Goal/Purpose of the Prototype

Daniel Murphy 11-June-2021

The goal of the prototype was research to learn the limitations of PWAs and to potentially provide an architectural software basis for a future Terraso PWA. The use case implemented by the prototype is to show a list of documents (which can be images, pdf files, html files, audio/video) that can be viewed within the application, and more importantly saved for offline reading, viewing, and sharing.

A unique challenge for the application is that most of the potential users will be living in world regions where access to the internet is limited, slow, unreliable or expensive (or some combination of thereof). So most of the work I have been doing regarding Progressive Web Apps in React involves utilizing the cache to give the application the functionality to operate within these limitations.

Technical Choices

This section describes some of the technical issues encountered during the creation of the prototype along with the solutions that were incorporated.

Device Interoperability: The application should run on computers, iPhones and Android phones with the same codebase. We briefly considered writing native applications but this would require multiple codebases. PWAs are well supported, lightweight and provide a rich set of libraries.

Internationalization: The application must operate in many languages, not only dealing with translation but also correctly representing date formats, currency, as well as modifying the appearance of the application based on regional conventions. For example, a PWA in Arabic would need to present text in a right to left format.

There are many different tools available in both Javascript in general and React that can be used for translation. I18next is a relatively lightweight and easy to use Javascript library that allows easy translations in the text of an application. The downside to i18next is that it requires creation of translation files (formatted in JSON) for every text field in the application. Once this has been setup all instances of translatable text need to be wrapped in the useTranslation hook.

While I think that I18next served the purposes of my prototype well, I think it would be beneficial to explore more advanced translation tools like Google's Translation API that allow for more automatic translations. For example, using Google's translation API will allow us to add new languages more quickly since the process is automated.

OffLine Operation: As mentioned, prospective users of the application will often be offline or will want to avoid using too much internet data usage. So the ability to cache assets of the application (both the application itself and the documents that are hosted on it) is very important. I have explored several different approaches to caching -

Service Workers:
A service worker is a script that the application runs in the background. While it can be used for things such as push notifications or background sync it can also be used to intercept and handle network requests thereby allowing a PWA to deliver content offline. The service worker is usually used alongside the CacheAPI.

Standard practice is to cache static files when the service worker is installed. It's also possible using EventListeners to Cache files on user interaction and this was implemented in the prototype.

There are several different "strategies" for caching that are commonly implemented - the network falling back to the cache, or cache falling back to the network are most commonly used.

For my prototype I decided to follow the Cache-Falling-Back-to-Network approach because it reduces the number of requests the application is making to the network. However this approach does come with certain drawbacks. For example, if a new version of the application is deployed, someone with the older version won't automatically get the new version when they access the application since they already have an older version cached. Luckily there is a solution for this in Workbox (below).

Workbox:
Workbox is a set of libraries that can power a service worker. Workbox provides many useful methods for writing custom caching logic. Workbox allows the developer to create a "PreCache manifest" of files that are cached when the service worker is installed. It also has the ability to filter what is automatically stored in the cache. For example I can set up a workbox event listener that only caches files with the ".png" extension or files that only match a REGEX expression. Workbox also provides many different useful modules.

Some examples of Workbox modules -

- Expiration Plugin: Allows the developer to create a cache with custom rules that automatically removes entries from the cache when certain criteria have been met. For example a developer can specify the maximum number of entries that can be stored in a cache. When that maximum number is reached Workbox will automatically remove the oldest entries in the cache. The Expiration Plugin also allows the developer to specify a value for "MaxAgeSeconds" which will automatically remove entries from the cache when they exceed that age.

- Workbox Broadcast Update: While responding to requests with cached entries can be fast and save bandwidth it runs the risk that users may end up seeing stale data since the cached content might be older than the version currently hosted on the PWA website. Workbox Broadcast Update works by comparing HTTP headers of the cached and the new responses. The values of header fields can be examined to check for new content. I currently have this module implemented in my code so that when a new version of the PWA is available the user receives a message “there is a new version of the Terasso PWA available” and is given the option to download the new version or continue using the old one.

After researching the above libraries I am confident that the rich functionality provided by them will be sufficient to implement the application’s offline requirements. During my research I also found that many developers are using the service worker/Workbox combination which shows these tools will have development momentum in the future.

GraphQL and External Storage For Documents:

The goal of the prototype is to have all the documents stored on an external repository and to have the PWA itself query information about these documents so it can display information about them. The assumption is that the number of documents will constantly be changing (with new documents being added) so the PWA will need the ability to constantly query the external document index (a database) to get the most recent set of documents. Currently in the prototype I have implemented the following steps towards achieving content delivery to meet these requirements.

Storage: I created an S3 bucket on AWS and dropped all the sample documents into it. The S3 bucket needs to have a cross-origin resource sharing (CORS) configuration enabled that gives the PWA application access to the bucket itself as well as HTTP operations (GET, POST, PUT). If this configuration is not properly enabled the PWA will be unable to cache the document.

GraphQL:

With the documents now stored on AWS S3, a method to query an index of these documents is needed. Researching this topic and comparing RESTful APIs vs GraphQL I decided that GraphQL was the better choice. GraphQL allows you to send a single query to a GraphQL that includes the concrete data requirements vs a REST API where you would typically need to gather the data by accessing multiple endpoints. GraphQL also avoids the issues of “overfetching” and “underfetching” that are common in REST APIS. Using a GraphQL API allows us to save on network bandwidth since all the information can be gathered in a single query while RESTful APIs typically requires queries to multiple endpoints to acquire all the content.

GraphQL was initially developed by Facebook in 2012 because they needed a data-fetching API that was powerful but also simple and easy to use. The following article here written by former Facebook engineer Lee Byron describes how GraphQL was originally created as part of an

effort to rebuild facebook's mobile applications. Specifically, GraphQL is designed to optimize data-fetching in mobile applications.

In creating my own GraphQL server I used several different libraries -

- Apollo Server: open-source graphQL server, compatible with any data source and GraphQL client
- Apollo GraphQL Client: compatible with multiplied devices

Since GraphQL APIs have more understanding of the underlying data structure than a RESTful API several "GraphQL Clients" have been created that can automatically handle batching, caching, and other operations.

The most commonly used GraphQL clients seem to be urql and the Apollo Client. For my prototype I used Apollo Client to learn more about it but then switched to querying the GraphQL server with simple fetch requests instead. I did this since at the moment my GraphQL API is relatively simple and doesn't have many complicated queries or any mutations. However the intention would be to use Apollo client in the future.

Additionally one of the big advantages of using the Apollo Client is the ability to cache queries. At the moment I can cache the query results manually (which also gives me more control/understanding of what's happening).

I believe that in the future when the application's GraphQL API becomes more complex (supporting many different queries as well as mutations) it would be beneficial to use the Apollo Client since it provides built in rich functionality to save time coding.

Design Issues Remaining:

Offline vs Online: (and Wifi vs Cellular): The PWA needs to be able to tell when the application is in offline mode or online mode. This state is required since the application will need to change behaviour based on offline vs. online. For example, if the application is offline we might want to hide documents that are not cached so users do not try to open them while they are offline. Additionally, it would be highly beneficial to be able to obtain information about the network, such as its available bandwidth. This knowledge would allow us to customize operation of the application based on the users current internet limitations. Another area of interest would be knowing when the user changes network connection, so we could re-ask questions about updating the application in case the new connection is cheaper or more expensive. The network information API makes tasks like these possible however at the time of this article (May 2021) the API is only supported in google chrome

Apple vs Android: PWAs have different limitations depending on the platform in use. Probably the most important difference is the storage limit differences between iOS and Android. (available storage for PWAs is considerably less on iOS devices vs. Android devices of similar

physical storage capacity). Also, some libraries that are supported on Android devices are currently not supported on iOS. In general, the iOS environment is much more restrictive for PWAs than Android.

One library that supports querying online state is the Network Information API which can be used to get information about the user's network (such as wifi, cellular, etc.)

Another difference in iOS vs. Android I have encountered during my research is that PWAs can be installed to the home screen from any browser in Android, but on an iPhone you must be using Safari to have the "add to home screen" button appear.

In exploring examples of PWAs I have found that many of them behave slightly differently when accessed on an iOS device. For example one PWA I found has a pop up window that prompts the user to install the PWA on their device when they first open it. Since this feature would not work on an iOS device (another apple-based restriction), when opened on an iOS device the PWA provides users with instructions on how to install the PWA to their home screen instead.

iframes and Supported Documents: An iframe or "Inline Frame" is an HTML document embedded in another HTML document. The Terraso Library prototype uses iframes for containing the documents themselves meaning it can support any audio/video documents that are in formats supported by the device's browser. One problem that arises is that certain browsers can't handle certain formats. For example, any browser can view WEBM files on a computer or android phone, but no browser can view WEBM files on an iOS device. I think it would be worth looking into alternative ways to display documents in order to have a more consistent expectation of what formats can be used for Documents on all devices.

Localizing Document Names: Localizing the names of the documents themselves is a relatively straightforward task when using DynamoDB or a similar noSQL database. It is possible to add fields in the database for the document name in different languages. A GraphQL query can then be written that queries the database for the name in a specific language. There are some use cases to consider, such as what should be displayed if a document does not have an entry for the language the client is running in.

Localizing Documents: If the documents themselves are available in different languages, it is possible to serve the user the appropriate language by expanding the GraphQL API to allow for retrieval. One way to do this would be to modify the GraphQL Schema so that every document object has a tree linking it to all equivalent documents in different languages. We could then write GraphQL queries that query a document in a specific language. The only problem with this approach is that if a document does not exist in a specific language, the GraphQL query would return an error and we would need to run another GraphQL to fetch the document in another Language. In general querying GraphQL for something that may or may not exist is a bad practice, so a more elegant approach would be preferable if possible. One implementation might be to *always* have an entry for each document in each supported language; if the document isn't actually available in the language, then the best available version could be inserted instead.

Alternatively, server side logic could handle a null table entry and algorithmically determine the best fall-back version to be returned in the query.

AWS Lambda: AWS Lambda is a serverless event-based compute service that lets you run code without provisioning or managing servers. The most relevant use for Lambda in the Terraso PWA is that we could use it to automatically maintain a DynamoDB table based on the contents of an S3 bucket. Doing this would allow for automatic synchronization between the GraphQL server and the S3 bucket, so whenever a new document is added to the S3 bucket, the GraphQL server will be updated and display that document within the application itself.
In the application's current state, documents must be added to the GraphQL server manually. It would be beneficial to learn/research more about Lambda and also to explore similar tools that do not require AWS integration.

Other initiatives

- Secure user authentication as required including possible 2FA authentication if required
- Develop automated regression tests so that each new version of the app can be guaranteed not to break functionality
- Setup QA, staging and production environments so that the release of new versions of the app can be controlled
- Backups of application data where required (e.g. S3 provides no guarantee of data persistence)
- Consider CDN’s that speed up content delivery for local regions