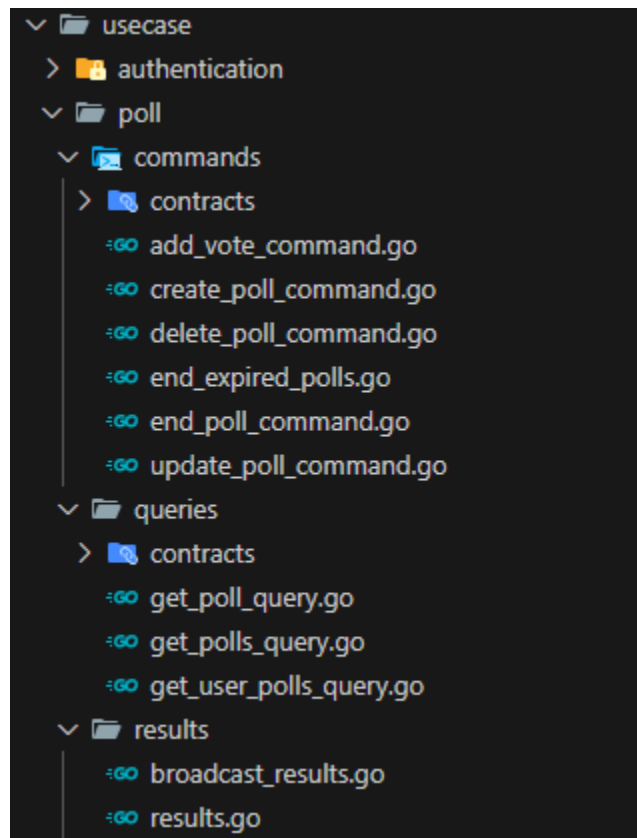
The ILogOutCommand and LogOutCommand are listed as commands because the use case causes changes in the database. The use case is responsible for user logout. A UnitOfWork and a validator are injected during initialization.

```
type ILogOutCommand interface {  
    LogOut(request *requests.LogOutRequest) (bool, *utility.ErrorCode)  
}
```

In the method, we validate the request using the validator and begin a transaction through the UnitOfWork. The user's refresh token is retrieved from the database to verify its existence. If the token is found, it is deleted to invalidate it. Lastly, we also set the "user" in the gin context to nil.

```
c.Set("user", nil)
```

2.2.2.2. Poll Use Cases



Here we implement the use cases that are concerned with the poll business logic. Concretely we have these use cases.

CREATE POLL USE CASE

The `ICreatePollCommand` and `CreatePollCommand` are listed as commands because the use case causes changes in the database. The use case is responsible for handling the creation of a poll. A `UnitOfWork` and a validator are injected during initialization.

```
type ICreatePollCommand interface {
    CreatePoll(request *requests.CreatePollRequest, user *entities.User) (*results.CreatePollResult, *utility.ErrorCode)
}
```

In the method, we validate the poll creation request. After the validation, a transaction is started. We check the expiry timestamp of the poll from the request and make sure it is not set in the past. We then create the `Poll` entity, together with the categories specified in the request, and persists it to the database using the `PollRepository`. Once the poll is successfully saved, the transaction is committed.

After committing, a message is broadcast to the `WebSocket` notifying all connected clients in real time about the newly created poll. The broadcast includes the poll's details, such as title, expiration date etc., and most importantly a broadcast-type, to help the client distinguish it among other broadcast types.

```

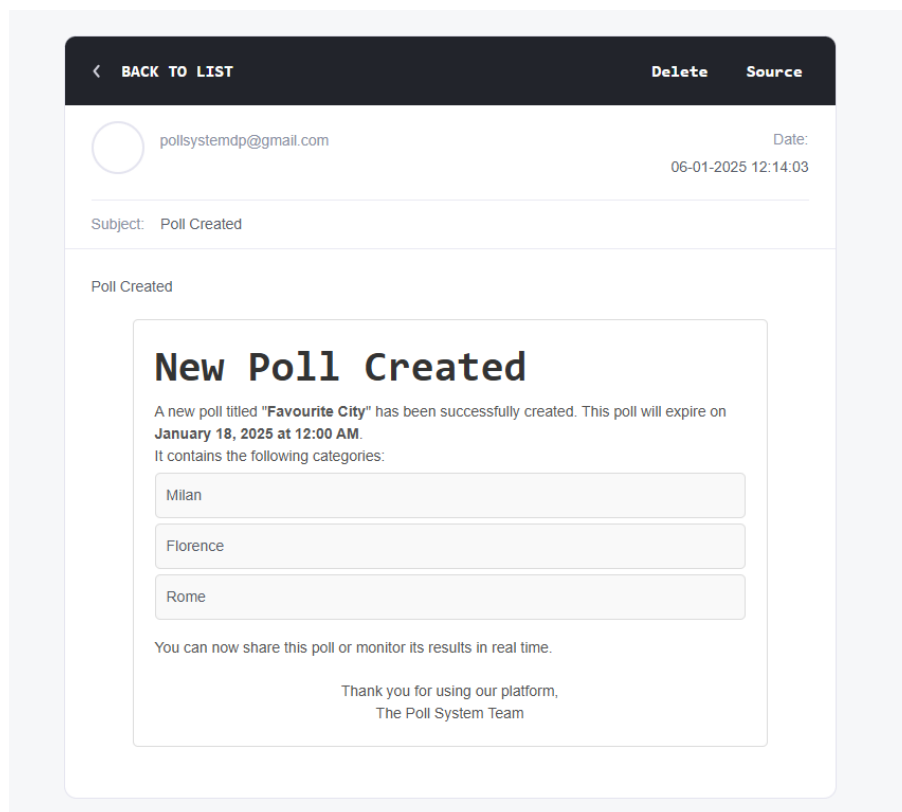
type BroadcastPoll struct {
    BroadcastType string `json:"type"`
    Data          struct {
        PollID      uint   `json:"poll_id"`
        Title       string `json:"title"`
        ExpiresAt   time.Time `json:"expires_at"`
        Ended       bool   `json:"ended"`
        Categories []struct {
            ID      uint   `json:"category_id"`
            Name   string `json:"category_name"`
            Votes  int    `json:"category_votes"`
        } `json:"categories"`
    } `json:"data"`
}

```

After the broadcast, an email notification is sent from this use case, to the creator of the poll, confirming its successful creation. A goroutine is spawned to deal with the sending of the email sending, thus ensuring responsiveness and avoidance of delays in returning the result to the client. Finally, the method then returns the created poll as a response.

When testing this endpoint, in the expires_at field, the date is expected in the following format:

"expires_at": "2025-01-18T00:00:00Z"



ADD VOTE USE CASE

The IAddVoteCommand and AddVoteCommand are listed as commands because the use case causes changes in the database. The use case is responsible for casting a vote. A UnitOfWork and a validator are injected during initialization.

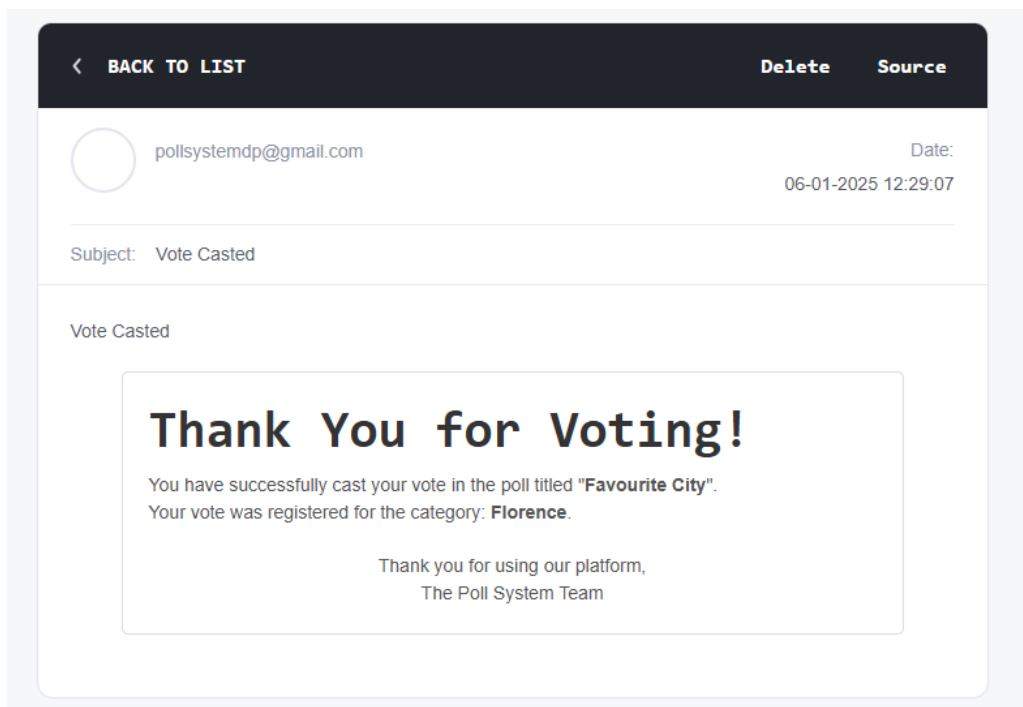
```
type IAddVoteCommand interface {
    AddVote(request *requests.AddVoteRequest, user *entities.User) (bool, *utility.ErrorCode)
}
```

In the method, we validate the request. A transaction is started using the UoW. In the method we first verify that the poll exists, has not expired, and that is also includes the specified category. Then we check whether the user has already voted in the poll to prevent duplicate votes. If all validations pass, a new Vote entity is created and stored in the database, and the transaction is committed.

Once the transaction is complete, we retrieve the updated poll data, including the latest vote counts for each category. This data is then broadcast to all connected clients.

```
type BroadcastVote struct {
    BroadcastType string `json:"type"`
    Data struct {
        PollID uint `json:"poll_id"`
        Categories []struct {
            CategoryID uint `json:"category_id"`
            Votes int `json:"category_votes"`
        } `json:"categories"`
    } `json:"data"`
}
```

Lastly, an email is sent to the user confirming that their vote has been received.



UPDATE POLL USE CASE

The IUpdatePollCommand and UpdatePollCommand are listed as commands because the use case causes changes in the database. The use case is responsible for handling the process of updating an existing poll. A UnitOfWork and a validator are injected during initialization.

```
type IUpdatePollCommand interface {
    UpdatePoll(userID uint, request *requests.UpdatePollRequest) (bool, *utility.ErrorCode)
}
```

In the method, we validate the request & a transaction is started using the UoW. In the method we retrieve the poll by its ID and verify its existence, we check that it has not already ended, we check that the new expiration date is a date in the future, and lastly, we ensure that the person performing the update is the owner of the poll. If checks pass, the poll fields are updated.

After committing, the updated poll is retrieved with a GET operation, a message is broadcast to notify all connected clients about the updated poll details. The type of the broadcast is update-poll, making it distinguishable among other broadcasts. But it uses the same broadcast struct used in the poll creation.

DELETE POLL USE CASE

The IDeletePollCommand and DeletePollCommand are listed as commands because the use case causes changes in the database. The use case is responsible for deleting a poll. A UnitOfWork and a validator are injected during initialization.

```
type IDeletePollCommand interface {
    DeletePoll(pollID uint, user *entities.User) (bool, *utility.ErrorCode)
}
```

In the method, we start a transaction & retrieve the poll by the given ID. Then we check if the poll exists and the requesting user is the creator of the poll. If the checks pass, the poll is soft-deleted from the database, and the transaction is committed. Once deleted, a message is broadcast to all connected clients about the deletion. The type of the broadcast is poll-deleted, making it distinguishable among other broadcasts.

```
type BroadcastDeletion struct {
    BroadcastType string `json:"type"`
    Data          struct {
        PollID uint `json:"poll_id"`
    } `json:"data"`
}
```


END POLL USE CASE

The IEndPollCommand and EndPollCommand are listed as commands because the use case causes changes in the database. The use case is responsible for ending a poll. A UnitOfWork and a validator are injected during initialization.

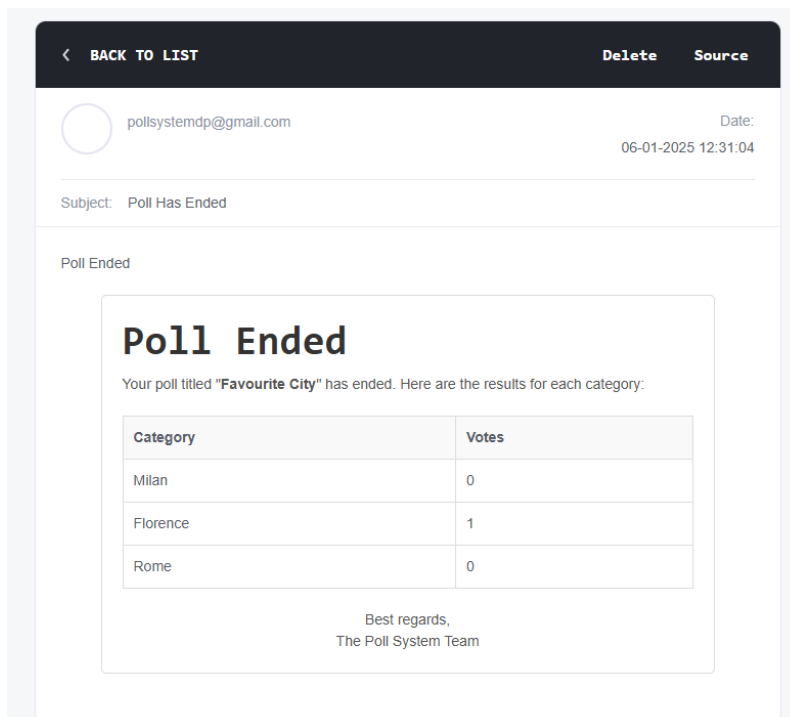
```
type IEndPollCommand interface {
    EndPoll(pollID uint, user *entities.User) (bool, *utility.ErrorCode)
}
```

In the method, we start a transaction, the poll is retrieved by its ID, and then we perform some checks. We verify that the poll exists, ensure that the requesting user is the poll's creator, and confirm that the poll has not already ended. If all checks pass, the poll is ended by setting the IsEnded field to true, and the changes are committed.

A message is broadcast to all connected clients that the poll has ended. The type of this broadcast is poll-ended making it distinguishable among others.

```
type BroadcastExpiry struct {
    BroadcastType string `json:"type"`
    Data          struct {
        PollID uint `json:"poll_id"`
    } `json:"data"`
}
```

Lastly, an email is sent to the owner of the poll, confirming its termination, and also providing the results of the poll.



GET POLL USE CASE

The IGetPollQuery and GetPollQuery are listed as queries because the use case causes no changes in the database. The use case gets the details of a single poll, including its categories & votes. The UnitOfWork is injected during initialization.

```
type IGetPollQuery interface {  
    GetPoll(pollID uint) (*results.GetPollResult, *utility.ErrorCode)  
}
```

In the method, the poll is retrieved using the PollRepository by its ID. We preload both categories and votes. We verify that the poll exists. Then we map the poll entity from the database to a GetPollResult “DTO” and return it.

GET POLLS USE CASE

The IGetPollsQuery and GetPollsQuery are listed as queries because the use case causes no changes in the database. The use case gets a list of polls in paginated format. In the request we specify if we want the active polls only, or all the polls. The UnitOfWork and a validator are injected during initialization.

```
type IGetPollsQuery interface {  
    GetPolls(request *requests.GetPollsRequest) (utility.PaginatedResponse[results.GetPollResult], *utility.ErrorCode)  
}
```

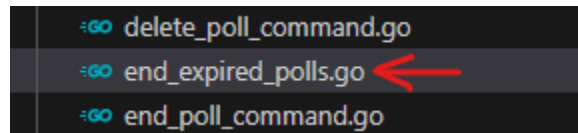
In the method, the request is validated using the validator, and the PollRepository is used to get the polls based on pagination parameters and filters, if any. We also preload both categories & votes. Then the result data is mapped to a GetPollResult DTO using a utility function. Finally, this mapping is returned.

GET POLLS OF USER USE CASE

The IGetUserPollsQuery and GetUserPollsQuery are listed as queries because the use case causes no changes in the database. The use case gets a list of polls in a paginated format. The polls created by the given user are fetched only. In the request we specify if we want the active polls only, or all polls. The UnitOfWork and a validator are injected during initialization.

```
type IGetUserPollsQuery interface {  
    GetPolls(userID uint, request *requests.GetPollsRequest) (utility.PaginatedResponse[results.GetPollResult], *utility.ErrorCode)  
}
```

In the method, the request is validated using the validator, and the PollRepository is used to get the polls based on pagination parameters and filters, if any made by the given user. Then the result data is mapped to a GetPollResult DTO using a utility function. Finally, this mapping is returned.

AUTOMATIC POLL EXPIRY

This is a use case that will be running in the background as a goroutine and it will not be available to be manually called by a user of our application. The purpose of the file, is to terminate all polls who are due.

In order to achieve this, in the method, we check if there are any expired polls, whose time is less than the current time. If yes, we start a transaction, where we set the `IsEnded` field of all expired polls to true, and commit this transaction.

For each ended poll, a message is broadcast to all the connected clients that the poll has ended. The type of this broadcast is the same used in the manual poll end use case.

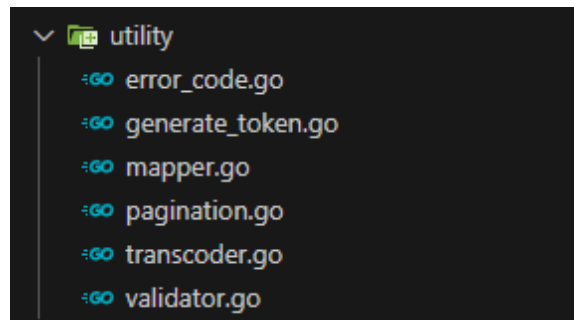
Lastly, an email is sent to the owner of each of the polls, confirming their termination, and also providing the results of each poll to the respective owner.

```
// goroutine for poll expiration
go func() {
    // every 1 minute - check for expiries
    ticker := time.NewTicker(1 * time.Minute)

    for range ticker.C {
        if err := commands.EndExpiredPolls(container.UnitOfWork); err != nil {
            fmt.Printf("Poll Expiry Error: %v", err)
        }
    }
}()
```

We define this goroutine in `main.go` and run this check every one minute.

2.2.3. Utility



In the utility folder we have put components and helper functions that are reused throughout the application. This design promotes modularity and reduces duplication.

Out of all the files we should mention the `error_code.go` file. The file contains the `ErrorCode` structure, which is a standardization of error handling across the application. The struct presents a uniform and clean interface that can be presented to the client, that contains only the most important stuff the client should know about.

```
type ErrorCode struct {
    Message      string `json:"message"`
    StatusCode    int    `json:"status_code"`
    Description   string `json:"description"`
}
```

For example, in case of an internal server error (some kind of database connectivity error etc.) we would return this message.

```
InternalServerError = NewErrorCode(
    "Internal Server Error.",
    http.StatusInternalServerError,
)
```

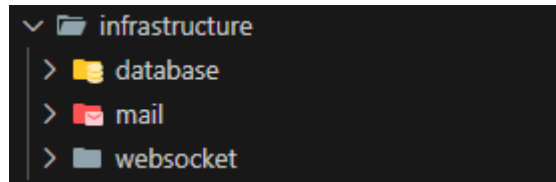
Another file that we should mention is the pagination.go file. In the file we declare two structs. One for holding the query parameters from an incoming request, where we store variables like the page size or the page that the user wants to see, together with filters if any. And then, in the second struct, we return a standardized format, for every paginated response.

```
You, 3 days ago | 1 author (You)
type QueryParams struct {
    Page      int    `validate:"omitempty"`
    PageSize  int    `validate:"omitempty"`
    Filter     string `validate:"omitempty"`
}

You, 5 days ago | 1 author (You)
type PaginatedResponse[T any] struct {
    Data      []T    `json:"data"`
    Page      int    `json:"page"`
    PageSize  int    `json:"page_size"`
    TotalCount int64  `json:"total_count"`
    TotalPages int    `json:"total_pages"`
}
```

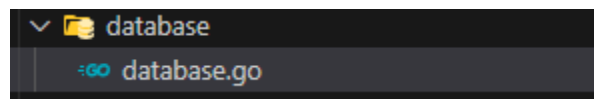
Lastly the last file we will touch is the generate_token.go. The file provides functions for generations refresh & JWT tokens, which are important for our authentication use cases. In this file we have the GenerateRefreshToken function which returns a random 16-byte token with an expiration time of 7 days. And then two other functions are provided in regards to the generation of JWTs each for different use cases.

2.3. Infrastructure Layer



The infrastructure layer, deals with practical things that the app needs but does not need to know how they work. In our application, for example, it performs the connection with the database, the sending of emails & the setting up of the WebSocket. This layer handles the technical stuff, leaving the other layers to focus on what they are best at.

2.3.1. Database



In the database folder we have a single file, the database.go file. The only thing the file does, is create a Database struct, that contains an instance of the `*gorm.DB`, and establish a connection to the database. That's it. We lastly have a getter for getting the `*gorm.DB` instance.

```
100, 3 weeks ago | 1 author (100)
type Database struct {
    DBContext *gorm.DB
}

func NewDatabase() (*Database, *utility.ErrorCode) {
    dsn := os.Getenv("CONNECTION_STRING")

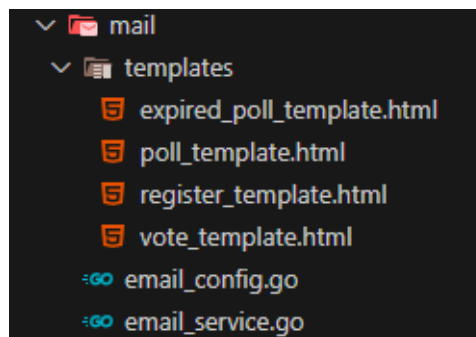
    context, err := gorm.Open(postgres.Open(dsn), &gorm.Config{})

    if err != nil || context == nil {
        return nil, utility.DatabaseConnectionError.WithDescription(err.Error())
    }

    return &Database{DBContext: context}, nil
}

func (database *Database) GetDBContext() *gorm.DB {
    return database.DBContext
}
```

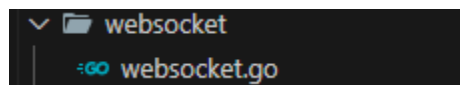
2.3.2. Mail



In the mail folder we have the logic for sending emails. To achieve this, we use the Gomail library for composing and sending emails through an SMTP server.

We have another folder, the templates folder, where we store the templates of our emails. We use the email_config.go file to get the email configuration details (SMTP host, port, etc.). And then we use the email_service.go file for sending the email through SMTP.

2.3.3. WebSocket



In the websocket folder, we have the websocket.go file, which manages the WebSocket functionality of the application. Here we use the Gorilla WebSocket library to upgrade the HTTP connection to the WebSocket protocol and handle messaging. In the file we have an UpgradeConnection that does this. The function handles the WebSocket handshake, upgrading an incoming HTTP connection to a WebSocket connection. Once this is done, we add this connection to a list of active clients.

Another function that the file has is the BroadcastMessage function. This function allows any part of the application to send a message to all connected clients by placing the message in a broadcast channel. Then, The HandleBroadcast function, which will be running in a separate goroutine, listens for messages on the broadcast channel and distributes them to all connected clients.

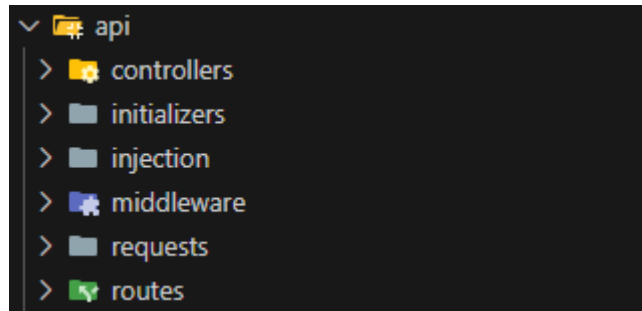
In main.go, a WebSocket route ["/ws"] for the client to connect to. Once connection to this endpoint, their HTTP connection is upgraded to a WebSocket connection using UpgradeConnection function. The application runs the HandleBroadcast function in a goroutine, allowing it to broadcast messages in real time without blocking other operations.

```
// WEBSOCKET
r.GET("/ws", func(c *gin.Context) {
    _, err := websocket.UpgradeConnection(c.Writer, c.Request)

    if err != nil {
        c.JSON(http.StatusInternalServerError, gin.H{"error": "WebSocket upgrade failed"})
        return
    }

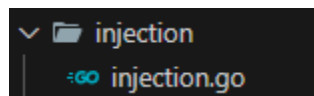
    // Block indefinitely
    select {}
})
```


2.4. API Layer



The API layer is responsible for presenting the endpoints, which the frontend will use to make communications. The layer will make requests to the Application layer, requesting the appropriate use-case be handled.

2.4.1. Dependency Injection



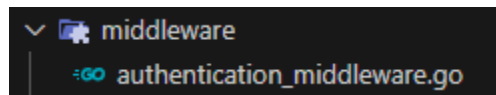
In the injection folder, we have the injection.go file. This file is responsible for managing dependency injection in the application, ensuring that all necessary components, such as repositories, commands, queries, and controllers, are correctly initialized and injected into each other. This design promotes modularity, testability, and most importantly separation of concerns.

Inside of this container, we get the instance of the database context, after having performed a connection with the database, and inject this instance into the unit of work. We also create an instance of the validator library. We inject the UoW, and the validator into the use cases (commands & queries), and finally, inject the commands and queries into the controllers.

```
type AppContainer struct {  
    UnitOfWork contracts.IUnitOfWork // needed in auth-middleware  
  
    AuthenticationController *controllers.AuthenticationController  
    PollController          *controllers.PollController  
}
```

The file returns a struct that contains references to the controllers and the UoW. The references are then passed onto the route declarations, which will be discussed in section 4.5.

2.4.2. Middleware



The middleware folder, is where we find the authentication middleware of our application. The authentication middleware is used to protect endpoints by validating the client JWT token before allowing access to the route. Thus, only authenticated users are allowed interaction with protected resources. The middleware has a dependency on the UoW, needing it to verify user details in the database. Here we use the Golang-JWT library to parse and validate tokens.

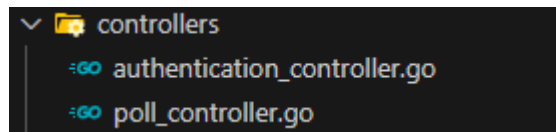
Concretely, the middleware extracts the Authorization header from the incoming HTTP request. In our application we expect the token to be in the headers. It then, parses the token using the secret signing key and validates its signature & claims.

Further on, we verify that the expiration field in the claims of the token to make sure that the token has not expired. If the token is expired, the request is rejected with a 401 status. We also check that the ID claim in the token, corresponds to a user in the database. If not, the request is rejected with 401.

If all is successful, the middleware attaches the authenticated user to the request context, making it accessible to subsequent handlers.

```
c.Set("user", user)
c.Next()
```

2.4.3. Controllers



The controllers folder is where the controllers of our application are found. In the application, controllers handle incoming HTTP requests, by binding the request and invoking the corresponding use case from the application layer. It was made sure, that the controllers be as minimal as possible. They contain no logic whatsoever apart from what is necessary. This makes sure that the principles of CLEAN are followed, and that we have a clear separation of concerns.

Each controller is initialized with dependencies such as commands and queries. For instance, the AuthenticationController is injected with the commands RegisterCommand & LoginCommand.

```

type AuthenticationController struct {
    RegisterCommand contracts.IRegisterCommand
    LoginCommand    contracts.ILoginCommand
    RefreshCommand  contracts.IRefreshCommand
    LogoutCommand   contracts.ILogOutCommand
}

func NewAuthenticationController(RegisterCommand contracts.IRegisterCommand,
    LoginCommand contracts.ILoginCommand,
    RefreshCommand contracts.IRefreshCommand,
    LogoutCommand contracts.ILogOutCommand) *AuthenticationController {
    return &AuthenticationController{
        RegisterCommand: RegisterCommand,
        LoginCommand:    LoginCommand,
        RefreshCommand:  RefreshCommand,
        LogoutCommand:   LogoutCommand}
}

```

Based on the result of the use case, we generate an appropriate HTTP response. Returning a 200 OK in case of success, or the respective error defined in our custom ErrorCode structure in the application's layer utility.

Something that should also be mentioned, is that on top of each method we have the Swagger annotations, which are necessary for generation the Swagger UI. For each method there is a

@Summary: What the API does.

@Description: Description of the API

@Tags: Tags that group the API endpoint into a specific category (Authentication & Polls).

@Accept: Specifies the content type the API expects (json).

@Produce: Specifies the content type the API produces in the response (json).

@Param: Defines the parameters accepted by the API, for example, their name, type, location (body, query), and whether they are required.

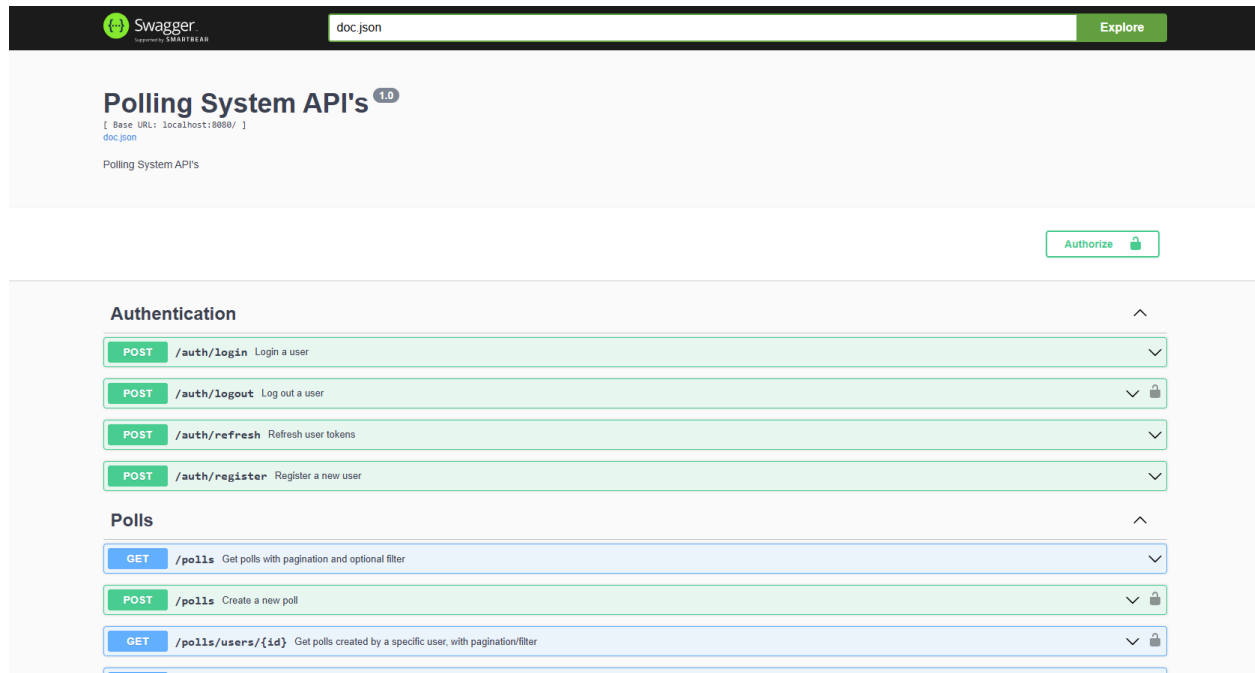
@Success: Describes the response structure and status code for successful requests.

@Failure: Describes the error response structure and corresponding status codes.

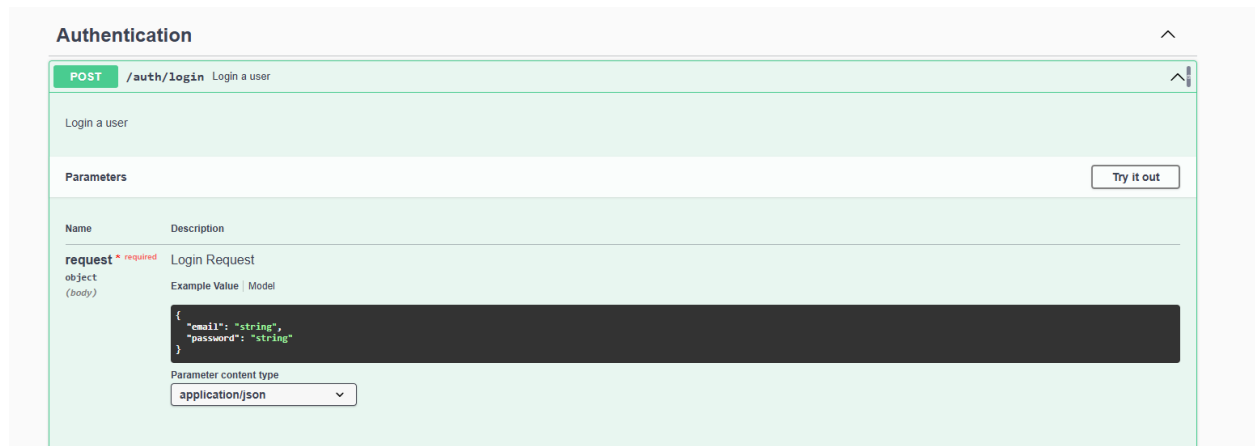
@Router: Specifies the HTTP method and endpoint path (/auth/login [post]).

@Security (Optional): Specifies the security requirements for the endpoint, such as requiring a Bearer token.

All of this produces the following Swagger UI



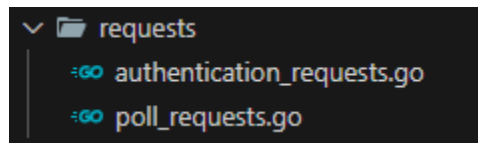
If we expand /login, we can make the request



And we also see the possible responses

| Responses | | Response content type |
|-----------|---|-----------------------|
| | | application/json |
| Code | Description | |
| 200 | JWT Token & Refresh Token | |
| | Example Value Model | |
| | <pre>{ "jwt_token": "string", "refresh_token": "string" }</pre> | |
| 400 | 4xx Errors | |
| | Example Value Model | |
| | <pre>{ "description": "string", "message": "string", "status_code": 0 }</pre> | |
| 500 | 5xx Errors | |
| | Example Value Model | |
| | <pre>{ "description": "string", "message": "string", "status_code": 0 }</pre> | |

2.4.4. Requests

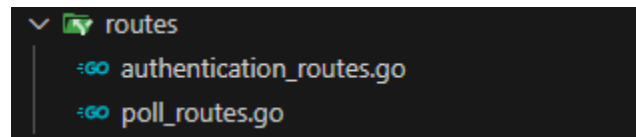


In the requests folder, we define the structs which we expect to receive as requests, in our APIs. For example, in the login API, we expect to have a request from body, so we create the following struct, in order to bind with the request.

```
type LoginRequest struct {
    Email    string `json:"email" validate:"required"`
    Password string `json:"password" validate:"required"`
}
```

Something that should be mentioned, is the “validate” annotation. This is something used by the validator library to perform the validations, which we do inside of the use cases.

2.4.5. Routes



In the routes folder we find the route declarations for the authentication & poll APIs. Each route maps a HTTP endpoint to a corresponding controller action, ensuring that incoming requests are routed correctly to the appropriate business logic. In some of the routes we also integrate our authentication middleware to protect the endpoints.

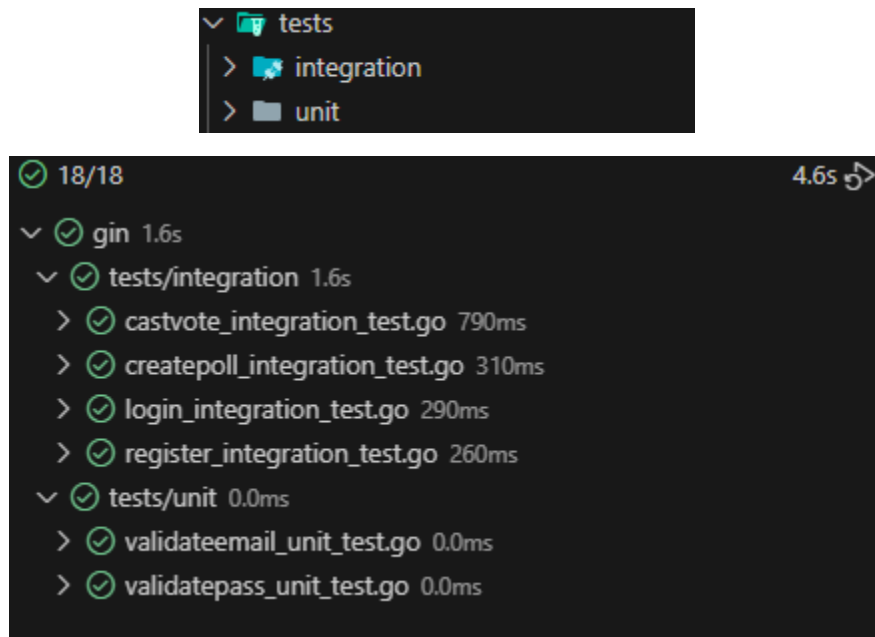
Endpoints are grouped by using Gin's Group method, which organizes routes under a common path (e.g., /polls) for better readability and maintainability.

```
func PollRoutes(r *gin.Engine, controller *controllers.PollController, UnitOfWork contracts.IUnitOfWork) {
    poll := r.Group("/polls")
    {
        poll.POST("", middleware.AuthenticationMiddleware(UnitOfWork), controller.CreatePoll)
        poll.POST("/:id/vote", middleware.AuthenticationMiddleware(UnitOfWork), controller.AddVote)
        poll.DELETE("/:id", middleware.AuthenticationMiddleware(UnitOfWork), controller.DeletePoll)
    }
}
```

Lastly, in the main.go file, the route declaration functions are called to register the routes with the Gin engine.

```
func main() {
    initializers.LoadEnvironmentVariables()
    container := injection.BuildContainer()
    r := gin.Default()
    r.GET("/swagger/*any", ginSwagger.WrapHandler(swaggerFiles.Handler))
    routes.AuthenticationRoutes(r, container.AuthenticationController, container.UnitOfWork)
}
```

3. Tests

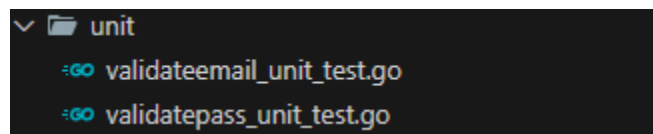


The image shows two screenshots. The top one is a file explorer view of a 'tests' directory containing 'integration' and 'unit' subdirectories. The bottom one is a terminal window showing the output of running tests. It indicates that 18/18 tests passed in 4.6 seconds. The tests are grouped under 'gin' (1.6s), 'tests/integration' (1.6s), and 'tests/unit' (0.0ms). Specific test files and their durations are listed: 'castvote_integration_test.go' (790ms), 'createpoll_integration_test.go' (310ms), 'login_integration_test.go' (290ms), 'register_integration_test.go' (260ms), 'validateemail_unit_test.go' (0.0ms), and 'validatepass_unit_test.go' (0.0ms).

```
✓ 18/18 4.6s
✓ gin 1.6s
  ✓ tests/integration 1.6s
    > ✓ castvote_integration_test.go 790ms
    > ✓ createpoll_integration_test.go 310ms
    > ✓ login_integration_test.go 290ms
    > ✓ register_integration_test.go 260ms
  ✓ tests/unit 0.0ms
    > ✓ validateemail_unit_test.go 0.0ms
    > ✓ validatepass_unit_test.go 0.0ms
```

In the tests folder we store the unit & integration tests of our application. In the application we make use of the Testify library to facilitate testing.

3.1. Unit Tests

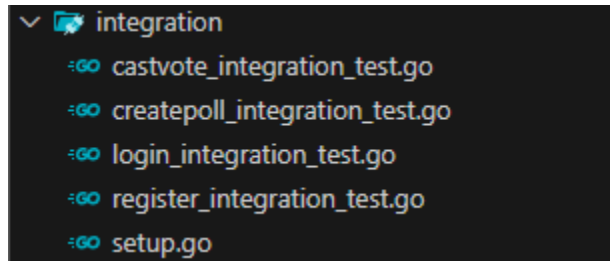


The image shows a file explorer view of the 'unit' directory, which contains two Go test files: 'validateemail_unit_test.go' and 'validatepass_unit_test.go'.

```
unit
  validateemail_unit_test.go
  validatepass_unit_test.go
```

The purpose of unit tests focus is on validating the functionality of individual components in isolation. In the context of our application, we define the following unit tests, that test our validation logic for email & password format, which is defined in the application's layer utilities.

3.2. Integration Tests



Integration tests validate the interactions between multiple components or layers of the application, ensuring that they work together as expected. In the context of our application, we define the following integration tests.

Something that we should mention, is the setup.go file. It spins up a database that will be used by us for these tests only. Then the tests are responsible themselves in the [Arrange] step for creating the tables they need in the database and test the use case.

4. Migrations



Our application has been developed using a code first approach, where the database schema is directly created from the domain entities defined in the domain layer. This approach ensures that the application's domain model drives the database structure, maintaining an alignment between the code and the database schema.

In the migrate.go file, we use of the GORM's AutoMigrate feature for database migrations. This automatically creates or updates the database tables based on the defined entities.

Before actually performing the migration, we make use of our ensureDatabaseExists function to check if the database exists. If it does not, it creates the database. It also adds some seed data for the purpose of this project.

The connection strings and database names are loaded from environment variables stored in the .env file.

To perform the migration, we must change directory to the file where the migrate.go is located, and then run it like a normal go file.

```
cd migration
```

```
go run migrate.go
```


5. Dependencies (Framework / Tools / Libraries)

GIN

go.mod: `github.com/gin-gonic/gin`

<https://github.com/gin-gonic/gin>

Gin is a web framework for Go. Gin provides features like routing, middleware, and utility methods to simplify HTTP based application development. Gin offers advantages compared to using only the vanilla net/http package.

Gin offers an in-built routing mechanism. In vanilla Go, we would need to manually implement our own routing logic, which can be error prone and less efficient. Another thing is middleware, in vanilla we would need to manually chain middleware functions, which adds significant complexity. On the other hand, Gin offers in-built middleware support.

Something else, that we should mention is the `*gin.Context`. Gin provides a `*gin.Context` object for each request, which combines query parameters, headers, form values, and JSON body data into one easily accessible structure. In vanilla we would need to manually parse and manage these components.

Validator

go.mod: `github.com/go-playground/validator/v10`

<https://github.com/go-playground/validator>

The Validator library is a tool used for the validation of structs and fields in Go. It is used for ensuring the integrity of user inputs in web applications.

The library offers in-built validation tags like `required`, `omitempty` etc. Validator simplifies and generalizes input validation for little things.

Golang JWT

go.mod: `github.com/golang-jwt/jwt/v4`

<https://github.com/golang-jwt/jwt>

The Golang JWT library provides an implementation of JSON Web Tokens (JWT) in Go. The library allows encoding and decoding of JWTs & also supports signing methods like HMAC and RSA for token integrity. It also provides tools to verify and validate the tokens.

Gorilla WebSocket

go.mod: `github.com/gorilla/websocket`

<https://github.com/gorilla/websocket>

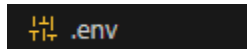
WebSockets are a communication protocol that enables duplex, real-time communication between a client and a server over a TCP connection. Go does offer a vanilla way for using WebSockets, but it is cumbersome and requires a lot of manual effort for features like connection handling, protocol compliance etc.

Gorilla WebSocket provides an easy, high-level API to upgrade HTTP connections and handle WebSocket communications. It also includes in-built support for handling multiple WebSocket connections with features like timeouts, and error handling etc. And due to its wide use, the library has wide documentation available online, making it easy to work with.

Go Environment Variables

go.mod: `github.com/joho/godotenv`

<https://github.com/joho/godotenv>



The godotenv library is a simple tool for the management of environment variables in Go. It separates sensitive configurations from the codebase, thus improving security & simplifying deployment.

Testify

go.mod: `github.com/stretchr/testify`

<https://github.com/stretchr/testify>

Testify is a testing library for Go that makes it possible to write tests. It provides a set of assertion methods to validate test conditions.

Crypto

go.mod: `golang.org/x/crypto`

<https://pkg.go.dev/golang.org/x/crypto/bcrypt>

The crypto package provides cryptographic utilities. Its bcrypt implementation is used for password hashing.

Swagger + Swagger GIN**go.mod: github.com/swaggo/files****go.mod: github.com/swaggo/gin-swagger****go.mod: github.com/swaggo/swag**<https://github.com/swaggo/gin-swagger>

The Swaggo library integrates Swagger with Go applications, enabling the generation of API documentation. The swag package parses code annotations to produce OpenAPI-compliant documentation.

Go Mail**go.mod: gopkg.in/gomail.v2**<https://github.com/go-gomail/gomail>

The gomail library is a simple tool for sending emails in Go applications. It supports SMTP servers and allows the sending of emails with features like attachments, custom headers etc.

Gorm + Postgres Driver**go.mod: gorm.io/gorm****go.mod: gorm.io/driver/postgres**https://gorm.io/docs/connecting_to_the_database.html#PostgreSQL

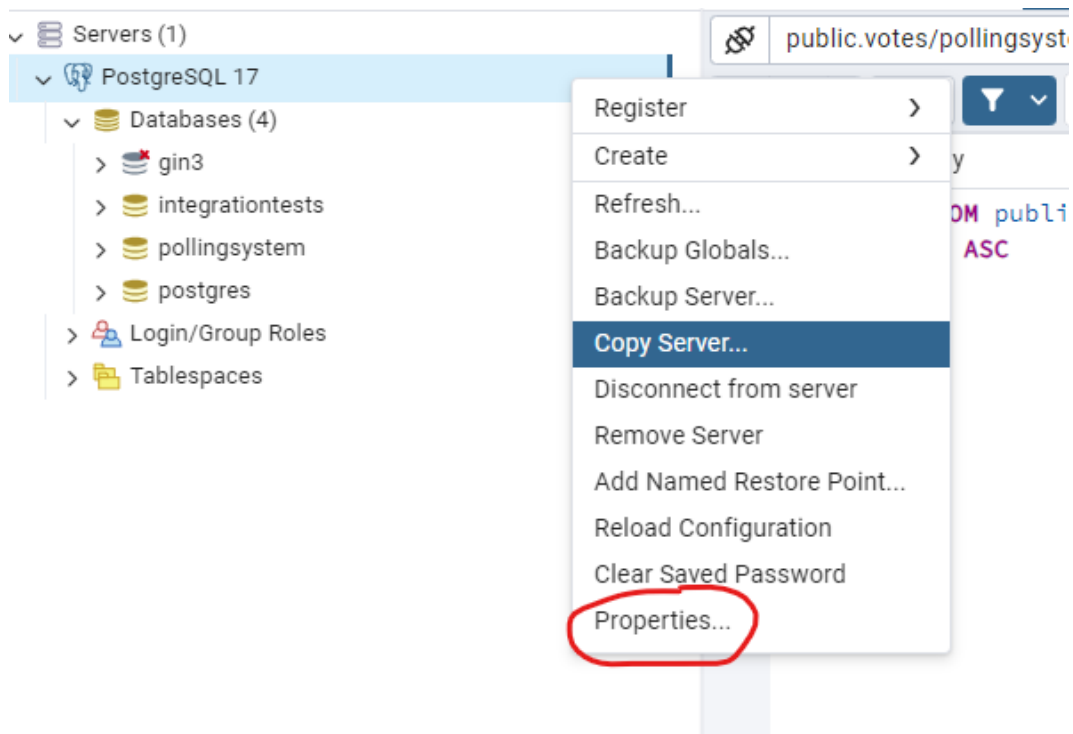
GORM is an Object Relational Mapper (ORM) library for Go. For the application we are using postgres so a postgres driver is also needed. GORM allows us to work directly with our structs instead of writing SQL queries for performing database operations.

GORM also supports automatic migrations, allowing us to sync any changes to our code directly to the database schema.

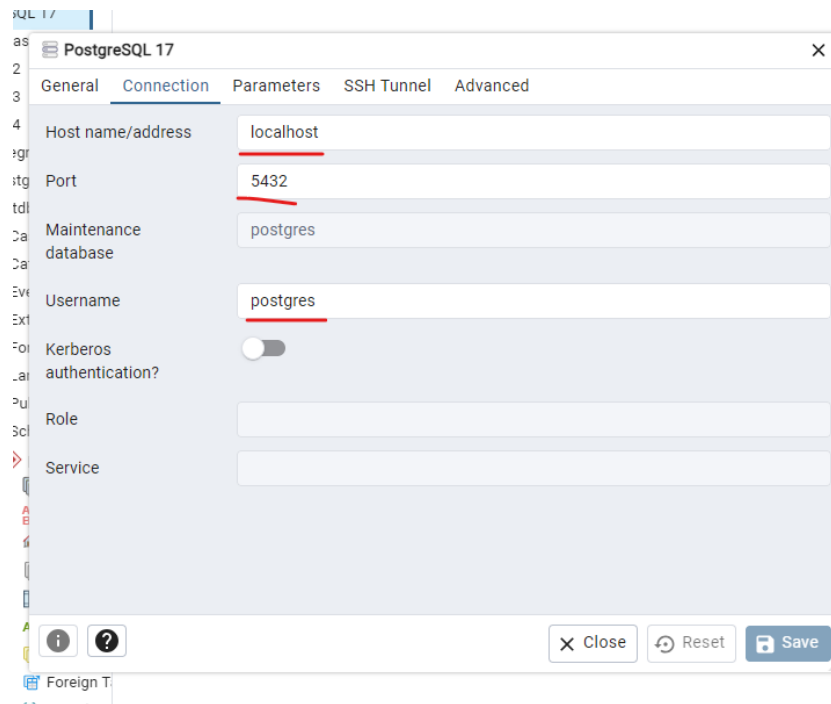
6. Instructions

Install Go & PostgreSQL & PGAdmin.

In PGAdmin after having chosen a password for your Postgres user, left clicking on the PostgreSQL 17 & choosing properties:



Will open up this window:



There will be available the hostname/address, port & username, which should be used to update the environment variables (located in .env file) if necessary. The password is the postgresql password, you chose during Postgres setup.

```
.env
You, last week | 1 author (You)
1 PORT=8080
2 CONNECTION_STRING="host=localhost user=postgres password=postgres dbname=gin3 port=5432 sslmode=disable"
3 SERVER_STRING="host=localhost user=postgres password=postgres port=5432 sslmode=disable"
4 DATABASE_NAME="gin3"
5 SECRET_JWT="OKE8z1LQypY77kjIMb7FZN50WkFhapUt0fsUaArLTQRvaMUXifmHghl8r00WZd5g"
6 EXPIRY_JWT=1
7 EXPIRY_REFRESH=7
8 SMTP_HOST="smtp.gmail.com"
9 SMTP_PORT=587
10 SENDER_EMAIL="pollsystemdp@gmail.com"
11 SENDER_PASS="jmizxwgcjbebsqe"
12 ACTUAL_PASS="Unifi2024"
13 CONNECTION_STRING_TEST="host=localhost user=postgres password=postgres dbname=testdb port=5432 sslmode=disable"
14 SERVER_STRING_TEST="host=localhost user=postgres password=postgres port=5432 sslmode=disable"
15 DATABASE_NAME_TEST="testdb"
You, last week * done
```

After that is done, the dependencies should be installed with:

go mod download

Run the database migrations and also some seed data:

cd migration

go run migrate.go

And finally, run the application:

go run main.go

After the application is ran, the following link should be available to reach the Swagger UI:

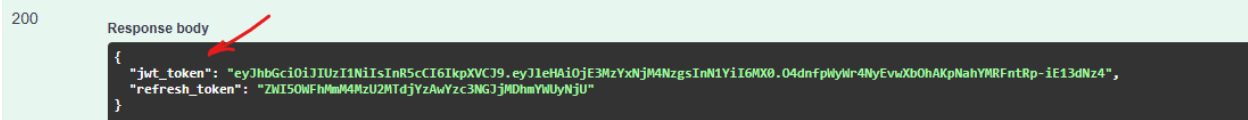
<http://localhost:8080/swagger/index.html>

After having reached the Swagger UI, either create a new user with the Register endpoint and then Login, or use the data of one of the seeded users in the Login endpoint.

"email": "user1@gmail.com",

"password": "password"

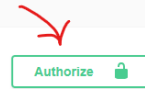
The login endpoint will return the jwt_token:



The token should be used in the Authorize button, which adds the JWT token in the header, for every request.

System API's ^{1.0}

:8080/ 1



ion



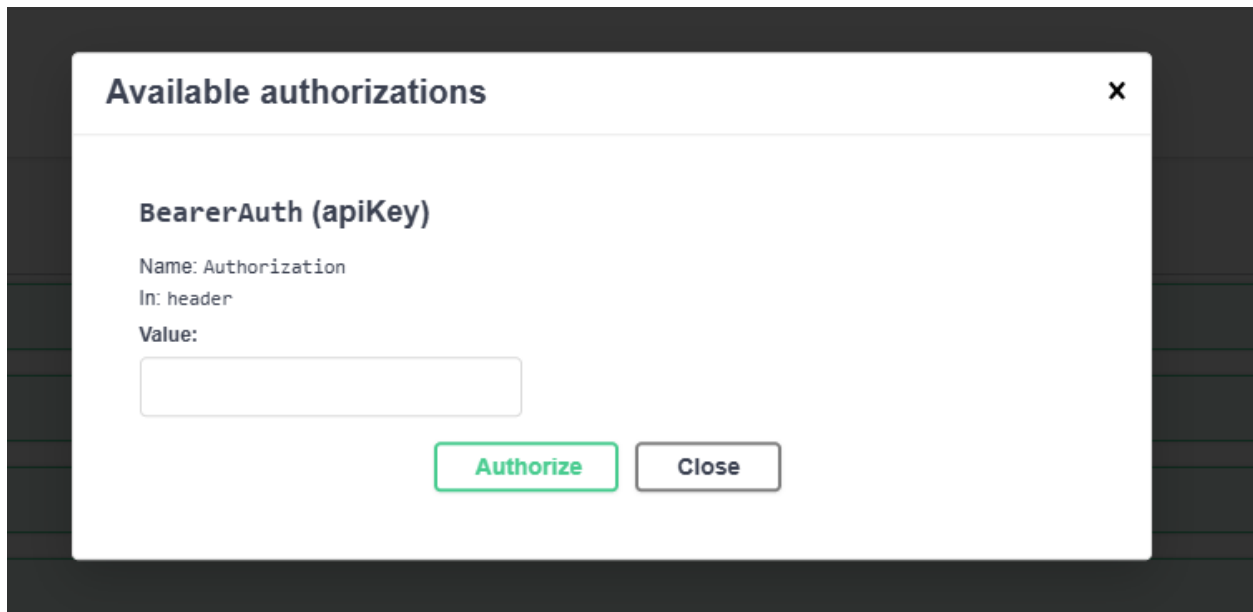
/login Login a user



/logout Log out a user



The token, should be pasted in the following popup window:



Now every endpoint is available both the ones that require authentication, and the ones that do not.

The other file **example client**, mimics how a client would make a connection to our websocket, in order to receive real-time updates. It is not necessary for the main application to work, but can be used to see the broadcasted messages. To run it, download the dependencies & run.

go mod download

go run main.go



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  COMMENTS
PS C:\Users\User\Desktop\Documents\UNIFI\Distributed\example client> go run .\main.go
2025/01/06 12:59:19 Connection error: failed to connect to WebSocket server: dial tcp [::1]:8080: connectex: No connection could be made because the target machine actively refused it.. Retrying in 5 seconds...
2025/01/06 12:59:24 Connection error: failed to connect to WebSocket server: dial tcp [::1]:8080: connectex: No connection could be made because the target machine actively refused it.. Retrying in 5 seconds...
Connected to WebSocket server!
Received message: {"type":"vote-cast","data":{"poll_id":1,"categories":[{"category_id":1,"category_votes":1},{"category_id":2,"category_votes":2},{"category_id":3,"category_votes":0}]}}
```