Week 9 Notes

Managing Modern Front-end workflow

A potential list of tasks that could need to happen every time something changes:

Lint CSS and Javascript to find any coding issues.

Run unit tests to make sure nothing got broken with your last changes

compile all SCSS/LESS to CSS

concatinate all CSS into one file for faster loading,

Minify the CSS to reduce file size.

Transpile Javascript with Babel for wider support for older browsers

Concatinate all JS files into one file for faster loading

Minify and Uglify Javascript to reduce size

Do the same for any 3rd party CSS or JS

Move all the production assets into a distribution directory to separate them from the development stuff

Tools To Help Manage Workflow

Package Managers: Keep track of external dependencies ex. development tools and libraries we are using. Knows which packages to download, but tracks versions too. **npm**

Bundlers: Handle the compiling, transpiling, concatenating, minifying, and moving around of assets in the project. **Parcel.** also Webpack

Task Managers: Keep track of what needs to be done and when. Scripts defined in the task manager for each phase of development. Also use **npm.** Can also use Grunt or Gulp.

Will need **NodeJS** installed on computer and the **boilerplate code.**

In boilerplate code:

node_modules: all 3rd party tools installed with **npm install** in command line.

src: all of the code to work with (html, js, css, sass)

package.json: info about the dependencies for the project. This tells what to get when running install command.

Under **scripts** in **package.json:**

prestart: This gets run automatically when you run start or you can run it by itself as well. It does a bit of cleanup for us.

start: This command starts up our bundler Parcel. This starts a webserver on port 1234, clears out the 'build' directory, builds the new files based on what is in 'src' (transpiles with babel, compiles our Sass if we are using it, and concatenates all of our CSS and JS into separate files.), watches our files for changes, re-compiles everything on a change and refreshes the browser. It places the code in a build directory. Quite a lot for one simple command!

prebuild: just like prestart this runs automatically when we run the build process.

build: This command gets our code ready for distribution. It runs a similar process to the start above, but also minifies the CSS and JS. It creates a dist directory. If you were publishing your app to a server that is the code you would move.

watch: is very similar to start

lint: runs a linter on your Javascript and will make recommendations and sometimes even changes to your code based on the rules it is configured with.

To run any of the scripts: **npm run *command***

example: to launch built in web server – npm run start

Under **devDependencies** in **package.json:**

List of the tools we are using to facililtate our development. Not too be part of distribution code (large files)

To start up your tools: run **npm run start** in the command line. You should see the build directory get created and populated with files. And if you point your web browser to **http://localhost:1234** you will see a simple page. Make a change to any of the files in the project and the browser will auto-refresh.

When you are done working you can hit **ctrl-c** in your command line to end that process.

The Window Object

Every JavaScript environment has a **global object.** Any variables that are created in the global scope are actually properties of this object, and any function are methods of it. The global object is the **window** object. This represents the browser window that contains a web page.

The Browser Object Model (BOM)

A collection of properties and methods that contain info about the browser and computer screen. Every browser window, tab, popup, frame, and iframe has a **window** object.

Only makes sense in a browser environment. Node.js has global object called **global**.

If you don't know the global object, use the keyword **this.**

const global = this;

Going Global

Global variables are access in all parts of the program. They are created without using **const, let,** or **var** and are properties of the global object (window object in browser environment).

x=6; //global variable created

window.x //variable accessed as a property of window object

window.x === x; //true – both variables are the same

Dialogs

Three functions that produce dialogs in the browsers: **alert(), confirm(),** and **prompt().**

window.alert() method pauses execution and displays a message. Message provided as argument to the method and **undefined** is returned:

window.alert('Hello'); //undefined

window.confirm() stops execution and displays a confirmation dialog and shows a message provided as an arg. Gives 'Ok' or 'Cancel' options. Returns true if user clicks OK and false if user clicks Cancel:

window.confirm('Do you wish to continue?');

<< undefined

window.prompt() stops executionand displays dialog that shows message and an input field. Input returned as a string when user clicks OK. null returned when Cancel is clicked:

window.prompt('please enter your name:');

Browser Information

Which Browser?

navigator property returns reference to **Navigator** object that contains info about the browser being used. It's **userAgent** property will return info about the browser and OS being used.

window.navigator.userAgent

*don't rely on this info because it can be changed

Location, Location, Location

window.location property is an object that contains info about the URL of current page. **href** property returns the full URL as a string:

window.location.href

protocol property returns string with protocol used (http, https, pop2, ftp) with colon at the end:

window.location.protocol // "https:"

host property returns string describing the domain of current URL and the port number(default 80 and may be omitted if same as default):

window.location.host // www.sitepoint.com

More properties:

- **hostname**
- **port**
- **pathname**
- **search**
- **hash**
- **origin**

Methods to window.location

- **reload()**
- **assign()**
- **replace()**
- **toString()**

The Browser History

window.history property used to access info bout previously visited pages

window.history.length shows how many pages visited before current page

window.history.go() method used to go to specific page. 0 is current page.

window.history.forward() and **window.history.back()** methods used to navigate by one page like forward and back buttons.

Controlling Windows

window.open() opens a new window with URL of page to be opened as first parameter, window title as second parameter, and a list of attributes as the third. Can be assigned to variable so it can be referenced later.

```
const popup = window.open('https://sitepoint.com','
SitePoint','width=400,height=400,resizable=yes');
```

close() method used to close a window:

```
popup.close();
```

Move and resize windows: **window.moveTo(x,y)** and **window.resizeTo(100,100)**

Screen Information

window.screen object has **height** and **width** properties: window.screen.height

> Also **availHeight** and **availWidth** to find height and width of screen minus any OS menus.

colorDepth used to find color bit depth of user's monitor.

The Document Object

Each **window** object contains a **document** object. Properties and methods deal with the page that has been loaded in the window.

document.write() writes a string of text to the page:

```
document.write(<h1>Hello World!</h1>);
```

Cookies

Small files saved locally on user's computer. Cookies used to sidestep problem not remembering anything about previous visits. Used to personalize experience. Store preferences, choices, authentication, tracking numbers, etc.

Cookies are a text file that contain a list of name/value pairs separated by semicolons:

"name=Superman; hero=true; city=Metropolis"

Creating Cookies

Assign to JavaScript's "cookie jar" using **document.cookie** property. Won't overwrite but will append to the end of a string.

```
document.cookie = 'name=Superman';
```

Reading Cookies

```
document.cookie;
```

Can use **split** method to break string into an array containing each name/value pair, then use **for of** loop to iterate through the array:

```
const cookies = document.cookie.split("; ");

for (crumb of cookies){
   const [key,value] = crumb.split("=");
   console.log(`The value of ${key} is ${value}`);
}
```

```
<< The value of name is Batman

The value of hero is true

The value of city is Gotham
```

Cookie Expiry Dates

Can make last longer than browser session with **"; expires=date"** at the end of the cookie when it's set.

```
const expiryDate = new Date();

const tomorrow = expiryDate.getTime() + 1000 * 60 * 60 * 24;

expiryDate.setTime(tomorrow);
```

document.cookie = `name=Batman; expires=${ expiryDate.toUTCString()}`;

Path and Domain

Secure Cookies

Deleting Cookies

Set it to expire at a time in the past

Also libraries to handle cookies like Cookies.js or jsCookie

Timing Functions

setTimeout() accepts a callback to a function as 1st parameter and a number of milliseconds as its second parameter. Pay attention to number returned.

window.setTimeout(() => alert("Time's Up!"), 3000);

<< 4

clearTimeout()

window.clearTimeout(4); //use number that was returned by window.setTimeout()

<< undefined

setInterval()

like setTimeout() but will repeatedly invoke the callback after every given number of milliseconds.

he previous example used an anonymous function, but it is also possible to use a named function like so:

function chant(){ console.log('Beetlejuice'); }

Now we can set up the interval and assign it to a variable:

const summon = window.setInterval(chant,1000);

<< 6

clearInterval()

Animation

setTimeout() and **setInterval()** can be used to animate elements.

```
const squareElement = document.getElementById('square');
let angle = 0;

setInterval( () => {
   angle = (angle + 2) % 360;
   squareElement.style.transform = `rotate(${angle}deg)`
}, 1000/60);
```

requestAnimationFrame() better performance than setTimeout()

HTML5 APIs

The data- Attribute

A way of embedding data in a web page using custom attributes that are ignored by the browser. Sole purpose is to be used by JavaScript program.

Names of attributes can be decided by developer, but must use the following format:

Start with **data-**

Contain only lowercase letters, numbers, hyphens, dots, colons or underscores

Include an optional string value

Examples:

data-powers = 'flight superSpeed'

data-rating = '5'

data-dropdown

data-user = 'DAZ'

data-max-length = '32'

Each element has a **dataset** property that can be used to access any **data-** attributes it contains:

```
<div id='hero' data-powers='flight superSpeed'>
   Superman
</div>
```

The data-powers attribute can be accessed using the following code:

```
const superman = document.getElementById('hero');
const powers = superman.dataset.powers;
<< 'flight superspeed
```

HTML5 APIs

HTML5 Web Storage

The Web Storage API provides a key-value store on the client's computer like using cookies but has fewer restrictions, more storage, and easier to use.

Differences:

Info stored is not shared with the server on every request

Info is available in multiple windows of the browser

Storage capacity – most browsers have a limit set at 5GB per domain

Any data stored does not automatically expire.

window object will have a property called **localStorage.** Also **sessionStorage,** but only saved for current session.

```
localStorage.setItem('name', 'Walter White');
```

To remove an entry from local storage, use the removeItem method:

```
localStorage.removeItem('name');
```

Alternatively, this can be done using the delete operator:

delete localStorage.name;

To completely remove everything stored in local storage, use the clear() method:

localStorage.clear();

To save an object as a JSON string:

localStorage.setItem('keyName', JSON.stringify(object);

To retrieve the string as an object:

name = JSON.parse(localStorage.getItem('keyName'));

Geolocation

Used to obtain the geographical position of the device. It will be a property of the **navigator** object that has **getCurrentPosition()** to return a **position** object to a specified callback function:

navigator.geolocation.getCurrentPosition(callbackFunctionName);

The **position** object passed to the **callbackFunctionName()** function has a **coords** property with a **latitude** and **longitude** property, which together give the coordinates of the device.

position object has other properties:

.speed

.altitude

.heading

.timestamp

.accuracy

.altitudeAccuracy

geolocation object has **watchPosition()** to call a callback every time the position of device is updated and returns an id that can be used to reference the position being watched:

const id = navigator.geolocation.watchPosition(youAreHere);

Web Workers

Allow processes to be run in the background, adding support for concurrency.

To start, use **Worker()** constructor function to make a new worker:

const worker = new Worker('task.js');

Web workers use messages to communicate between the main script and worker script. **postMessage()** method used to send a message and start worker working. To post a message to the worker:

worker.postMessage('Hello');

To post message from the worker:

self.postMessage('Finished');

To stop, use **terminate()**:

worker.terminate(); or self.close();

Websockets

A new protocol that allows two-way communication with a server – aka push messaging. Connection kept open and responses 'pushed' to client as soon as they are received.

Notifications

API allows you to show messages using the system's notifications.

Need to request permission of user. Use **requestPermission()** method of **Notification** global object.

Multimedia

<audio> and <video> tags:

<audio src='/song.mp3' controls>

Your browser does not support the audio element.

</audio>

<video src='http://movie.mp4' controls>

Your browser does not support the video element.

</video>

.play()

.pause()

.volume

.muted

.currentTime

.playbackRate

.loop

.duration

Events:

play

pause

volumechange

loadedmetadata

Other APIs

Canvas

Shims and PolyFills

The terms shim and polyfill are often used interchangeably. The main difference between them is that a shim is a piece of code that adds some missing functionality to a browser, although the implementation method may differ slightly from the standard API. A polyfill is a shim that achieves the same functionality, while also using the API commands that would be used if the feature was supported natively.