Phase 4 Design Report

Canvas Group: morning-1

Project Name: Game-DB

Website URL: http://gamedb.us-east-1.elasticbeanstalk.com/

GitHub Repo Link: https://github.com/numan201/game-db

Phase Lead: David Wolf

Members:

Albert Garza

- **GitHub ID:** possumrapture
- **Email:** albertsgarza@utexas.edu

Numan Habib

- **GitHub ID:** numan201
- **Email:** numanhabib@gmail.com

John Nguyen

- **GitHub ID:** johnnguyen3196
- **Email:** johnnguyen3196@gmail.com

Alejandro Rodriguez

- **GitHub ID:** JustAlejandro
- **Email:** rodriguezalejandro@utexas.edu

David Wolf

- **GitHub ID:** rambisco
- **Email:** david.wolf@utexas.edu

<u>Information Hiding</u>:

Within model:

```
getGameData() {
  async getTwitchIntegration()
  async getSteamPlayerCount()
  async getYoutube()
  //one async call per API
  return combinationOfAllAsync()
}
```

Figure 0: Async Pseudo-code

In our design we refactored the specific game page to no longer be a JavaScript promise chain, where if one promise were to fail the whole chain may fail. Instead we moved all the API calls to separate methods that run independently of each other, and passed the raw data object to each method and allowed them to modify the fields as they wished. To implement Information Hiding however, we had to change this and have methods only take in the information they required to retrieve their data. Also, methods now return their data through key-value pairs so no method can modify the base object.

This simplifies the expansion of the game page since we just need a method to return the key-value pair that the game rendering page expects and add it to our list of promises chained in the game page model.

A clear disadvantage of our modular approach is that if an API call requires multiple pieces of data they have to be passed individually. Also if a method wants to return multiple pieces of data, it has to wrap all its data into one object. This implementation also doesn't have an easy framework for bundling API calls if order is important.

Within controller:

```
render( getGameData() )
```

The Controller calls the proper Model to render a given page and then passes that information to the View. Because of this, we can hide all implementation details from the Controller and keep the code in that section simple.

<u>**Design Patterns**</u>:

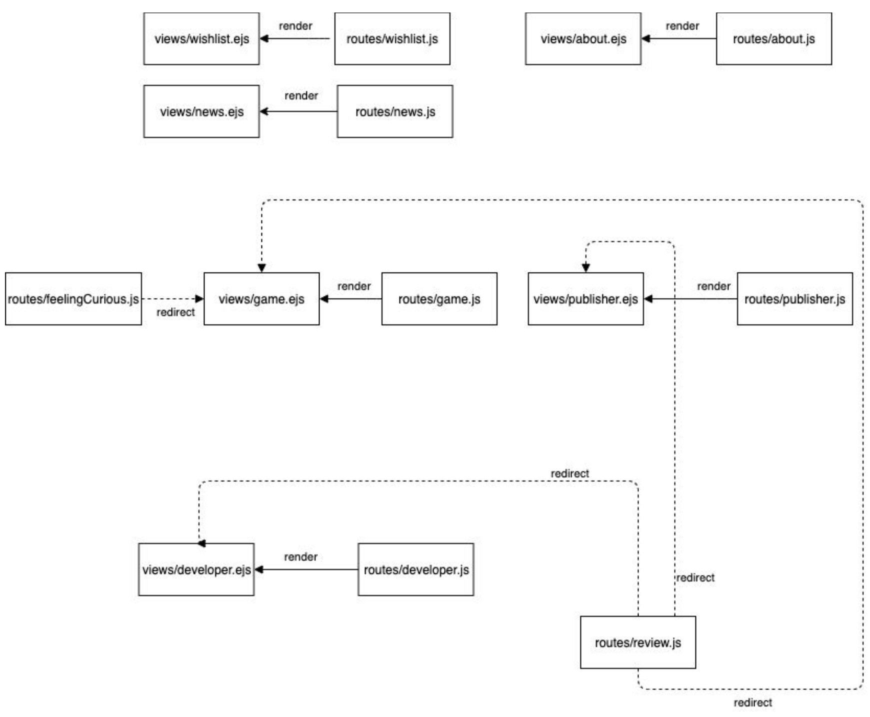


Figure 1: Before MVC

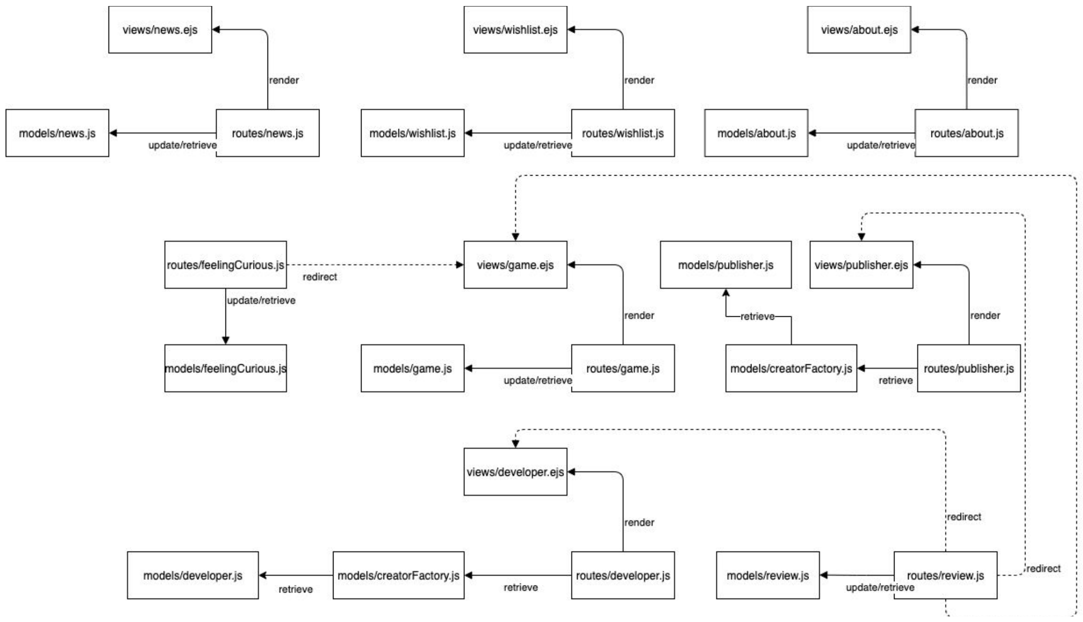


Figure 2: After MVC

MVC:

The MVC design pattern stands for model/view/controller. This design pattern fit well into our application since it divided responsibility of each file, easing development. The Model is the first part of the design pattern. The Model contains information about the current page. In our case, this includes the information we have in our database, and the information we gain from our APIs. Once all the information has been gathered inside the model, it is sent over to the view through the controller. The view is responsible for displaying the data given by the controller. In our case, the view is, logically, in the views/ directory and the controller in the routes/ directory. Finally, the controllers are our routers. UI elements are linked to various URLs and GET parameters, which in turn map to routers. The routers call methods on the models, thereby manipulating them. Users can change between themes, add items to their wishlist, log in, and many other things that change database entries and request different models and views. When a user clicks on a game page, the controller in routes/game.js calls functions in the model in models/game.js to get the information needed for views. Then after all the information is compiled, the data is sent to the view in view/game.ejs to display to the user. This is the biggest difference from before MVC. In the past, the js file would compile information from the database, update the database with anything new from the request, and then call for the page render, all in one file.

The advantages of the Model/View/Controller design pattern are multiple. Firstly, it offers a good conceptual way of thinking about the behavior of our application. When planning the implementation of a new feature, the requirements and behavior of that new feature can be separated and more easily processed. Another advantage is that functionality is segregated into maintaining data, displaying data, and manipulating data. Because of this, interactions are more streamlined between the database, the application, and the web view. Changes can be made to the way things are stored in the database, but as long as the model returns information as expected, no other module has to be changed. For example, if we wanted to start caching Twitch usernames, we could implement the module that returns a username either from Twitch, or from the cache if it is empty. There would be no required change for the view, which expects the username in a certain format, and would still receive it as expected.

The disadvantage of using MVC is that it adds complexity to our project. Rather than being able to just add logic to our controller to handle specific cases, we have to add this functionality to the model. For example, redirection logic is more difficult to implement, such as our Wish List add/remove functionality. Within MVC we cannot redirect in the Model, so we have to update database entries in a more complex way.

<u>Factory Pattern</u>:

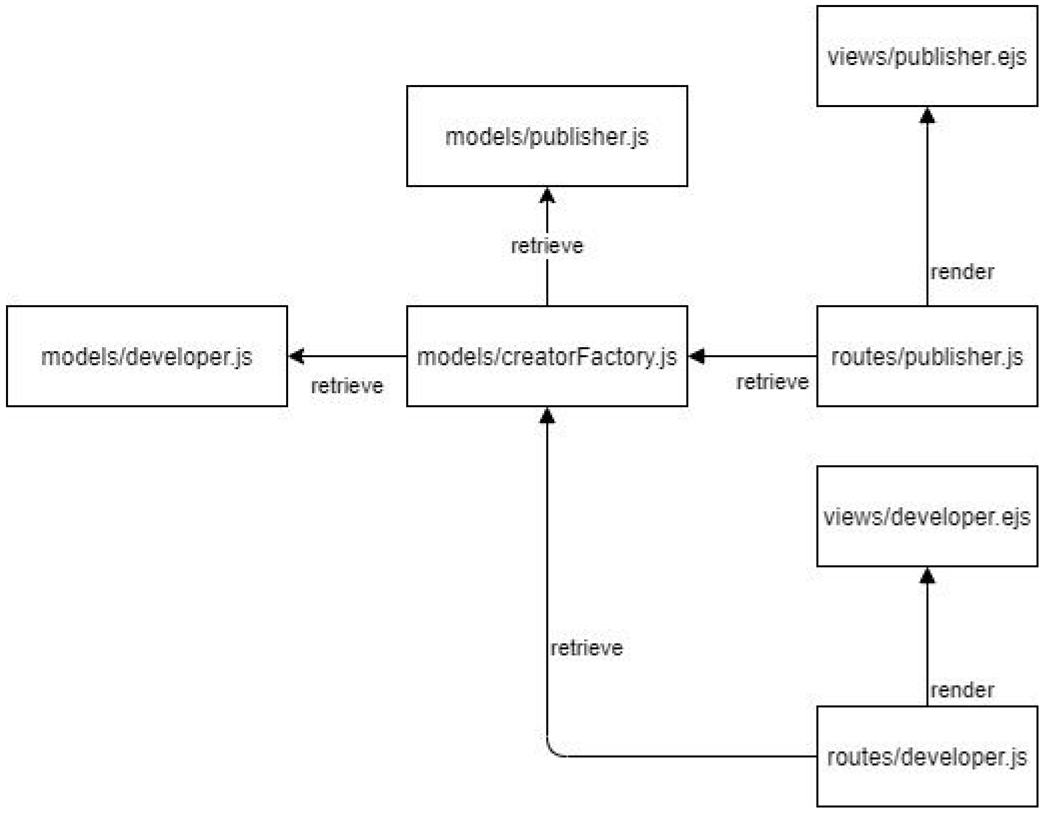


Figure 3: Factory Pattern with MVC

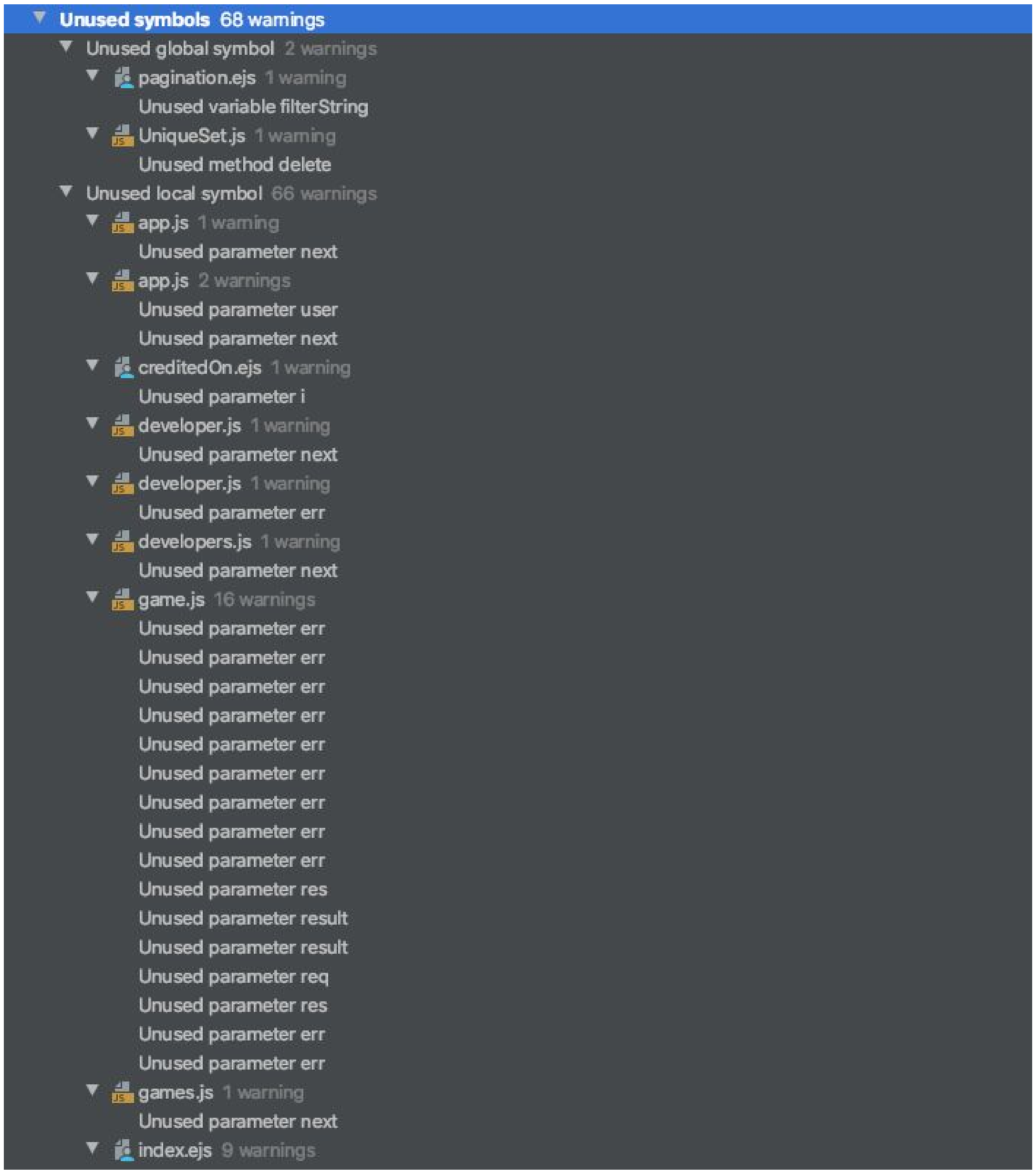
Since the developer and publisher model are very similar, it makes sense to use the factory pattern. When a user clicks on a developer or publisher, the factory provides the appropriate model and the controller retrieves the information from the factory to render the views.

The advantage of using this pattern is that we can easily add more creator models to our website.

Since the factory only creates two different models, the amount of development time needed to implement this pattern doesn’t justify the added complexity. Without the factory pattern, each controller could simply retrieve information from the model instead of retrieving it from a middleman.

Refactoring:

Example 1: Simplifying method calls by removing parameters



```
▼ Unused symbols 68 warnings
  ▼ Unused global symbol 2 warnings
    ▼ pagination.ejs 1 warning
        Unused variable filterString
    ▼ UniqueSet.js 1 warning
        Unused method delete
  ▼ Unused local symbol 66 warnings
    ▼ app.js 1 warning
        Unused parameter next
    ▼ app.js 2 warnings
        Unused parameter user
        Unused parameter next
    ▼ creditedOn.ejs 1 warning
        Unused parameter i
    ▼ developer.js 1 warning
        Unused parameter next
    ▼ developer.js 1 warning
        Unused parameter err
    ▼ developers.js 1 warning
        Unused parameter next
    ▼ game.js 16 warnings
        Unused parameter err
        Unused parameter err
        Unused parameter err
        Unused parameter err
        Unused parameter err
        Unused parameter err
        Unused parameter err
        Unused parameter err
        Unused parameter err
        Unused parameter res
        Unused parameter result
        Unused parameter result
        Unused parameter req
        Unused parameter res
        Unused parameter err
        Unused parameter err
    ▼ games.js 1 warning
        Unused parameter next
    ▼ index.ejs 9 warnings
```

Figure 4: A few unused parameters in methods

Using the Webstorm's Code Inspection tool, we were able to find numerous parameters in our functions that were not used. We removed the parameters to make the method calls easier to understand and overall shorter. Figure 4 shows a few unused parameters with the file associated with the parameter.

Example 2: Promise chain refactoring

Before

```
.then((data) => {

  data.videos = [];

  let options = {
     q: data.game.name,
     part: 'snippet',
     type:' video'
  };

  return youtubeApiV3Search(youtubeKey, options)
     .then((response) => {
        response.items.forEach((item) => {
           data.videos.push(item.id.videoId);
        });

        return data;
     })
     .catch(err => data);

})
.then(data => {
  let reviews =
req.app.locals.db.collection('reviews').find({id:id.toLoc
aleString()}).toArray();
  return Promise.all([reviews]).then(([reviews]) => {
     data.reviews = reviews;
     return data;
  });
})
  .then(data => {
     /* Continues with many more promises */
  }
```

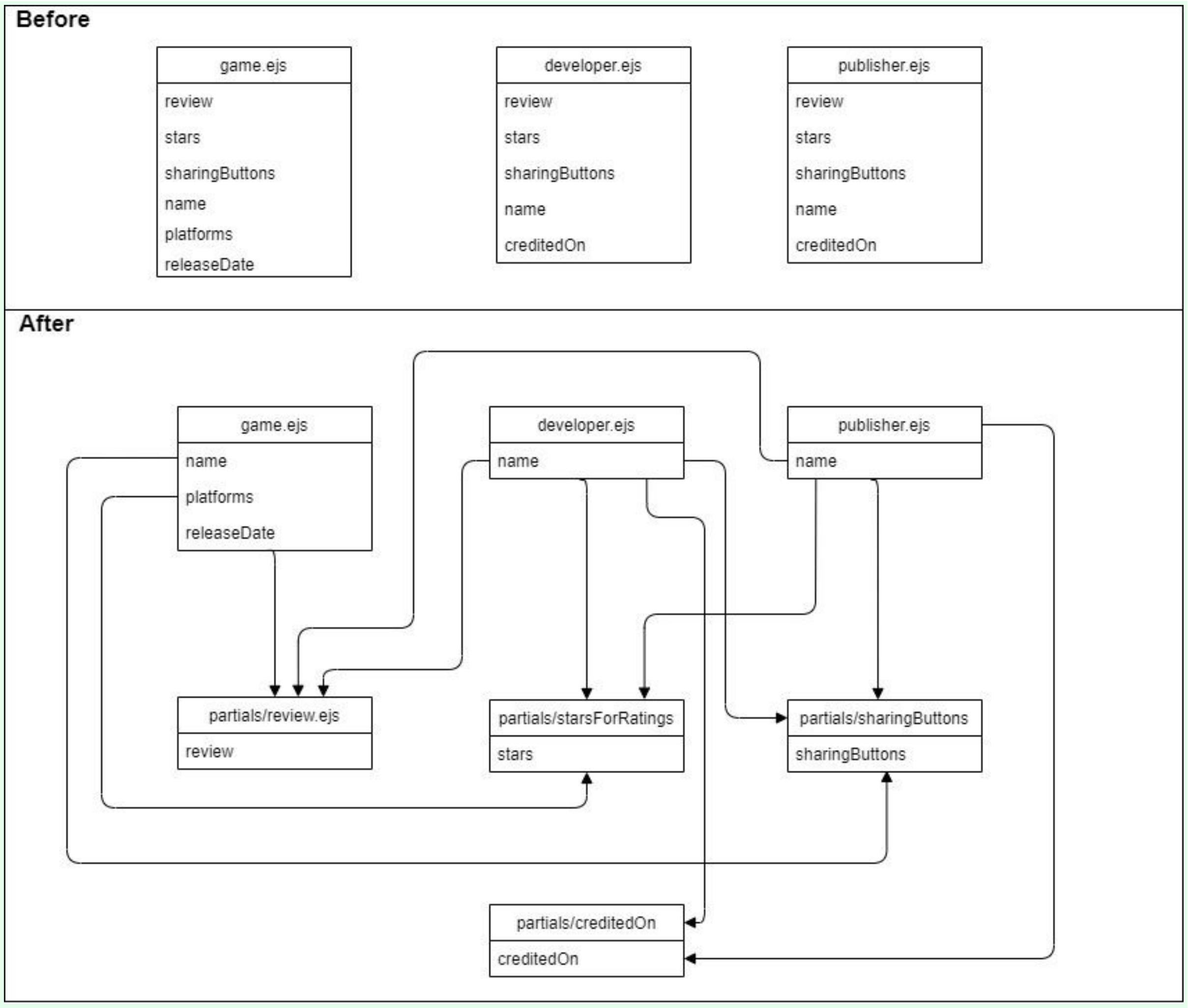
After

```
function getGameData(promise, req, res) {
  const cachePromise = new Promise(finish =>
checkCached(promise, finish, req, res, id))
  req.app.locals.db.collection('games').findOne({_id :
id})
     .then(game => {
        let data = {title: game.name, game, page:
req.baseUrl};
        getSteamAppId(data.game.stores);
        const hiddenPromises = [
           new Promise(resolve =>
getSteamAchievements(resolve, steamAppId, req)),
           new Promise(resolve => getWishlist(resolve,
uniqueId, req)),
           new Promise(resolve =>
getReviewsCounts(resolve, ratings)),
           new Promise(resolve =>
getReviews(resolve, req, id)),
           new Promise(resolve =>
getPublishers(resolve, gameId, req)),
           new Promise(resolve =>
getDevelopers(resolve, gameId, req)),
           new Promise(resolve =>
getSteamPrice(resolve, steamAppId)),
           new Promise(resolve =>
getTwitchIntegration(resolve, name)),
           new Promise(resolve => getHLTB(resolve,
name)),
           new Promise(resolve =>
getYoutube(resolve, name))];
        Promise.all(hiddenPromises).then(output => {
           let game = data;
           output.forEach(entry => {
              game[entry.key] = entry.value;
           });
           game.date = new Date();
           return data;
        });
     });
}
```

Figure 5: Code snippet of promises in models/game.js

Previously, each API call was chained in a single method. This chaining causes an issue in long page loads as each promise waits until a previous promise is resolved, and if one promise fails, the entire page load fails. We refactored this code to separate each promise returned from the call into a new promise array, called hiddenPromises, as seen in Figure 5. Because of this we're able to have all our API calls run asynchronously and as an added benefit cut our page load times by more than half.

Example 3: Extracting duplicate code from ejs files and adding them to the partials/ directory



This was a very important step in cleaning up our code. If in the future, we needed to make changes to any of these modules, we would not have to go through our code and search for every use of them. For example, the sharing buttons were the same for the developer, publisher, and game pages. If we had wanted to remove Facebook, and add LinkedIn, it would have been a mess to find every page the share buttons were located. They can now be changed simply and easily.