**Progressive Web App Documentation**

**\* It’s expected that HTML, CSS and Javascript are known \***

**or familiar while reading this document**

-------------------------- DEVELOPER WARNING START --------------------------

**IMPORTANT:**

Reading this document and the articles attached to it is imperative to working on 360’s PWA. Not understanding a concept can and will lead to bad coding, new bugs and long hours of failure. We know it’s a lot of reading but it’s worth it in the end.

It’s recommended to use a Mac due to it’s elementary setup of applications and libraries. This documentation is based on Mac.

---------------------------- DEVELOPER WARNING END ----------------------------

**Getting Started:**

Applications that you will need to download and install:

Google Chrome: [Download](https://www.google.com/chrome/?brand=CHBD&gclid=CjwKCAjw6vvoBRBtEiwAZq-T1duuLc5upnZRTrLkopGLS7eiJwbyLW7X88epgMUlYyp7IYH5ij1f2hoCayAQAvD_BwE&gclsrc=aw.ds)

Visual Studio Code: [Download](https://code.visualstudio.com/)

Once installed: Open VS Code. On the left hand side, click on square “” which is the tab to install extensions. Search and install the extensions:

“Prettier - Code formatter” - Puts your code in a consistent readable format every time you save a file

“Live Share” - Allows you to simultaneously work on a document with another person that also had VS Code. This extension is not perfect though and may be a little buggy.

“stylelint” - Checks syntax correctness for Sass (`.scss` files)

“EditorConfig for VS Code” - Ensures that indentation style, line endings, and file endings are consistent across editors and operating systems

“ESLint” - Checks syntax correctness for JavaScript (`.js` files)

Node.js: [Download](https://nodejs.org/en/download/)

Git: [Install Tutorial](https://hackernoon.com/install-git-on-mac-a884f0c9d32c)

Android Studio: [Download](https://developer.android.com/studio)

Xcode: [Download](https://apps.apple.com/us/app/xcode/id497799835?mt=12) ( From the App Store. An Apple ID is required )

After getting the applications above, go to [360’s UI GitHub repo](https://github.com/gordon-cs/gordon-360-ui). On GitHub, the PWA branch is jjPWA. Clone this repo. Afterwards, open the folder in VS Code. Open the built in terminal and make sure that you’re in the repo folder. If not, “[cd](http://www.linfo.org/cd.html)” your way there. Type, “npm install”. After npm finishes installing, the console might ask you to type “npm audit fix”. This is to fix any vulnerabilities that were caught. Once that’s done, you can start up the UI by running the command, “npm start”. This will start up the server and open a page in your default browser to the URL of the UI (aka, 360’s start page).

**Debugging:**

To debug, we will be using Chrome Developer Tools. After running “npm start”, if 360’s webpage didn’t open in Chrome, in VS Code’s console, copy the link that has “http://localhost:xxxx”. The ‘xxxx’ is the port number that your server is running on. Open Chrome and go to this link. Once, 360’s homepage loads, right click the page and click on “Inspect”. This will open Chrome’s Developer Tools. It would be best if you familiarize yourself with the [Application](https://developers.google.com/web/tools/chrome-devtools/progressive-web-apps), [Performance](https://developers.google.com/web/tools/chrome-devtools/evaluate-performance/), [Network](https://developers.google.com/web/tools/chrome-devtools/network/) and [Audits](https://developers.google.com/web/tools/lighthouse/) tabs. Each of these serves its own purpose in debugging 360.

**Troubleshooting:**

If you are reading this section, you are probably experiencing something weird either with Chrome, service worker, or with PWA in general. These are some ways that you can try to solve before you change your working code:

* After opening up Gordon 360 locally, right click “inspect” then go to “Application” tab, go under “clear storage” and click “clear site data” button.
* In the upper left corner of web browser, click “Chrome”, then click “clear browser data”. Then go ahead to delete any cached files and cookies.
* If these methods did not work, go ahead and reboot your whole computer. (This method has been working greatly for practicum 2019!)

List of symptoms that is caused by Chrome:

1. If localhost starts with 6XX,XXX, instead of 3000, reboot your computer
2. Sometimes, a new implemented function might look as broken, but it could be a service worker that is behaving weirdly, so if “clear site data”
3. If you see a message saying “no matching service worker...”
4. If you do not see any icon images under the “manifest” tab

**Testing on Devices:**

Real Devices: You can use a real android device to debug. Click [here](https://developers.google.com/web/tools/chrome-devtools/remote-debugging/) for more info. Testing on iOS is possible but not guaranteed to work. It’s better to use an emulator.

Emulators:

Android ( MUST HAVE ANDROID STUDIO INSTALLED ):

1. Open Android Studio
2. If you have already created a dummy project, it should appear on the left hand side in a list. You can click on it and skip to step 5. If not, Click on “Start a new Android Studio project”
3. Click on “Add no activity” and then “Next”
4. Click on “Finish”
5. Once an empty project loads, on the top bar click, “Tools”
6. Click on “AVD Manager”
7. Click on “+ Create Virtual Device”
8. Choose any android device you want to emulate and click “Next”. The Pixel 3 is recommended
9. Choose any OS. Q is recommended.
10. Click “Next” once more
11. Click “Finish”
12. Your emulator should appear in the list of virtual devices. Click the green play button or double click the device’s info in order to run the emulator.
13. DEBUGGING: If you used the recommended settings above, you will have Chrome already installed on the emulator. Open it and the emulator should now be visible in Chrome Developer Tools: Remote Devices on your computer . (Note: Read the article above dealing with “Real Devices” in order to know how to forward your local host so that the emulator can use it.

iOS ( MUST HAVE XCODE INSTALLED ):

1. To run an iOS simulator
   1. Open Spotlight by pressing “cmd + spacebar” and type “Simulator”
   2. Click on the application “Simulator” to open it.
   3. An iPhone XR or whatever the default device is will appear. It should automatically turn on and boot up.
   4. When boot up is complete, click on the application “Safari” in the simulator.
2. To create spotty network connections during simulation
   1. Open Xcode
   2. At startup there should be a window that says “Welcome to Xcode”. Make sure that window is the focus of your screen
   3. On the top bar of your screen (Where the apple icon is to the left), click on “Xcode”
   4. Go to “Open Developer Tool”
   5. Click on “More Developer Tools”. This will open up your default browser to an Apple login page. Login using your Apple credentials.
   6. After signing in successfully, on the left hand side, you will see a search bar and under it “Categories”
   7. Click on the search bar and search for “Additional Tools for Xcode”
   8. Download the latest version there is and open it
   9. Once the dmg opens, click the folder “Hardware”
   10. Open the application “Network Link Conditioner”. This will install the application right into your System Preferences. After agreeing to do so, you may eject the dmg (can be found on desktop) and move the file to trash
   11. IMPORTANT: Whenever you close out of System Preferences, the Network Link Conditioner will disappear as an icon. If you want to use it again, you have to open Spotlight by pressing “cmd + spacebar” and typing “Network Link Conditioner.prefPane”. Click on Network Link Conditioner when it pops up in Spotlight’s search result in order to open it in System Preferences
   12. Open “Network Link Conditioner.prefPane” and enable it by toggling it on. With this program, you can simulate different network connections to test out how your program will respond. For example, while connected to GordonNet (Gordon College’s Wifi) you can simulate being connected to LTE, 3G, Edge, etc. This allows you to test your program against spotty network connections
3. DEBUGGING: On your mac, open Safari. Then enable [Developer Tools](https://support.apple.com/guide/safari/use-the-developer-tools-in-the-develop-menu-sfri20948/mac) if you have not already.

(Note: Read the article above dealing with “Real Devices” in order to know how to forward your local host so that the emulator can use it.

**PWA Requirements**

Web - App minimum requirements

* Manifest file with the properties:
  + Short\_name
  + Name
  + Start\_url
  + Icons with various pixel sizes
* Registered service worker
* The webpage must be served over https (or local host when developing)

**Manifest:**

**What it is:**

**A web app manifest provides information about a web application in a** [**JSON**](https://developer.mozilla.org/en-US/docs/Glossary/JSON) **text file, necessary for the web app to be downloaded and be presented to the user similarly to a native app (e.g., be installed on the homescreen of a device, providing users with quicker access and a richer experience). PWA manifests include its name, author, icon(s), version, description, and list of all the necessary resources (among other things).**

**Properties ( that our own manifest has ) :**

**background\_color** :

A placeholder background color for the application page to display before the stylesheet is loaded. Color is written like a CSS property. Examples are:

* "background\_color": "red"
* "background\_color": "#7FC5E8"

**categories**:

An array of strings defining the names of the categories that the application supposedly belongs to. Examples are:

* "categories": "business"
* "categories": "[“business”, “books”, “medical”]

**description**:

A string in which developers can explain what the application does. Examples are:

* "description": "A planner for your homework"

**dir**:

The base direction in which to display direction-capable members of the manifest. Together with the lang member, it helps to correctly display right-to-left or left-to-right languages. Examples are:

* "dir": "auto"
* "dir": "ltr”
* "dir": "rtl”

**display**:

Determines the developers’ preferred display mode for the website. The display mode changes how much of browser UI is shown to the user and can range from "browser" (when the full browser window is shown) to "fullscreen" (when the app is full-screened). Examples are:

* "display": "browser”
* "display": "fullscreen”
* "display": "standalone”

**icons**:

Specifies an array of objects representing image files that can serve as application icons for different contexts. It’s required to have at least one picture that’s 144x144. An example is:

* "icons": [

{

"src": "src/images/icons/app-icon-48x48.png",

"type": "image/png",

"sizes": "48x48",

"density": "0.75",

},

**lang**:

Specifies the primary language for the values of the manifest's directionality-capable members, and together with dir determines their directionality. Examples are:

* "lang": "en-US”
* "lang": "fr-CA”

**name**:

A string that represents the name of the web application as it is usually displayed to the user (e.g., amongst a list of other applications, or as a label for an icon). It’s directionality-capable, which means it can be displayed left-to-right or right-to-left based on the values of the dir and lang manifest members. Examples are:

* "name": "Homework Planner”
* "name": "Gordon 360”

**orientation**:

Defines the default orientation for all the website's top-level browsing contexts. Examples are:

* "orientation": "portrait”
* "orientation": "landscape”

**screenshots**:

Defines an array of screenshots intended to showcase the application. These images are intended to be used by progressive web app stores. An example is:

* "screenshots": [

{

"src": "app-screenshot-one.jpg",

"sizes": "1280x720",

"type": "image/jpg"

}

]

**short\_name**:

A string that represents the name of the web application displayed to the user if there is not enough space to display name (e.g., as a label for an icon on the phone home screen). It’s directionality-capable, which means it can be displayed left-to-right or right-to-left based on the value of the dir and lang manifest members.

Examples are:

* "short\_name": "G360”
* "short\_name": "Spotify”

**start\_url**:

A string that represents the start URL of the web application — the preferred URL that should be loaded when the user launches the web application (e.g., when the user taps on the web application's icon from a device's application menu or homescreen). Examples are:

* "start\_url": "https://example.com”

**theme\_color**:

A string that defines the default theme color for the application. This sometimes affects how the OS displays the site (e.g., on Android's task switcher, the theme color surrounds the site). Color is written like a CSS property. Examples are:

* "theme\_color": "blue”
* "theme\_color": "#4644D5”

**Service Worker Usage:**

There are listenable events:

* Fetch - Browser initiates a fetch (is an http request)
* Push - Service worker receives push notifications from server
* Notification interaction - Action of user interacting with displayed notification
* Background sync - Service worker receives background sync event

Registration involves three states: Install -> Waiting -> Active

Service worker Lifecycle

* Installing - service worker being registered but not yet activated

a) event.waitUntil()

b) self.skipWaiting()

* Installed - service worker has finished its setup and is waiting
* Activating - intended to allow the service worker to finish the setup

a) event.waitUntil()

b) self.skipWaiting()

* Activated - ready to handle functional events

**PWA Concepts**

**Promises:**

What they are:

The **Promise** object represents the eventual completion (or failure) of an asynchronous operation, and its resulting value. The parameter of a **Promise** is a callback function that takes two parameters, resolve and reject. Be careful though, these are no ordinary parameters. They are actually methods.

let promise1 = new Promise(function(resolve, reject) {

// Some code to run

});

These two parameters have different actions. In the area where “// Some code to run” executes, we would call either resolve() or reject() depending on the code we ran. If our code ran successfully, we would call resolve() passing in whatever we want as a parameter. Of course oppositely, if our code fails, we would call reject() and pass in whatever we want as a parameter. Both of these returns a **Promise** object that has two properties:

* *[[PromiseStatus]]* :The status of the Promise object which can either be resolved (the operation was successful) or rejected (the operation failed). Another status a Promise object can have is pending (waiting for the operation to be successful or a failure) but this would be the initial status and not of a returned Promise object
* *[[PromiseValue]]* :The result of the Promise object which will be what you passed in as a parameter to the resolve() or reject() method (whichever one is returned)

Promise Object Example One:

let promise1 = new Promise((resolve, reject) => {

if (5 > 3) {

resolve({ items: 5, message: "Hello" });

}

});

console.log(promise1);

/\* A Promise object will be logged to the console with the **[[PromiseStatus]]** as **resolved** and the **[[PromiseValue]]** as the object **{ items: 5, message: “Hello” }** \*/

------------------------------------------------------------

Promise Object Example Two:

let promise2 = new Promise((resolve, reject) => {

if (5 !== 3) {

reject("5 is not equal to 3!");

}

});

console.log(promise2);

/\* A Promise object will be logged to the console with the **[[PromiseStatus]]** as **rejected** and the **[[PromiseValue]]** as the string **"5 is not equal to 3!"** \*/

Note: The Promise object might be accessible but its two properties [[PromiseStatus]] and [[PromiseValue]] are not without handlers. They both result as undefined when logged to the console.

How they are handled:

Promises aren’t simple objects that we can read data straight from. They can be handled with two methods known as then() and catch(). In addition to that, there’s one other way to handle promises that is easier to understand than using the methods then() and catch(). This easier method, is await.

For Promise objects:

* then() : Used to handle Promise objects whose status is resolved. In other words, this method is used when the Promise object successfully ran an operation. Inside of this method, we pass in a parameter which is a callback function that contains a single parameter (should have a reasonable name such as “**data**” below). This parameter will be set equal to whatever the *[[PromiseValue]]* of the Promise object is.

Example:

/\* Using the **promise1** variable from the above example which is a Promise object whose status is resolved \*/

promise1.then(function(data) {

console.log(data);

});

/\* The parameter "**data**" is equal to the **[[PromiseValue]]** of the Promise object **promise1**.

When we log this to the console, we will get the object **{ items: 5, message: "Hello" }** \*/

* catch() : Used to handle Promise objects whose status is rejected. In other words, this method is used when the Promise object failed to run an operation. Inside of this method, we pass in a parameter which is a callback function that contains a single parameter (should have a reasonable name such as “**error**” below). This parameter will be set equal to whatever the *[[PromiseValue]]* of the Promise object is.

Example:

/\* Using the **promise2** variable from the above example which is a Promise object whose status is rejected \*/

promise2.catch(function(error) {

console.log(error);

});

/\* The parameter "**error**" is equal to the **[[PromiseValue]]** of the Promise object **promise2**.

When we log this to the console, we will get the string “**5 is not equal to 3!"** \*/

Note: It’s necessary in most cases to have both methods chained to each other when receiving a Promise if it’s not guaranteed to pass or fail. An example of this is when doing fetch request.

/\* Using a combination of the **promise1** and **promise2** variables from above as **promise3** for an example of chaining \*/

promise3.then(function(data) {

console.log(data);

}).catch(function(error) {

console.log(error);

});

* await : Unlike then() and catch(), await is a keyword and not a method. It makes JavaScript wait until the Promise finishes and returns its result. This doesn’t cost performance because while Javascript waits, the CPU runs other tasks such as scripts and event handlers. The data that’s returned from the Promise is stored inside the Promise’s property, *[[PromiseValue]]* . Note: await only works in async (asynchronous) functions. This also means for functions that returns a Promise object such as fetch. The keyword async means that whatever is the result of a regular function will be wrapped inside of a Promise object and returned as that.

/\* Since we declared **async** at the beginning of the function, we can use the keyword **await** inside and outside the function wherever we call it \*/

async function fetchThenCache(request) {

return await fetch(request)

.then(fetchResponse => {

if (fetchResponse) {

return fetchResponse.clone();

}

})

/\* We declare **async** once more in the catch function because inside we have a callback function. The async that’s declared above isn’t passed into the callback function \*/

.catch(async () => {

const response = await caches.match(request);

if (response)

return response;

else

console.log(`Failed to get ${request.url} from cache`);

});

}

**Caching Static/Dynamic Files:**

This is important to note since it requires code outside the service worker

1. When you type in your credentials and click on “login”, on line 81 in ”gordon-360-ui/src/views/Login/index.js” you will see that we post a message to the service worker containing an object that has a message, the user’s token, and the current term inside. The message is to let the service worker know that we’re ready to cache all dynamic files. The current term is used in the links since we cannot import the current term from the service worker (more info about this below). The token is the most important of those three because when you do a fetch, it’s required that you show authentication. Without it, you will get the:

Message: "Authorization has been denied for this request."

error for each dynamic file you try to cache. We get the token in ”gordon-360-ui/src/views/Login/index.js” instead of doing it in the service worker because the service worker doesn’t have the functionality to do that. You can’t import things into the service worker (aka no access to [localStorage](https://stackoverflow.com/questions/40887635/access-localstorage-from-service-worker) (<-- Click for more info).

1. Once the service worker receives the message “cache-static-dynamic-files”, it will execute several functions to cache all required files.
2. In cacheDynamic(token, arrayOfLinks) and getUserInfoForLinks(token, termCode)**,** a special header is created in order to authorize the request. This is where the token is placed. After the header is created, we do a fetch for each url that’s needed. These steps occur for each url:
   1. A new Request object is made
      1. The url is the first parameter and the second is a special object that contains the special header
   2. A fetch is done for the recently created request
      1. If there’s a response, we cache it. If not, we log that the url failed to the console

Note: If all is successful, you will see the static files have the response type of “basic” while the dynamic files should have “cors”. If you want to learn more about “CORS” click [here](https://www.codecademy.com/articles/what-is-cors). To know more about Response types, click [here](https://developer.mozilla.org/en-US/docs/Web/API/Response/type).

**Errors and Warnings:**

1. “Getting example-URL.com from cache instead…”

This warning occurs in fetchThenCache() in “gordon-360-ui/public/sw.js”. This is due to a fetch error from a server possibly due to a connection that cannot be established or file is not available.

1. “Caching Static Files Failed”

This warning occurs in preCacheStatic() in “gordon-360-ui/public/sw.js”. This is due to a failure to cache all of the static files in the array staticCache.

1. “Failed to fetch and cache Dynamic File: example-URL.com

This warning occurs in preCacheDynamic() in “gordon-360-ui/public/sw.js”. This is due to a failure to fetch a specific dynamic file.

1. “Failed to get example-URL.com from cache”

This warning occurs in fetchThenCache() in “gordon-360-ui/public/sw.js”. This is due to a failure to fetch a specific dynamic file from the cache.

**How to view all files in cache (Google Chrome):**

Copy and paste this function into the console.log of Chrome’s Developer Tools. Make sure to change ‘cache-1.0’ to the correct name of the cache.

**caches.open('cache-1.0').then(cache => {**

**cache.keys().then(items => {**

**let num = 1;**

**items.map(item => {**

**console.log(`${num}) ${item.url}`);**

**num++;**

**})**

**})**

**});**

**Q&A:**

Why do things need to be cached in the first place? What does it really mean for something to be cached?

We cache things in order to have access to data that we don't need to fetch for. For example, if we required a specific piece of data that never changes as the user browses the site, it would be more efficient to cache it so that we don't have to fetch the data from the server. Therefore, increasing performance.

Caching is one of the fundamentals of PWA. Its use is to gather all data required for a site to run without error offline. In other words, we save data to our local storage so that if we lose internet connection, our site will still work since the data is being served from the local storage (aka the cache). Without it, PWA are impossible.

Is all the Promise code done in the service worker or do we use promises throughout the front-end javascript?

Promises are done throughout the front-end of javascript. They guarantee that an operation either passed or failed.