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
}
```

3) GameRanking Struct

Purpose

This is the structure for our overarching rating database for the games on our website. Every time a new game is added to our website or rated for the first time, an entry in the UserGameRanking database is made using this scheme. The GameName is the name of the video game, the AverageRating is an average calculated by dividing the overall sum of all user scores for a game by the number of reviews. NumReviews is the number of reviews a game has.

Format

```
type GameRanking struct {
    gorm.Model
    GameName      string  `gorm:"uniqueIndex"` // Name of game
    AverageRating float32 // Average Rating (out of 5) of the game
    NumReviews    int     // Number of times a game has been reviewed
}
```

Databases Used Throughout the Backend

1) User Database: “currentlyUsers.db”

Purpose

Stores and catalogs all the existing users that Quest List has. Users are stored in rows formatted using the “User” struct. Existing users can be added using the “sign-up” function and retrieved using “getusers.”

2) Review Database: “reviews.db”

Purpose

Stores and catalogs every review made by a user of the website. Each row is formatted using the “Review” struct. Each entry is for a different game by a different user. Reviews for an existing game by an existing user can be overwritten to limit memory use. New reviews can be added using “writereview” and retrieved using “getreview.”

3) Quest List Average Ratings Per Game: “UserGameRankings.db”

Purpose

Stores and constantly updates the average ratings each game on the Quest List website has. Each game has one entry and has an average rating score and a count of the total reviews made. Each row is formatted using the “AverageUserRating” struct. This database is updated automatically when a new review is created or updated.

Handler Functions and their Functionality

1) HELLO

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) SPECIFIC-GAME

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: **{An Array of}**

```
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) GAMES

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: **{An Array of}**

```
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) SIGN-UP

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 Expected & Format:

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

5) SIGN-IN

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 Expected & Format:

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

6) WRITE A REVIEW

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 Expected & 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 A REVIEW

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) RECENT GAMES

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: **{An Array of}**

```
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"`
}
```

9) TOP RATED GAMES

GET: {Get a json of the top 5 games based on QuestList user ratings}:

http://localhost:8080/topgames

Functionality

The function of this method is to return the RAWG game information for the top 5 games that have the highest average user rating based on ratings from QuestList itself. Once a user makes a review, the average rating for a game based on all user input is recalculated and stored. This function opens the “UserGameRankings.db”

database and utilizes Gorm's "Where" function. It searches the database of all games' average ratings and uses a quick sort algorithm to order the result in descending order based on ratings. The top 5 game names are searched using RAWG SDK's NewGamesFilter(), SetPage() and SetSearch() functions using the game names and a page size of 1. The array of ordered RAWG games are then marshaled and packaged into a json file which is sent to the header.

Status Returned:

If Successful: http.StatusOK

If Unsuccessful: http.StatusInternalServerError

Json Returned & Format: **{An Array of:}**

```
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"`
}
```

10) UPCOMING GAMES

GET: {Get a json, of the requested size, of games that will release within the next month}:

http://localhost:8080/upcominggames

Functionality

This function returns an array of RAWG games that have not been released yet, but will be available within the next month, starting tomorrow. This uses a simple time calculation to calculate the next month's time frame and pass that information to the RAWG SDK. An SDK filter is used with the NewGameFilter(), SetPageSize(), SetDates(), and SetOrdering function to find games within the time frame that will release soon, ordered in terms of pre-existing anticipation of the game retrieved from RAWG itself. The array of games are then marshaled and packed into a json file and sent to the header.

Status Returned:

If Successful: http.StatusOK

If Unsuccessful: N/A

Json Returned & Format: **{An Array of:}**

```
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"`
}
```

11) GET USER

GET: {Returns a json file of an array of user struct's whose names somewhat match the passed in name}:

http://localhost:8080/getuser

Functionality

This function returns an array of User objects most similar to the username provided in the URL contents. The function begins by opening the user database, "currentUsers.db", and pulling the username with parameter name "user" from the URL. GORM's AutoMigrate is then called to format the database to accept the User{} struct. Next, GORM's db.Where() function is called with functionality "LIKE" to find all users with a name similar to the username pulled from the URL previously and stores it in the array titled users of type []User also storing an error in the hasUsers

variable of type err. Lastly, if there are no users with a name similar to the query parameter, hasUsers will store an error and http.StatusInternalServerError will be thrown. Otherwise, if there are users in the "users" array, those objects will be converted to a byte array using json.Marshal() and written to the header along with a http.StatusOK. Additionally, the function returns the users array to be used for Unit Testing.

Status Returned:

If Successful: http.StatusCreated

If Unsuccessful: http.StatusInternalServerError

Json Returned & Format: **{An Array of:}**

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

12) RECENTLY MADE REVIEWS

GET: {Returns a json file of an array of review structs added within the past month from most recent to least recent}:

http://localhost:8080/recentreviews

Functionality

This function returns all of the reviews from the last month in order of recency. This function starts by opening the "Reviews.db" database and setting the time frame for the function starting from the current time (start) to a month ago(end). Next, GORM's AutoMigrate is then called to format the database to accept the

Review{} struct. GORM's db.Where() function then uses the ">" functionality with the updated_at column to find all reviews with an updated time greater than a month ago. The reviews are then stored in latestReviews of type []Review and recentReviews of type err. If recentReviews has a non-nil error, a http.StatusInternalServerError is written to the header, else latestReviews is reversed using the reverseArray([]Review) function to order the reviews from most to least recent and latestReviews is converted into a json and written to the header along with the status http.StatusOK. Additionally, if latestReviews has elements in it, they are returned for Unit Testing, else nil is returned.

Status Returned:

If Successful: http.StatusOK

If Unsuccessful: http.StatusInternalServerError

Json Returned & Format: **{An Array of:}**

```
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
}
```

13) FEATURED GAMES

GET: {Returns a json of the game with the most number of reviews in the last month}:

http://localhost:8080/featuredgame

Functionality

This function returns a RAWG game object that corresponds to the element in UserGameRankings.db with the most reviews. This function starts by opening the "UserGameRankings.db" and setting the time frame for the function starting from the current time (start) to a month ago(end). Next, GORM's AutoMigrate is then called to format the database to accept the GameRanking{} struct. GORM's db.Where() function then uses the ">" functionality with the updated_at column to find all reviews with an updated time greater than a month ago. The game rankings are then stored in gameRankings of type []GameRankings and recentRankings of type err. If recentRankings has a non-nil error, a http.StatusInternalServerError is written to the header. Otherwise, the name of the game with the highest number of reviews is stored in featuredGame of type string. The rawg function chain NewGamesFilter().SetPageSize(int pageSize).SetSearch(string gameName) is then used with parameters 1 and featuredGame respectively and stores a []*rawg.Game object in games. The array games is an array of size 1 containing the game with name featuredGame. For functionality purposes, this game is stored in variable "game" of type *rawg.Game by defining it with games[0]. This game is then converted to a json file and sent to the header along with the status http.StatusOK. Additionally, if game is defined, it is returned for Unit Testing, else nil is returned.

Status Returned:

If Successful: http.StatusOK

If Unsuccessful: http.StatusInternalServerError

Json Returned & Format: **{An Array of}**

```
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"`
}
```