4-17-2018

Lee Honors College Mobile Application

James Ward
Western Michigan University, jamesignatius54@gmail.com

The
Lee Honors College Mobile App
Guide to Everything

This document does its best to describe the setup, usage and maintenance of the Lee Honors College mobile application. This app was created in the Spring of 2018, by Benjamin Campbell, James Ward, and Peter Shutt for their senior design project.

Front End

The front end is a mobile application made with the React Native framework. This framework is mostly written in the ES6 standard of javascript, with some special syntax unique to React Native. Documentation for this framework can be found here: https://facebook.github.io/react-native/ . Along with the base libraries, several dependencies have been added. Below is a list of these dependencies and their current documentation links.

lodash: ^4.16.4, https://lodash.com/docs/4.17.5
react-native-elements: ^0.19.0, https://github.com/react-native-training/react-native-elements
react-native-vector-icons: ^4.5.0, https://github.com/oblador/react-native-vector-icons
react-navigation: ^1.0.0-beta.22, https://github.com/react-navigation/react-navigation
react-native-dotenv: ^0.1.1, https://github.com/zetachang/react-native-dotenv
react-native-scripts: 1.8.1,
https://github.com/react-community/create-react-native-app/tree/master/react-native-scripts

Note: These dependencies are listed in the source code in the file 'Client/LHCApp/package.json'. Updates to these libraries can be done automatically but with caution. Updating a library to a new version than what was used for this project may break some functionality. The package manager npm can be used to update and install all of the dependencies listed in a directory. Documentation on npm can be found here: https://docs.npmjs.com/

> **With npm installed, to install the dependencies:**
> Open a terminal, navigate to the directory with the package.json file, run "npm install"
> **to update the dependencies <<caution>>:**
> Open a terminal, navigate to the directory with the package.json file, run "npm update"

Coding was done using notepad, sublime text, and other text document writers. Development was conducted with the use of the Expo toolchain. Expo was created specifically for React Native and provides several functionalities for building, running, testing, and publishing apps. Documentation on Expo can be found here: https://expo.io/

Pages

The app is broken up into four main pages: **Home, Contacts, Resources**, and **Courses.** All of the pages except for home have sub-pages that display data from one of the main pages in greater detail. The pages are listed below, with their subpages --also referred to as expanded pages-- and a brief description of what they display.

Home: no expanded pages. Shows a social media news feed similar to what is found on Twitter and Facebook. The displayed data can be filtered by the originating social media feed by pressing on the gear icon in the banner, and selecting one of the options. Clicking on the social media post will navigate away from the app to the phone's default browser to open the post in the actual social platform.

Contacts: expanded page **ContactExpanded**. Displays a list of faculty members in a searchable contact list. To search the list, press on the magnifying glass, and enter the search word into the text box that appears. Pressing on a contact with navigate the user to the expanded page where details including: name, title, office phone, email, and pronouns will be displayed.

Resources: expanded pages **EventExpanded, GeneralInfo.** This page has three selectable tabs, Building Information, LHC Sponsored Events, and Links. Pressing building info, or links will navigate the user to the GeneralInfo page where a list of information cards are displayed. For example, pressing link will navigate to a page with useful links displayed on cards. Pressing one of these links will function like a regular link. Pressing the LHC Sponsored Events tab will navigate to the EventExpanded page. This page is similar to the home screen except that it only contains information cards on LHC sponsored events.

Courses: expanded page **CoursesExpanded.** This page functions like the contacts page. A list of courses is displayed in a searchable list. To search the list, use the search icon like before. This page also includes a disclaimer for the accuracy of the course listing. It can be accessed via the *Disclaimer button towards the bottom of the screen. Pressing on a course will navigate the user to the course expanded page. This page displays details about the course including: subject, number, title, description, and sections.

Implementation

Here you will find implementation documentation of the front end. The root directory of the front end is "Client/LHC_App/". Generally, a page is defined as a component of the larger app. These components may have functions inside of them that aid the main function **render().** This function is what controls what gets displayed on the screen for that page (component). What page gets displayed is controlled by the main file **App.js.** This file contains all of the navigation routes for the app, and serves as the entry point for when the app starts up. The styling of the pages is stored in the file **styles.js.** This file can be found with the other pages under the directory "Client/LHC_App/src/"

App.js: "Client/LHC_App/" mentioned above, holds the navigation logic of the app. All pages and subpages must be imported to this page to be added to the navigation routes. Pages that require sub-pages must be defined as a const StackNavigator. This allows the per-page navigation alongside the encapsulating TabNavigator that manages the four main pages. The

const StackNavigators are added to the TabNavigator definition like a normal page after it has been defined; see code for examples.

HomeScreen.js: “Client/LHC_App/src/” Displays a social media news feed with data pulled via an asynchronous api fetch request. This is a request sent to the backend in the form of a url, and responded to with a json file containing the requested data. This file is then parsed and displayed in the page’s view in the render function. All pages use this style of data fetching and displaying. This page, and the EventExpanded page use a technique called ‘paging’ where not all of the data is requested at once. Instead the data is returned in pages, one for each request. This is useful for when loading all of the data at once would be costly. This paging is handled by the fetchData function in tandem with the handleEnd function. handleEnd is called once a user has scrolled to the bottom of the already loaded data, this triggers another page request, which returns another page and adds it to the end of the loaded data. This page also filters this data post-request with the functions Twitter, Facebook, and All. Each of these filter the loaded data based on their search value i.e Twitter filters and returns all posts from Twitter.

Contact.js: “Client/LHC_App/src/” Displays a listing of LHC contacts, all with an associated onPress function. Using the onPress function, the page will navigate to the ContactExpanded page, while passing the specific user data to display. Contains the function _handleResults used to set the displayed data to a search result filter returned by the SearchBanner component.

> **ContactExpanded.js:** “Client/LHC_App/src/ExpandedViews/” Used to display detailed contact information. This information need to be passed to it as an object. This page has no other functionality other than text display.

Resources.js: “Client/LHC_App/src/” Simple three card page. Once a card is pressed, it will navigate via the StackNavigator to the corresponding expanded page, passing along what data is being requested to be displayed.

> **EventExpanded.js:** “Client/LHC_App/src/ExpandedViews/” handles requests to display a list of events on the database. Executes API fetch data calls with the method “fetchData” in a async fashion. The data returned from this is appended to the end of the current event list. This allows for page loading like the Home page. Once no more data is returned from the API call, the page with set the “EOF” --End Of File-- flag and will no longer attempt to load more until the page is reloaded.
>
> **GeneralInfo.js:** “Client/LHC_App/src/ExpandedViews/” handles requests to display general key-value pairs of data in a list. This page loads all pairs at once since this is a small volume. May need to be updated to use the page system if the database grows too large. The pairs are loaded onto floating cards. If the pair represents a link, pressing on that card works like a link. This is made possible by the function “Linking.openURL” that takes a link, and uses the phone’s own ability to open the link in a browser.

Courses.js: “Client/LHC_App/src/” Displays top level data about courses. All course titles are loaded at once like the GeneralInfo page and has the same warning. Using the onPress method, the page will navigate to the CourseExpaned page, while passing it the course id to to fetch the details on. The disclaimer is controlled by a component called a Modal. This component is show and hid by a state set by the button, located at the bottom of the file, and on the modal. Contains the function _handleResults used to set the displayed data to a search result filter returned by the SearchBanner component.

> **CoursesExpanded:** “Client/LHC_App/src/ExpandedViews/” Used to display detailed information on a the course identified by the couse_id value passed to it. Uses a fetch api to grab the single course and its details from the backend and displays them in a card format. This page has no other functionality other than text display.

Banner.js/SearchBanner.js: “Client/LHC_App/src/” These are the helper components that are included into the other pages to display the banner that sits on the top of each main page. The base component Banner, takes in a title to display on the banner. The more specialized banner SearchBanner, starts with this and then adds on the required logic to allow the banner to function as a search bar for the page that is including it.

Example usage:

```
<Banner title='Resources'/>
<SearchBanner
    title='Courses'                          >> Optional title for the banner
    handleResults={this._handleResults}  >> Ref to local method that displays results
    data={this.state.data}                    >> The base data to start search on
/>
```

The handle results function is called by the SearchBanner component and is passed the search results in the form of an array. In the pages, this array is what gets passed to the flat list to be displayed.

Publishing and Distributing the App

The Expo toolchain is the main point for creating, running, publishing, and distributing the app. This may sound like a difficult task, but Expo makes it as simple as possible. After Expo creates the app binaries they can be sent to the Android Play Store or the Apple App Store.

Publishing is the act of making the app available through Expo. This is not the same as distributing the app to app stores, but can be a useful stepping stone to see how the app will look and run. Published apps can only be accessed via links to the Expo community repository, usually distributed as invites from the publisher. The published app can also be used with a simulator to view changes with hot reloading.

With Expo it is possible to generate the binaries for the iOS and the Android versions of the app. View this link to reference the intricacies of creating the binaries with Expo: https://docs.expo.io/versions/v27.0.0/guides/building-standalone-apps.html. The basic method that we used was to issue a command that would create the binaries and store them in the cloud on an Expo sever. The command line would show a link to the binaires where it can be downloaded to the phone. The command that was used for Android was: “exp build:android”. For iOS the command was “exp build:iOS”. Installing an app this way also prompts the phone’s security as the app is not coming from a trusted site. This can be overridden by turning off the security check in the phone’s setting.

To distribute the app to app stores, there are several steps and requirements to fulfil. The Expo toolchain comes to the rescue here again. Using it, we can easily compile our javascript code into the Android .apk, and the Apple .ipa binary files that for all intent and purposes are the app. Documentation for this process can be found here: https://docs.expo.io/versions/v27.0.0/distribution/index.html. Of course there are still other requirements such as sample images, accessibility, content control, and many others. Check with the specific app store for their requirements, as they can change from store to store. For most of the requirements, Expo can help.

Sending the app to the iOS App Store and the Android Play Store necessitates having a developer account for the respective stores. For this project we worked with WMU’s University Relations in order to be added to their developer accounts to deploy the app. This guide can be followed for pushing the app to the Android Play Store: https://docs.microsoft.com/en-us/xamarin/android/deploy-test/publishing/publishing-to-google-play/?tabs=vswin. The guide for publishing the app to the iOS App Store can be found here: https://help.apple.com/itunes-connect/developer/.

Back End

The back end of the LHC Mobile App is running a Django web framework in conjunction with an Apache Web Server. Django provides all of the tools necessary to store and retrieve data from a SQLite database, and Apache serves this data. A few quick notes on what versions these were developed on:

Ubuntu Server 16.04
Django: 2.0.1
Python: 3.5.2
Apache: 2.4.18

Setup and Installation

Assuming that you are starting from a fresh server, the first order of business is to clone the repository from Bitbucket. Enter the project's page and click on the "Clone" button near the upper right. A window will pop up with some selected text; this text is the command you will run in command line to clone the repository down to the machine. Simply open a command line (if on a server with a GUI) and paste this in the desired directory to download.

After this has happened, a few steps need to be done to initialize the project:

1.) Make sure required libraries/packages are installed
2.) Make sure the SQLite Database is set up and static files collected
3.) Make sure Apache and Mod_WSGI are installed and configured

1.) There are not many libraries/packages required, the most important is of course Python 3 and Django being the most obvious:
    python3.6
    python3-pip
    django
    Pillow
    Twitter
    Facebook

Python 3 should already be installed, but it and Pip can be installed with apt-get:

```
sudo apt-get install python3.6
sudo apt-get install python3-pip
```

Pip is used to install Python packages.

The next four should be installed with pip

```
sudo pip3 install django
sudo pip3 install pillow
sudo pip3 install python-twitter
sudo pip3 install facebook-sdk
```

Pillow is used by Django for handling the uploading of image files, it is required as well.

2.) This is a simple step, there are two commands that need to be ran by Django in order to make sure the database is set up (these are ran from the base project folder that has Django's manage.py file):

```
python3 manage.py makemigrations
python3 manage.py migrate
```

(The 3 may not be necessary, but it guarantees that it will run the file in Python 3 and not 2)

The first command will search through the various models.py files in the Django app folders and construct the migration files that will create tables in the databases. The second command will actually create the proper tables in the database.

Next, we need to collect static files into the STATIC_ROOT folder. First, go into lhc_app_server/lhc_app_server/settings.py and make sure there is a string variable declared at the bottom called STATIC_ROOT that is set to a string representing the absolute path to a folder on the server that will contain your static files. In our project, this path is:

```
STATIC_ROOT = "/home/seniorDesignApp/static/"
```

This will be important in the next step.

The last step here is to make sure there is a superuser defined, and it's easy enough to do (run this also from the Django project root folder that contains manage.py):

```
Python3 manage.py createsuperuser
```

This will prompt you for a username and password for a user account that has access to add, delete, and modify the objects in the Django database, so one of these is necessary.

3.) First of all, this is a very helpful reference page in Django's documentation:
https://docs.djangoproject.com/en/2.0/howto/deployment/wsgi/modwsgi/
Django's documentation contains tons of great info, including basic Apache set up configurations. There isn't much that needs to be done, but it is important to be done correctly.

The first step is to make sure that Apache is installed. This is is done simply enough with `apt-get`. The command to update Apache2 is:

```
sudo apt-get update && sudo apt-get install apache2
```

Also install Mod WSGI for Python 3

```
sudo apt-get install libapache2-mod-wsgi-py3
```

After Apache is installed, you're going to need to add some configurations to the main config file. On our server (Ubuntu 16.04) this is located in /etc/apache2/apache.conf. You'll need to add a few lines to this file (at the bottom is fine) to make Django work properly, I've literally copy and pasted them below:

```
Alias /static/ /home/seniorDesignApp/static/
```

```
WSGIScriptAlias /
/home/seniorDesignApp/SeniorDesign_LHCApp/Server/lhc_app_server/lhc_app_se
rver/wsgi.py
WSGIPythonPath
/home/seniorDesignApp/SeniorDesign_LHCApp/Server/lhc_app_server

<Directory
/home/seniorDesignApp/SeniorDesign_LHCApp/Server/lhc_app_server/static>
Require all granted
</Directory>

<Directory
/home/seniorDesignApp/SeniorDesign_LHCApp/Server/lhc_app_server/lhc_app_se
rver>
<Files wsgi.py>
Require all granted
</Files>
</Directory>

<Directory /home/seniorDesignApp/static>
Require all granted
</Directory>
```

I'll describe what this does below:
The first line routes any traffic going to lhcapp.honors.wmich.edu/static/ to the folder /home/seniorDesignApp/static/. This is the absolute path to the folder that Django puts all of its static files (css, javascript, images) inside of. This directory is defined in the `lhc_app_server/lhc_app_server/settings.py` file as such:

```
STATIC_ROOT = '/home/seniorDesignApp/static/'
```

Notice that the paths are the same for both the STATIC_ROOT and the Alias in the apache.conf file, they must match. When you run the command `python3 manage.py collectstatic`, all of the static files needed by the server are collected from their various folders and put inside the folder defined in STATIC_ROOT. We then alias any traffic with a /static/ in the URL to go to that folder; simple.

The next line, WSGIScriptAlias, routes traffic of the url / to the file lhc_app_server/lhc_app_server/wsgi.py. This file is what allows Django to work within Apache. As long as this Alias is in the configuration, everything will work just fine.

WSGIPythonPath ensures that any code calling `import lhcapp` or `from lchapp import x` will work properly.

The next three chunks give request to the necessary directories that Django files are kept in.

You also need to make sure that Ubuntu's default firewall software, wtf, is allowing traffic for Apache. This can be achieved simply:

Check firewall-allawable applications: `sudo ufw app list`
Allow Apache Full: `sudo ufw allow 'Apache Full'`
Verify Apache Full is Allowed: `sudo ufw status`

The last step is to verify that the service itself is running:

```
sudo systemctl status apache2
```

The server should be ready to go at this point. There are a few commands you'll want to remember for starting, stopping, restarting and reloading Apache. Anytime the Django code is changed, the Apache service will need to be reloaded:

sudo systemctl start apache2
sudo systemctl stop apache2
sudo systemctl restart apache2
sudo systemctl reload apache2

General Outline

The Django project is broken into a few folders, called Apps: **courses**, **events**, **social**, and **staff**. There is also another folder with the same name as the project: **lhc_app_server**. This folder contains a few special files.These apps each contain more or less separate data, each relating to a part of the app:

> **Lhc_app_server**: The entry point for the project.
> **Social**: The social media feed
> **Staff**: The faculty/staff page
> **Events**: The events page
> **Courses**: Courses and sections

Each of these folders are almost identical in how they are set up, with the exception of **lhc_app_server**. I'll cover the important files (there are a few other files you won't need to deal with).

```
lhc_app_server/
      |-settings.py
      |-urls.py
```

```
|-views.py
|-wsgi.py
```

The `settings.py` file contains information crucial to the Django project (more here https://docs.djangoproject.com/en/2.0/ref/settings/). There are only a few lines you should need to worry about that you'll see:

> `DEBUG = False`
> This mode will show detailed error information when receiving HTTP error codes in your browser. It is HIGHLY recommended that this is ALWAYS set to False when in production, so people cannot get sensitive information.
>
> `ALLOWED_HOSTS`
> This is a list of IP addresses that are allowed to access the application without getting a 404 error. For this app, what's important is that `'lhcapp.honors.wmich.edu'` is among these, as that is what the app uses to make requests to the server. I did, however, leave the actual IP address and local IP address here just in case.
>
> `STATIC_ROOT`
> As explained above, this variable is set to the folder that `collectstatic` will place all static files into.
>
> The other settings shouldn't really ever need to be messed with, but Django has a massive host of documentation should the need arise.

`Urls.py` is the first file that, along with the other urls.py files in the other app folders, directs the traffic coming through to the server. Here's the great Django documentation on it: https://docs.djangoproject.com/en/2.0/topics/http/urls/. Essentially these `urls.py` files take the URL requests from the app, such as `lhcapp.honors.wmich.edu/api/courses/all/`, and break it down and route it properly.

A quick look at the `lhc_app_server/urls.py` file will show you that there is a line for several different URLs; any with `api/events/` are then routed to the `events/urls.py` file, any with `api/staff/` are routed to the `staff/urls.py` file, etc. These `urls.py` files will then route these various URLs to methods that are called Views (more on those here: https://docs.djangoproject.com/en/2.0/topics/http/views/) As you can see, a few of the lines in the `lhc_app_server/urls.py` file do not route to other files, some call a view directly, such as for `api/info/building/`, and `api/home/`. A better-designed app would've avoided this, but it's fine.

`Views.py` contains the code that retrieves data from the database using Django's model interface, and returns the JsonResponses used by the app.

The other app folders, such as `social/` and `staff/`, contain another important file, called `models.py` that defines how the database tables are set up, they define what information we hold for each Object that we create, such as a staff member, a course, a section, or an event. More on models here: https://docs.djangoproject.com/en/2.0/topics/db/models/

So, at a base level: the models.py files define Models in the Database. An HTTP request comes in for some url, say `/api/courses/all`. First, Django checks the `lhc_app_server/urls.py` file for any matching url, which it does: `/api/courses/`. So it sends the request to the `courses/urls.py` file with the remaining part of the url: `all`. This url says to call the `all_courses` view, which is defined in `courses/views.py`. That view will gather whatever data is necessary from the database, create a JSON object, and return it as a JsonResponse.

Again, the Django documentation is **fantastic** and I **highly** recommend taking a night or two to go through the tutorial: https://docs.djangoproject.com/en/2.0/intro/tutorial01/. It's probably one of the best programming/framework tutorials I've ever gone through, and Django at its core is pretty simple, so everything you need in order to understand how the backend words will surely be found in there. Especially since our webapp does not serve webpages; only JsonResponses. The only logic is pulling items from the database and putting them in a JSON object to return.

Social Media Script:

Introduction:
The social media script will pull posts from Twitter and Facebook and will add them into the database. This is done through a script written in Python3. The libraries should be included in the above installation so the script should be ready to go. This script is run from a cron job every night so a new installation will mean that this needs to be reinstated.

Running the Script:
The command to run this script is "python3 manage.py socialAPI". The cron line looks like this: "30 2 * * * cd /home/seniorDesignApp/SeniorDesign_LHCApp/Server/lhc_app_server/ && python3 manage.py socialAPI >> socialAPIOutput.txt &>> socialAPIError.txt && python3 manage.py collectstatic <<< yes && chgrp -R www-data /home/seniorDesignApp/static/".

Location:
The script is located here:
"SeniorDesign_LHCApp/Server/lhc_app_server/social/management/commands/socialAPI.py"

Token:
The APIs use a token system that is connected through a social media application. We created an application on the Lee Honors Colleges Facebook and Twitter accounts. Once we had the

application we received a token which is stored in the script to pull the social posts. The Facebook token will expire every 60 days unless it is used. Since our cron job will run everyday the token will be refreshed and we won't have to worry about it expiring. There is no expiration for the Twitter token. In the event that a token needs to be refreshed manually, a system administrator will need to modify the social script to enter the new token.

Documentation Links for The Social Media APIs:
Twitter API documentation: https://github.com/bear/python-twitter
Facebook API documentation: http://facebook-sdk.readthedocs.io/en/latest/index.html