Backend Team Report

7.04.2023

—

Members

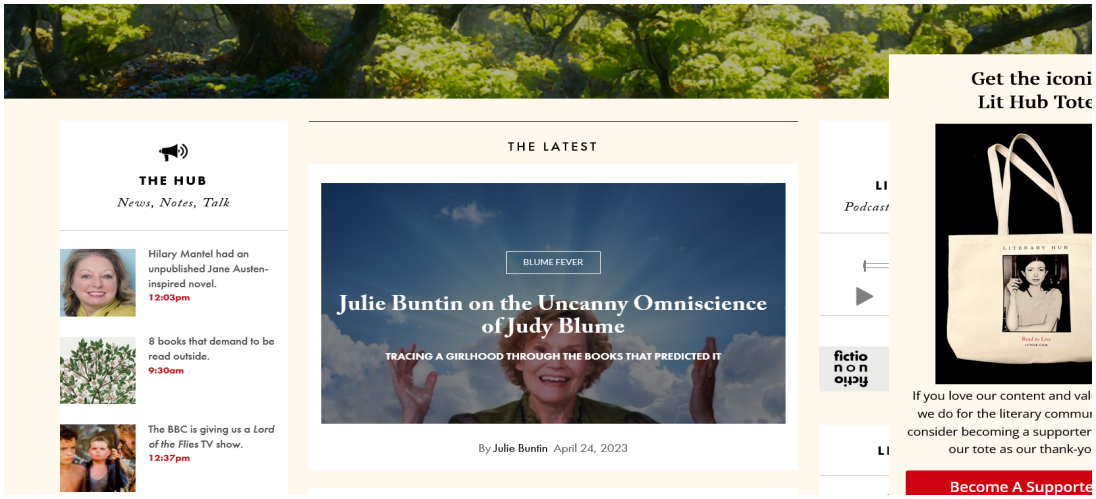
18b9062 MR CHEW KENG SIONG

19b2064 MR LIM BENG LEI

19b2100 AWG MOHAMMAD AMIR IZZUDDIN BIN HAJI NORHAMIDI

19b9057 MR MUHAMMAD MIRZA AKMAL BIN AWANG ROSLIE

Introduction



Project Overview

*The aim of the project is to create a website similar to the website "**LitHub**". As the backend team, We are responsible for ensuring that the database and host server are functional and ready to be deployed. Some of the main goals of our team are to create a database using NoSQL, hosting the website via GitHub and ensure a successful connection between the front end and the back end side of the website. By the end of this project, after having reached and completed all the goals listed in the Goals section. We will have a fully functioning website that is not only superficially but also functionally similar to the LitHub website.*

Technologies and tools used

The Backend team made use of several technologies and tools during the development cycle of this project. The first being *Firebase*, which was used to create and manage the database of the project.In addition to *Firebase*, the team utilized *PHPStorm IDE* as their primary development environment. MongoDB was used to showcase Database Switching using IoC Dependency Injection. Which required the installation of other tools such as *XAMPP PHP (Version 8.1.12)*, *Xdebug*, *Composer*, *gRPC (Version 1.43.0) and MongoDB(Version 1.16)*. Furthermore, the team was able to obtain a free IDE student license through JetBrains, which helped to reduce costs and improve productivity.

The team also utilized *GitHub* as their version control system, which allowed for seamless collaboration and version tracking. By connecting to the project repository to*PHPStorm*, the team was able to manage the project's codebase and track changes over time through the IDE.

Overall, the use of these technologies and tools helped the backend team to work efficiently and effectively, resulting in a successful project delivery. By leveraging these technologies and tools, the team was able to develop a high-quality system that met the project's requirements and exceeded expectations.

Installation guide for PHP IDE

Below is a step by step guide to install the software used by the backend team during this project

Please ensure Software VERSION ARE ALL COMPATIBLE or adheres to the versions listed below

1. Install PHPStorm IDE
2. Install XAMPP PHP 8.1.12
3. Install Xdebug (Use the wizard)
    a. Open your CMD terminal
    b. Then using XAMPP command and copy/ paste the output into the Xdebug webpage
    c. Follow the instructions outputted by the Xdebug wizard
4. Install Composer (use installer)
5. Install gRPC 1.43.0
6. Connect to the GitHub project through PHPStorm
    a. This requires access from the repository owner
    b. After that, simply git pull the project
7. Integrate Apache Server with IDE for seamless deployment
8. Setup IDE's CLI Interpreter
9. Setup composer by running 'composer install' in terminal
10. Install MongoDB
    a. Browse the MongoDB PHP driver's latest and most stable version from the official website.
    b. Download the Thread Safe(TS) zip file according to the PHP version and machine configuration.
    c. After extraction, copy the php_mongodb.dll file and paste it to this path "C:\xampp\php\ext".
    d. Open the php.ini file and add this file "extension = php_mongo.dll" to the series of extensions.

Rest API

REST API, or Representational State Transfer API, has gained popularity as a preferred option for creating web services owing to its flexibility and simplicity. The primary objective of using REST API is to enable smooth communication between clients located on the frontend and servers on the backend. The RESTful design permits various components of a web application to interact seamlessly, thus making it uncomplicated for developers to build applications that can function on any platform. With REST, developers can establish web services that are easily accessible through standard HTTP methods, thus enabling users to communicate with the server through simple and instinctive requests. This expedites the development process and enhances user experience. In conclusion, REST API streamlines communication between the frontend and backend, making it an ideal choice for developing contemporary web applications.

Dependency Injection

The container.php file, also known as an Inversion of Control (IoC) Container, is a valuable tool for PHP developers as it allows for looser coupling of classes in their applications. By using an IoC container, developers can create and manage dependencies between objects in their code more easily, resulting in a more modular and flexible codebase. The container.php file implements the functionalities for defining and resolving dependencies, which are then injected into classes when needed, removing the need for tightly-coupled dependencies. This makes code easier to maintain and test, as changes to one class won't have unintended consequences on others. In addition, using an IoC container can lead to faster development and easier scalability. Overall, the container.php file is a crucial component for developers who prioritize modular, testable, and maintainable code.

NoSQL Databases

NoSQL databases have become increasingly popular in recent years, particularly for web applications. One of the main reasons for this is scalability. NoSQL databases are designed to handle large amounts of data and traffic, which makes them ideal for websites that need to be able to handle a high volume of traffic.

In the case of this website, a NoSQL database can handle the large number of articles, comments, and user data that will be generated. Traditional SQL databases can struggle with this type of workload, particularly when it comes to scaling up to handle increased traffic.

Another advantage of NoSQL databases is their ease of use. Unlike traditional SQL databases, NoSQL databases do not require a fixed schema. This means that changes can be made to the database structure without the need to update the entire schema. This makes it easier to add new features or make changes to the website as needed.

In addition, NoSQL databases are designed to be highly available and fault-tolerant. This means that if one server fails, the database can continue to operate without interruption. This is particularly important for websites that need to be available 24/7.

Composer

Composer is a widely adopted tool in PHP development that offers effective dependency management. The tool's popularity stems from its capability to manage third-party dependencies through the vendor folder. The vendor folder contains all the necessary external libraries and packages required for a PHP application to function accurately. Developers can install, upgrade, and remove these dependencies with ease, thanks to Composer's efficient mechanism. This feature simplifies the process of managing and tracking the external libraries and packages in the application. Besides, Composer comes with an autoloader feature that facilitates the development process by automatically loading classes and files without needing explicit inclusion or require statements. Overall, Composer's ability to manage third-party dependencies and provide an efficient autoloading mechanism makes it a vital tool for modern PHP development and this project.

Database design and configuration

Below outlines how the database would theoretically be structured, along with a proposed pseudo code of how other classes could interact with the database through the interface without being tightly coupled to the code structure of any specific database designs.

Design of Firestore Database Structure

(Root Document)
> Articles (Collection)
    > Opinion (Document) (maybe can include data relating to opinion articles, e.g. Chief Editors, Number of articles, etc)
    > Articles (Collection)
        > Article 1 (Document)(Can include data such as title, content, author, publish date etc)
        > Article 2 (Document)
    > News (Document)
    > Reviews (Document)
> Podcasts (Collection)
    > Podcast1 (Document)
    > Podcast2 (Document)

> Podcast3 (Document)

Design of MongoDB Database Structure

> Articles (Database)
> > Articles (Collection)
> > > Article 1 (Document)(Can include data such as title, content, author, publish date etc)
> > > Article 2 (Document)
> > > Article 3(Document)

> Podcasts (Database)
> > Podcast1 (Collection)
> > > Audio 1(Document)
> > > Audio 2(Document)
> > Podcast2 (Collection)
> > Podcast3 (Collection)

Example of Pseudocode of a function

E.g. Get viewable article from opinions should be:

```
$db = new class associated with DatabaseInterface.php;
$db->getArticle(type = Opinions, viewable = true, articleID = 1256);
```

Instead of:

```
$db = new FirestoreClient();
$db->get(collection(articles).document(opinions).collection(viewable).document(162356));
```

Design of Database Structure

Using NoSQL Database Structure as an Example.

- Collections is similar to an array of objects (Documents)
- Documents is similar to an object file which stores variables(Such as string, int, dates, etc
- Collections links to multiple child Documents
- In Firestore:
  - Documents can link to multiple child Collections

Database could implement the use of **Roles**.

- Site privileges and functionalities could be restricted by the account type that is currently logged in.
- E.g.
    - Roles such as **Admins**, who would have the ability to access everything. They would also be expected to have privileges that allow them to conduct administration related tasks such as approving or rejecting articles or other content that is uploaded to the site.
    - Other roles such as **Editors**, would be able to upload and even edit their own articles.
    - Whereas the regular **Users** would be only able to leave comments on these articles.

Code Justification and Structure

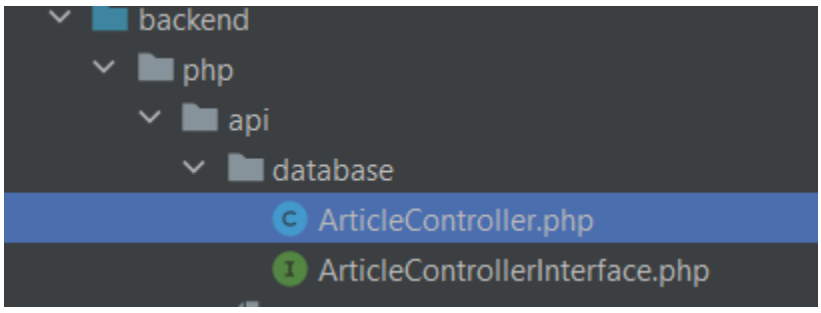
Backend folder

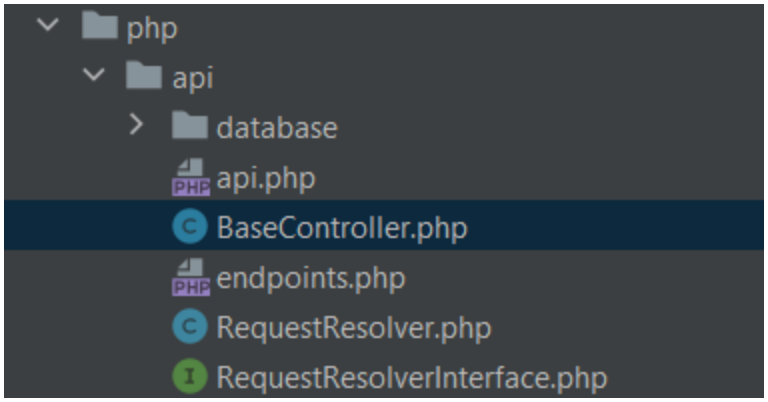
All the folders below (except the test folder) are found under the Backend folder,

api Folder

This Section will cover all the files and folders under the api folder.

api/database folder





api.php

All REST API requests for uri of /api/* is rerouted here and the requests will be resolved based on endpoints.php. This rerouting is achieved by using the .htaccess file used by Apache servers. An example available in *dummy_index2.html.*

Things to take notes of , there are many services and 3rd party dependencies that we can install to handle. REST API functions seamlessly. Team member Beng Lei created this because he did not wish to walk the other team members through installing and understanding another 3rd party dependency.

BaseController.php

The Base Controller class is expected to be inherited by all future Controller classes, Controller classes are classes that will take the REST API request by the client and will resolve said request by calling the functions of the interface associated with the Controller class .

protected function response()

This function constructs the API response to send to the client, It first checks if the strErrorDesc variable is set,

- If not, we'll send out the response data.
- If it is, then we'll send out the error response

protected function setErrorDesc()

Controllers will use this function to set the strErrorDesc variable, but currently it just returns a simple string.

protected function sendOutput($data, $httpHeaders=array())

This function is used to send the API response. The response() will call this method when it wants to send the API response to the user.

endpoints.php

Returns an array linking endpoints to their appropriate function calls.

- Each endpoint will have REST API methods associated with it
- Each of the REST API methods will have a function associated with it.

Some of our functions that should've been linked to GET requests according to best practices are instead linked to POST requests, this is because our functions have the option to do lengthy requests which would clutter the url if we use GET, so our "GET" request is converted to a POST request instead. The controllers array will then list out all the available controllers.

RequestResolver.php

Acts as the resolver for API requests, this class is the middleman that will link all valid endpoints to all the different controller classes. Valid endpoints and methods will be directed to this class by api.php. This class will call the appropriate controller class to carry out the API request and functions that will be passed here can be referred to in endpoints.php.

Functions will have two parts separated by "/": the Class to be used, and the function of that class to be called.

e.g. "ArticleControllerInterface/getArticles"

Means that from the class ArticleControllerInterface, Call getArticles()

RequestResolverInterface.php

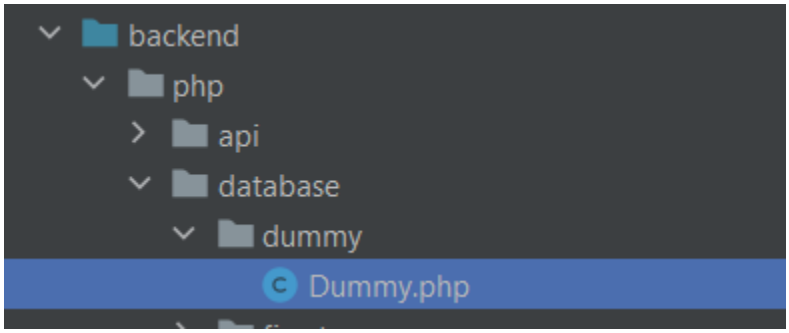
Has the **public function resolve(string $function);** function responsible for choosing the correct class.

ArticleControllerInterface.php

This is where all article database API functions should be implemented. The function is used to complete the REST API client for the Articles Database, It talks to the appropriate database class functions and sends the return to the client.

database folder

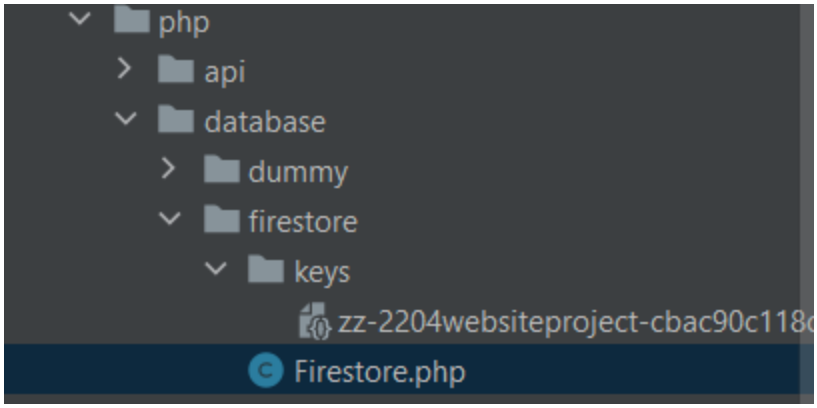
This section will cover all the files and folders under the database folder. This is not to be confused with the database folder mentioned previously under the api folder.



Dummy.php

The code creates a class named "Dummy" in the "backend\php\database\dummy" namespace, which is designed to implement the "DatabaseInterface" interface. This interface includes several methods that enable the retrieval, addition, updating, and deletion of articles. However, the Dummy class is only intended to provide temporary, non-functional implementations of these methods to facilitate testing and demo.

The main objective of this code is to offer a testing implementation of the "DatabaseInterface" interface, to be used until a fully functional implementation can be developed and integrated with an actual database.



Firestore.php

Here is the Firestore implementation of the DatabaseInterface.

```
public function getArticles(string $json): string
{
    $jsonArr = json_decode($json, associative: true);

    $ref = $this->firestoreClient->collection( name: ARTICLE_RO
```

This code defines a public function called *"getArticles*" that takes a JSON string as input and returns a JSON string as output. The function does the following:

1. Decodes the input JSON string into an associative array called $jsonArr using the json_decode() function.
2. Creates a reference to a Firestore collection based on the "from" field of the $jsonArr associative array.
3. Sets the number of articles to be returned based on the "noOfArticles'' field of the $jsonArr associative array. If this field is not present, it defaults to 5.
4. Sets the field to sort the articles based on the "sortBy '' field of the $jsonArr associative array. If this field is not present, it defaults to "title".

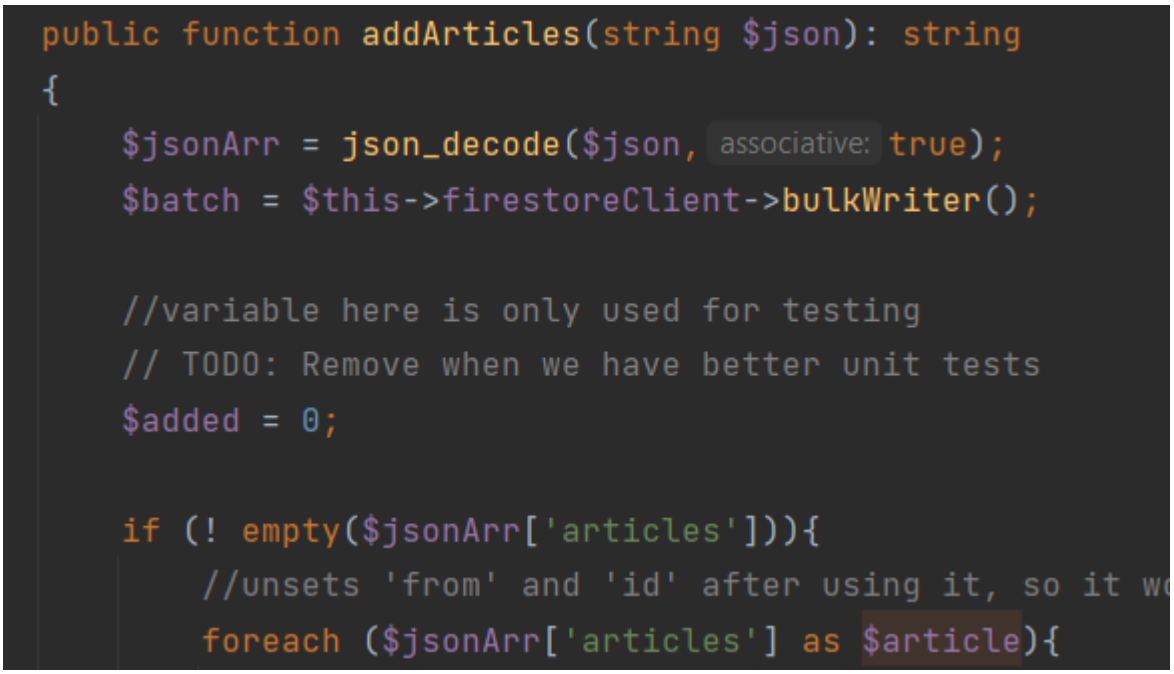
5. Sets the order in which to sort the articles based on the "order" field of the $jsonArr associative array. If this field is not present or its value is not "descending", the articles are sorted in ascending order.
6. Creates a Firestore query that orders the articles based on the specified sort field and order and limits the result set to the specified number of articles.
7. Executes the query and retrieves a list of documents that match the query criteria.
8. Loops through the documents, creates an array of articles, and adds each article to the array using its ID as the key. If a document does not exist, it prints an error message.
9. Returns a JSON string that contains an array of articles with their respective IDs as keys, wrapped in an outer "articles" object

```
3 usages   BengGit
public function getArticlesByID(string $json): string
{
    $jsonArr = json_decode($json, associative: true);
    $articles = array();

    foreach ($jsonArr['articles'] as $request){
        $docRef = $this->firestoreClient->collection( name: ARTICLE_ROOT."/".$request['from']."/".ARTICLE_SUB_C
        $snapshot = $docRef-> snapshot();

        if ($snapshot->exists()){
            $article = array();
            $article['id'] = $snapshot->id();
            $article = array_merge($article,$snapshot->data());
            $articles[$article['id']] = $article;
        }
    }
```

This code defines a public function called "getArticlesByID" that takes a JSON string as input and returns a JSON string as output. The function does the following:

1. Decodes the input JSON string into an associative array called $jsonArr using the json_decode() function.
2. Creates an empty array called $articles.
3. Loops through the "articles' ' field of the $jsonArr associative array, which is an array of article IDs and their corresponding "from" fields.
4. For each article in the loop, creates a reference to the document in the Firestore collection that corresponds to the article's "from" field and ID.
5. Retrieves a snapshot of the document using the snapshot() function.
6. If the snapshot indicates that the document exists, it creates an array called $article.

7. Adds the ID of the document to the $article array.
8. Merges the data from the snapshot into the $article array.
9. Adds the $article array to the $articles array, with the article ID as the key.
10. Returns a JSON string that contains an array of articles with their respective IDs as keys, wrapped in an outer "articles" object. If an article is not found, it will not be included in the output JSON string.



This code defines a public function called "addArticles" that takes a JSON string as input and returns a JSON string as output. The function does the following: Decodes the input JSON string into an associative array called $jsonArr using the json_decode() function. Creates a Firestore bulk writer object. Initializes a variable called $added to zero for testing purposes. If the input JSON array has articles, loops through each article in the array and performs the following actions:

- Creates a reference to a Firestore collection based on the "from" field of the article.
- Removes the "from" and "id" fields from the article array so that they are not included in the new document.
- If the article has no "id" field, create a new Firestore document reference using the collection reference.
- If the article has an "id" field, retrieve the Firestore document reference using the collection reference and the "id" field.

- Sets the Firestore document reference to the new article data using the bulk writer's set() method.
- Increments the $added variable by 1. Commits the batch writer to Firestore. Returns a JSON string with a message indicating the number of articles added to Firestore.

```
public function updateArticles(string $json): string
{
    $jsonArr = json_decode($json, associative: true);
    $batch = $this->firestoreClient->bulkWriter();

    //variable here is only used for testing
    // TODO: Remove when we have better unit tests
```

This code defines a public function called "updateArticles" that takes a JSON string as input and returns a JSON string as output. The function does the following:

1. Decodes the input JSON string into an associative array called $jsonArr using the json_decode() function.
2. Creates a BulkWriter instance for batch updates.
3. Initializes a variable called $updated to zero. This variable is only used for testing and will be removed later.
4. Checks if the $jsonArr associative array has an "articles' ' key with non-empty value. If yes, it loops through each article in the array.
5. Creates a Firestore reference to the article document based on the "from" and "id" fields of the article.
6. Unsets the "from" and "id" fields from the article array since they won't be updated.
7. Loops through each field of the article and creates an array of update operations. Each operation consists of a field path and a new value.
8. Executes the update operation using the update() method of the BulkWriter instance.
9. Increments the $updated variable.
10. Commits the batch update using the commit() method of the BulkWriter instance.
11. Returns a JSON string that contains a message with the number of articles updated.

```php
    public function deleteArticles(string $json): string
    {
        $jsonArr = json_decode($json, associative: true);
        $batch = $this->firestoreClient->bulkWriter();

        //variable here is only used for testing
        // TODO: Remove when we have better unit tests
        $deleted = 0;

        if (! empty($jsonArr['articles'])){
            foreach ($jsonArr['articles'] as $article){
                $ref = $this->firestoreClient
                    ->collection( name: ARTICLE_ROOT."/".$article['from']."/".ARTICLE_SUB_COLLECTION)
                    ->document($article['id']);

                $batch->delete($ref);
                $deleted = $deleted + 1;
            }
            $batch->commit();
        }
        return json_encode(array("deleted " . $deleted . " article(s)"));
    }
}
```

This is a public function called "deleteArticles" that takes a JSON string as input and returns a JSON string as output. The function does the following:

1. Decodes the input JSON string into an associative array called $jsonArr using the json_decode() function.
2. Creates a reference to a Firestore collection based on the "from" field of each article in the $jsonArr associative array.
3. Deletes the document with the ID specified in the "id" field of each article in the $jsonArr associative array.
4. Uses a bulkWriter to perform the delete operations in batches for better performance.
5. Returns a JSON string that contains a message indicating the number of articles that were deleted.

MongoDBHandler.php

Here is the MongoDB implementation of the DatabaseInterface.

```php
3 usages   SS4301 Backend Team
public function addArticles(string $json): string
{
    // Decode the JSON input
    $articles = json_decode($json,  associative: true);

    // Initialize the count of added articles
    $count = 0;

    // Loop through each article and add it to the database
    foreach ($articles['articles'] as $article) {
        $collection = $this->mongoClient->selectCollection($this->databaseName, $article['from']);
        unset($article['from']);

        if(array_key_exists( key: 'id', $article)){
            $article['_id'] = $article['id'];
            unset($article['id']);
        }

        $result = $collection->insertOne($article);
        if ($result->getInsertedCount() > 0) {
            $count++;
        }
    }

    // Return the result as a JSON string
    return json_encode(['message' => "added $count article(s)"]);
}
```

This PHP function accepts a JSON string as an argument, decodes it, and adds the articles contained in it to a MongoDB database. The function does the following:

1. The json_decode() function is used to decode the JSON string passed as an argument to the function. The second parameter true is passed to indicate that the result should be returned as an associative array.
2. A variable $count is initialized to keep track of the number of articles added to the database.
3. The function then loops through each article in the $articles array and adds it to the MongoDB database.
4. Inside the loop, the selectCollection() method of the MongoDB\Client class is used to select the appropriate collection in the database based on the from field of the article.
5. The unset() function is used to remove the from field from the $article array.

6. If the id field is present in the $article array, its value is copied to the _id field and the id field is removed from the array.
7. The insertOne() method is used to insert the $article array into the selected collection. The result of the insertion is stored in the $result variable.
8. If the insertion was successful (i.e., the number of inserted documents is greater than 0), the $count variable is incremented.
9. After all articles have been added to the database, a JSON string is returned containing a message indicating the number of articles added.

```
3 usages   SS4301 Backend Team
public function updateArticles(string $json): string
{
    $articles = json_decode($json,  associative: true)['articles'];
    $updatedCount = 0;

    if(!empty($articles)){
        foreach ($articles as $article) {
            $collection = $this->mongoClient->selectCollection($this->databaseName, $article['from']);
            $id = $article['id'];
            unset($article['from']);
            unset($article['id']);

            $updates = array();
            foreach ($article as $path => $value){
                $updates[$path] = $value;
            }

            $collection->updateOne(
                ['_id' => $id],
                ['$set' => $updates],
                ["upsert" => true]
            );

            $updatedCount = $updatedCount + 1;
        }
    }

    return json_encode(array("result message" => "updated " . $updatedCount . " article(s)"));
}
```

This is a PHP function that updates articles in a MongoDB database based on a JSON string that contains the updates. The function does the following:

1. The `json_decode()` function is used to decode the JSON string passed as an argument to the function. The `true` argument is passed to indicate that the result should be returned as an associative array, and the `['articles']` index is used to extract the array of articles to be updated.
2. A variable `$updatedCount` is initialized to keep track of the number of articles updated in the database.
3. If the array of articles is not empty, the function enters a loop to update each article.
4. Inside the loop, the `selectCollection()` method of the `MongoDB\Client` class is used to select the appropriate collection in the database based on the `from` field of the article.
5. The `id` field is extracted from the `$article` array and stored in the `$id` variable. The `from` and `id` fields are then removed from the `$article` array using the `unset()` function.
6. A new array `$updates` is initialized to hold the update operations for the article.
7. A loop is used to iterate through the remaining fields in the `$article` array, and each field's path and value are added to the `$updates` array.
8. The `updateOne()` method is used to update the article in the collection. The first argument to `updateOne()` is a filter that matches the document to be updated, based on its `_id` field. The second argument is an update document that uses the `$set` operator to specify the fields to be updated and their new values. The `upsert` option is also set to `true` to insert the document if it doesn't exist.
9. If the update was successful, the `$updatedCount` variable is incremented.
10. After all articles have been updated, a JSON string is returned containing a message indicating the number of articles that were updated.

```php
3 usages   SS4301 Backend Team
public function deleteArticles($json): string
{
    $data = json_decode($json,  associative: true);

    $deletedCount = 0;

    if (! empty($data['articles'])){
        foreach ($data['articles'] as $article) {
            $collection = $this->mongoClient->selectCollection($this->databaseName, $article['from']);
            $collection->deleteOne(['_id' => $article['id']]);
            $deletedCount = $deletedCount+1;
        }
    }

    return json_encode(array("result message" => "deleted " . $deletedCount . " article(s)"));

}
```

This is a PHP function that deletes articles from a MongoDB database based on a JSON string that contains the list of articles to be deleted. The function does the following:

1. The `json_decode()` function is used to decode the JSON string passed as an argument to the function. The `true` argument is passed to indicate that the result should be returned as an associative array, and the `$data` variable is used to store the decoded array.
2. A variable `$deletedCount` is initialized to keep track of the number of articles deleted from the database.
3. If the `$data` array contains a non-empty `articles` field, the function enters a loop to delete each article.
4. Inside the loop, the `selectCollection()` method of the `MongoDB\Client` class is used to select the appropriate collection in the database based on the `from` field of the article.
5. The `_id` field of the article to be deleted is extracted from the `$article` array, and the `deleteOne()` method is used to delete the article from the collection.
6. If the deletion was successful, the `$deletedCount` variable is incremented.
7. After all articles have been deleted, a JSON string is returned containing a message indicating the number of articles that were deleted.

```php
public function getArticles(string $json): string
{
    // Parse the JSON request
    $requestData = json_decode($json, associative: true);

    // Get the "from" field from the request
    $from = $requestData["from"];

    // Get the "noOfArticles" field from the request
    $noOfArticles = isset($requestData["noOfArticles"]) ? $requestData["noOfArticles"] : 5;

    // Get the "sortBy" field from the request
    $sortBy = isset($requestData["sortBy"]) ? $requestData["sortBy"] : "title";

    // Get the "order" field from the request
    $order = isset($requestData["order"]) ? $requestData["order"] : "ascending";

    // Check if the number of articles is greater than 0
    if ($noOfArticles <= 0) {
        // Return an empty array
        return json_encode(array("articles" => array()));
    }

    // Query the database to get the article
    $collection = $this->mongoClient->selectCollection($this->databaseName, $from);
    $query = $collection->find([], [
        'limit' => $noOfArticles,
        'sort' => [$sortBy => ($order === 'ascending' ? 1 : -1)]
    ]);

    $query = json_encode(iterator_to_array($query));
    $queryArr = json_decode($query, associative: true);
    // Create an array to store the results
    $results = array();

    // Loop through the articles and add them to the result array
    foreach ($queryArr as $article) {
        $result = array(
            "id" => $article["_id"],
        );
        unset($article['_id']);
        $result = array_merge($result, $article);
        $results[$result['id']] = $result;
    }
```

This is a PHP function that retrieves articles from a MongoDB database based on the provided parameters. The function does the following:

1. It first decodes the JSON input to get the request data.
2. It retrieves the "from" field from the request, which is the name of the collection to query.
3. It retrieves the "noOfArticles" field from the request, which specifies the maximum number of articles to retrieve. If this value is less than or equal to 0, an empty array is returned.
4. It retrieves the "sortBy" and "order" fields from the request, which specify the field to sort by and the order of the sorting (ascending or descending), respectively. If these fields are not provided in the request, "title" and "ascending" are used as the default values, respectively.
5. It queries the database using the provided collection name, "noOfArticles", "sortBy", and "order" to retrieve the specified number of articles sorted by the specified field in the specified order.
6. It loops through the retrieved articles, formats them as an array, and adds them to the result array.
7. It encodes the result array as JSON and returns it.

```php
3 usages   SS4301 Backend Team
public function getArticlesByID(string $json): string
{
    $requests = json_decode($json,  associative: true);
    $articles = array();


    foreach ($requests['articles'] as $request){
        $collection = $this->mongoClient->selectCollection($this->databaseName, $request['from']);
        $query = $collection->findOne(['_id' => $request['id']]);

        //Change to array
        $query = json_encode(iterator_to_array($query));
        $query = json_decode($query, associative: true);

        $article = array();
        $article['id'] = $query['_id'];
        unset($query['_id']);
        $article = array_merge($article,$query);
        $articles[$article['id']] = $article;
    }

    return json_encode(array("articles" => $articles));

}
```

This function takes a JSON string as input and retrieves articles from a MongoDB database by their IDs. It returns a JSON string containing the retrieved articles. The function does the following:

1. The function takes a JSON string as input.
2. The JSON string is decoded into an associative array using `json_decode`.
3. An empty array is initialized to store the retrieved articles.
4. A `foreach` loop is used to iterate over each article ID in the request.
5. For each article ID, the corresponding article is retrieved from the MongoDB database using the `findOne` method of the `MongoCollection` object.
6. The retrieved article is converted from a `MongoDB\Model\BSONDocument` object to an associative array using `json_encode` and `json_decode`.
7. The `_id` field of the article is copied to a new `id` field for compatibility with the output format of other functions in this class.
8. The article is added to the array of retrieved articles.
9. The array of retrieved articles is encoded as a JSON string and returned. The JSON string is wrapped in an associative array with a single key, `articles`.

DatabaseInterface.php

The database class will accept json strings and return the results in json.

Here is where the functions that would be implemented into the database specific PHP files are specified in.

public function getArticles(string $json):

Would return articles from a section based on a given query.

Query fields:

- from: the Article Section requested
- (optional) noOfArticles: the number of articles to be returned by default: *5*
- (optional) sortBy: the field used to sort the articles by default: "*title*"
- (optional) order: the order in which the articles are sorted by default: *"ascending*"

Returns:

* Json array of Article arrays,

* Each article array will have a key associated with it based on the article's ID

* Article Array will contain the Article ID as its first field,

* subsequent fields will depend on the fields that exists in that article

```
* e.g.
* Request JSON:
*  {
*      from: "Crimereads",
*      noOfArticles: 2,
*      sortBy: "publish_date",
*      order: "ascending"
*  }
```

An example

```
*
* Return JSON:
*  {
*      articles: {
*          YhC9FJUY03km13UWybCJ:{
*              id: "YhC9FJUY03km13UWybCJ",
*              title: "Crimereads Title 1"
*              img_url: "https://pbs.twimg.com/media/DXtHp7zXcAIlO_n?format=jpg&name=4096x4096",
*              author: "Beng",
*              publish_date: "2023-04-07T09:58:23.687000Z",
*              content: "Crimereads"
*          },
*          vG7GatbnFqHds1SiTtnB:{
*              id:  "vG7GatbnFqHds1SiTtnB",
*              title:  "Crimereads Title 2",
*              img_url:  "https://pbs.twimg.com/media/DXtHp7zXcAIlO_n?format=jpg&name=4096x4096",
*              author:  "Beng",
*              publish_date:  "2023-04-07T09:59:18.789000Z",
*              content:  "Crimereads"
*          }
*      }
*  }
*
```

This would be the returned JSON array

public function getArticlesByID(string $json):

Returns articles based on article IDs given

Query Fields:

- from: The article section of the article
- id: The ID of the article

Returns Json array of Article arrays,

- Each article array will have a key associated with it based on the article's ID
- Article Array will contain the Article ID as its first field, subsequent fields will depend on the fields that exists in that article

```
* e.g.
* Request JSON:
*  {
*      articles: {
*          {
*              from: "Crimereads",
*              id: "vG7GatbnFqHds1SiTtnB"
*          },
*          {
*              from: "Fiction and Poetry",
*              id: "ex7UanwL6Pf5dWUKTw90"
*          }
*      }
*  }
*
* Return JSON:
*  {
*      articles: {
*          vG7GatbnFqHds1SiTtnB:{
*              id:  "vG7GatbnFqHds1SiTtnB",
*              title:  "Crimereads Title 2",
*              img_url:  "https://pbs.twimg.com/media/DXtHp7zXcAIlO_n?format=jpg&name=4096x4096",
*              author:  "Beng",
*              publish_date:  "2023-04-07T09:59:18.789000Z",
*              content:  "Crimereads"
*          },
*          ex7UanwL6Pf5dWUKTw90:{
```

An Example

public function addArticles(string $json):

Add articles based on JSON given.

Add Articles JSON:

from: The article section that the article will be added to

- (optional) id: The id of the new article
    - (this is not recommended because it could generate user errors if id match a pre-existing id) by default: *random id generated by the database*
- (optional others): other fields that the article will contain

Returns Json array with a simple array stating the number of articles added

- Future plans should rework or remove the return all together

```
* e.g.
* Add Articles JSON:
*  {
*      articles: {
*          {
*              from: "Crimereads",
*              title: "New Crimereads",
*              content: "Newly added article"
*          },
*          {
*              from: "Fiction and Poetry",
*              title: "New Fiction and Poetry",
*              content: "Newly added article"
*          },
*          {
*              from: "Fiction and Poetry",
*              id: "FictionAndPoetryArticle2"
*              title: "New Fiction and Poetry 2",
*              content: "Newly added article"
*              author: "Beng"
*          }
*      }
*  }
*
* Return JSON:
*  {
*      result message: "added 3 articles"
*  }
```

An Example

public function moveArticles(string $json):

Moving articles around Firestore collections is not natively supported, Firestore has no way to easily move articles. From some research, we should probably try to work around this rather than forcefully create a function to do this.

A possible way to implement this would be to:

1. Copy the contents of the whole document,
2. Create a new document at intended location with the content of the previous document,
3. Delete the old document

It is important to take note that: metadata and etc would probably be lost in the process

public function updateArticles(string $json):

Update articles based on JSON given.

Update Articles JSON:

- from: The article section that the article is located in
- id: The ID of the article to be updated
- (optional others): other fields that will be added/changed in the article

Return Json array with a simple array stating the number of articles updated

- Future plans should rework or remove the return all together

```
*
* e.g.
* Updated Articles JSON:
*  {
*      articles: {
*          {
*              from: "Crimereads",
*              id: "vG7GatbnFqHds1SiTtnB",
*              content: "Updated content"
*          },
*          {
*              from: "Fiction and Poetry",
*              id: "ex7UanwL6Pf5dWUKTw90",
*              title: "Updated title",
*              note: "newly added 'note' field"
*          }
*      }
*  }
*
* Return JSON:
*  {
*      result message: "updated 3 articles"
*  }
*
```

An Example

public function deleteArticles(string $json):

Deletes articles based on JSON given

Delete Articles JSON:

- from: The article section that the article is located in
- id: The ID of the article to be deleted

*

Return Json array with a simple array stating the number of articles deleted

- Future plans should rework or remove the return all together

```
*
* e.g.
* Updated Articles JSON:
*  {
*      articles: {
*          {
*              from: "Crimereads",
*              id: "vG7GatbnFqHds1SiTtnB",
*          },
*          {
*              from: "Fiction and Poetry",
*              id: "ex7UanwL6Pf5dWUKTw90",
*          }
*      }
*  }
*
* Return JSON:
*  {
*      result message: "deleted 2 articles"
*  }
*
```

An Example

Util folder

This section covers all the files under the util folder.

Bootstrap.php

The handling of the secret key in this file is probably very insecure, as the secret key should be a secret, the key could be potentially linked to payment and admin access to the database, security of this key should be handled more seriously before putting our website out to the public. Using containers and microservices is another way of passing the (secret) key securely.

```php
if(!defined( constant_name: 'FIRESTORE_PROJECT_ID')){
    define("FIRESTORE_PROJECT_ID", "zz-2204websiteproject");
}
```

Firestore database project ID

```php
require_once PROJECT_ROOT_PATH . '/vendor/autoload.php';
```

Composer autoloader

```php
$config = require PROJECT_ROOT_PATH . '/backend/php/util/config.php';
```

Configuration array, currently only used by the dependency injection contianer, Container.php

```php
$container = Container::getInstance();
foreach ($config['interfaces'] as $interface => $concrete){
    $container->bind($interface, $concrete);
}
```

Binding all the interfaces to the classes based on $config

Config.php

Contains information linking interface files to their corresponding classes. This linking can also include information on how the class instance is to be constructed as seen by the Firestore linking.

This file also attempts to construct the FirestoreClient and connect to the Firestore Database.

```php
while(!$firestoreClient instanceof FirestoreClient){
    try {
        $firestoreClient = new FirestoreClient([
            "keyFilePath" => FIRESTORE_KEY,
            "projectId" => FIRESTORE_PROJECT_ID
        ]);
    } catch (\Google\Cloud\Core\Exception\GoogleException $e) {}
```

Loops until we successfully get a FirestoreClient instance

```php
if(!$firestoreClient instanceof FirestoreClient){
    sleep( seconds: 1);
```

Wait 1 sec and try again if it fails

Container.php

Dependency Injection Container

It binds the abstract and concrete, typically Interfaces(abstract) to their respective Classes(concrete), the abstract can be anything: *classes, interfaces, string, int.* Concrete has to be something that can be instantiated (functions or classes).

Creates Class(concrete) instances by resolving the abstract given, has the ability to pass parameters and instances needed by the instance being constructed.

This class is fully based on this YouTube video (https://www.youtube.com/watch?v=HOVWXa7HBZY)

```php
protected function __construct(){
```

Constructor is protected so that other files cannot create multiple instances of this class

```php
public function bind($abstract, $concrete): void
{
    $this->bindings[ $abstract ] = $concrete;
}
```

Binds the abstracts with their respective concretes.

Bindings are saved to an array with the $abstract as the key and the $concrete as the value.

```php
public function resolve($abstract): mixed
{
    if(!isset($this->bindings[$abstract])){
        $this->ensureClassIsInstantiable($abstract);
    }

    $concrete = $this->bindings[ $abstract ] ?? $abstract;

    if(is_callable($concrete)){
        return call_user_func($concrete);
    }

    return $this->buildInstance($concrete);
}
```

When given an $abstract,

- if $abstract is not bound,
    - Checks if the $abstract itself can be instantiated.
    - else error
- $concrete will be the bound instantiable,or the instantiable $abstract itself
    - if $concrete is a function, calls the function

- else build the class instance using buildInstance()

```php
protected function ensureClassIsInstantiable($class){
    $reflection = new ReflectionClass($class);

    if(!$reflection->isInstantiable()){
        throw new Exception( message: "{$class} is not instantiable");
    }
}
```

Uses the ReflectionClass to check if $class can be instantiated. Throws an error otherwise.

```php
3 usages   BengGit
protected function buildInstance($class){
    $reflection = new ReflectionClass($class);
```

Builds the instance

Checks if this class is bound to a function, return and call the function

- Else, if there's no constructor for this class, return a new instance of this class
- Else, if this class is bound to another class, return and build an instance of the new class
- Else, if there's no constructor and is not bound to something else and is still not instantiated, then throw exception
- Else, the instance must have a constructor.

Create instances of all the required parameters

- If the parameters for the constructor is not optional and cannot be instantiated, the return error
- Return the new class instance with the instantiated parameters in the constructor

Notice how we can chain the container bindings, I have some examples in *backend_dummy_index.php*

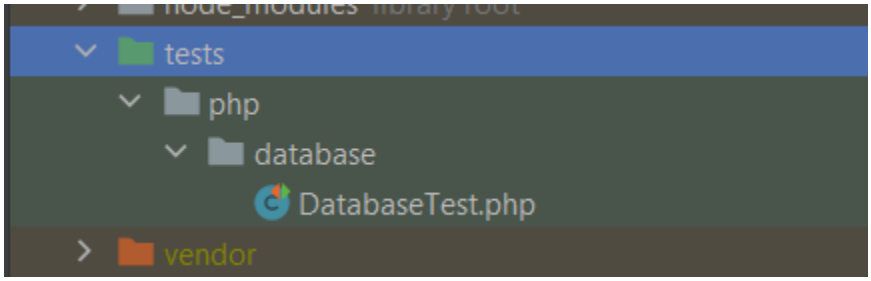
```php
public static function getInstance(){
    if(is_null(self::$instance)){
        self::$instance = new static();
    }

    return self::$instance;
}
```

Checks if an instance of this Class exists, create a new instance if not .

Then returns an instance of this class

Tests folder

This section covers all the files under the tests folder.



Currently, it contains only the*DatabaseTest.php,* this is where the test cases for the database functions are found, testing for functions that add,delete,update and get articles from the database. The tests done here are rudimentary and more robust tests could be created in the future

Conclusion

Lessons Learned

Using Methods learned and covered in the coursera program, such as the courses specifically regarding Test Driven development, DevOps and Containers.

- Test-driven development taught us the benefits and methods of writing automated tests before writing any of the actual production code, enabling us as developers to catch errors early in the development process while also understanding what we were building and if it was being built correctly. As a guideline to create code that passes the test cases created beforehand.
- DevOps taught us a variety of important concepts, the most applied concept being pair programming. Where we would take turns writing then switching to observing code, allow us to understand and look out for potential errors before they were integrated or pushed.
- Introduction to Containers taught us the consistency, portability, and scalability that it brings as an advantage. Most importantly, the scalability that it enabled for our project.

FUTURE IMPLEMENTATION

Implement account creation and login using Firebase Authentication.

Summary

The project at this current state is able to add,create, delete and update articles from the database (both mongoDB and Firestore) through REST API and the dependency injection container. Though clearly not a perfect implementation and with serious security risks, this project still provides a resourceful insight into software development. Hopefully future students enrolled in this module will be able to not only resolve all our issues but be able to fully finalize this project.