Project Delta

Handbook

Important guidelines and hopefully the answer to everything

v. 2019-10-17

Contents

- Configuration
  - VSCode Extensions
  - Linting
- Figma
- Commenting code
  - Sass
- CSS (Sass)
  - Creating a new component
  - Creating the component class
    - But why?
  - Using media queries
  - Grid (Flexbox)
- Git
  - Branches
    - Getting started
      - Cloning the repository
      - Connect to the repository
    - Creating a branch, making changes and pushing them to GitHub
  - Cheat sheet & FAQ
    - fetch
    - pull

Configuration

VSCode Extensions

You should have the following extensions installed:

- Live Server
- Live Sass Compiler
- W3C Validation (Validates HTML according to W3C)
  Requires setup, please see the instructions on the extension details page
- CSSTree validator (Validates CSS according to W3C)

Optional extensions:

- GitLens

Linting

*"**Linting** is the process of checking the source code for Programmatic as well as Stylistic errors. This is most helpful in identifying some common and uncommon mistakes that are made during coding."*
https://stackoverflow.com/questions/8503559/what-is-linting/30339671#30339671

We are going to have to use a linter (such as Prettier), but it needs further investigation in config setup.

More content will be added here later …

Figma

File naming convention: **myh_<view>_<page>.fig**
Project directory: **design/**

Example:
myh_mobile_contact.fig
myh_laptop_contact.fig

Commenting code

Comments should be used as much as possible when something in the code is not self-explanatory.

The default hotkey for comments in most editors (including VSCode):
Windows: CTRL + *
Mac: CMD + *

Sass

Learn more: https://sass-lang.com/documentation/syntax/comments

Sass comments SHOULD use double slashes (//) and should NOT use the slash-asterisk style (/* */).

The reason for this is that when our Sass code goes through the compiler, the comments written in the “double slash”-style will not be present in the compiled .CSS file.

Comments inside the folder **www/css/tools** MUST be written in SassDoc style in order to generate a documentation page.
Too confusing? Just write regular slash comments and someone else can fix it later ¯_(ツ)_/¯
Learn more: https://sass-lang.com/documentation/syntax/comments#documentation-comments

CSS (Sass)

File naming convention: **_*.scss** *(do not use the .sass extension)*
Class naming convention: **Block Element Modifier (BEM)**
Project directory: **www/css/**

NEVER manually modify the **main.css** file

We are using Sass and not regular CSS, Sass compiles all of our SCSS files into a single **main.css** file - meaning that we may ONLY edit the SCSS files.

If you modify the **main.css** file **YOU WILL LOSE ALL OF YOUR CHANGES** once any SCSS file is compiled! Be afraid!

Read about Sass: https://sass-lang.com/documentation
Read about BEM: http://getbem.com/naming/

Creating a new component

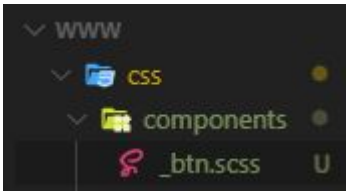
Instead of keeping all of our CSS (Sass) code in one file, we are splitting it all up into separate files (components). This helps us stay organized and prevents merge conflicts.

To create a new component, make a new file in the **/www/css/components/** directory.

The filename should be prefixed with an underscore (_), followed by the name of your component and suffixed with **.scss**. The name of your component should preferably explain what the component is/does.

Try to avoid giving your components the exact same name as existing HTML tags.

Example:



See: https://sass-lang.com/documentation/at-rules/import#partials to understand why the underscore is important.

Creating the component class

Again, read about BEM: http://getbem.com/naming/
Or watch this video if you're feeling lazy: https://www.youtube.com/watch?v=SLjHSVwXYq4

Instead of directly targeting and modifying HTML elements, we are creating **reusable** component classes.

```
// Instead of this
button {
    font-size: 14px;
    font-weight: bold;
    text-transform: uppercase
}

// Do this
.btn {
    font-size: 14px;
    font-weight: bold;
    text-transform: uppercase
}
```

This allows us to style the button (in this case) without interfering with the original HTML tag style.

But why?

Picture this scenario
You have three <button> elements. Two of the buttons should be black with white text and come in two different sizes (small and medium).
The third button should have a large size and an orange background with black text.

How would you solve this problem?

This is the approach we should be taking - using BEM:
https://codepen.io/hexat/pen/WNNGrJj
we can even do this (notice the change in CSS):
https://codepen.io/hexat/pen/dyypMqN

By working this way we are not only redesigning a website, we are also creating our very own reusable CSS framework.

Now that’s something to put on your resumé - even if the end result may not be the greatest!

It should also help in showing how CSS frameworks work and why they can be a great asset in web development.

Using media queries

Instead of creating a media query the CSS way, use our very own mixin **responsive** inside of your component classes. It should throw an error at you if used incorrectly.

Example:

```scss
// This example component sets font size based on screen width
.custom-component {
    // When width is 0px or greater
    font-size: 12px;

    // When width is 578px or greater (sm breakpoint)
    // This is what you should be using in 99% of cases!
    // Converts to:
    // @media only screen and (min-width: 578px)
    @include responsive('sm') {
        font-size: 14px;
    }
    // the same thing can also be written as
    @include responsive($min: 'sm') {
        font-size: 14px;
    }

    // When width is 578px or less (sm breakpoint)
    // Converts to:
    // using @media only screen and (max-width: 578px)
    @include responsive($max: 'sm') {
        font-size: 16px;
    }

    // When width is between 578px and 789px (sm and md breakpoints)
    // Converts to:
    // @media only screen and (min-width: 578px) and (max-width: 789px)
    @include responsive($min: 'sm', $max: 'md') {
        font-size: 16px;
    }
}
```

Grid (Flexbox)

The core functionality, such as the grid system is based on the Bootstrap 4 **container -> row -> column** method.
Meaning that if you read the Bootstrap Grid documentation you'll know how our grid system works as well - same shit, different code!

https://getbootstrap.com/docs/4.3/layout/grid/

Git

Branches

Naming convention: **<type>/<issue id>/<feature>**
Types: **feature**, **fix**

The naming convention may resemble GitFlow, however, we are using our own standard.

NEVER commit to the *master* branch!

Getting started

1. ### Cloning the repository

   To get started you first need to clone the projects GitHub repository.

   In your terminal, use the command:
   git clone https://github.com/chas-academy/u02-redesign-delta.git

   This will download all of the remote project files to your computer.

2. ### Connect to the repository

   Even though you've now downloaded the project, you still haven't made a connection to the project repository. In order for you to be able to send changes back to the repository you need to make that connection.

   In your terminal, use the command:
   git remote add origin https://github.com/chas-academy/u02-redesign-delta.git

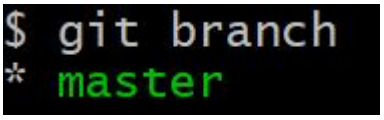
   Read more: https://www.atlassian.com/git/tutorials/syncing

Creating a branch, making changes and pushing them to GitHub

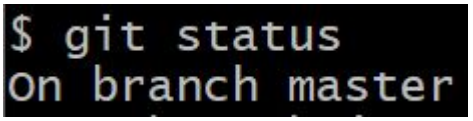
Instead of committing new changes to the master branch - create a new branch!

1. Before doing so, ensure that you are on the master branch.

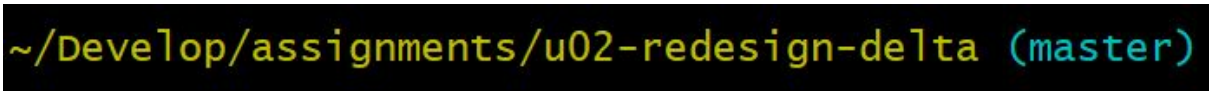
   In your terminal, use the command **git branch** to check. The branch with an asterisk is the one you are currently on.

   You can also use the command **git status**.

   Or perhaps just check the end of your working path.

2. Ensure that your master branch is up to date by using the command
   git pull
   Note: This command downloads the most recent files and updates your local files. If you have work that has not yet been committed or stashed it will all be lost.
3. Use the command **git checkout -b <type>/<issue id>/<feature>** to create a branch
   a. Replace **<type>** with the change type
   b. Replace **<issue id>** with the GitHub issue ID number
   c. Replace **<feature>** with the a very short description of the feature or fix that you are implementing
4. Do whatever change you were going to do
5. Use the command **git add .** or **git add -A** to add all your changes to the staging area.

6. To commit your changes, use the command **git commit -m “<message>”** where **<message>** is replaced with your actual commit message. Remember to use imperative form!
7. To push your changes, use the command:
   git push -u origin <branch name>
   Where **<branch name>** is the name of the branch you created in step 2.

Cheat sheet & FAQ

Cheat sheet page
https://gitsheet.wtf/
FAQ - When you've messed up and don't know what to do (Be careful with some of these)
https://ohshitgit.com/

fetch

git fetch really only downloads new data from a remote repository - but it doesn't integrate any of this new data into your working files. Fetch is great for getting a fresh view on all the things that happened in a remote repository.
Due to it's "harmless" nature, you can rest assured: fetch will never manipulate, destroy, or screw up anything. This means you can never fetch often enough ...
https://www.git-tower.com/learn/git/faq/difference-between-git-fetch-git-pull

pull

git pull, in contrast, is used with a different goal in mind: to update your current HEAD branch with the latest changes from the remote server. This means that pull not only downloads new data; it also directly integrates it into your current working copy files. This has a couple of consequences ...
https://www.git-tower.com/learn/git/faq/difference-between-git-fetch-git-pull