# QuestList Backend API Documentation

## Introduction

The backend API for the Quest List project specializes in retrieving and storing data passed into it by the front-end servers. It aims to allow the front-end to properly retrieve and store information about individual users in the form of sign up (storage of information) and sign in (information retrieval). When dealing with games, our API supports a variety of function calls that can provide the front-end developers with different game information depending on the specifications passed into our function calls.

## How is Information Shared?

When communicating between the front and back-end, two different procedures are employed depending on whether information is being requested by the front or being sent to the front.

### *How can the front-end call the backend?*

To call any of the functionality from the backend, the front must make a URL request to the server hosting the backend (in our case http://localhost:8080/). When calling these URLs, certain handlers must be specified after the final "/" to attain the desired results. The handlers and their functions are listed below.

### *What data can be sent to the backend and in what form can it be sent?*

Information can be sent to the backend through json files. Json files can be created and formatted in a way that matches one of the structs outlined below. Structs are basically objects, so in order to properly have all values intended to be sent to the

back be interpreted correctly, a similar object must be created in the front and packaged into a json file. All structs used in the back are listed below and the documentation for each handler will advise you about the requirements of json files. When certain parameters are required by the backend in the URL, we expect the front-end to use query parameters to add the necessary information which the backend then reads from the URL

*How is data sent from the backend to the front-end?*

Once an operation is completed,  the backend will return two forms of data. The first return is found in every operation handled by the backend: an http status. If a function call is successful and everything was retrieved, stored, or modified correctly, the backend will write to the header: “http.StatusOK.” If something went wrong, each function call will write a different http status to alert to potential errors.

The second type of data returned is a json. Though not all operations return a json file, the ones that do will be specified and the layout of the file will be shown.

*Where does the game content come from?*

The API we used is titled RAWG API. RAWG is a database of games storing a variety of information including but not limited to: name, image, different ratings, release date, and developers. We have used this data to organize and traverse the database to acquire desired information. In order to easily use the RAWG API, we found “RAWG SDK GO”, a client with built in functions to traverse the RAWG database. The documentation for both RAWG API and RAWG SDK GO are linked below.

- RAWG API: https://api.rawg.io/docs/

- RAWG SDK GO: https://pkg.go.dev/github.com/dimuska139/rawg-sdk-go#section-documentation

## Structs Used Throughout the Backend

### 1) User Struct

*Purpose*

Used to store and communicate information about users of the website. Stores a username (string), an email address (string), and a password (string).

*Format*

```
type User struct {
    gorm.Model
    Username string `gorm:"uniqueIndex"`
    Email    string `gorm:"uniqueIndex"`
    Password string
}
```

### 2) Review Struct

*Purpose*

Used to store and communicate information about the reviews users have made about a specific game. It stores the name of the game (string), the rating of the game (float), the description of the review a user has commented (string), the username (string), and the play status of the game in question (string).

Note: the playstatus can only be one of the following strings: `PLAYING, DROPPED, COMPLETED, ON HOLD`

*Format*

```
type Review struct {
    gorm.Model
    GameName    string  //Names of game being reviewed
    Rating      float32 //Rating (out of 5) of the game
    Description string  //Description of the game played
    Username    string  //Name of the account
    PlayStatus  string  //PLAYING, DROPPED, COMPLETED, ON HOLD
}
```

## Handler Functions and their Functionality

**1) GET: {HelloWorld}: http://localhost:8080/**

*Functionality*

This url serves as a test. It prints out a message to the backend page if the server is up and running. This is not meant to return nor do any meaningful computations and can be used to test if the backend servers are running

*Status Returned:*

If Successful: http.StatusOK

If Unsuccessful: N/A

*Json Returned & Format:*

N/A

**2) GET: {Get a json of 10 games most similar to provided name}:**

**http://localhost:8080/specific-game**

*Functionality*

This url retrieves the 10 games most similar to the name provided in the query parameters provided and sends a json of these 10 games. We use RAWG SDK's NewGamesFilter() function along with SetPageSize(int numElements) and SetSearch(string gameName) to retrieve an array of *rawg.Game objects. SetPageSize(int numElements) takes in the parameter 10 to set the number of returned elements to 10. SetSearch(string gameName) then takes in the query parameter name as a parameter and retrieves the 10 most similar elements to that name. From there, we json.Marshal() these games and write these games to the header.

### *Status Returned:*

If Successful: http.StatusOK

If Unsuccessful: N/A

### *Json Returned & Format:*

```
type Game struct {
    ID              int            `json:"id"`
    Slug            string         `json:"slug"`
    Name            string         `json:"name"`
    Released        DateTime       `json:"released"`
    Tba             bool           `json:"tba"`
    ImageBackground string         `json:"background_image"`
    Rating          float32        `json:"rating"`
    RatingTop       int            `json:"rating_top"`
    Ratings         []*Rating      `json:"ratings"`
    RatingsCount    int            `json:"ratings_count"`
    ReviewsTextCount int           `json:"reviews_text_count"`
    Added           int            `json:"added"`
    AddedByStatus   *AddedByStatus `json:"added_by_status"`
    Metacritic      int            `json:"metacritic"`
    Playtime        int            `json:"playtime"`
    SuggestionsCount int           `json:"suggestions_count"`
    ReviewsCount    int            `json:"reviews_count"`
    SaturatedColor  string         `json:"saturated_color"`
    DominantColor   string         `json:"dominant_color"`
    Platforms       []*struct {
        Platform       *Platform    `json:"platform"`
        ReleasedAt     DateTime     `json:"released_at"`
        RequirementsEn *Requirement `json:"requirements_en"`
        RequirementsRu *Requirement `json:"requirements_ru"`
    } `json:"platforms"`
    ParentPlatforms []*struct {
        Platform struct {
            ID   int    `json:"id"`
            Slug string `json:"slug"`
            Name string `json:"name"`
        }
    } `json:"parent_platforms"`
    Genres []*Genre `json:"genres"`
    Stores []*struct {
        ID    int    `json:"id"`
        Store *Store `json:"store"`
        UrlEn string `json:"url_en"`
        UrlRu string `json:"url_ru"`
    } `json:"stores"`
    Clip             *Clip  `json:"clip"`
    Tags             []*Tag `json:"tags"`
    ShortScreenshots []*struct {
        ID   int    `json:"id"`
        Name string `json:"name"`
    } `json:"short_screenshots"`
}
```

**3) GET: {Get a json of all games in a specific page of the database}:**

**http://localhost:8080/games**

*Functionality*

This function gets a json of all the games in a page specified through the url query parameters. These parameters specify the page and page size which we then pass into SetPage(int page) and SetPageSize(int numElements) respectively along with RAWG SDK's NewGamesFilter() function. The combination of these functions returns an array of size "numElements" from the page "page" in the game database. From there, we json.Marshal() these games and write these games to the header.

*Status Returned:*

If Successful: http.StatusOK

If Unsuccessful: N/A

*Json Returned & Format:*

```
type Game struct {
    ID               int            `json:"id"`
    Slug             string         `json:"slug"`
    Name             string         `json:"name"`
    Released         DateTime       `json:"released"`
    Tba              bool           `json:"tba"`
    ImageBackground  string         `json:"background_image"`
    Rating           float32        `json:"rating"`
    RatingTop        int            `json:"rating_top"`
    Ratings          []*Rating      `json:"ratings"`
    RatingsCount     int            `json:"ratings_count"`
    ReviewsTextCount int            `json:"reviews_text_count"`
    Added            int            `json:"added"`
    AddedByStatus    *AddedByStatus `json:"added_by_status"`
    Metacritic       int            `json:"metacritic"`
    Playtime         int            `json:"playtime"`
    SuggestionsCount int            `json:"suggestions_count"`
    ReviewsCount     int            `json:"reviews_count"`
    SaturatedColor   string         `json:"saturated_color"`
    DominantColor    string         `json:"dominant_color"`
    Platforms        []*struct {
        Platform       *Platform    `json:"platform"`
        ReleasedAt     DateTime     `json:"released_at"`
        RequirementsEn *Requirement `json:"requirements_en"`
        RequirementsRu *Requirement `json:"requirements_ru"`
    } `json:"platforms"`
    ParentPlatforms []*struct {
        Platform struct {
            ID   int    `json:"id"`
            Slug string `json:"slug"`
            Name string `json:"name"`
        }
    } `json:"parent_platforms"`
    Genres []*Genre `json:"genres"`
    Stores []*struct {
        ID    int    `json:"id"`
        Store *Store `json:"store"`
        UrlEn string `json:"url_en"`
        UrlRu string `json:"url_ru"`
    } `json:"stores"`
    Clip             *Clip  `json:"clip"`
    Tags             []*Tag `json:"tags"`
    ShortScreenshots []*struct {
        ID   int    `json:"id"`
        Name string `json:"name"`
    } `json:"short_screenshots"`
}
```

**4) Post: {Create a new user and add it to the database}:**

**http://localhost:8080/sign-up**

*Functionality*

This function creates a user and adds it to the database if it is not already there. This function starts by opening the user database and migrating the format of the user struct to Gorm's database. The function then pulls the body content from the front-end using json.Decode() to put this struct in a user object of type struct User. The function then validates the uniqueness of the email and username. If either the username or email exist in the database already, the user will not be added to the database and the http.StatusInternalServerError status will be returned. Otherwise, the user is added to the database and the http.StatusCreated status is returned.

*Status Returned:*

If Successful: http.StatusCreated

If Unsuccessful: http.StatusInternalServerError

*Json Returned & Format:*

```
type User struct {
    gorm.Model
    Username string `gorm:"uniqueIndex"`
    Email    string `gorm:"uniqueIndex"`
    Password string
}
```

**5) POST: {Sign a user in and assure their account exists}:**

**http://localhost:8080/sign-in**

*Functionality*

This function finds a user in the database and logs that user in if the user is found in the database. Similar to the SignUp() function, this function starts by opening the user database and migrating the format of the user struct to Gorm's database. The function then pulls the body content from the front-end using json.Decode() to put this struct in a user object of type struct User. The function then validates that the specified user exists in the database. If it doesn't, the http.StatusInternalServerError status is returned. However, if the username exists in the database, the password is then verified, returning the Internal Server Error status if it doesn't match the one in the database. Lastly, if the password matches the user is logged in by storing that username in a currentlyActiveUser variable and writing the http.StatusOK to the header telling the front-end that the user was found.

*Status Returned:*

If Successful: http.StatusOK

If Unsuccessful: http.StatusInternalServerError

*Json Returned & Format:*

```
type User struct {
    gorm.Model
    Username string `gorm:"uniqueIndex"`
    Email    string `gorm:"uniqueIndex"`
    Password string
}
```

**6) POST: {Create a new review for a user}: http://localhost:8080/writeareview**

*Functionality*

This function creates a new review for a user and overwrites the user's previous review if they have already created one for this game. This function starts by opening the review database and migrating the format of the review struct to Gorm's database. The function then pulls the body content from the front-end using json.Decode() to put this struct in a review object of type struct Review. The function then checks if the user has already created a review for this game. If they have, the rating, description, and play status are changed for that specific review in the database and the http.StatusOK status is returned. If they have not created a review yet for this game, a new review object is created in the database in the Review struct format and the http.StatusCreated status is returned. It should be noted that the user does not have to add every element to the review (e.g. a user may create a review with no description).

*Status Returned:*

If Successful: http.StatusOK or http.StatusCreated

If Unsuccessful: N/A

*Json Returned & Format:*

```
type Review struct {
    gorm.Model
    GameName    string  //Names of game being reviewed
    Rating      float32 //Rating (out of 5) of the game
    Description string  //Description of the game played
    Username    string  //Name of the account
    PlayStatus  string  //PLAYING, DROPPED, COMPLETED, ON HOLD
}
```

**7) GET: {Get a json with a list of a specified user's review structs}:**

**http://localhost:8080/getreviews**

*Functionality*

This function takes in a parameter of type user struct and retrieves all of the reviews created by that user. This function starts by opening the review database and migrating the format of the user struct to Gorm's database. The function then pulls the body content from the front-end using json.Decode() to put this struct in a user object of type struct User. The function then searches in the database for all instances of user.Username and returns an array of reviews of type *Review.

*Status Returned:*

If Successful: http.StatusOK

If Unsuccessful: http.StatusInternalServerError

*Json Returned & Format:*

```
type Review struct {
    gorm.Model
    GameName    string  //Names of game being reviewed
    Rating      float32 //Rating (out of 5) of the game
    Description string  //Description of the game played
    Username    string  //Name of the account
    PlayStatus  string  //PLAYING, DROPPED, COMPLETED, ON HOLD
}
```

**8) GET: {Get a json of the 4 most recent games}: http://localhost:8080/recent**

*Functionality*

This function returns a json of the 4 most recent games added to the games database. The function starts by using RAWG SDK's NewGamesFilter() function along with SetPageSize(int numElements) with parameter 4 and SetOrdering(string order) with parameter released. The combination of these functions returns one page of games of size 4 in order of release starting with the most recent game added to the database. These games are then packaged using json.Marshal and sent to the header.

*Status Returned:*

If Successful: http.StatusOK

If Unsuccessful: N/A

*Json Returned & Format:*

```
type Game struct {
    ID                int            `json:"id"`
    Slug              string         `json:"slug"`
    Name              string         `json:"name"`
    Released          DateTime       `json:"released"`
    Tba               bool           `json:"tba"`
    ImageBackground   string         `json:"background_image"`
    Rating            float32        `json:"rating"`
    RatingTop         int            `json:"rating_top"`
    Ratings           []*Rating      `json:"ratings"`
    RatingsCount      int            `json:"ratings_count"`
    ReviewsTextCount  int            `json:"reviews_text_count"`
    Added             int            `json:"added"`
    AddedByStatus     *AddedByStatus `json:"added_by_status"`
    Metacritic        int            `json:"metacritic"`
    Playtime          int            `json:"playtime"`
    SuggestionsCount  int            `json:"suggestions_count"`
    ReviewsCount      int            `json:"reviews_count"`
    SaturatedColor    string         `json:"saturated_color"`
    DominantColor     string         `json:"dominant_color"`
    Platforms         []*struct {
        Platform        *Platform    `json:"platform"`
        ReleasedAt      DateTime     `json:"released_at"`
        RequirementsEn *Requirement `json:"requirements_en"`
        RequirementsRu *Requirement `json:"requirements_ru"`
    } `json:"platforms"`
    ParentPlatforms []*struct {
        Platform struct {
            ID   int    `json:"id"`
            Slug string `json:"slug"`
            Name string `json:"name"`
        }
    } `json:"parent_platforms"`
    Genres []*Genre `json:"genres"`
    Stores []*struct {
        ID    int    `json:"id"`
        Store *Store `json:"store"`
        UrlEn string `json:"url_en"`
        UrlRu string `json:"url_ru"`
    } `json:"stores"`
    Clip             *Clip  `json:"clip"`
    Tags             []*Tag `json:"tags"`
    ShortScreenshots []*struct {
        ID   int    `json:"id"`
        Name string `json:"name"`
    } `json:"short_screenshots"`
}
```