UI Framework

Summary

In order to have an easy solution to develop a simple and consistent UI, we decided to use React as our UI library of choice. React has a strong community that has been providing support since 2013 and has only been growing since then. Many popular social media websites such as Instagram and Facebook incorporate the use of React, as well other known learning sites such as Khan Academy and Codecademy.

Problem

Picking a framework/library is like choosing the material to build: choose something that is cheap and easy, like wood, then it'll work for a while, but is still vulnerable to a variety of external factors. In the same way, choosing a material that's over the top for the product can be too offputting; you wouldn't try to use nice, contemporary marble for a simple country house.

To pick a framework/library is to keep this - and more - in mind. It has to be the best fit for the product, needs to work well with existing products, has to achieve a good level of optimization and performance, and has to be relatively easy to pick up. However, even if something may be best fit in all these categories, feasibility under a certain time limit can throw all of this away. In this sense, picking framework/library is really a trade-off among all these categories.

Constraints

- Clients want to keep the scope as a webapp, not intending for native apps.
- Different browsers have various ways of handling element rendering. This creates various complications and forces the programmer to potentially use older versions of a feature.
- Time to learn a framework/library.

Options

Framework and Libraries:

- Angular 2
    - Pros:
        - Currently gaining popularity due to its high performance rate.
        - Created and supported by Google, a massive and well known company that prides in pushing quality products.
        - Created with modern web practices in mind (lazy loading, nested components, etc.)
    - Cons:
        - High learning curve that may require too much time for such a short semester.
        - Requires knowledge of Typescript, a relatively new programming language compared to JavaScript.
        - Relatively new framework (2016) that still has some areas that do not have the necessary support yet.
- React
    - Pros:
        - Familiarity working with React within group
        - Medium learning curve.
        - Backed and supported by Facebook.
        - Incredible reusability with its “component” approach to modern web pages.
        - Search engine optimization (SEO) and React Virtual DOM have led to performance enhancement.
        - Browser-friendly debugging tools by Chrome and Firefox allows a better environment to examine React’s complex hierarchy.
        - Built upon the already familiar language, JavaScript.
    - Cons:
        - Relatively new (2013), so it faces the same issue as Angular of having areas that aren’t documented too well.
        - JSX (Javascript XML) is an initial barrier and takes time to be familiarized with.
        - Constantly changing environment may lead to entire changes to a product’s existing hierarchy (e.g. React’s “hook” functions).
- Vue
    - Pros:
        - Low learning curve compared to Angular and React.
        - Backed by a dedicated community that provides constant support.
        - Contains a built-in router as well as in-state management.
    - Cons:

        - Relatively new (2014), so it faces the same issue as Angular and React as mentioned before.
        - Development is slower since it's backed by a community rather than a large company.
        - Deals with the controversial idea of being overly flexible and containing too many options.
        - Literal language barrier: seeking advice from Vue.js experts is a bit harder as it is gaining more popularity in China than other countries.

Design Tools:

- Bootstrap
    - Pros:
        - Low learning curve.
        - Still supported and built by an active community with an active release just a little over a month ago.
        - Relative ease to import and use - really just a matter of using the right id for each tag.
    - Cons:
        - Selection of styles is limited compared to those of Material.
        - Reliance on Bootstrap can lead to relatively similar results; more customization is required in order to produce intended effects and appearance.
        - Relatively infamous for its 'too verbose' style in HTML.
- MaterialUI
    - Pros:
        - Provides support design components for Angular, React, and Vue.
        - Most websites use Material to design their own website - this helps capture the idea of familiarity among websites.
        - Creates a more tactile and friendly user interface.
        - Idea of not reinventing the wheel.
    - Cons:
        - Over-reliance can lead to excessive importations.
        - Since Material is used for familiarity, it's not for those who wish to create a unique user experience.

Rationale

In the end, we chose to use React as our library of choice for frontend development. Out of the three, React has been released the longest, allowing more support in case of unsuspecting problems. It ranks as intermediate difficulty out of the three, but its advantages far outweigh this shortcoming: it's reusability and in-browser bug support allows for a faster development pace while also preserving the quality of the product. In addition, while JSX may prove to be difficult at first, the fact that it's really just the consolidation of HTML and JavaScript, a familiar markup language and programming language, respectively, means that its learning curve is not that high.

While Vue does have an easier learning curve, it's outmatched by React and Angular's performance, and since it's not backed up by a major company like the other two, it is more prone to more bugs and issues in the future. Some of its advantages - such as in-state management and built-in routers - are also available in React, albeit with a little more implementation effort required.

Angular - while powerful and effective - the amount of work required to learn both a programming language and an entire framework may prove to be too time-consuming. Its size is also not fit for the product at hand, and the use of Angular may be more than necessary.

As for the design tools, we will primarily use MaterialUI's components with the usage of Bootstrap for pages that are relatively more static.

Web Framework

Summary

In order to have an out of box solution for developing a scalable and secure webapp, we decided to use a web framework, Django. Django has a suite of tools essential for backend architecture along with an active developer community. Many companies use Django in their web development stack, showcasing its usefulness and reliability.

Problem

Creating API endpoints, designing backend databases and user authorization is a tedious and critical process of creating a good webapp. If done wrong, security breaches and mishandled data can easily arise. Utilizing a framework gives us a toolkit to design these structures

Constraints

- Any and all tools/software used in the solution must be free/open-source

Options

- Django
    - Pros:
        - Object Relational Model gives development process flexibility in database choice. There is also no need for Query Language syntax
        - Strong libraries for user authentication, admin interfacing, and other solutions
        - Easy API development and routing
        - Model, View, Template structure makes it easy for templating views
        - Team has familiarity working with Django
        - Active development and support community
        - Python
    - Cons:
        - A bit of a dinosaur when it comes to size and dependencies being pulled in

        - Wandering off of the “blessed path” can be painful or require extra effort
- Ruby on Rails
    - Pros:
        - Small learning curve
        - Convention-Over-Configuration makes development to deployment quick
        - Strong libraries for user authentication, JSON API building, etc.
        - ORM
        - Rails 6 has good security and offers protection against a lot of beginner developer mistakes.
    - Cons:
        - Ruby language unfamiliar to the team
- Flask
    - Pros:
        - Microframework with less abstraction between backend and frontend, increasing performance
        - Flask is simpler and more of a minimalist to Django, allowing for creation of more flexible solutions
        - ORM support and simple database management
    - Cons:
        - Since it is a microframework, there are less tools and libraries supported for Flask
        - User authentication will need to be implemented separately
- Express
    - Pros:
        - Javascript is a language both frontend and backend developer have some familiarity with
        - Node.js makes it easy to add extra resources for project
        - Could go down a fullstack JS route so team can be specialized in JS
        - Simultaneous request handling is better in JS than other languages due to the V8 engine(this isn’t really a big problem in a production setting)
    - Cons:
        - Standard library of Express/Javascript is not great, so a lot of dependencies are needed; a left-pad incident never seems to be too far around the corner
        - Async programming can create inefficient code if done wrong, so knowledge is needed.

Rationale

In the end, we chose to use Django as our web framework of choice for development. Django not only provides an extensive suite of libraries to handle critical functionality such as user authorization and API endpoint creation, but it also utilizes an ORM, providing flexibility for developing backend code for any database management system. In development work, we now have the flexibility to write models for a lightweight DB such as SQlite, and seamlessly deploy the model to production for use by DBMS such as PostGreSQL/MySQL. While Ruby on Rails also provides an “out-of-box” solution and has a small learning curve like Django, our team collectively has familiarity working with Django and Python from other personal projects.

Database

Summary

In order to have an efficient way to handle data transactions, we decided to use PostgreSQL as our DBMS of choice. Having a DBMS is essential for querying the data in a secure and concurrent manner.

Problem

Our web app will have relational data that will need to be displayed on the frontend. The data also needs to be queried and catered depending on the end users. To handle all transactions related to this data, leveraging a DBMS is critical for efficiency and security of the data.

Constraints

- Data is relational
- Any database choice should be open source/non-enterprise
- ibiblio staff receives data from all radio stations
- Radio stations users only view their billing transit
- Data will be stored in a mount location and will be added to the DB

Options

- SQLite
    - Pros:
        - “Serverless” database and doesn’t run as a server process
        - No configuration
        - Portable and lightweight DB solution that is still ACID compliant
    - Cons:
        - Concurrency is constrained, as only one change to the database can happen at any time
        - SQLite is files-based DBMS, so with large datasets, performance will drop drastically
        - Cannot make access permissions for different users

        - Data access and protection from bugs is not as strong as DBs that utilize servers
- PostgreSQL
    - Pros:
        - Optimizes complex queries well
        - Efficient parallel processing capabilities
        - Utilizes Multi-Version Concurrency Control, which is extremely powerful and efficient for concurrency control; each transaction views a snapshot of data
        - Can handle working with large amounts of data (big data)
        - Database can leverage all cores in processors
    - Cons:
        - Expensive on memory when there are alot of client connections, as each new client connection uses up to 10 MB/connection
        - Not efficient it expecting very fast read operations
        - If database is simple, PostgreSQL can be overkill
- MySQL
    - Pros:
        - One of the most popular DB systems, so a lot of support and documentation with active community
        - Most benchmarks show that MySQL is the fastest database system
        - Offers database replication
        - MySQL has built-in scripts and processes to optimize database security.
    - Cons:
        - Not entirely SQL compliant, PostgreSQL is more strict.
        - Not fast when multiple users are reading and writing. Transactions may slow down overall performance
        - MySQL does not handle large database sizes efficiently

Rationale

In the end, we chose to use PostgreSQL as our DB engine of choice. We decided against SQLite for production, since we do not need the portability SQLite offers. The value of using a DBMS for maintaining ACID and concurrency is more valuable than using SQLite. Although our schema is small and the data we will be working with is not on the scale of big data, the performance discrepancies between PostgreSQL and MySQL will be minimal. We also believe PostgreSQL is a great choice for future scalability, and has an active and growing community. Our group also has familiarity working with PostgreSQL, therefore the tradeoff is in favor of PostgreSQL.

Backend Service

Summary

In order to have a backend service for code deployment and testing, we decided to set up our own deployment and staging environments on server environments provided by ibiblio/UNC ITS.

Problem

All of the components of the application need to be hosted in an environment that provides compute and network resources to ensure that the application remains reliable, secure, and accessible. The environment also needs to be serviceable by operators and system administrators after the initial deployment and it would ideally allow for streamlined integration and deployment processes to make updating the application easier and less risky.

Constraints

- Production service must be hosted on-site at UNC
- Debian 10 OS environment
- All provisioning and deployment steps should be scripted, automated, or replicable to allow for the entire solution to be re-provisioned from a fresh OS image as needed
- Client would prefer a containerized solution as that is the general direction that software solutions at ibiblio are moving towards; however, client would like the option of running the solution in alternative OCI-compliant container engines, such as Podman
- All solution components should be easily decoupled from containerization infrastructure (that is, it should be able to run outside of the containerized environment and not depend on any particular feature provided by the container engine)

Options

- Amazon ECS (Elastic Container Service)
    - Pros:

        - Well-documented, with plenty of learning resources provided by Amazon and third parties
        - Default VPC security groups and network ACLs make it security-related misconfigurations more difficult
        - Existing instances can be cloned, making it easy to replicate a container environment
    - Cons:
        - Limited to only Docker as the container runtime and only Docker Swarm as the orchestration platform, unless we choose to spin up our own instance of Podman or some other runtime in an EC2 instance (in which case much of the AWS integrations provided by ECS are not available)
        - Hosting is remote (no on-site hosting option available)
- Azure Container Service
    - Pros:
        - Supports both Docker Swarm and Apache Mesos as orchestration platforms
        - More platform-agnostic integration options are available (e.g. most of Amazon ECS's integrations are with Amazon products; this is not the case here)
    - Cons:
        - Not as well-documented as some other hosting platforms
        - Hosting is remote (no on-site hosting option available)
- Google Container Engine
    - Pros:
        - Powered by Kubernetes, one of the leading open-source orchestration platforms in use on the market today
        - Building upon Kubernetes allows for deployed solutions to be moved away from Google Container Engine in the future if desired
        - Full flexibility of Kubernetes cluster configuration options are available at administrators' disposal
    - Cons:
        - Full extent of Kubernetes complexity is exposed to administrators, which could be overkill and also make debugging difficult
        - Hosting is remote (no on-site hosting option available)
- DigitalOcean
    - Pros:
        - Offers regular virtual server hosting in addition to a new managed Kubernetes service; the latter offers the portability options and avoids vendor lock-in due to the use of Kubernetes

    - Cons:
        - DigitalOcean is newer to the Kubernetes hosting scene than other companies and is a smaller company, so fewer documentation and support options are available
        - Full extent of Kubernetes complexity is exposed to administrators, which could be overkill and also make debugging difficult
        - Hosting is remote (no on-site hosting option available)
- UNC ITS/ibiblio datacenter
    - Pros:
        - ibiblio administrators retain full control over the full tech stack, streamlining future maintenance or migration work
        - We would have the freedom of using whatever container and orchestration platforms would be most suitable or desired by the client
        - Client can offer feedback and make accommodations for specific needs in the hosting environment
    - Cons:
        - Support and documentation found online may not always be accurate, as the environment that we work with will not be an exact, standardized match with someone else's
        - Support response time is dependent on clients' availability

Rationale

In the end, we chose to use the server environment provided by UNC ITS and ibiblio for our hosting purposes. This is mostly due to the client's requirement that the service ultimately be deployed on ibiblio servers. While this does result in more legwork on our part for initial environment setup/getting everything to work, it provides greater flexibility to the client for expanding the application in the future or migrating it into a different environment.