Week 6 (User Interfaces)

1. Introduction

This week will focus on how to create good user interfaces. Previously, we would have a multi-paged application where whenever we want to load HTML we would need to query the server and the server would return a html page. However, possible to do the rendering on the client side instead of the server side.

1. Javascript AJAX calls

Javascript can query an API with the fetch API. To use Fetch API use the fetch() method. Has 1 mandatory argument which is the path of resource to fetch. Returns a promise that resolves to the Response to that request. This is true even if the server responds with a HTTP error status.

Note: As response is returned in a Promise, this can be chained with other promises.

Normally you can use it to query an API to obtain data in a JSON format.

If the API is designed with RESTful principles – which most are now – then API endpoints should correspond to HTTP methods. Similarly what is happening is that JS is sending a HTTP request to API directly without having to refresh the page.

1. Pros

The above approach is useful for the following reasons:

1. You do not need to reload the entire page like you would if you are going to a new URL. This is especially important as most pages are quite similar in any web app. This leads to faster load speeds as this means that you only need to download and load the necessary HTML/CSS/JS
2. Leads to a more reactive looking UI leading to better UI/UX experience. This is called the Single Page Application and is highly popular with consumer facing applications
3. Unable to give users a real time dynamic look on their applications as they have to keep refreshing.
4. Cons
5. Lose state in URL. When JS queries API there is no change in URL.

To solve this, you can manipulate JS API to push a request to the user’s browsers history.

                        history.pushState({section: section}, "", `section${section}`);

history.pushState takes the following arguments

* State -> JSON object associated with new history entry created by pushState(). I.E. it is the event’s state
* Title -> String most browsers ignore this param so can be empty string
* URL -> String. New History URL to be pushed to browser

1. User cannot press back cause there is no URL

Web API has methods to fix this too.

Window.onpopstate = funcRef

Where funcRef is a handler function

            window.onpopstate = function(event) {

                console.log(event.state.section);

                showSection(event.state.section);

            }

Window.onpopstate is an event handler that runs when you pop something off your history. Not here, event.staste.section would say section1 only because we have used history.pushState to push a state on our events whenever they click on a section. That is where event.state is set. Section is part of the JSON object that was set to it.

Note: The window object here is powerful as it represents the physical window on the user’s screen and displays all content on the user’s screen.

1. Properties of Windows and Document object

Windows.innerWidth = Width of their screen

Windows.innerHeight = Height of the screen

Windows.scrollY = how far down the page user has scrolled in pixels

Document object represents the entire web page. Note documents are typically long and they do not fit on the window itself at once.

Documents.body.offsetHeight = entire height of the document

1. Infinite Scroll

To create infinite scroll we have to know when the user reaches the bottom of the page. This is simply when

Documents.body.offsetHeight = windows.scrollY + Windows.innerHeight

In index.html in pages app in scroll

1. Animation by CSS

Found in animate1 to animate3

1. Animate with Infinite scroll in Django template

Hide -> Index template

Shows the following

1. How to create HTML element with JS
2. How to append it to the DOM
3. How to use fetch to query APIs
4. How to dynamically load content to page
5. React

React based on declarative programming as compared to imperative programming which is what we have been practicing so far.

Imperative programming = step by step instruction

Declarative programming = Just specify what is the end state

React divides application into components an each component has an underlying state and states can be manipulated with react generating the HTML and CSS to show that change

1. Packages that we have to download to work with React

* React -> Library to define components and how they act
* ReactDOM -> Package that takes components and insert them into the DOM
* Babel -> Package to translate code from one to another. React is actually not written in JS but rather written in JSx which is a JS extensions but has more features and special syntax for representing HTML. Hence, need something to translate JSx to JS as browsers don’t support JS.

Project Notes

Outline of SPA

* HTML is divided into 2 main elements. First, email-views which is meant to display emails sent to the user. Second, compose-view which allows the user to send an email.
* Note that compose-views contain forms which are not Django forms but forms rendered normally in HTML
* Buttons along the top selectively show and hide views.
  + Compose button should hide emails-view and show compose-view
  + Inbox button should hide compose-view and show emails-view
* 3 types of mailboxes (i) inbox (ii) sent (iii) archived. Same function load\_mailbox to display them. It needs to display the name of the selected mailbox by updating the innerHTML of emails-view

Outline of API

* API endpoints are:

**/emails/<mailbox> where mailbox can be inbox, sent or archive.**

* Sending a GET request would return a JSON of a list of all emails in that mailbox in reverse chronological order.
* To access values in JS, use fetch to send a GET request to that endpoint

**/emails/email\_id where email\_id is an Int**

* Will return JSON representation of the email
* If don’t exists or user don’t have access to email, will return 404 error

**POST /emails**

* Send POST request to that route will send an email. 3 main arugments for this endpoint (i) recipients value which is a Comma Separated string of all users to send an email to (ii) subject string (iii) body string
* Success code is 201 and JSON response of message Email Sent Successfully

**PUT /emails/<int:email\_id>**

* Route meant to mark email as read/unread or archived/unarchived. Send PUT request to end point where email\_id is id of email to modify

Notes

**Adding Event Handlers**

Document.addEventListner(“click”, function () {

….

}

This is the generic way to add an event listener with vanilla JS. This method adds an event handler to the document.

Syntax:

Document.addEventListner(event, function, useCapture)

Event = A string that specifies the name of the event such as onclick

Function -> Specifies the function to run when the event occurs

**Showing HTML elements**

Elements in HTML are mostly “inline” or “block” elements. Former refers to element that has floating contents on left and right side. Latter means element fills the entire line and nothing can be displayed on the left and right side of it.

Document.getElementById(“myDIV”).style.display = “none”

Above means hide the element

Document.getElementById(“myDIV”).style.display = “none”

Above means show the element

**Changing JS string to JSON**

**JSON Structure**

**Difference between ID and Class**

ID is used to identify 1 single element in our HTML. Class can be used to identify more than 1 HTML element

**Setting classes, id or roles in JS**

Just use element.setAttribute(name, value)

**Promises Javascript**

Then() method returns a Promise. Takes 2 arguments (1) onFulfilled -> callback function for success. The function has 1 argument called fulfilment value. If not a function, internally replaced with an “Identity” function which returns the received argument (2) onRejected -> A function called if the Promise is rejected. Function has 1 argument the reject reason. If not a function, interally replaced with a Thrower function i.e. it throws an error it received as an argument

**Truthy and Falsy JS**

Truthy value is a value that is considered true when encountered in a Boolean context. All values are truthy unless defined as falsy.

Such values are

if (true)

if ({})

if ([])

if (42)

if ("0")

if ("false")

if (new Date())

if (-42)

if (12n)

if (3.14)

if (-3.14)

if (Infinity)

if (-Infinity)

**CSS Flexbox Tutorial**

**Grid Tutorial Bootstrap**

**CSS Selectors**

Selectors are basically ways for CSS to find HTML elements to apply styling. 5 types of selectors

* Simple selectors

Selectors that select based on name, id and class.

Examples are:

1. Select by element i.e. whether they are a paragraph or anchor
2. Select by ID e.g. #para1. Uses the id attribute of an HTML element to select a specific element. Each element only has 1 id – or is supposed to.
3. Select by class e.g. [.className]. Use classes to select
4. Combine class and element selector like this -> p.center

An element with multiple classes will be styled according to both classes.

* Combinator Selectors

Select elements based on specific relationship between them

* Psuedo-class sectors

Select elements based on a certain state

* Pseudo-elements selectors

Select and style part of an element

* Attribute selectors

Select elements based on attribute or attribute value

**Difference between onlick and addEventListener**

Onclick is an object property. This means if there is more than 1 such property, it will be overridden. In contrast, addEventListener binds an event handler ot the element and we can call it each time without being worried of it being overwritten.

Onclick is similar to inline Javascript i.e. <button onclick=”doSomething()”>. This is a practice that is discouraged.

AddEventLIstener does not work on old browsers namely i.e. below version 0.

addEventListner syntax

.addEventLIstener(event, functionToExecute)

There are the following type of events:

* Mouse Events -> Anything related ot the mouse
* Touch Events -> Only triggered by touch-enabled devices like smartphones. Note in smartphones, the mouse over event isn’t triggered at all because they can’t detect a finger hovering over a phone.
* Keyboard Events
* Form Events
* Window Events

Note passing by reference vs. calling a function

AddEventListener accepts a function reference as its 2nd argument not a function itself. A function reference is the memory address of that function rather than the function itself. If you add the function itself, it will just automatically call the function without the event triggering.

Example

Do this:

document.getElementById("tooltip-link1").addEventListener("click", function(){

displayTooltip(2)

});

OR

document.getElementById("tooltip-link1").addEventListener("click", () =>

displayTooltip(2)

);

Don’t do this

document.getElementById("tooltip-link1").addEventListener("click",displayTooltip(2));

The first example adds a function reference to displayTooltip(2) to those events while the 2nd example directly calls the function.

Project Specific Information

**Show email of mailbox**

Mailbox -> GET request to get all emails of a specific mailbox -> Would get an array of JSON objects where each JSON object is an email object. -> Wrap each email in a div and each div should contain the following information (i) sender (ii) subject (iii) timestamp

Each row should contain the following

1 div of class container -> 1 div of class row -> 3 div of class col-sm

Must many row be in the same container?

Yes. There should be 1 div of row container. In each row there is 1 div of class row and 3 divs of class col-sm

Problem: Right now all the div of class col-sm is being added to 1 class row instead of being separated

**View Email**

Idea here is when user click on anything within div class = “row”, to do so. Basically addEventHandler click around a div row. The event click should send an API call to get that particular email to retrieve details of it. Note that API call would need email\_id which can be found in email. Which means you need to add something to the div? **[How to add arguments into event handler function]**

Onclick should also change the status of the email to read. Means when load emails need to set state for the div -> Maybe hidden attribute determining whether it is read or unread.

Also need to add event handler such that when cursor hovers around the div it will change shape from cursor to like a finger pointing to show that you can click it.

Would get a JSON object if that email exists and will return 404 with JSON response error if it does not or user does not have access to that email.

Need to add new div to indbox.html to display the email. Need to hide email-views and compose-views.

HTML to display email needs something like (1) Email subject first line (2) Sender second line (3) Email content 3rd line. All should be in their own divs.

**Archive and Unarchive**

1. Need to add archive and unarchive to email-container dataset. Need to add email\_id to each email-container in dataset as well.
2. Create button that when clicked, would trigger sending an API call to change a button to be archived or unarchived for that particular email. Means api call needs to pass in email\_id. Button should also automatically load user inbox.

**Reply Email**

* Add reply button to email view
* When this button clicked should take them to compose view
* Recipient field of compose form should be automatically set
* Subject line should be set with Re:
  + Need logic to detect whether the subject line already starts with Re if it does then can just pass it as is if it does not then we need to create a new one for it
* Body should be prefilled
  + Need timestamp and sender and then prefill with information