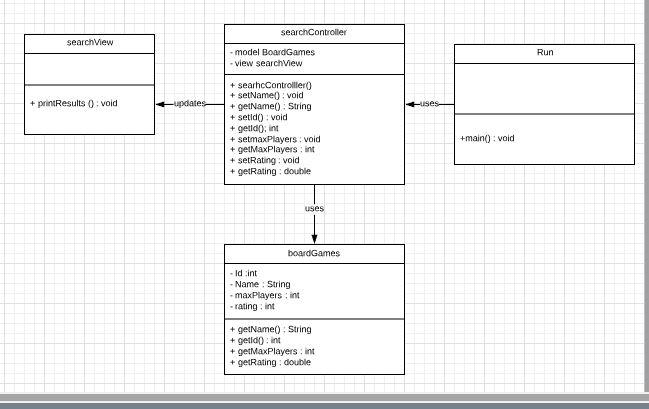
1. Team Members and Roles
   1. Regina Locicero
      1. UI designer, Regina has a strong background of creating beautiful UI’s from previous classes and has experience using graphic design software.
   2. Cauldierre McKay
      1. Backend-developer, Our strongest php developer, he's plays a pivotal role creating PDO’s and CRUD capabilities. He will also be helping with the database because that will not be an easy task.
   3. Stevie Ricci
      1. Front-end developer, Steven will be helping regina with the UI and Views but also will be utilizing his minimal php knowledge to manage user settings and profiles. He has a strong jquery background and can add some functionality to our UI.
   4. Alex Talbert
      1. RIP.
2. Background
3. Project Description
4. Project Requirements
   1. For this project we need to get certain features running. Logging in, offloading user ratings, setting up a personal library, and creating collections. Logging in is essential to the experience and differentiates this from the website as the site is more tailored to you, instead of any random user coming to the site. Since this is an application, you can have an offline connection so you can still view information about the board games on your wishlist, even when on the go.
5. Business Rules
   1. We believe that this service should be an application that can be incorporated into the BoardGameGeek's future desktop application. As a website, BGG has limited connectivity and must always adapt to the different web browsers attempting to open it. One design for Firefox, one for Opera, etc. With a desktop application, there can be a unified interface, and more tailored to the individual user. This should also be able to work with other board game websites with the setup we have decided on.
6. Design Patterns
   1. DBs - use our ERD for our end and [link to the API docs](https://boardgamegeek.com/wiki/page/BGG_XML_API2)
   2. MVC :

* With our application being primarily involved with searching through a database, MVC is always a strong option to go with. It’s definitely not the most creative approach and we have all used if before, when it comes down to completing this project in a timely manner along with us being in other high level classes, MVC just seems like the way to go.
  + 1. Model - pSQL
    2. View - html/css javascript
    3. Controller - php



* 1. Observer
     1. This application must use the Observer pattern in our opinion. Based on the user input for ratings, and adding board games. This will help normalize the database design and help the separation of concerns. Having the model listen to the view creates a more cohesive program.

1. Technologies Used
   1. LAMP: We don’t see any need to make things more complicated than they need to be. We are all familiar with using a lamp stack and these technologies are more than capable of getting the job done in a timely and efficient manner. A major plus to using a lamp stack is that we can connect our project serenity allowing easier access for us and those who will be using our application.
      1. Linux
      2. Apache
      3. Mysql
      4. PHP
   2. Jquery
   3. html/css
   4. Serenity
   5. Bootstrap
2. Timeline
   1. We have created a trello page for this project. This allows us to split up tasks in a organized and visual manner. Sections start our as broad as either front-end, backend. Then individual tasks are assigned with in each section. We have the ability to edit these tasks and assign each other to them or to certain sections of the tasks. For example, front-end development has all the views as individual tasks. This helps us get a general idea of how many different pages we will have and can easily check off what has been created and what still has to be developed.
   2. Dropbox milestones
      1. Milestone 4 - Exception Handling - October 19th
      2. Milestone 5 - Performance and refactoring - November 2nd
      3. Milestone 6 - Testing - November 16th
      4. Milestone 7 - Deployment and Packaging - November 30th
      5. Final Code - December 6th
3. Layering (https://www.jinfonet.com/resources/bi-defined/3-tier-architecture-complete-overview/)
   1. Data Layer **Cauldierre**
      1. The data layer is essentially handling most of the input and output involving the database. Creating enough abstraction where the other layers can use this layer without much information of this layer specifically. It is designed to handle any language tossing variables and thingymajigs its direction.
      2. Input data validation/sanitization of all information passed in through the UI.
      3. Search function to gain access to different information found in the database and take the user there.
      4. Create data objects for easy information input into the database.
   2. Presentation Layer **Steven**
      1. The top-most level of our application where our user interface is has to translate tasks and results from our business layer into something our user will easily understand with little to no previous knowledge of our application. Basically the user needs to enter information and see results. Basics of the UI will be completed using html/css/bootstrap along with javascript/jquery for functionality and form validation. Our data layer will mostly consist of PHP objects so html will be calling our php scripts to receive and sent data. Below are a few functions our Presentation layer will be involved in.
         1. The user needs to be displayed with a simple login page that is not cluttered with other aspects of our application. A simple form is required with brief directions. There are functions going on behind the scenes with our business layer and data layers communicating as the user logs in but they don’t need to be aware of that. The UI just needs to simply display success or and error and why there was an error using non technical terms that non technology experts will understand.
         2. A search function needs to communicate with the business layer as the user completes a book search. Information such as filters and strings need to be sent to the business layer in a format which our database will understand and be able to quickly return results to all the way buck up to the UI.
         3. Displaying a users profile will require a check of the user currently logged from our business layer who is receiving the users information from the data layer. Of course the users session will still have to be active as the data is pulled up then displayed simply for the user to quickly confirmed their information is correct.
      2. Form validation
   3. Business/Application Layer **Genie**
      1. Login authorization and authentication
      2. Our business layer is where most of our authentication and authorization will happen. There will be a way for users to login. This will be handled by a BusinessUser class that will have a login and logout functions. This class will make sure that the user is logged in and has the appropriate access. There will be a way to check that the user is logged in and store or references to a User class from the data layer with appropriate information. This class will be checked whenever we want to display a personal collection and especially when a user wants to modify their collection by adding or removing games.
4. Possible Functions
   1. Log in
   2. Log out
   3. View collection
   4. Rate Game
   5. View ratings
   6. Search
   7. Display profile
   8. Display forum
   9. Display game data
   10. Display collection
   11. Homepage display (not a function but whatevs)
5. Exceptions Expected
   1. Connecting to Database: We will be using the “magic” \_\_construct() method to establish a connection to mySql. This requires creating a new PDO class with our connection attributes as parameters. We will try to connect by inserting our server, DB name, User and Password as parameters. Upon failure, we will catch a PDOexception.
      1. Mysql host errors
      2. Can’t find server
      3. Can’t find user
      4. Can’t find password
   2. Login
   3. Null Pointer exceptions
      1. For when the user searches for a boardgame or some other value where it’s expected to have a filled in value, and is returned null the exception should be caught. When there are no boardgames it should display that there are no boardgames matching that name. When a user attempts to add a boardgame to their library that doesn’t exist it will give the user feedback that the game doesn’t exist. We don’t want our application to crash because a game doesn’t exist which is why it’s important to catch these null pointer exceptions and give the user feedback that the value they are entering isn’t valid. We are not necessarily showing these errors to the users, just giving them only the feedback they need.
   4. Unexpected Values
      1. Though most of the form fields will be sanitized, some things might slide through the cracks because hackers exist. So we will throw an exception whenever EXACTLY what we want isn’t passed in.
6. Exceptions in some code snippets
   1. ![construct method for DB](construct.png)
   2. ![recover all objects from the DB](getAllObjects.png)
   3. There are other simple methods that are very general for retrieving data from database but we don’t think you need to see all the basic CRUD functions
7. Performance and Refactoring for API part of the backend
   1. Already done:
      1. There are two model classes for the api: GameAbbrv and Game. The GameAbbrv class is for the abbreviated data that is returned when searching for a game. The Game class has all the details a user iswould need when looking at a single game. There is an ApiGrabber class that created the connection to the api and makes calls to get data. There is a method that takes in a string to search for games and a different method for getting game details using a single game id. These calls have been refactored to use the Guzzle library instead of using curl to make the calls more efficient. Tests have also been extracted out of this class to a different file for better coding practice.
   2. Still to be done:
      1. The tests need to be more extensive and more clearly formatted. The model classes should also be altered to have getters and setters in some cases. More comments need to be added for clarity. More data will be added, the API is difficult to work with as the xml returned is poorly formatted.
8. Performance and Refactoring - for other parts of the code
   1. Should be in the comments of the code
9. Testing
   1. API
      1. The code that pulls from the BoardGameGeek API is in api\_grabber.php in the classes folder
      2. In the tests folder there is an ApiGrabberTest.php class that uses PHPunit to run tests
      3. To run tests run the command

./vendor/phpunit/phpunit/phpunit tests/ApiGrabberTest.php

* 1. Database functionality
     1. Well we currently fail the connecting part of the database. The port to PostgreSQL was not handled very well at all creating the database. The DB.sql file is really messing up for me even when running it on localhost.
     2. As for the functionality, all of the tests pass in MySQL on Serenity. These tests include:
        1. CRUD with Users, Library
        2. Creating and reading from the database
     3. Not currently implemented but needs to be tested is handling dropdown menus with foreign keys to make easier for usage. No one should be allowed to type in a random username when dealing with connected data. Should only bring up usernames that are in the database.

1. Code should be all in a folder called iste432 -master - there was a lot of code we felt it would be much more readable to keep in it’s folder than put it into the document
   1. All the code is ours expect for the composer files, which is the dependency installer we used for installing Guzzle to increase API efficiency.
   2. Assets folder has the front end code
   3. Classes has some php classes for the api
   4. Vendor is not ours and is part of the composer and Guzzle files