# Homework 1 - Practice with Human-Centered Design & Progressive Web Apps

By Constantin Miranda & Kyle Harms

## 1. Overview

In Homework 1 you'll design and build an app for a client. You will follow the human-centered design process to build this app. You'll first identify the client's criteria for the app and plan the design using sketches. Lastly, you'll code the app using Vue.js.

### 1.1. Learning Objectives

- Employ human-centered design processes.
- Learn how to build a progressive web app using Vue.js
- Practice discovering users' needs through interviews.
- Critically evaluate what a user says versus what they actually need.
- Explore the potential design space using manual/hand-based techniques like sketching.
- Improve programming confidence and programming through practice.
- Learn to reuse existing work and re-purpose it for your needs through refactoring.
- Think critically about how to evaluate your designs.

### 1.2. Deadlines & Credit

| Part | Deadline | Credit |
|---|---|---|
| Part I | Tue 9/24, 5:00pm | 0 points |
| Part II | Tue 9/24, 5:00pm | ~25 points |
| Part III | Tue 9/24, 5:00pm | ~25 points |
| Part IV | Tue 9/24, 5:00pm | ~75 points |
| Part V | Tue 9/24, 5:00pm | ~25 points |
| Part VI | Tue 9/24, 5:00pm | 0 points |

**There will be no deadline extension.** Start early, get help early, submit early!

## 1.3. Git Repository & Submission

Clone `git@github.coecis.cornell.edu:info4340-fa2019/YOUR_GITHUB_USERNAME-hw1.git`. Replace **YOUR_GITHUB_USERNAME** in the URL with **your actual GitHub username**. This is usually your NetID.

Submit **all** materials to your GitHub repository for this assignment.

**Tip**: Commit and push your changes every time you work on your project. Every time you commit and push you store your changes on the GitHub server. This acts as a back-up for your work. It also means that if you forget to submit before the deadline, there's something already on the server that the TAs can grade.

# 2. Part I: Progressive Web Apps with Vue.js

You'll be building a Progressive Web App (PWA) for this homework.

> Progressive Web Apps are web applications that are regular web pages or websites, but can appear to the user like traditional applications or native mobile applications.
> Source: https://en.wikipedia.org/wiki/Progressive_web_applications

Because PWAs are built with standard HTML5 technologies you don't need any special tools to build an app. You can build a PWA with just the technologies and techniques you learned in INFO 1300. However, using a framework, like Vue.js, can save you a lot time and effort.

We'll be using the Vue.js framework to build our apps this semester.

As UX designer, you'll need to work with the software engineers to design interfaces that they can actually code. Before designing your app, you should first understand the tools that will be used to build the app and any technical possibilities and limitations of these tools. If you don't know the limitations of these tools, you may produce designs that are technically unfeasible or extremely difficult to code.

**In this part, you will study how to work with the Vue.js framework.**

## 2.1. Vue.js Resources

Web frameworks constantly change. What's hot now, won't be hot in 5 years. However, conceptually they all work similarly. The best way to learn about a framework is to utilize its reference documentation.

**You need to use this documentation to complete this homework.**

- Vue Documentation: https://vuejs.org/v2/guide/syntax.html
- Vue CLI 3 guide: https://cli.vuejs.org/config/
- Moment.js guide: https://momentjs.com/
- `v-if` Documentation: https://vuejs.org/v2/guide/conditional.html#v-if
- `v-on` Documentation: https://vuejs.org/v2/guide/events.html
- TODO: `$emit` documentation (need to find a good guide - official documentation that I found looks sparse)

## 2.2. Getting Started with a Vue.js Project

**You should have already cloned your repository for this assignment and opened the project in Visual Studio Code.**

Vue.js projects are simply node.js packages (libraries) that import the Vue.js packages (libraries). Node.js is simply a JavaScript runtime that you can use outside of a web browser. All node.js packages have a **package.json** which tells

node.js about your package. **Take a look at *package.json*.** Notice that the *vue* package is specified in the *dependencies* section. This tells Node that your app (package) will use the *vue* package.

In order for you to run your app, you'll need to install the dependent packages. First run `npm install` inside of your repository. **npm** is the **N**ode **P**ackage **M**anger. This will tell Node.js to install all of the dependent packages for your project into the *node_modules* directory.

To run a Vue.js app, you need issue the `npm run serve` command. Check out the *scripts* section in **package.json**. Observe that *serve* will simply run the `vue-cli-service serve` command for you. This command launches a local web server written in JavaScript via Node.js. To stop the local web server, press **control + c**.

## 2.3. How are Vue.js PWAs Structured?

Let's take a look at the files that make this application work. We won't explain everything, so you'll need to refer to Vue's documentation or ask for help if you don't understand a piece of code. Remember, we are all here to learn, and when we help each other, we all learn more.

- **main.js**: This is the main JavaScript file that drives your Vue application. The Vue.js build tool looks here to figure out how to build the rest of the application. This file imports Vue.js and creates the Vue instance. It also tells the application to render **App.vue** on-screen.
- **App.vue**: This is the main single-file component for your application. As you can see, the only element we're using here is `<router-view>`. This allows us to place a "view" on-screen and switch between "views" as a user navigates our application.
- **router.js**: Defines what view to display. Notice that the path is set to `/`. This is the root path (http://localhost:8080/) and currently is pointing to the **Announcements.vue** view. When a user navigates to our application, the first thing they will see is what we coded into **Announcements.vue**.
- **Announcements.vue**: This is the main view of the application. It imports and places the two *components* for our application on-screen: **InputAnnouncement.vue** and **ViewAnnouncement.vue**. The **Annoucements.vue** view also handles passing data from the **InputAnnouncement.vue** component to the **ViewAnnouncement.vue** component.
- **InputAnnouncement.vue**, **ViewAnnouncement.vue**: These are the two components for our application.

  One handles all of the elements and logic for collecting information from the user. It uses `$emit` and `submit()` to send the data back to **Announcements.vue**.

  The other handles all of the elements and logic for displaying the input announcement in a table. It receives the input data via **Announcements.vue** (remember that **InputAnnouncement.vue** sent the data to **Announcements.vue** to then get passed to this component).

## 2.4. Building a User Interface in Vue.js

To help you see how these views and components come together to build app, we're going to add some input components to the interface and display them back to the user.

1. We'll need a form to enable users to input an announcement, just like we learned in INFO 2300. Let's start by adding the input components to the user interface.

   In the `<template>` section of **InputAnnouncement.vue** add the following input components after the `<h4>`:

   ```
   <input
     placeholder='Enter Text Here'
     v-model='tempAnnouncement'
     @keyup.enter='submit'
   />
   <button v-on:click='submit'>Set Announcement</button>
   ```

2. When a user submits the form, we'll want to store the data they entered so we can show it back to them later.

   The `v-model='tempAnnouncement'` in the input component tells Vue to store the user's data for this input component in the `tempAnnouncement` property. We need to create this property.

   In the `<script>` section, add the following code block to create the `tempAnnouncement` property.

   ```
   data () {
     return {
       tempAnnouncement: ''
     }
   },
   ```

3. Next, we want to send the information that the user input in `InputAnnouncement.vue` to the `ViewAnnouncement.vue` component.

   We will do so using the `$emit` feature in Vue. Copy-paste this code block below the `data()` block:

   ```
   methods: {
     submit: function () {
       this.$emit('inputData', this.tempAnnouncement)
       this.tempAnnouncement = ''
     }
   }
   ```

4. Before we start connecting everything, we need to import the components.

   At the top of our `<script>` tag in the **Announcements.vue** view add the following code:

   ```
   import InputAnnouncement from '@/components/InputAnnouncement.vue'
   import ViewAnnouncement from '@/components/ViewAnnouncement.vue'
   ```

   And we need to register them. Paste this code block directly below `name: 'Announcements'`:

   ```
   components: {
     InputAnnouncement,
     ViewAnnouncement
   },
   ```

5. We need to capture the information sent from `InputAnnouncement.vue`.

   You can access the data you `$emit`-ed by referencing the key ( `inputData` ) and calling a method to handle the passed data.

   Add the following component tag to **Announcements.vue**. This pulls the data from the **InputAnnouncement** component into our main view.

```
  <InputAnnouncement @inputData='addAnnouncement' />
```

   Now, we will want to store the data somewhere, so add the following code block below the `components:` property:

```
data: function () {
  return {
    announcementData: ''
  }
},
```

   Now, we want to write the method, `addAnnouncement()` that we are calling in our component tag. This method will take the passed data and assign it to our `announcementData` data property.

```
methods: {
  addAnnouncement (variable) {
    this.announcementData = variable
  }
}
```

6. We've now captured the data from **InputAnnouncement.vue** and stored it in a data property in our main view. How do we send data from our main view to another component? We'll use a `prop:` in conjunction with a `watch` .

   In **Announcements.vue**, add the following component tag. We are passing `aanouncementData` as a `prop:` titled 'announcementPassed'.

   ```
   <ViewAnnouncement :announcementPassed='announcementData' />
   ```

   We need to register our `prop:` in **ViewAnnouncement.vue**. Copy-paste the following code block below the `name:` property:

   ```
   props: {
     announcementPassed: {
       type: String
     }
   },
   ```

   We now want to watch our `prop:` for any changes. We'll achieve this using a `watch:` property. Copy-paste this code block below your `props:` section:

   ```
   watch: {
     announcementPassed: function () {
       var inputData = { 'date': 'Latest Announcement', 'announcement': this.announcementPassed }
       this.announcementList = inputData
     }
   }
   ```

7. Finally, we need to control when the components display.

   When an announcement has not been entered, we want **InputAnnouncement.vue** to display. When an announcement has been entered, this component should disappear and **ViewAnnouncement.vue** should display.

   In **Announcements.vue**, add `inputData: false` to your `data:` property. It should now look like this:

```
data: function () {
  return {
    announcementData: '',
    showInput: true
  }
},
```

   When users input data, we want the `showInput` boolean to change to `false`. Modify your `addAnnouncement()` method to match the following:

```
methods: {
  addAnnouncement (variable) {
    this.announcementData = variable
    this.showInput = false
  }
}
```

   Finally, we want to control the `<template>` elements using a `v-show` directive. Modify your component tags to match the following:

```
<InputAnnouncement v-show='this.showInput' @inputData='addAnnouncement' />
<ViewAnnouncement v-show='!this.showInput' :announcementPassed='announcementData' />
```

8. Refresh the app in your browser. Clicking the 'Set Announcement' button should now display the user's announcement in a table.

## 2.5. Using Node.js Libraries

If you notice, the date stamp is hardcoded to say "Latest Announcement". Our users will want something more specific. We are going to use Moment.js to generate a time stamp in a text string.

First, stop your development server using **control + c**. Install Moment.js using `npm install --save moment`. Notice your **package.json** and **package-lock.json** files have been modified to include Moment.js.

Import the library at the top of your `<script>` tag in `ViewAnnouncement.vue`. Modify the `inputData` associative array to use Moment.js to generate a time stamp. Assign the time stamp to the `date:` field in the array.

# 3. Part II: Discovering Your Users' Needs

Before you can build something for your client, you need to understand their needs and the needs of their users. For this project, your client is Barb. She's also the user. Note that sometimes the client and the users are different audiences.

To learn about Barb's needs you interviewed her during class. You then analyzed her interview responses and developed a persona and requirements.

For Part II, create a **documents** directory in your assignment's repository. Add the following files to this directory:

- **persona.jpg**: Finalize your draft persona from class and then take a photo of it.
- **interview.md** or **interview.jpg**: Take a photo of your interview notes or type them up.
- *(optional)* ***requirements.md***: You may optionally include the original requirements that you prepared for class.

## 3.1. Criteria

As you may remember, we switched from using requirements to criteria because I was concerned that the rigid nature of requirements might limit your design ideas. As a class we then documented the criteria for our client's app.

As a reminder, our user's current process for managing tasks is to place a post-it notes in a highly visible area in her home. This system breaks down when a post-it note gets lost.

We also learned that, in fact, this system is highly usable and works well for our user... mostly. This system should be...

- **Visible** due to it being in a high traffic area.
- Permits **prioritization** through spacial location and *colored* post-its.
- **Collaborative** through being readily adoptable and available to others.
- Provides **temporal awareness** of up coming and highly prioritized tasks.
- On an average day, **gently and non-aggressively reminds** the user twice a day: when she walks by in the morning before work and walks by again when she get's home.

Other relevant information:

- Our user places post-its on her computer monitor for a **desktop computer**.
- The desktop's screen is **always visible**; it does not sleep and there is no screen saver.
- In order to focus on the design process and learning Vue.js, **you may assume:**
  - **There is only one task at a time.** Though the user should be able to change the task.
  - The app will always be the **only thing on the screen** at any given point in time. The app is set to be always on top. etc.
  - The app need only work on **desktop computers**. No mobile, responsive design required.
- The app should **remember its data** when it's closed and re-opened.
- Any **information displayed** in the app **must always be current**!

# 4. Part III: Exploring Design Solutions

In Part I, you explored the possibilities and limitations of building a PWA with Vue.js. Then in Part II you explored your user's needs. Now in Part III, you'll design a PWA.

Get your persona out. As your designing, constantly reference it. Place yourself in your persona's shoes and ask yourself if *your* needs are being met.

**First, explore your ideas. Sketch a bunch of designs on paper by hand.** Feel free to collaborate with your peers: get feedback from one another, critique each others designs, etc. When you're done, take a photo of each sketch and place them in a **sketches** folder inside of your **documents** folder.

Once you have explored your ideas through sketching, **finalize your design by producing wireframe(s)**. For time's sake, I would prefer that you author your wireframes by hand on paper. However, you may produce a digital drawing. If you produce a digital drawing, it must be a drawing and not a **mock-up**. Wireframes are *polished sketches* that should be mostly black and white line drawings. Place your wireframes in a **wireframe** folder inside of the **documents** folder.

# 5. Part IV: Build a High Fidelity Prototype!

With your design in hand, it's time to **code a high fidelity prototype** of the app using Vue.js. For this part, you will **re-use the Announcements app that you built earlier**.

1. Refactor the Announcements app's code and customize it to build our client's app.

   **You'll need to rename your components and views.**

   For example, consider renaming `ViewAnnouncement.vue` to `Countdown.vue`.

   **Make sure** that this name change is applied throughout your file structure. We recommend combing through all of the files in your app using VS Code's project search.

   Hint: You may need to change the `name:` property within your component and search for references to the old component in **router.js**.

2. The first things you'll likely implement are the user input fields in the former **InputAnnouncement.vue** component. When you collect information from the user, you temporarily store the information in the component's `data()` property.

   To make the data persistent between opening and closing the app, store this information using the Web Storage API. This will ensure that the data persists even if our user closes their browser window.

   Use `localStorage.setItem()` to create and set a `localStorage` key.

3. Just like in our announcements application, once information is input into the former **InputAnnouncement.vue** component, we need to tell the **main view** that is is time to switch to the former **ViewAnnouncement.vue** component in order to show the user's data.

   You should also include the following code block in your **main view**. The `mounted` Vue Lifecycle Hook will ensure that our application checks whether data has already been set in `localStorage` (i.e. when our user reopens the app):

   ```
   mounted: function () {
     if (localStorage.date) {
       this.inputDate = true
     }
   },
   ```

4. Finally, you'll want your data to always be current.

   Use the following code block to ensure that your data is updated frequently:

   ```
   mounted () {
     this.interval = setInterval(this.time, 1000)
   },
   beforeDestroy () {
     clearInterval(this.interval)
   }
   ```

5. **Implementation restrictions:**

   - When importing libraries, use a node.js package. (Added to package.json)
     - Do not hotlink JavaScript libraries in the HTML.
     - Do not import with a CDN.
   - You must develop a PWA using Vue.
   - Your app doesn't need to look perfect. But it should look nice and be usable.
     - Users consistently rate attractive designs as more usable.
     - Take some care to make the final app attractive. It's doesn't need to be *beautiful*.
     - It should not look *bare bones*; default HTML styling is not good enough.

# 6. Part V: Evaluate the Prototype

Typically we evaluate our design during every phase. Due to time constraints, we skipped that. Shame on us. We should have prepared a test plan and scenarios during discovery and design.

In the document's folder create a 'test-plan.md' file. In this Markdown file, document a test plan for evaluating whether your design meets the needs of the user.

Your test plan should include a **description of how you plan to conduct user testing** and the **scenarios** (not tasks) that you will use during testing.

# 7. Part VI: Check your Submission

You should commit and push everything to your GitHub repository.

Once you have pushed everything, **test your submission**! If your submission is incomplete or doesn't work, you will likely receive a very low score. It is your responsibility to test that your submission works. There are no redos. There are no valid excuses for submitting broken code.

To test, clone your repository to a new location on your computer. Run `npm install` and then `npm run serve`. Scroll up in the console window and check for errors. If you see errors here, then your code is broken! Test your app in a web browser and make sure it works!