Hughes Fieldhouse Project Artifacts

Developers: Matt Woolery, Dattu B Medarametla
Clients: Greg Hansen, Brooke Byland
GDP 1 Instructor: Dr. Denise Case
GDP 2 Instructor: Dr. Ajay Bandi

Preface

This document includes information and artifacts from the Hughes Fieldhouse Project Team's work. It is intended to be used as a reference to help understand the way we worked on the project and information that is useful for getting an understanding of how it works. Further explanations on how to use the application can be found in the User's Manual. Information about how to work on the application's various features can be found in the Developer's Manual.

TOC

Project Introduction

Last August, we were assigned to work on the Hughes Fieldhouse Website. We were approached by Greg Hansen, Brooke Byland, and the Campus Recreation team to be taken on as clients for the project. The clients wished to have an updated site for the Hughes Fieldhouse, because at the time they had a site with soon to be outdated information on nwmissouri.edu. Going forward, they wanted to have a site that conveyed the information about the Hughes Fieldhouse in an easy to use and user friendly experience, with everything found in ideally less than 3 clicks for users. They eventually decided that they would also like the option to post updates to the site from an editor on the site and from social media. They also wanted to have additional information about the Fieldhouse and the current happenings on the site with a Map, Astra Calendar, and a contact form. We proposed a solution to them in the form of a Progressive Web App, a web app designed to feel like a Native app. In the long run we did not follow through with the PWA functionality due to some complications with its functionality and the database, but we did manage to provide an app like experience by using Bootstrap to resize the pages to mobile screens. We also proposed a Banner Item editor that would allow them to post scheduled messages to the site on a banner at the top.

Methodology

To understand how we worked on this project week by week, it is important to discuss the development methodologies that we followed. In this section we will cover the methodology that we followed and discuss why we choose it and why it worked best for us. We followed the Agile/Scrum development methodology. We classify it as this because we followed primarily an Agile approach, but we did follow some elements of Scrum, but we did not follow every single element of Scrum due to the size of our group that worked on the project changed from GDP 1 to GDP 2 to a smaller size that does not allow for true Scrum.

Agile Explanation: We primarily focused on the Agile methodology. Since it is a website and not software with many different algorithms, there is always going to be some sort of deliverables as you develop, be it a new functionality or a new web page. Because we focused on creating working Software (Web app pages) we were able to release new content as soon as we completed it. Also this allowed us to complete periodic User Acceptance Tests with the client to verify our work and progress on the app. Agile is the best way to describe the methodology for the most part, however there was certain components of Agile that we failed to complete due to low motivation by those who worked on it early on. An example of how we failed to follow the methodology to a better extent early on was that we didn't have set deadlines for certain pages or features to be released, mainly whoever was willing to work on a feature was responsible for it to get released by a milestone. Also another example is that testing other than User Acceptance Testing early on was not done since no one put effort into testing until GDP 2.

Scrum Explanation: During most of the project's life, we followed a Scrum approach. Although it was not perfect Scrum, it was somewhat of a take on Scrum. In GDP 1, we had a larger team and had a Scrum Master that managed the Sprints that we were working on for the milestones and managed the backlog and daily scrum in Jira. Eventually, we were separated into a smaller group of developers, and while we ended up continuing to do something similar to Scrum, it was not fully Scrum since there was not a Scrum Master. We continued to have daily meetings and discuss tasks that we were going to be completing in the week and tried our best to remember to go in and keep track of our tasks in Github's Issue Tracker, however without a dedicated Scrum Master, that proved to be difficult to remember to do at times.

Figure 1.

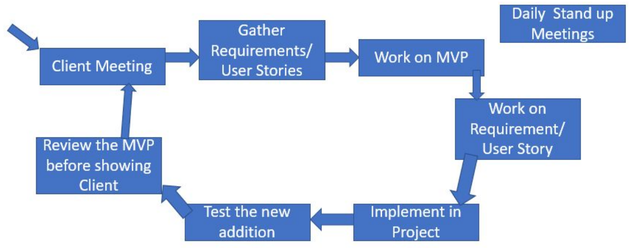

Client Meeting
Gather Requirements/ User Stories
Work on MVP
Daily Stand up Meetings
Work on Requirement/ User Story
Implement in Project
Test the new addition
Review the MVP before showing Client

Figure 1 shows an example of the methodology we used in general throughout the entire project. We start each iteration with the client meeting, from which we gather requirements and decide on the next Minimally Viable Prototype that we will demonstrate the next time that we meet. After we created the MVP, we work on individual User stories and implement them in the project. On each addition, we test the functionality manually since our project doesn't have individual units to test or we do some User Acceptance testing if we have doubts on appearances of the site. After we have our additions implemented in the project and we have approached near the time of showing the MVP to the client, we code review each other's work until we are satisfied to show the MVP to the client. After showing the client the MVP, we can run through the cycle again. I also included the daily meetings out to the side, since they don't have a definite place to be set in the life cycle since they occur at any time, but in general they occur between the beginning of work on the MVP through the time we show the client the MVP. In the daily meeting we decide who will work on what requirements and discuss any issues that we have, after this discussion we begin working on the tasks needed to get to the next steps.

Requirements/User Stories

User:

1. As a user, I want to see the contact information related to the Hughes Fieldhouse.
2. As a User, I want to view a grid of images that helps navigate to each functional area of the app.
3. As a member of the Marketing Team, I want to ensure that the design reflects the Northwest brand.
4. As a user, I want to contact the persons involved in case of any inquiry regarding the events being conducted.
5. As a user, I want to see and click on the map in the homepage and view the location of the Hughes Fieldhouse.
6. As a user, I want to click on the stories on the storyboard so that I will be redirected to the page related to that (clicked) story.
7. As a user, I want to find information about the lockers by clicking on the lockers image on the storyboard.
8. As a user, I want to make a reservation at the Hughes Fieldhouse for an event by viewing the current events on the calendar and then be able to contact to inquire about reserving the field
9. As a user, I want to click on the map, so that I can get driving directions to the Field House from my current location.
10. As a user, I want to view and fill out the contact form and be able to send feedback, inquire about the fieldhouse and other actions so that I can reach out to the Fieldhouse Team.
11. As a user, I want to see the twitter/Facebook/Instagram feed on the home page in order to check out recent happenings.
12. As a user, I want to see the announcement banner when I open the website/home page.
13. As a user, I want to click on the contact number in the contact info, so that I can call the responsible person.
14. As a user, I want to click on the email link, so that I will be navigated directly to the send email to the contact person.
15. As a user, I want to navigate to other pages from the Home page, that includes, - - About Us page, Facilities page and Media page.

Admin:

1. As an admin, i should be able to access the login page from the homescreen.
2. As an Admin, i should be able to login to the banner item page.
3. As an admin, i should be able to create the banner item.
4. As an admin, i should be able to view the banner item list.
5. As an admin, i should be able to edit the the banner item.
6. As an admin, i should be able to delete the the banner item.

7. As an admin, I want to see the number of people who have viewed the page in Google Analytics.

Basic Functional Requirements

1. Useable on any device with a responsive design.
2. Users should be able to utilize Google Maps, Astra, Social Media, and Contact Form to be able to get and understand information
3. Only authorized users are able to use the Database

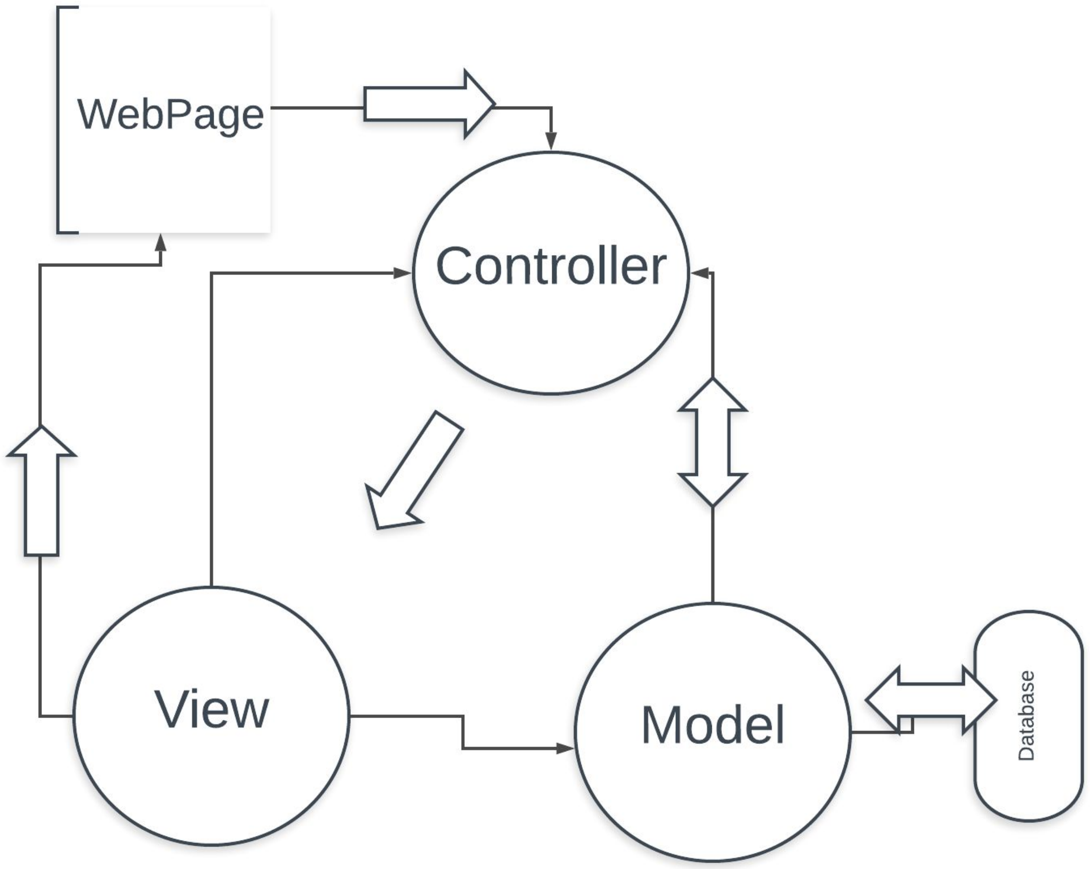
Basic Non-Functional Requirements

1. The app should have some standard of performance by loading quickly
2. The pages of the app should be accessed in 3 clicks or less
3. Data is stored reliably

Architectural Design

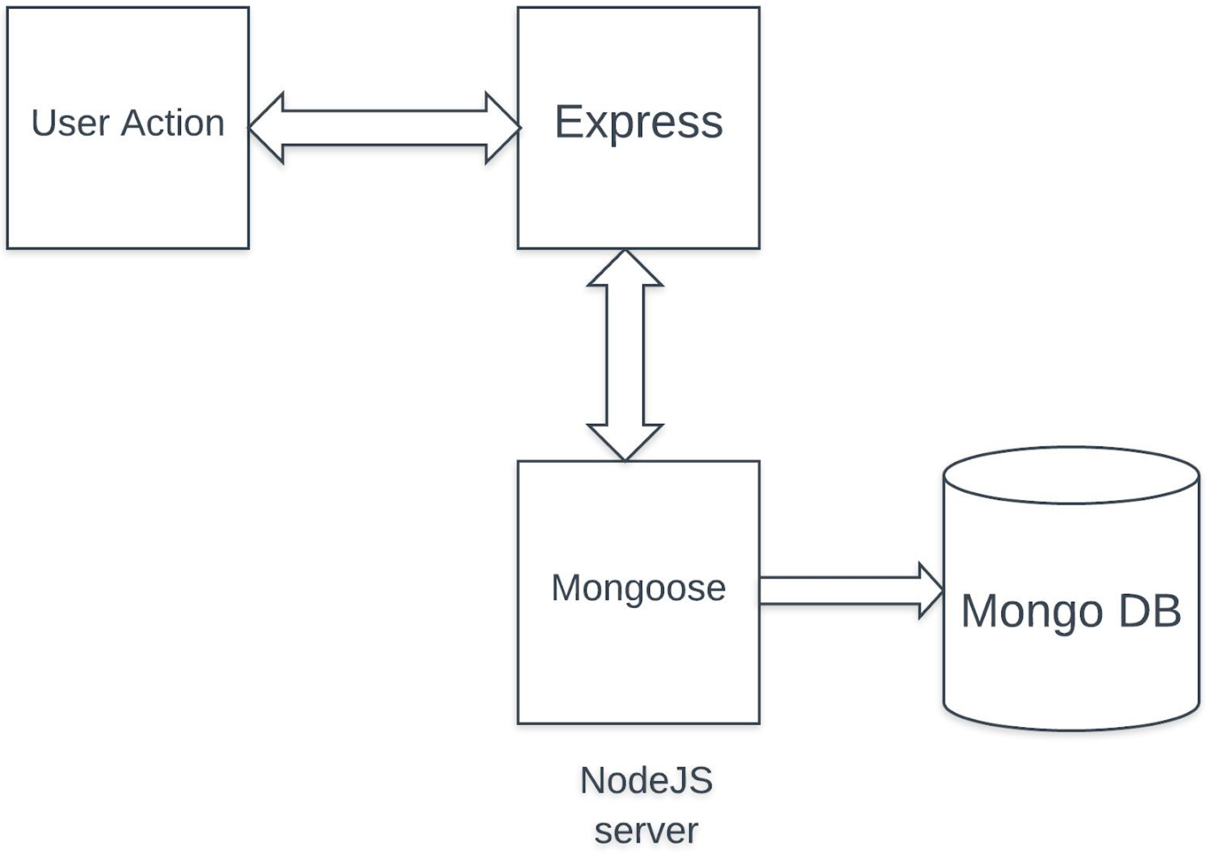
Our project is an Model View Controller web app that is running on a Node.js server. It also uses MongoDB and a few cloud services to perform certain tasks. In this section we will discuss what our app's architecture is and provide a graphic that explains it.

Model View Controller: The app uses a Model View Controller architecture to handle data modifications and displaying of pages. The Model component uses Mongoose models to create the Banner Item and perform data modifications with MongoDB. The Views are EJS templates that are displayed by the controllers and allows the user to communicate to the Model via the Banner Item editor. The Controller handles the requests made by the users, it returns the views for the pages that the user requests for.

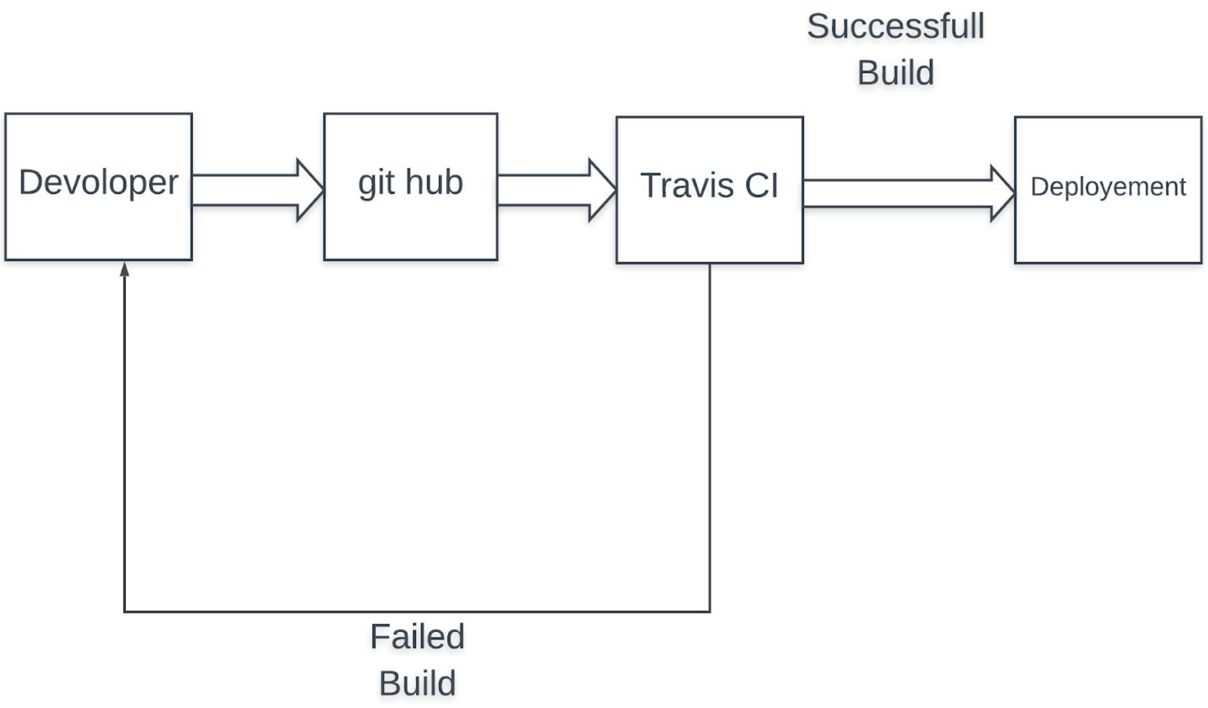


Node.js: was the server of choice that runs the application. It is good for development because there is many applications and services that are easy to add to your application via an npm install command. This allowed us to include Passport authentication and Mailgun mailing easily.

MongoDB: This is a Nosql database that we used because it is free and easy to use. Our application uses Mongoose to create the Banner Item models so that it can send the data changes to our Atlas cloud MongoDB database. The Node.js app communicates with the MongoDB when a data modification change is performed, also on app start where it connects to the database to get all of the current records in the collection.



Deployment: On deployment, the code goes to Github, where it is then pulled into Travis CI where it builds, on a successful build it moves to Heroku to be deployed.

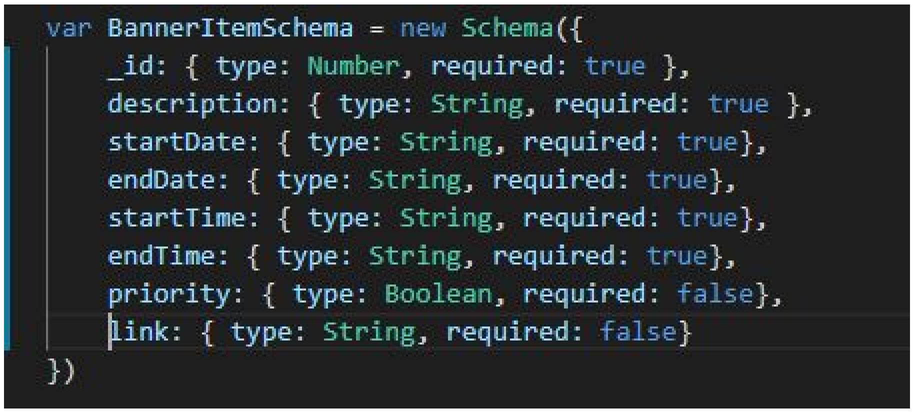

Successfull
Build
Devoloper
git hub
Travis CI
Deployement
Failed
Build

Dummy Data

The data is stored in a Cloud Atlas MongoDB cluster.   There are 3 collections stored in the database that our application uses. Dummy data and explanation for what is being stored for each of these collections will be given in this section. The 3 collections are not related to each other in any way so a traditional ER diagram would not be too helpful. Instead we will show (if applicable) the MongoDB Schema setup used in app, provide explanations for the collection and the fields it has, and provide example dummy data that is used by the app.

BannerItem

- Schema



```
var BannerItemSchema = new Schema({
    _id: { type: Number, required: true },
    description: { type: String, required: true },
    startDate: { type: String, required: true},
    endDate: { type: String, required: true},
    startTime: { type: String, required: true},
    endTime: { type: String, required: true},
    priority: { type: Boolean, required: false},
    link: { type: String, required: false}
})
```

- Explanations for Fields

_id is the id of the banner item. The database starts at one, takes the maximum id from the database and adds 1 to it each time a new item is added. This way there is not any repeated id's and each item is listed in order in a easy to navigate way.

Description: This is what is displayed in the text of the banner item on the homepage. This is where the announcement is put.

startDate: This is a string that is the start date that the banner begins to display. It is stored in string format instead of date format since it is easy to add the start time with it and use the combined string to compare to now to decide if it needs to be displayed. Also there was inconsistencies with the JavaScript format and the MongoDB date format so it was easier to do it this way.

startTime: This is a string that is the start time that the banner begins to display. It is stored in string format so it could be easily added to the startDate field to do the comparison to see if the Banner Item should be displayed or not.

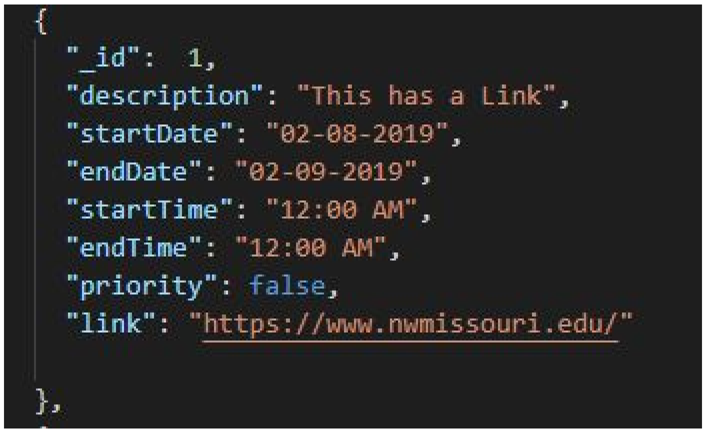
endDate: This is a string that is the end date that the banner is to stop displaying on. It is stored in string format instead of date format since it is easy to add the end time with it and use the combined string to compare to now to decide if it needs to be displayed. Also there was inconsistencies with the JavaScript format and the MongoDB date format so it was easier to do it this way.

endTime: This is a string that is the end time that the banner is to stop displaying. It is stored in string format so it could be easily added to the startDate field to do the comparison to see if the Banner Item should be displayed or not.

Priority: This is a boolean that decides is used to decide if a Banner Item is of a higher priority and needs to be displayed above the rest of the other items. It is not required because if it is not specified as true, it is to be assumed as false which means it is not a priority item.

Link: This is a string that is the link to an external url that will be used to make the Banner Item a link to an external site. This is optional since it is not necessary that the Banner Item links to another site each and every time.

- Dummy Data

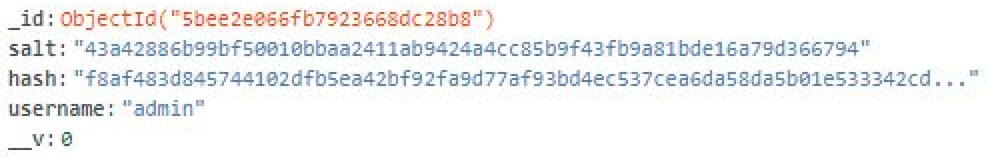


```
{
  "_id":  1,
  "description": "This has a Link",
  "startDate": "02-08-2019",
  "endDate": "02-09-2019",
  "startTime": "12:00 AM",
  "endTime": "12:00 AM",
  "priority": false,
  "link": "https://www.nwmissouri.edu/"

},
```

User

- Dummy Data:



```
_id: ObjectId("5bee2e066fb7923668dc28b8")
salt: "43a42886b99bf50010bbaa2411ab9424a4cc85b9f43fb9a81bde16a79d366794"
hash: "f8af483d845744102dfb5ea42bf92fa9d77af93bd4ec537cea6da58da5b01e533342cd..."
username: "admin"
__v: 0
```

_id: Unique id created by MongoDB
Salt: String added to the Hashing function to help improve the hashing, needed to translate password back for password to authenticate
Hash: The password after it was ran through the hashing function

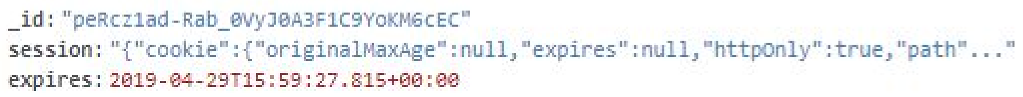
Username: the username of the Banner Item editor user.

Session

- Explanation

The session collection is created by the passport whenever a user successfully logs in. If a user logs in and leaves the BannerItem site without logging out and the app remains open, the session will remember that the user actually logged in and let them come back without having to resign in. The id is handled by MongoDB, session is the data that is created by the Passport Session module and expiration date is the date that Atlas will remove the session from the database.

- Dummy Data



```
_id: "peRcz1ad-Rab_0VyJ0A3F1C9YoKM6cEC"
session: "{"cookie":{"originalMaxAge":null,"expires":null,"httpOnly":true,"path"..."
expires: 2019-04-29T15:59:27.815+00:00
```

Tools & Technology

In this section, we are going to highlight the tools used in development and explain why we decided to use these tools. We will discuss the different services that we used, the dependencies, and any other development tools that will be useful to highlight for those who ask about the project.

HTML/CSS/JavaScript

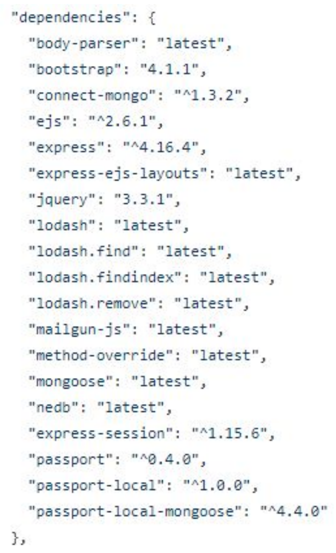
Since this is a web application, these 3 are synonymous with websites. Although there is additional tools at play with the creation of the pages, in general they are entirely HTML pages, that get the styles from the style sheet. Since we also used a view engine, there were times when we were not able to have javascript in a .js file, for example setting the title on the page for the tests. Other times the we opted against having a individual js file for certain codes so we were able to group like items together such as the calendar jquery script for the calendar pop out being grouped with the form items that used those scripts. These are the primary front end development tools.

Node.js

Node.js was used as the server of the application. The reason we chose this as our server tool is because it is easy to start up and develop for and it is also free. Also Node.js has tons of development dependencies that you can add in to perform certain tasks, each of these dependencies are often well documented or easy to figure out online.

Node.js Dependencies

The Node.js dependencies can be found in package.json, we also included the dependencies listing below. We will briefly cover what each dependency is used for in our app if applicable, the version numbers do not have too much to do with the overall functionality most of the time but developers looking to use these tools as well might want to stick with the latest supported versions.



```
"dependencies": {
  "body-parser": "latest",
  "bootstrap": "4.1.1",
  "connect-mongo": "^1.3.2",
  "ejs": "^2.6.1",
  "express": "^4.16.4",
  "express-ejs-layouts": "latest",
  "jquery": "3.3.1",
  "lodash": "latest",
  "lodash.find": "latest",
  "lodash.findindex": "latest",
  "lodash.remove": "latest",
  "mailgun-js": "latest",
  "method-override": "latest",
  "mongoose": "latest",
  "nedb": "latest",
  "express-session": "^1.15.6",
  "passport": "^0.4.0",
  "passport-local": "^1.0.0",
  "passport-local-mongoose": "^4.4.0"
},
```

"body-parser": "latest" - This package is middleware used to parse the json into the controller. Often in the controller, you would see res.req or other lines of code like that, that was body parser handling text that the user entered and made it into JSON so that it could be saved into mongoose.

"bootstrap": "4.1.1" - Bootstrap is key to the responsive design. Although it was included in the project src files, it was important to add this in for the Heroku deployment so it could use Bootstrap to handle the resizing to mobile screens.

"connect-mongo": "^1.3.2" - This was used to connect to Atlas MongoDB. When provided a connection string, it handled the connection to the database so that data modification would be reflected in the database.

"ejs": "^2.6.1" - This was the templating engine that we used. Often in MVC, it is good to use a templating engine such as ejs to produce consistent layouts. Having a consistent layout ensures all pages look similar and also it allows for the only part of the page that we have to change is the body, this way any changes that we make to the header or footer will be the same on all pages.

"express": "^4.16.4",
"express-ejs-layouts": "latest" - This was the layout tool used for ejs. Having this allowed us to define a layout page and individual header, footer, and bodies for the pages. This dependency is similar to having an "includes" function that is often seen in programming languages to include code from a specified file.

"jquery": "3.3.1" - Although we have jQuery in the src files as well, we included it here since it is used by the Heroku deployment. jQuery is a javascript library that allows you to perform more complex javascript, in fewer lines of code. Notable items in our site that use jQuery are the date/time pickers.

"lodash": "latest","lodash.find": "latest", "lodash.findindex": "latest","lodash.remove": "latest", The lodash is used to make handling arrays, string, and objects easier. Most notably in our project, lodash was used to handle data modification in the nedb database which was used for development. It was used in a few other places as well but that was the main purpose.

"mailgun-js": "latest" - mailgun was the email service used in the contact form. It is a service that allows those who set up an account and obtain an api key to send up to 10,000 emails for free.

"method-override": "latest" - this was used in case we needed to override a method such as the get and post methods to the database. The final version of the app, does not use anything from here.

"mongoose": "latest" - Mongoose is used in Node.js to communicate with the Mongo database. Our app uses it to create the Schema's for the models and in the controllers to directly communicate with the MongoDB.

"nedb": "latest", - This is a local database that we used for development, it is still working in the app, however all of the current live data is read from Mongo. It was useful to have this when we were still working out the Schemas for the Models since it allowed for quick and easy changes on the startup of the app.

"express-session": "^1.15.6", - This was not originally part of the requirements, but the passport example that was found included a means for creating sessions when logging in so the user could potentially be logged in for an extended period of time without having to reenter the password every time to login. It creates a session when the user logins, then it stores the session, the session can be used until it expires or the user logs out.

"passport": "^0.4.0","passport-local": "^1.0.0","passport-local-mongoose": "^4.4.0" - Passport is used by the application for authenticating the user. This allows the user to use a username and password to securely login. The mongoose version means that the credentials can be saved in a User collection in the cloud.

<u>Github Version Control</u>
We used github as our means of version control. We choose github because a lot of other services nicely integrate with it such as Travis and Heroku, also it is easy to collaborate with others on it.

<u>Heroku Deployment</u>
Our app is deployed on Heroku. We set up the deployment to read from the Master branch of our github repository and on every push to master, Heroku redeploys the app. We chose Heroku because it is easy to set up and is free to use.

<u>Travis CI</u>
Although we did not implement the full DevOps cycle, we did work through each of the areas and the CI tool is an important step in DevOps. We chose Travis CI as the tool of choice to do this as it is free and is set up to work with Github well. The CI is not setup to run the tests before Deployment because we used Selenium for the Integration testing, and to run Selenium Tests in Travis, you must pay money to cloud host the automation tests. If Travis is turned on for our app, it will build the application, and if it can build it deploys to Heroku.

<u>Mailgun</u>
We used mailgun for sending emails in the contact form. It is a useful service that we chose because it allows us to send up to 10,000 emails for free with a free domain. Paid options are inexpensive and we also provided instructions in the developers manual on how to set up accounts to possibly switch domains if the client hits the 10,000 email limit.

Progressive Web App

Progressive Web App was the original goal of the application since it allowed for a native app like experience with a Web App. By using a combination of Bootstrap to make it look like a mobile app, using a web manifest, and using a Service Worker, you can get the option from Google to download the app to your home screen of your phone so you can view it offline. This was a good idea at the beginning of the project, but as the requirements changed to include the database, we found that it was not reliable to count on the caching features of the web app to reflect changes made in the database. So while it is not something that is delivered with the final project, it is something that can be easily enabled by reading the code comments and uncommenting the service worker registration found in index.ejs for the homepage.

Selenium

We used Selenium for the Automation/Integration testing of the site. Since websites often do not have individual units to be tested, Unit Testing is often not possible. Websites are often made up of multiple pages that automation testing can test for the interaction between pages. We choose Selenium since it is a common tool used in DevOps and automation testing, as well as it is free and easy to code in Java. Our Selenium tests of our project test the controller's, test logging into the banneritem editor, tests the various data modifications, and tests to see if the site resized properly for mobile.

Visual Studio Code

This was our IDE of choice when creating a majority of the site

Test Cases

We provided numerous test cases using Selenium to prove that our app holds some sort of reliability, that way when users end up using this app, we can be certain that thought of everything during development to have quality in our end product. We created 17 test cases, in this section we will cover what the test's intended outcome is and why we chose to do that test. In the Results & Analysis of Test Cases section, we will discuss the outcome that we got, any issues that we faced, and what conclusions we can make from the results of the test.

controller_get_index(WebDriver browser)- this test was chosen to test the controller. When a user enters https://localhost:3000/ or the corresponding deployment url, we expect that the index page is returned. We choose this test because users often enter in the url at the top and our app uses the "/" path to go to the index page. To verify a successful load of the page, the test gets the title of the page which is unique to that page, if Selenium navigated to the page that was specified it is a success.

controller_get_login(browser) - this test was chosen to test the controller. When a user enters https://localhost:3000/login or the corresponding deployment url, we expect that the login page is returned. We choose this test because users often enter in the url at the top and our app uses the "/login" path to go to the login page. To verify a successful load of the page, the test gets the title of the page which is unique to that page, if Selenium navigated to the page that was specified it is a success.

controller_get_awards(browser) - this test was chosen to test the controller. When a user enters https://localhost:3000/awards or the corresponding deployment url, we expect that the awards page is returned. We choose this test because users often enter in the url at the top and our app uses the "/awards" path to go to the awards page. To verify a successful load of the page, the test gets the title of the page which is unique to that page, if Selenium navigated to the page that was specified it is a success.

controller_get_contact(browser) - this test was chosen to test the controller. When a user enters https://localhost:3000/contact or the corresponding deployment url, we expect that the contact page is returned. We choose this test because users often enter in the url at the top and our app uses the "/contact" path to go to the contact page. To verify a successful load of the page, the test gets the title of the page which is unique to that page, if Selenium navigated to the page that was specified it is a success.

controller_get_events(browser); - this test was chosen to test the controller. When a user enters https://localhost:3000/events or the corresponding deployment url, we expect that the events page is returned. We choose this test because users often enter in the url at the top and our app uses the “/events” path to go to the events page. To verify a successful load of the page, the test gets the title of the page which is unique to that page, if Selenium navigated to the page that was specified it is a success.

controller_get_lockers(browser) - this test was chosen to test the controller. When a user enters https://localhost:3000/locker or the corresponding deployment url, we expect that the lockers page is returned. We choose this test because users often enter in the url at the top and our app uses the “/locker” path to go to the locker page. To verify a successful load of the page, the test gets the title of the page which is unique to that page, if Selenium navigated to the page that was specified it is a success.

controller_get_meetingrooms(browser); - this test was chosen to test the controller. When a user enters https://localhost:3000/meetingRooms or the corresponding deployment url, we expect that the meeting rooms page is returned. We choose this test because users often enter in the url at the top and our app uses the “/meetingRooms” path to go to the meetingRooms page. To verify a successful load of the page, the test gets the title of the page which is unique to that page, if Selenium navigated to the page that was specified it is a success.

login_to_admin_page(browser) this test was chosen to test the controller through the login process and verifies that an authorized user is permitted to login. We chose this test because it is important to ensure that only authorized users can use the Banner Editor and it shows that they can get to the Banner Editor. To verify a successful load of the page, the test gets the title of the page which is unique to that page, if Selenium navigated to the page that requires login it is a success.

connected_to_db(database) - this test was chosen to test the database connection to MongoDB. We chose this test because the app needs to be able to connect to the cloud MongoDB service to perform data operations. To verify a successful connection, we try to get the name of the database that all of our data is stored in. If we can get the title of the database, that means we are connected to the cloud MongoDB service.

banneritem_create(database,browser) - This test was chosen to test the creation of a Banner Item using the Editor’s form. We chose this test because we need to make sure that we can navigate to the create portion of the editor and are able to use the form to create a BannerItem. This test actually tests that we can get to the create portion of the editor app, another test actually tests that the item is successfully created after using the create form. If we can get the title of the create page, that means we navigated to it which is a success.

Banner ItemCreated Posted(database) - This test was chosen to test that the Create form successfully posts a new Banner Item to MongoDB. We choose this test because we need to make sure that the information that was entered in the form is actually saved in the database and that we can retrieve it from the database. If we are able to get back a JSON response from the database with the correct values that we entered into the form, it was successfully created.

banneritem_details(database,browser) - This test was chosen to test that using the Details/:id route returned the correct details for that item. We chose this test because it is important that we can access the Details portion of the app and that the correct information is displayed for the id of the object that you specify. If we are able to navigate to the correct page with the id specified, we can get the title of the page, which means that the page successfully loaded.

banneritem_edit(database,browser) - this test was chosen to test that we can get the edit page for the item id that is specified. We chose this test because it is important to know that the controller method to get to the edit page for the specified id works. If we are able to get the title of the page, we know it successfully loaded which means it was a success.

banneritem_delete(database,browser) - this test was chosen to test that we can get the delete page for the item id that is specified. We chose this test because it is important to know that the controller method to get to the delete page for the specified id works. If we are able to get the title of the page, we know it successfully loaded which means it was a success.

BannerItemEditSuccess(database,browser); - this test was chosen to test that we can edit an existing Banner Item in the database, and that the values from the edits are saved to the database. We chose this because it is important to know that using the edit portion of the app is able to take what the user enters and updates the original record to show the new data. If we are able to get the edited JSON data to match what we are expected, it successfully edited the data.

Delete_BannerItem_From_Database(database,browser); - this test was chosen to test that we can delete an item from the database. We chose this because it is important to know that we can use the delete portion of the app to get the specified id and remove it from the database if we no longer need it. If we try and get the data for a specified id and nothing is returned after the deletion, we know that the deletion had occurred.

mobile_design_displays(mobilebrowser) - this test was chosen to test that when the screen is at a certain size (the size of a mobile device), the site should change how it is displayed to accommodate that device. This is important to test for because the client wanted a web app that appeared as a native app, so we designed the app to look a certain way on a phone. To test that it resized properly I placed a div inside of the screen that is invisible at all sizes, however the media query that the css uses to resize to a mobile screen removes this div from the DOM. If Selenium can’t find the visibility of this div in the DOM, we know that the media query occured and that the resizing for mobile screens happened.

Results & Analysis of Test Cases

In this section we will cover if the tests were successful and why. If they weren't always successful, an explanation as to what happened and the impact of the failure. Many of these tests have notes that they have rare failures due to Selenium running too fast, there are thread sleeps in the tests that are particularly prone to this problem but it can still happen if the internet connection is too slow to finish loading the JavaScript. The thread sleeping was the best work around this most of the time since there necessarily a one best way to account for slow internet connection. You can view the successes as a table in the figure below the description list.

1. controller_get_index(browser) - Success: This test was able to get the unique title of the expected document after the page had been loaded. Note: Very Rarely will this fail and this is due to Selenium sometimes trying to get the document title before the document is fully loaded

2. controller_get_login(browser) - Success: This test was able to get the unique title of the expected document after the page had been loaded. Note: Very Rarely will this fail and this is due to Selenium sometimes trying to get the document title before the document is fully loaded

3. controller_get_awards(browser) - Success: This test was able to get the unique title of the expected document after the page had been loaded. Note: Very Rarely will this fail and this is due to Selenium sometimes trying to get the document title before the document is fully loaded

4. controller_get_contact(browser) - Success: This test was able to get the unique title of the expected document after the page had been loaded. Note: Very Rarely will this fail and this is due to Selenium sometimes trying to get the document title before the document is fully loaded

5. controller_get_events(browser) - Success: This test was able to get the unique title of the expected document after the page had been loaded. Note: Very Rarely will this fail and this is due to Selenium sometimes trying to get the document title before the document is fully loaded

6. controller_get_lockers(browser) - Success: This test was able to get the unique title of the expected document after the page had been loaded. Note: Very Rarely will this fail and this is due to Selenium sometimes trying to get the document title before the document is fully loaded

7. controller_get_meetingrooms(browser) - Success: This test was able to get the unique title of the expected document after the page had been loaded. Note: Very Rarely will

this fail and this is due to Selenium sometimes trying to get the document title before the document is fully loaded

8. login_to_admin_page(browser) - Success: This test was able to properly login an authorized user and directs to the index of the Banner Item editor. Notes: Sometimes this test fails if it ends up running too fast, what happens is sometimes Selenium doesn't get the sendKeys values entered correctly so when it submits it, the app handles it as an incorrect user and directs back to the login page, which is a proper outcome with no negative impact, but does cause the test to fail sometimes.

9. connected_to_db(database) - Success: This test was able to connect to the database and get the title of the database that we are working in.

10. banneritem_create(database,browser) - Success: This test was able to get the unique title of the expected document after the page had been loaded. Note: Very Rarely will this fail and this is due to Selenium sometimes trying to get the document title before the document is fully loaded

11. Banner ItemCreated Posted(database) - Success: This test was able to check that the values sent by Selenium in the banneritem_create method did in fact create a new Banner Item in the database with the proper values.

12. banneritem_details(database,browser) - Success: This test was able to get the unique title of the expected document after the page had been loaded. Note: Very Rarely will this fail and this is due to Selenium sometimes trying to get the document title before the document is fully loaded.

13. banneritem_edit(database,browser) - Success: This test was able to get the title of the document for the test item's id, after the page had been loaded. Note: Very Rarely will this fail and this is due to Selenium sometimes trying to get the document title before the document is fully loaded.

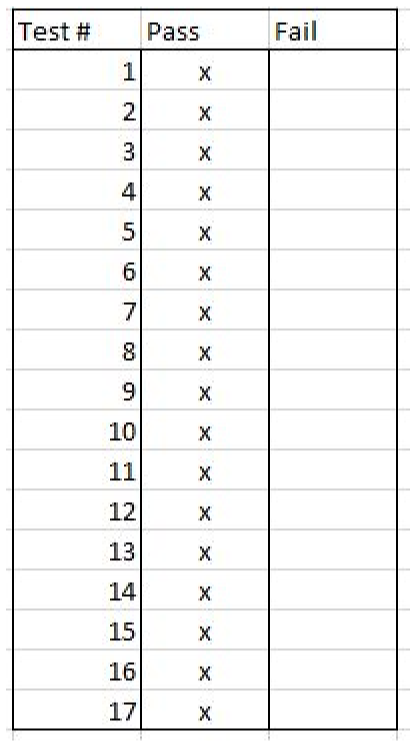
14. banneritem_delete(database,browser) - Success: This test was able to get the title of the document for the test item's id, after the page had been loaded. Note: Very Rarely will this fail and this is due to Selenium sometimes trying to get the document title before the document is fully loaded.

15. BannerItemEditSuccess(database,browser) -Success: This test was able to get the JSON string back from the database with the updated information that was entered by Selenium into the edit form for the test item's id.

16. Delete_BannerItem_From_Database(database,browser) - Success: This test was able to get an empty JSON string back when searching for the test item’s id, meaning that it was removed from the database successfully.

17. mobile_design_displays(mobile browser) - Success: This test was able to confirm that an element that should be removed from the DOM with a CSS media query did in fact get removed, so it was resized to fit a mobile screen.

Table of Passes and Fails



Test #	Pass	Fail
1	x	
2	x	
3	x	
4	x	
5	x	
6	x	
7	x	
8	x	
9	x	
10	x	
11	x	
12	x	
13	x	
14	x	
15	x	
16	x	
17	x	

Limitations

The client expressed that they would like to be able to easily change the hours, images, contact information, and other things without having to code. Since we are not able to recreate a Content Management System, this is not possible. They will have to have someone to go in and change things manually in the code. This has been addressed in the Developer’s manual in which we mentioned that we have an images folder in the public folder. The developer should be able to find the images and go to the code where it is documented to make these changes.

Deployment

Live Deployment: https://hughesfieldhouse.herokuapp.com/
Code Repository: https://github.com/mwoolery/project-charter-template
Test Code Repository: https://github.com/mwoolery/HughesFieldhouseTests
Travis CI: https://travis-ci.org/mwoolery/project-charter-template

User Manual:
https://docs.google.com/document/d/1QEIwG14MR_RyRn_h9wo9dXy6MYG-UV3TI4VVSJzP5po/edit?usp=sharing

Developers Manual:
https://docs.google.com/document/d/19MAh3oS5gMTZ7Ip2oIPKf3O6tNiIPYC3TU_0t7-A3EI/edit?usp=sharing

All of the relevant code and deployment information can be found above. Links to view the User Manual and the Developers Manual also included above. Source code Documentation can be found in each of the code repositories.