Magpie CMS API Developer Documentation

PHP Version (Continued from W17 team)

Date Last Modified: 5. Jun. 2018

API code: https://github.com/rkwitz/Magpie-CMS-API
Frontend code: https://github.com/EthanTuning/Magpie-CMS-Website

API user documentation: https://documenter.getpostman.com/view/4418001/RW87rVjU

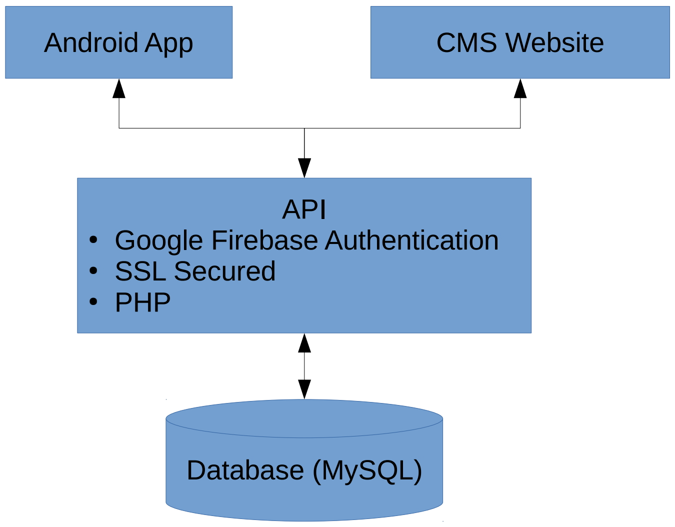
Table of Contents

Introduction

This is documentation for developers working on the Magpie API codebase. For the API user documentation, see the link at the bottom of the document.

Magpie is a geocaching application for phones. For information about geocaching, see https://en.wikipedia.org/wiki/Geocaching. The web-based Application Programming Interface (API) is used to serve both the phone application (currently for Android systems) and the Content Management System (CMS). The CMS was created to allow users to create and manage their data.

The API attempts to adhere to an architectural style known as “Representational State Transfer.” This style applies object-orientated paradigms to web-enabled resources. Traditional web APIs emulated Remote Procedure Calls (RPC). The RESTful API aims for more flexibility and does not distinguish between client types. The Android application and browser-based CMS utilize the same API endpoints. A list of resources on the topic of RESTful API design can be found at the end of this document.



Frameworks and Tools:

- PHP v7
  - Composer
    - to manage dependencies and 3rd party PHP libraries
  - Slim v3
    - A very useful mini-framework for mapping URL endpoints to Controllers written in php
    - Also, Slim Middleware is used to ensure authentication when users connect to the API
  - Kreait’s firebase-php library
    - Google does not provide php libraries for their Firebase project (probably to force everyone to use Go or Python), this 3rd party library used instead.
  - PhpMailer

        - Used to send emails to users. Note: Currently disabled, however it did work, just not reliably enough to push live.
    - phpMyAdmin
        - To manage the MySQL easier
- Google Firebase
    - Google OAuth2 tokens provided by Firebase Authentication
- MySQL
    - Opensource relational database used for data storage
    - MariaDB could probably be used with zero configuration changes
- SSL/TLS
    - Since the API uses a Bearer token to authenticate, SSL is an absolute must to prevent session hijacking and credential leakage

Constraints:

- API and CMS website must deploy to Bluehost shared hosting site already owned by client
- CMS website must allow a single user both admin and regular user functionality

Data Models

The API exists to store and retrieve data. The first step is defining what kinds of data we need to store and their relationship with each other. Each data structure is tied to it’s own table in the database. The data fields of an object are stored in the database. However, it’s representation is converted to JavaScript Object Notation (JSON) by the API before being returned to the client.

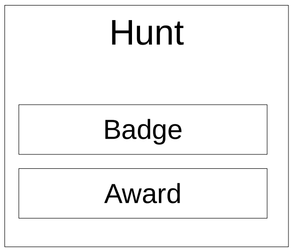


Illustration 1: Hunt Abstraction

Hunt

The core unit of this entire project is a *Hunt*. A Hunt is the parent object in Magpie. It contains all other objects, usually referred to as “sub-resources” or “children.” Currently there are only two types of sub-resources, *Badges* and *Awards*.

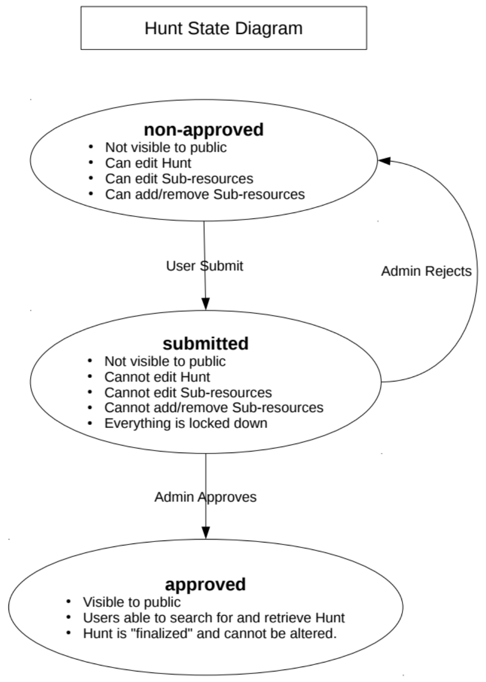
```
{
  "class": "hunt",
  "data": {
    "hunt_id": "671",
    "abbreviation": "string",
    "approval_status": "non-approved",
    "audience": "string",
    "date_end": "1902-01-10",
    "date_start": "2120-12-25",
    "name": "string",
    "ordered": "0",
    "summary": "string",
    "sponsor": "string",
    "super_badge": "string",
    "uid": "jaksEPf7HjPI2uRA1gnQ0yhYJyy1",
    "city": "City",
    "state": "WA",
    "zipcode": "99227"
  },
  "href": "https://magpiehunt.com/api/v1/hunts/671",
  "subresources": [
    {
      "class": "badges",
      "href": "https://magpiehunt.com/api/v1/hunts/671/badges",
      "type": "json"
    },
    {
      "class": "awards",
      "href": "https://magpiehunt.com/api/v1/hunts/671/awards",
      "type": "json"
    }
  ]
}
```

The “class” attribute contains the name of the object (in this case a Hunt). The “data” attribute contains the actual data about a Hunt. Name, location, dates, etc are in here. The “href” is a self-referencing link. The “subresources” is a list of sub-resources.

The ‘subresources’ list doesn’t contain a list of actual resource objects, it contains links to them. This is a “recommended” way to handle sub-resources according to the REST style, but there is no hard rule saying you have to do this. This did save on some complexity though, by enforcing modularity.

Hunt Status

To ensure that users don’t create Hunts that could lead people to their doom, all Hunts must be reviewed by an Administrator before they can be published for public consumption. There is a field in the Hunt called “approval_status” that keeps track of this. Below is a state diagram that covers the different states:



Badge

A Badge is a location in the real world. It contains the geographic coordinates of the location, as well as other data about the location.

```
{
  "class": "badge",
  "data": {
    "badge_id": "7",
    "description": "string",
    "icon": {
      "href": "https://magpiehunt.com/api/v1/uploads/f01d5bb02739777c.jpg"
    },
    "image": {
      "href": "http://www.customURL.com/image.jpg"
    },
    "landmark_name": "string",
    "lat": "4.54",
    "lon": "-30.43",
    "name": "string",
    "qr_code": "qr_code string",
    "hunt_id": "678"
  },
  "href": "https://magpiehunt.com/api/v1/hunts/678/badges/7"
}
```

This object has the same style as a Hunt. It has a "class", "data", and "href" as top-level attributes. There is no list of subresources, as the API currently can only handle one level of sub-resources. Notice how images contain an {"href" : URL} pair. This is to aid in parsing the Badge on the client side. While iterating through the attributes, if "attribute.href" exists, then it is a link.

Award

An Award is an achievement for completing a Hunt. It is structurally similar to a Badge, and on the backend, it is treated the same as a Badge. It has it's own table in the database.

```
{
  "class": "award",
  "data": {
    "award_id": "1",
    "address": "string",
    "description": "string",
    "lat": "45.435435435",
    "lon": "-39.54543",
    "name": "string",
    "redeem_code": "string",
    "award_value": "string",
    "hunt_id": "671"
  },
  "href": "https://magpiehunt.com/api/v1/hunts/671/award/1"
}
```

Notes

Images are currently referred to by URL in objects. When images are uploaded, a PHP controller saves the image to disk with a pseudo-random filename in the uploads/ directory. The saved filepath is converted to a URL and stored in the appropriate column in the database. This is not ideal.

The last team used a separate database table to store image data. This table could store the owner of the image, the name, the type, and any other data associated with it. The image would still be saved on the local filesystem, but the database would hold additional information. This is probably a better way to handle image files. We started getting crunched for time, hence our current approach of throwing all files into the uploads/ folder and having absolutely no storage structure.

API Overview

There’s three main parts to the API:

1. The Slim application
2. The Controllers
3. The Mapper class

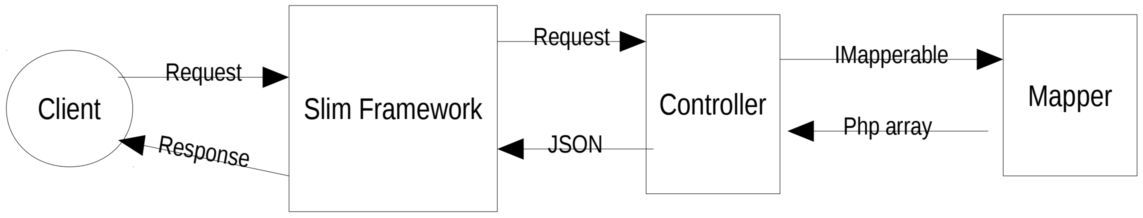
Routing

The Slim application is responsible for taking a HTTP request, routing it to the correct Controller, and returning a response. Added on to the Slim application are Middleware layers that handle Authenticating the client with Firebase, making sure the client exists in the database, and checking if the client is an administrator. There is also a middleware layer that handles CORS compatibility, which is a massive pain.

Once the request is routed to the appropriate Controller class, the controller creates an instance of a Mapper, and uses the Mapper to interface with the database. In some cases the controller performs SQL queries directly (The AdminController does this). The controller also has to send client input to the Mapper by using an object such as Hunt, Badge, or Award. These objects implement an IMapperable interface for use by the Mapper. They also only allow certain fields to be entered, and will ignore any data not allowed (see comments in source code).

Once the IMapperable object reaches the Mapper, a series of checks are performed to determine whether or not the object can be placed in the database, or if data can be returned. The Mapper then returns data in an associate array (or an array of associative arrays depending on the function performed).

This data is then JSON encoded by the Controller, and sent back to the client.



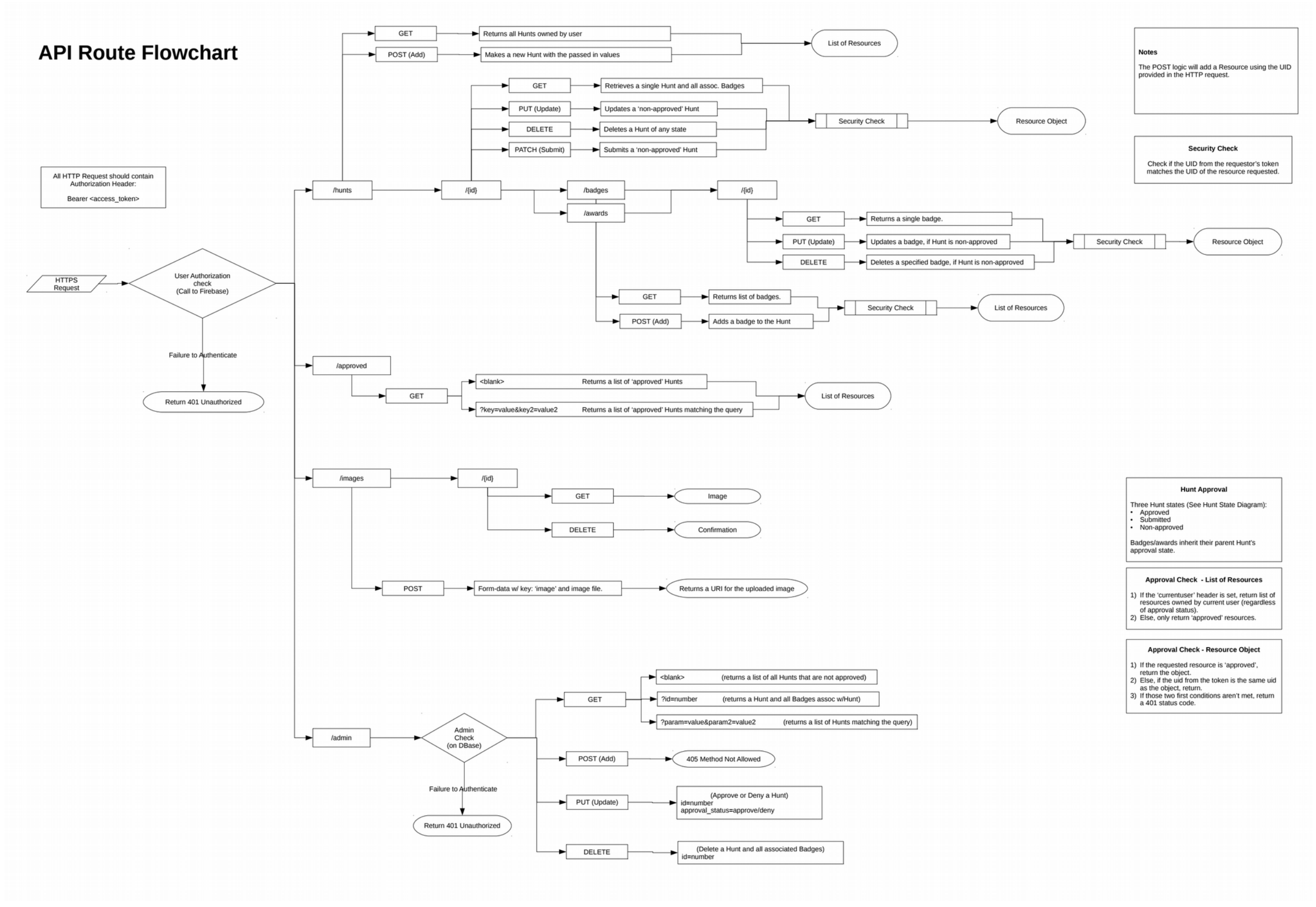
Endpoints

In general a URL specifies a resource, so “magpiehunt.com/api/v1/hunts” references the Hunts resource. The HTTP verbs like GET, POST, and DELETE are actions to perform on that resource. A POST sent to /hunts means that a client wants to create a Hunt.

If there’s a number following the resource, it’s referring to that specific resource, ex: “magpiehunt.com/api/v1/hunts/2322”. A GET send to that URL indicates someone wants to retrieve that resource.

Sub-resources are tacked on to their parent resource IDs. A badge for example is “magpiehunt.com/api/v1/hunts/2322/badges/3”.

This is a pdf file in the documentation/ folder. This flowchart depicts the endpoints of the API that a client would use. The ‘approved’ endpoint was never completed.

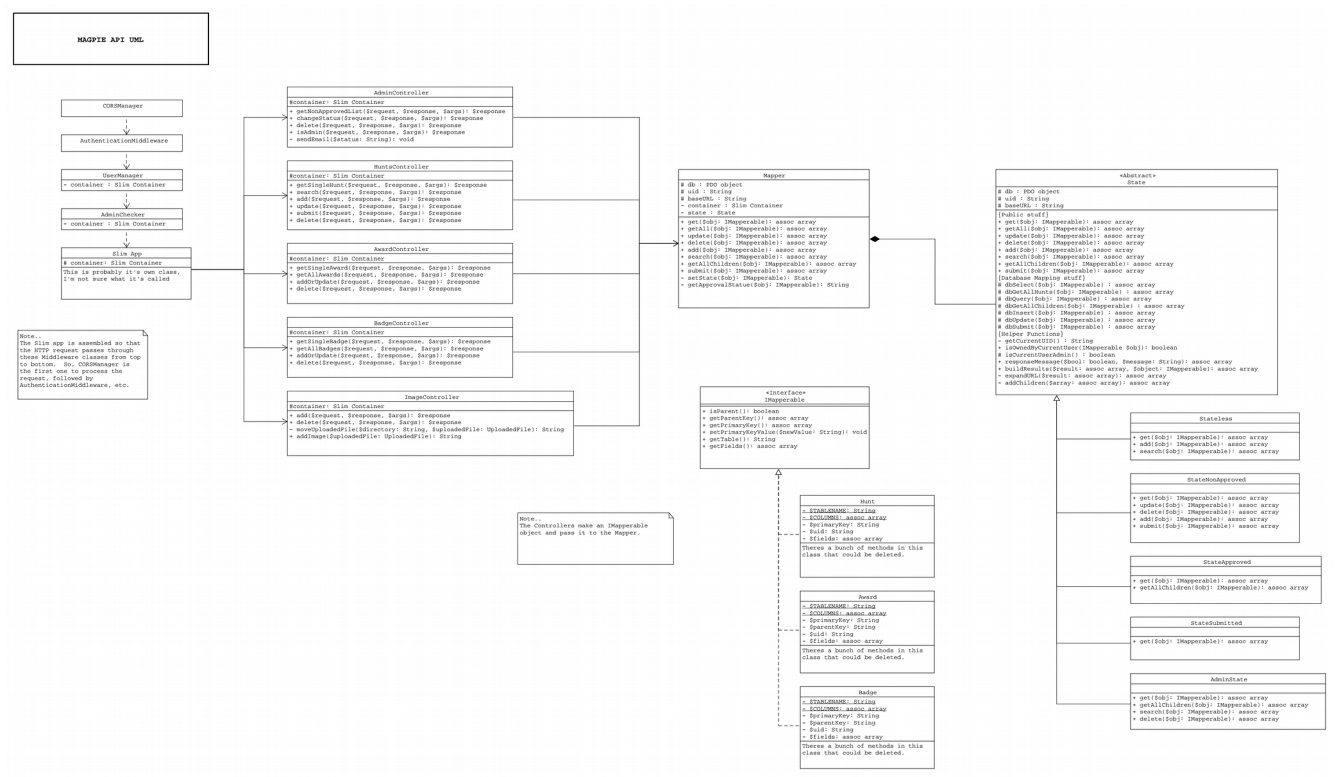

API Route Flowchart
All HTTP Request should contain Authorization Header:
Bearer <access_token>
HTTPS Request
User Authorization check (Call to Firebase)
Failure to Authenticate
Return 401 Unauthorized
/hunts
GET
Returns all Hunts owned by user
POST (Add)
Makes a new Hunt with the passed in values
List of Resources
/{id}
GET
Retrieves a single Hunt and all assoc. Badges
PUT (Update)
Updates a 'non-approved' Hunt
DELETE
Deletes a Hunt of any state
PATCH (Submit)
Submits a 'non-approved' Hunt
Security Check
Resource Object
/badges
/awards
/{id}
GET
Returns a single badge.
PUT (Update)
Updates a badge, if Hunt is non-approved
DELETE
Deletes a specified badge, if Hunt is non-approved
Security Check
Resource Object
GET
Returns list of badges.
POST (Add)
Adds a badge to the Hunt
Security Check
List of Resources
/approved
GET
<blank>
Returns a list of 'approved' Hunts
?key=value&key2=value2
Returns a list of 'approved' Hunts matching the query
List of Resources
/images
/{id}
GET
Image
DELETE
Confirmation
POST
Form-data w/ key: 'image' and image file.
Returns a URI for the uploaded image
/admin
Admin Check (on DBase)
Failure to Authenticate
Return 401 Unauthorized
GET
<blank> (returns a list of all Hunts that are not approved)
?id=number (returns a Hunt and all Badges assoc w/Hunt)
?param=value&param2=value2 (returns a list of Hunts matching the query)
POST (Add)
405 Method Not Allowed
PUT (Update)
(Approve or Deny a Hunt) id=number approval_status=approve/deny
DELETE
(Delete a Hunt and all associated Badges) id=number
Notes
The POST logic will add a Resource using the UID provided in the HTTP request.
Security Check
Check if the UID from the requestor's token matches the UID of the resource requested.
Hunt Approval
Three Hunt states (See Hunt State Diagram):
• Approved
• Submitted
• Non-approved
Badges/awards inherit their parent Hunt's approval state.
Approval Check - List of Resources
1) If the 'currentuser' header is set, return list of resources owned by current user (regardless of approval status).
2) Else, only return 'approved' resources.
Approval Check - Resource Object
1) If the requested resource is 'approved', return the object.
2) Else, if the uid from the token is the same uid as the object, return.
3) If those two first conditions aren't met, return a 401 status code.

UML Diagram

This is a pdf export of the “documentation/UML_overview.uxf”, which is editable by UMLet. UMLet is a really neat UML creator.

To save on clutter I didn’t mention the namespaces of the classes. Namespaces are the PHP equivalent of Java packages.

For more information on the classes, see the comments at the top of the PHP file for each class.

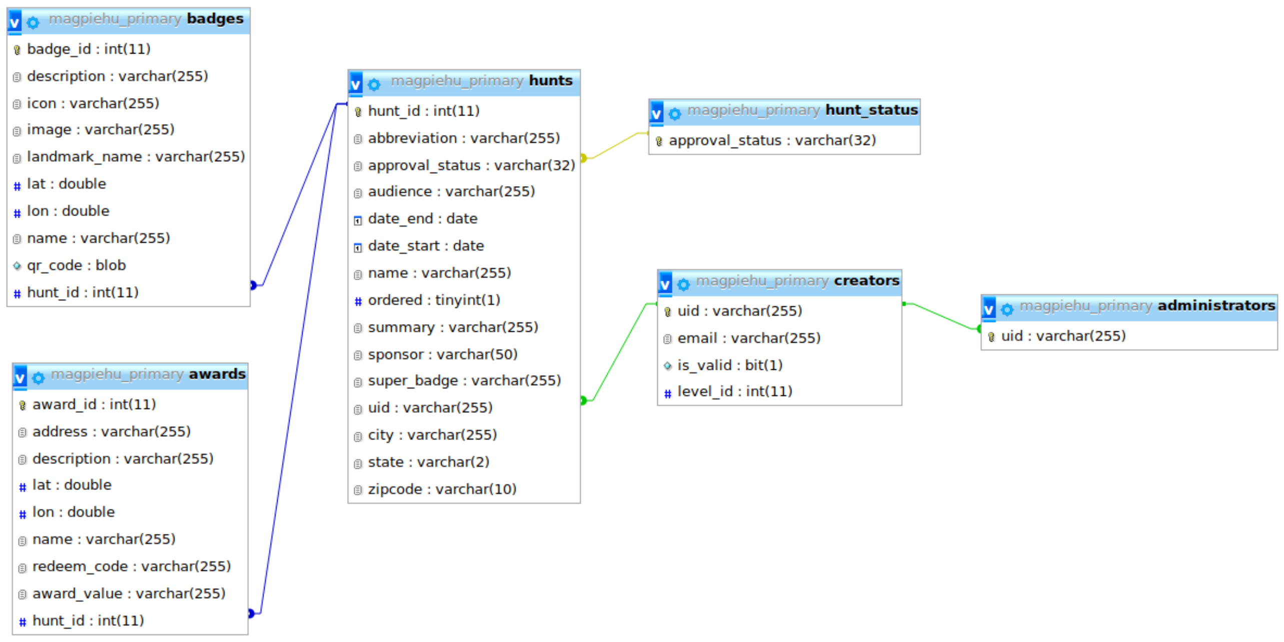

MAGPIE API UML
CORSManager
AuthenticationMiddleware
UserManager
- container : Slim Container
AdminChecker
- container : Slim Container
Slim App
# container: Slim Container
This is probably it's own class, I'm not sure what it's called
Note.. The Slim app is assembled so that the HTTP request passes through these Middleware classes from top to bottom. So, CORSManager is the first one to process the request, followed by AuthenticationMiddleware, etc.
AdminController
#container: Slim Container
+ getNonApprovedList($request, $response, $args): $response
+ changeStatus($request, $response, $args): $response
+ delete($request, $response, $args): $response
+ isAdmin($request, $response, $args): $response
- sendEmail($status: String): void
HuntsController
#container: Slim Container
+ getSingleHunt($request, $response, $args): $response
+ search($request, $response, $args): $response
+ add($request, $response, $args): $response
+ update($request, $response, $args): $response
+ submit($request, $response, $args): $response
+ delete($request, $response, $args): $response
AwardController
#container: Slim Container
+ getSingleAward($request, $response, $args): $response
+ getAllAwards($request, $response, $args): $response
+ addOrUpdate($request, $response, $args): $response
+ delete($request, $response, $args): $response
BadgeController
#container: Slim Container
+ getSingleBadge($request, $response, $args): $response
+ getAllBadges($request, $response, $args): $response
+ addOrUpdate($request, $response, $args): $response
+ delete($request, $response, $args): $response
ImageController
#container: Slim Container
+ add($request, $response, $args): $response
+ delete($request, $response, $args): $response
- moveUploadedFile($directory: String, $uploadedFile: UploadedFile): String
+ addImage($uploadedFile: UploadedFile): String
Mapper
# db : PDO object
# uid : String
# baseURL : String
- container : Slim Container
- state : State
+ get($obj: IMapperable): assoc array
+ getAll($obj: IMapperable): assoc array
+ update($obj: IMapperable): assoc array
+ delete($obj: IMapperable): assoc array
+ add($obj: IMapperable): assoc array
+ search($obj: IMapperable): assoc array
+ getAllChildren($obj: IMapperable): assoc array
+ submit($obj: IMapperable): assoc array
- setState($obj: IMapperable): State
- getApprovalStatue($obj: IMapperable): String
«Abstract» State
# db : PDO object
# uid : String
# baseURL : String
{Public stuff}
+ get($obj: IMapperable): assoc array
+ getAll($obj: IMapperable): assoc array
+ update($obj: IMapperable): assoc array
+ delete($obj: IMapperable): assoc array
+ add($obj: IMapperable): assoc array
+ search($obj: IMapperable): assoc array
+ getAllChildren($obj: IMapperable): assoc array
+ submit($obj: IMapperable): assoc array
{Database Mapping stuff}
# dbSelect($obj: IMapperable) : assoc array
# dbGetAllHunts($obj: IMapperable) : assoc array
# dbQuery($obj: IMapperable) : assoc array
# dbGetAllChildren($obj: IMapperable) : assoc array
# dbInsert($obj: IMapperable) : assoc array
# dbUpdate($obj: IMapperable) : assoc array
# dbSubmit($obj: IMapperable) : assoc array
{Helper Functions}
- getCurrentUID() : String
+ isOwnedByCurrentUser(IMapperable $obj): boolean
# isCurrentUserAdmin() : boolean
+ responseMessage($bool: boolean, $message: String): assoc array
+ buildResults($result: assoc array, $object: IMapperable): assoc array
- expandURL($result: assoc array): assoc array
- addChildren($array: assoc array): assoc array
«Interface» IMapperable
+ isParent(): boolean
+ getParentKey(): assoc array
+ getPrimaryKey(): assoc array
+ setPrimaryKeyValue($newValue: String): void
+ getTable(): String
+ getFields(): assoc array
Note.. The Controllers make an IMapperable object and pass it to the Mapper.
Hunt
- $TABLENAME: String
- $COLUMNS: assoc array
- $primaryKey: String
- $uid: String
- $fields: assoc array
Theres a bunch of methods in this class that could be deleted.
Award
- $TABLENAME: String
- $COLUMNS: assoc array
- $primaryKey: String
- $parentKey: String
- $uid: String
- $fields: assoc array
Theres a bunch of methods in this class that could be deleted.
Badge
- $TABLENAME: String
- $COLUMNS: assoc array
- $primaryKey: String
- $parentKey: String
- $uid: String
- $fields: assoc array
Theres a bunch of methods in this class that could be deleted.
Stateless
+ get($obj: IMapperable): assoc array
+ add($obj: IMapperable): assoc array
+ search($obj: IMapperable): assoc array
StateNonApproved
+ get($obj: IMapperable): assoc array
+ update($obj: IMapperable): assoc array
+ delete($obj: IMapperable): assoc array
+ add($obj: IMapperable): assoc array
+ submit($obj: IMapperable): assoc array
StateApproved
+ get($obj: IMapperable): assoc array
+ getAllChildren($obj: IMapperable): assoc array
StateSubmitted
+ get($obj: IMapperable): assoc array
AdminState
+ get($obj: IMapperable): assoc array
+ getAllChildren($obj: IMapperable): assoc array
+ search($obj: IMapperable): assoc array
+ delete($obj: IMapperable): assoc array

Database

The file “database/MagpieDB.sql” is a SQL script to create the database. It will also create a user (with password) for the PDO container in Slim to use to access the database. The file “database/old-MagpieDBv1.sql” is an older version of the database that I ended up using quite a bit for reference, so I’ve included it for future reference.

I recommend using phpMyAdmin to manage the database and import the SQL script.

Below is a diagram of the database:



Recommendations

The Mapper is a bit messy. It may be easier to just switch to a PHP Object-Relational Mapper (ORM) like Propel or Doctrine. I would at least look into them more. Due to the nature of the approval and ownership checking, it might be easier to keep a Mapper to do those checks, and then instead of the hand-written PDO, use an ORM to do the MySQL interfacing. So the ORM would exist in the State classes.

Firebase is being improved very quickly. It might be worth looking into all the services it provides. Right now, we're just using it for authentication. However they have database and file storage services as well. Basically, the whole backend could probably exist on Firebase. No php, sql, any of that crap.

Our group underperformed significantly due to having absolutely no planning process (or plans even). The process used in the Software Engineering class may seem tedious, but it works.

1. Define the problem
2. Divide the problem into smaller chunks (that connect to each other via interfaces)
3. Plan the approach (make sure what you're making can actually deploy)
4. Do the plan (and test as you go. Also document as you go, makes it easier.)

References

List of References by topic:

PHP

- Documentation: http://php.net/manual/en/
- W3 Schools Tutorial: https://www.w3schools.com/php/default.asp
- Other: http://www.phptherightway.com/
- Autoloading: https://phpenthusiast.com/blog/how-to-autoload-with-composer

PDO

- Intro: https://phpdelusions.net/pdo
- Slim also has a good PDO snippet

Slim:

- Documentation: https://www.slimframework.com/docs/
- IBM Tutorial: https://www.ibm.com/developerworks/library/x-slim-rest/
- Another tutorial: https://scotch.io/tutorials/getting-started-with-slim-3-a-php-microframework
- Passing Variables from Middleware: http://help.slimframework.com/discussions/questions/8833-slim-3-pass-variables-from-middleware

Firebase

- Authentication: https://firebase.google.com/docs/auth/
- Google sign-in: https://firebase.google.com/docs/auth/web/google-signin
- Verify ID Tokens: https://firebase.google.com/docs/auth/admin/verify-id-tokens
- Verify ID Tokens w/3rd party: http://firebase-php.readthedocs.io/en/latest/authentication.html#verify-a-firebase-id-token
- Firebase on the server: https://firebase.google.com/docs/admin/setup
- Firebase on the server 2: https://github.com/kreait/firebase-php/blob/master/docs/authentication.rst

- Not sure how userful but here ya go: https://stackoverflow.com/questions/42098150/how-to-verify-firebase-id-token-with-phpjwt

Postman

- Authorization in Postman: https://www.getpostman.com/docs/v6/postman/sending_api_requests/authorization

REST API

- The actual Dissertation: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
- Rest and PHP: http://coreymaynard.com/blog/creating-a-restful-api-with-php/
- Linking: https://stormpath.com/blog/linking-and-resource-expansion-rest-api-tips

Basic Create, Read, Update, Delete (CRUD) stuff

- Tutorial: https://www.taniarascia.com/create-a-simple-database-app-connecting-to-mysql-with-php/

Other

- Force SSL: https://my.bluehost.com/hosting/help/599
- JSON: https://www.json.org/
- JSON Viewer: http://jsonviewer.stack.hu/
- JSON Web Tokens: www.jwt.io
- UML: http://www.cs.bsu.edu/~pvg/misc/uml/ and http://pages.cs.wisc.edu/~hasti/cs302/examples/UMLdiagram.html