

`async/await`

Two types of asynchrony

We have been working with two broad types of asynchronous events:

1. **Inherently asynchronous events**

- Example: `addEventListener('click')`. There is no such thing as a synchronous click event.

2. **Annoyingly asynchronous events**

- Example: `fetch()`. This function would be easier to use if it were synchronous, but for performance reasons it's asynchronous

Asynchronous fetch()

The usual
asynchronous
fetch() looks like
this:

```
function onJsonReady(json) {  
    console.log(json);  
}
```

```
function onResponse(response) {  
    return response.json();  
}
```

```
fetch('albums.json')  
    .then(onResponse)  
    .then(onJsonReady);
```

Synchronous fetch()?

A hypothetical synchronous fetch() might look like this:

// THIS CODE DOESN'T WORK

```
const response = fetch('albums.json');  
const json = response.json();  
console.log(json);
```

This is a lot cleaner code-wise!!

However, a synchronous fetch() would freeze the browser as the resource was downloading, which would be terrible for performance.

async / await

What if we could get the best of both worlds?

- Synchronous-*looking* code
- That actually ran asynchronously

// THIS CODE DOESN'T WORK

```
const response = fetch('albums.json');  
const json = response.json();  
console.log(json);
```

async / await

What if we could get the best of both worlds?

- Synchronous-*looking* code
- That actually ran asynchronously

// But this code does work:

```
async function loadJson() {  
  const response = await fetch('albums.json');  
  const json = await response.json();  
  console.log(json);  
}  
loadJson();
```

async / await

What if we could get the best of both worlds?

- Synchronous-*looking* code
- That actually ran asynchronously

// But this code does work:

```
async function loadJson() {  
  const response = await fetch('albums.json');  
  const json = await response.json();  
  console.log(json);  
}  
loadJson();
```

???

async functions

A function marked `async` has the following qualities:

- It will behave more or less like a normal function if you don't put `await` expression in it.
- An `await` expression is of form:
 - `await promise`

async functions

A function marked `async` has the following qualities:

- If there is an `await` expression, **the execution of the function will pause** until the `Promise` in the `await` expression is resolved.
 - Note: The browser is not blocked; it will continue executing JavaScript as the `async` function is paused.
- Then when the `Promise` is resolved, the execution of the function continues.
- The `await` expression evaluates to the resolved value of the `Promise`.

```
function onJsonReady(json) {  
  console.log(json);  
}  
function onResponse(response) {  
  return response.json();  
}  
fetch('albums.json')  
  .then(onResponse)  
  .then(onJsonReady);
```

The methods in
purple return
Promises.

```
async function loadJson() {  
  const response = await fetch('albums.json');  
  const json = await response.json();  
  console.log(json);  
}  
loadJson();
```

```
function onJsonReady(json) {  
    console.log(json);  
}  
function onResponse(response) {  
    return response.json();  
}  
fetch('albums.json')  
    .then(onResponse)  
    .then(onJsonReady);
```

The variables in
blue are the values
that the Promises
"resolve to".

```
async function loadJson() {  
    const response = await fetch('albums.json');  
    const json = await response.json();  
    console.log(json);  
}  
loadJson();
```

async functions

```
async function loadJson() {  
  const response = await fetch('albums.json');  
  const json = await response.json();  
  console.log(json);  
}
```

 `loadJson();`

async functions

```
async function loadJson() {
```

```
   const response = await fetch('albums.json');  
  const json = await response.json();  
  console.log(json);  
}
```

```
 loadJson();
```

async functions

```
async function loadJson() {  
  ➡ const response = await fetch('albums.json');  
    const json = await response.json();  
    console.log(json);  
}  
➡ loadJson();
```

Since we've reached an `await` statement, two things happen:

1. `fetch('albums.json');` runs
2. The execution of the `loadJson` function is paused here until `fetch('albums.json');` has completed.

async functions

```
async function loadJson() {  
  ➡ const response = await fetch('albums.json');  
    const json = await response.json();  
    console.log(json);  
}  
➡ loadJson();  
  console.log('after loadJson');
```

At the point, the JavaScript engine will return from `loadJson()` and it will continue executing where it left off.

async functions

```
async function loadJson() {
```

```
  ➡ const response = await fetch('albums.json');  
    const json = await response.json();  
    console.log(json);  
}
```

```
➡ loadJson();  
  console.log('after loadJson');
```

async functions

```
async function loadJson() {
```

```
  ➡ const response = await fetch('albums.json');  
    const json = await response.json();  
    console.log(json);  
}
```

```
loadJson();
```

```
➡ console.log('after loadJson');
```

async functions

```
async function loadJson() {  
  ➡ const response = await fetch('albums.json');  
    const json = await response.json();  
    console.log(json);  
}  
loadJson();  
console.log('after loadJson');
```



async functions

```
async function loadJson() {  
➡ const response = await fetch('albums.json');  
  const json = await response.json();  
  console.log(json);  
}  
loadJson();  
console.log('after loadJson');
```

If there are other events, like if a button was clicked and we had a event handler for it, JavaScript will continue executing those events.

async functions

```
async function loadJson() {  
➡ const response = await fetch('albums.json');  
  const json = await response.json();  
  console.log(json);  
}  
loadJson();  
console.log('after loadJson');
```

When the `fetch()` completes, the JavaScript engine will resume execution of `loadJson()`.

Recall: `fetch()` resolution

```
function onResponse(response) {  
    return response.json();  
}  
fetch('albums.json')  
    .then(onResponse)
```

Normally when `fetch()` finishes, it executes the `onResponse` callback, whose parameter will be `response`.

In Promise-speak:

- The return value of `fetch()` is a `Promise` that **resolves to** the `response` object.

async functions

```
async function loadJson() {  
  ➡ const response = await fetch('albums.json');  
    const json = await response.json();  
    console.log(json);  
}  
loadJson();  
console.log('after loadJson');
```

The value of the `await` expression is the value that the Promise resolves to, in this case `response`.

async functions

```
async function loadJson() {  
    const response = await fetch('albums.json');  
    ➡ const json = await response.json();  
    console.log(json);  
}  
loadJson();  
console.log('after loadJson');
```

async functions

```
async function loadJson() {  
  const response = await fetch('albums.json');  
  ➡ const json = await response.json();  
  console.log(json);  
}  
loadJson();
```

Since we've reached an `await` statement, two things happen:

1. `response.json();` runs
2. The execution of the `loadJson` function is paused here until `response.json();` has completed.

async functions

```
async function loadJson() {  
    const response = await fetch('albums.json');  
    ➡ const json = await response.json();  
    console.log(json);  
}  
loadJson();
```

If there are other events, like if a button was clicked and we had a event handler for it, JavaScript will continue executing those events.

async functions

```
async function loadJson() {  
    const response = await fetch('albums.json');  
    ➡ const json = await response.json();  
    console.log(json);  
}  
loadJson();
```

async functions

```
async function loadJson() {  
    const response = await fetch('albums.json');  
    ➡ const json = await response.json();  
    console.log(json);  
}  
loadJson();
```

When the `response.json()` completes, the JavaScript engine will resume execution of `loadJson()`.

Recall: json() resolution

```
function onJsonReady(jsObj) {  
    console.log(jsObj);  
}  
function onResponse(response) {  
    return response.json();  
}  
fetch('albums.json')  
    .then(onResponse)  
    .then(onJsonReady);
```

Normally when `json()` finishes, it executes the `onJsonReady` callback, whose parameter will be `jsObj`.

In Promise-speak:

- The return value of `json()` is a Promise that **resolves to** the `jsObj` object.

async functions

```
async function loadJson() {  
    const response = await fetch('albums.json');  
    ➡ const json = await response.json();  
    console.log(json);  
}  
loadJson();
```

The value of the `await` expression is the value that the Promise resolves to, in this case `json`.

async functions

```
async function loadJson() {  
  const response = await fetch('albums.json');  
  const json = await response.json();  
➡ console.log(json);  
}  
loadJson();
```

async functions

```
async function loadJson() {  
  const response = await fetch('albums.json');  
  const json = await response.json();  
  console.log(json);  
  }  
loadJson();
```

async functions

```
async function loadJson() {  
  const response = await fetch('albums.json');  
  const json = await response.json();  
  console.log(json);  
}  
loadJson();
```

Note that the JS execution does **not** return back to the call site, since the JS execution already did that when we saw the first `await` expression.

Returning from async

Q: What happens if we return a value from an async function?

```
async function loadJson() {  
    const response = await fetch('albums.json');  
    const json = await response.json();  
    console.log(json);  
    return true;  
}  
loadJson();
```

Returning from async

A: async functions must always return a Promise.

```
async function loadJson() {  
    const response = await fetch('albums.json');  
    const json = await response.json();  
    console.log(json);  
    return true;  
}  
loadJson();
```

If you return a value that is **not** a Promise (such as `true`), then the JavaScript engine will automatically wrap the value in a Promise that resolves to the value you returned.

Returning from async

```
function loadJsonDone(value) {  
  console.log('loadJson complete!');  
  console.log('value: ' + value);  
}
```

```
async function loadJson() {  
  const response = await fetch('albums.json');  
  const json = await response.json();  
  console.log(json);  
  return true;  
}  
loadJson().then(loadJsonDone)  
console.log('after loadJson');
```

REST

Restful API

Une API web est dite “**Restful**” si elle a les caractéristiques suivantes:

- Les requêtes sont envoyées sous forme de requêtes HTTP:
 - Méthodes HTTP: GET, POST, PUT, PATCH, DELETE..
- Un **Endpoint**: <http://example.com/api/ressources>
- Les requêtes doivent être envoyées avec un MIME*/Content-type spécifique tel: JSON, XML, HTML, etc.

* Multipurpose Internet Mail Extensions

Contraintes d'une API REST

Une API Restful doit respecter certaines contraintes d'architecture:

1. **Archi client-serveur:** chacun évolue indépendamment.
2. **Serveur sans états (stateless):** pas de sessions.
3. **Utilisation du cache:** le client sait combien de temps il peut garder les données qu'il reçoit avant expiration.
4. **Une interface uniforme:** un seul id, contient l'URL des ressources suivantes..
5. **Archi en couches:** Réduire la complexité de l'architecture globale de l'API.
6. **Code à la demande:** étendre les fonctionnalités du client.

Ressources

En REST tout est resources, on interagit avec les ressources à travers différents endpoints (URLs):

- <http://example.com/products>
- <http://example.com/products/1>
- <http://example.com/products/1/description>

Action sur les ressources

Pour interagir avec une ressource, il suffit d'utiliser la méthode HTTP (verbe) adéquate :

- GET <http://example.com/products/1> (Récupérer)
- POST <http://example.com/products> (Créer)
- PUT <http://example.com/products/1> (Mettre à jour)
- DELETE <http://example.com/products/1> (Effacer)

API Endpoints

Tweets	Retweets	Likes (formerly favorites)
• POST statuses/update • POST statuses/destroy/:id • GET statuses/show/:id • GET statuses/oembed • GET statuses/lookup	• POST statuses/retweet/:id • POST statuses/unretweet/:id • GET statuses/retweets/:id • GET statuses/retweets_of_me • GET statuses/retweeters/ids	• POST favorites/create/:id • POST favorites/destroy/:id • GET favorites/list

API de Twitter

API Endpoints

Resource URL

<https://api.twitter.com/1.1/statuses/update.json>

Resource Information

Response formats	JSON
Requires authentication?	Yes (user context only)
Rate limited?	Yes

Parameters

Name	Required	Description	Default Value	Example
status	required	The text of the status update. URL encode as necessary. t.co link wrapping will affect character counts.		

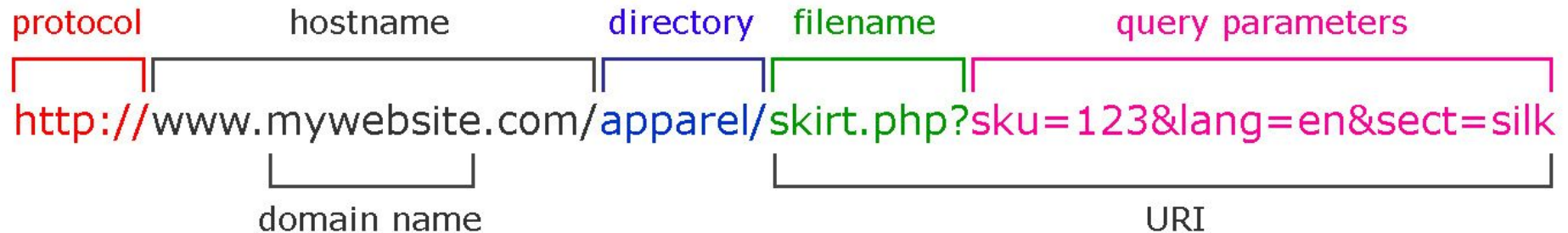
POST statuses/update

Paramètres des Endpoints

Un endpoint peut accepter un ensemble de paramètres pour effectuer sa tâche. Ces paramètres peuvent être passés de **trois manières** différentes:

- Paramètres dans l'URL.
- Headers dans la requête.
- Corps (body) de la requête.

Paramètres des Endpoints - paramètres dans l'URL



Paramètres des Endpoints - headers de la requête

▼ Request Headers

```
:authority: www.google.fr  
:method: GET  
:path: /  
:scheme: https  
accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8  
accept-encoding: gzip, deflate, br  
accept-language: fr-FR,fr;q=0.9,en-US;q=0.8,en;q=0.7  
cache-control: no-cache  
pragma: no-cache  
upgrade-insecure-requests: 1  
user-agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/65.0.3325.162 Safari/537.36
```

Paramètres des Endpoints - corps de la requête

```
1  POST /login HTTP/1.1
2  Host: foo.com
3  Content-Type: application/x-www-form-urlencoded
4  Content-Length: 37
5
6  username=foo@foo.com&password=123_foo
```

Codes d'état HTTP (Status code)

Il existe plus de 40 codes HTTP, chacun ayant une signification bien précise. Voici les grandes catégories:

- **1xx**: Information (e.g. 100 attente de la suite (continue))
- **2xx**: Succès (e.g. 200 tout s'est bien passé (OK))
- **3xx**: Redirection (e.g. 301 document déplacé (redirection))
- **4xx**: Erreur du client web (e.g. 404 document introuvable)
- **5xx**: Erreur du serveur (e.g. 500 Erreur interne)

API Restful et cache (E-tags)

Les E-tags permettent d'implémenter la contrainte de cache pour une API Restful. Il existe également d'autres méthodes qui sont l'utilisation des headers **Expires** et **cache-control**.

```
HTTP/1.1 200 OK
Date: Sat, 09 Feb 2013 16:09:50 GMT
Server: Apache/2.2.22 (Ubuntu)
Last-Modified: Sat, 02 Feb 2013 12:02:47 GMT
ETag: "c0947-b1-4d0258df1f625"
Content-Type: application/json

{
  id: 4,
  item: "take out the trash",
  created: "Sat, 02 Feb 2013 08:29:53 GMT",
  updated: "Sat, 02 Feb 2013 12:02:47 GMT",
}
```

API Restful et cache (E-tags)

Une fois que je connais l'E-tag, je peux à l'avenir demander s'il a changé ou non en envoyant une requête avec comme header:

If-None-Match: c0947-b1-4d0258df1f625

```
HTTP/1.1 304 Not Modified
Date: Sat, 09 Feb 2013 16:09:50 GMT
Server: Apache/2.2.22 (Ubuntu)
Last-Modified: Sat, 02 Feb 2013 12:02:47 GMT
ETag: "c0947-b1-4d0258df1f625"
```

API Restful et cache (E-tags)

```
HTTP/1.1 200 OK
```

```
Date: Sat, 09 Feb 2013 16:29:24 GMT
```

```
Server: Apache/2.2.22 (Ubuntu)
```

```
Last-Modified: Sat, 02 Feb 2013 14:33:21 GMT
```

```
ETag: "c7493-d7-a6b64d37f6cc3" # New ETag!
```

```
Content-Type: application/json
```

```
{  
  id: 4,  
  item: "Take out the trash, TODAY!",  
  created: "Sat, 02 Feb 2013 08:29:53 GMT",  
  updated: "Sat, 02 Feb 2013 14:33:21 GMT",  
}
```

OpenAPI

C'est un ensemble de spécifications permettant de décrire une API REST.

Cette description peut être comprise aisément à la fois par les développeurs ainsi que par les ordinateurs.

L'API est décrite sous forme d'un fichier JSON ou YAML.

Un ensemble d'outils permet de facilement convertir ces fichiers de description en documentation web ou encore en bibliothèques pour directement utiliser l'API décrite sans avoir à l'implémenter.

OpenAPI

```
1  swagger: '2.0'
2  schemes:
3    - https
4  host: api.twitter.com
5  basePath: /1.1
6  info:
7    contact:
8      email: support@twitter.com
9      name: Twitter support
10     url: 'https://dev.twitter.com'
11     x-twitter: twitter
12   title: Twitter
13   version: '1.1'
```

OpenAPI

```
29  paths:
30    /account/settings.json:
31      get:
32        description: |-
33          Returns settings (including
34          current trend, geo and sleep time information) for the authenticating user.
35        externalDocs:
36          url: 'https://dev.twitter.com/docs/api/1.1/get/account/settings'
37        operationId: account.settings.get
38        responses:
39          '200':
40            description: Successful Response
41        parameters:
42          - description: |-
43            The Yahoo! Where On Earth ID to use as the user's default trend location. Global informat
44            available by using 1 as the WOEID. The woeid must be one of the locations returned by GET
45            trends/available.
46
47            Example Values: 1
48            in: query
49            name: trend_location_woeid
50            required: false
51            type: string
52          - description: |-
53            When set to true, t or 1, will enable sleep time for the user. Sleep time is the time when
54            SMS notifications should not be sent to the user.
55
56            Example Values: true
57            in: query
58            name: sleep_time_enabled
59            required: false
60            type: string
```

Strict mode

Restricted dialect of Javascript

Not a subset — different semantics

Granularity: function (or file)

Strict mode

Restricted dialect of Javascript

Not a subset — different semantics

Granularity: function (or file)

Strict mode

Restricted dialect of Javascript

Not a subset — different semantics

Granularity: function (or file)

Strict mode differences

More errors are throw errors (or syntax errors)

Changes semantics for optimisation

Prohibits «likely» future keywords as identifiers

Strict mode differences

More errors are throw errors (or syntax errors)

Changes semantics for optimisation

Prohibits «likely» future keywords as identifiers

Strict mode differences

- More errors are throw errors (or syntax errors)
- Changes semantics for optimisation
- Prohibits «likely» future keywords as identifiers

Throw errors

Catchable errors/exceptions

```
function getRectArea(width, height) {  
  if (isNaN(width) || isNaN(height)) {  
    throw new Error("Parameter is not a number!");  
  }  
}  
  
try {  
  getRectArea(3, "A");  
} catch (e) {  
  console.error(e);  
  // Expected output: Error: Parameter is not a number!  
}
```

Strict mode changes

Stricter assignments:

- No accidental global creation

- Assignments to non-writeable values

- Deletion of non-removable fields

- Duplicate function parameters `function (a,a,c)`

- Duplicate field names `{"a": 1, "a": 2}`

- Global function calls have undefined `this`

Strict mode changes

Stricter assignments:

- No accidental global creation

- Assignments to non-writeable values

- Deletion of non-removable fields

- Duplicate function parameters `function (a,a,c)`

- Duplicate field names `{"a": 1, "a": 2}`

- Global function calls have undefined `this`

Strict mode changes

Stricter assignments:

- No accidental global creation

- Assignments to non-writeable values

- Deletion of non-removable fields

- Duplicate function parameters `function (a,a,c)`

- Duplicate field names `{"a": 1, "a": 2}`

- Global function calls have undefined `this`

Strict mode changes

Stricter assignments:

- No accidental global creation

- Assignments to non-writeable values

- Deletion of non-removable fields

- Duplicate function parameters `function (a,a,c)`

- Duplicate field names `{"a": 1, "a": 2}`

- Global function calls have undefined `this`

Strict mode: function

```
function myStrictFunction() {  
    // Function-level strict mode syntax  
    "use strict";  
    function nested() {  
        return "And so am I!";  
    }  
    return `Hi! I'm a strict mode function!` + nested();  
}  
function myNotStrictFunction() {  
    return "I'm not strict."  
}
```

Can also apply for the entire file

Modules

Scope

Exports

Imports

Module exports

```
export const name = "square";

export function draw(ctx, length, x, y, color) {
  ctx.fillStyle = color;
  ctx.fillRect(x, y, length, length);

  return { length, x, y, color };
}
```

```
export { name, draw, reportArea, reportPerimeter };
```

Can export: variables, functions, classes

Module exports

```
export const name = "square";

export function draw(ctx, length, x, y, color) {
  ctx.fillStyle = color;
  ctx.fillRect(x, y, length, length);

  return { length, x, y, color };
}
```

```
export { name, draw, reportArea, reportPerimeter };
```

Can export: variables, functions, classes

Module exports

```
export const name = "square";

export function draw(ctx, length, x, y, color) {
  ctx.fillStyle = color;
  ctx.fillRect(x, y, length, length);

  return { length, x, y, color };
}
```

```
export { name, draw, reportArea, reportPerimeter };
```

Can export: variables, functions, classes

Module imports

```
import { name, draw, reportArea, reportPerimeter } from
    "./modules/square.js";
```

Always by an URL

```
<script type="importmap">
{
  "imports": {
    "shapes": "./shapes/square.js",
    "shapes/square": "./modules/shapes/square.js",
    "https://example.com/shapes/square.js":
      "./shapes/square.js",
    "https://example.com/shapes/": "/shapes/square/",
    "../shapes/square": "./shapes/square.js"
  }
}
</script>
```

App Organization and History API

Michael Chang
Spring 2023

Full stack topics

(Today) Web app organization

(Thursday) Authentication and users

CSS techniques and mobile styling

Accessibility

Week 10

Frontend frameworks, deployment, advanced techniques, wrap-up

Plan for today

Web app structure

Definition: single page application

Different strategies and their tradeoffs

location and history API

Changing URL without reloading page

Storing data across HTML pages

Note: route ordering

Express checks routes one by one

First added -> first checked

Uses first match

Example

```
api.post("/binky", ...);
```

```
api.use(bodyParser.json());
```

req.body not available in POST /binky route

```
api.all("/*", ...); /* handle all requests */
```

```
api.get("/", ...);
```

Second handler never called

Web app structure

How many HTML pages?

Traditional websites: one page per "view"

Pages might have dynamic content filled in by server

Single page application (SPA)

Only one HTML file; elements shown/hidden via JS

Dynamic content via fetch

Popular with frontend frameworks

Modern non-SPAs

Use fetch for dynamic content

Some showing/hiding to respond to events (user) or data (server)

If page sufficiently different, can make a new HTML file

Browser URL bar

Current model

URL shows exact file name the server will return

E.g. `http://localhost:1930/social.html`

Links point to specific HTML pages

E.g. `<a href="social.html">Part 3</a>`

Limitations

No dynamic URLs, sharable links

E.g. `http://localhost:1930/profile/mchang`

Changing URL -> reload from server

No smooth/fancy transitions, loading bars...

When page is an SPA, no "deep links"

Can't link to parts of app; can't use back button

Multi-step approach

0. Stick to current organization. It works fine.

For informational pages, pointing at HTML files is fine

If only a couple views, no need for deep links

0.5. Use URL hash

Hash part never sent to server; changing won't reload

E.g. `http://localhost:1930/social.html#mchang`

1. Server maps multiple URLs -> same HTML

Use location in frontend to fetch/display content

2. Use history API to change URL

Change URL without reloading page

Frontend: location

window.location: info about the loaded URL

.href: "http://localhost:1930/index.html?binky=winky#foo"

.protocol: "http:"

.host: "localhost:1930"

.pathname: "/index.html"

.search: "?binky=winky"

.hash: "#foo"

.assign(url)

Navigate to this URL (loaded from server)

.replace(url)

Replace URL (load from server, can't "Back" to current URL)

Backend: sending files

```
const PUBLIC_PATH = path.join(process.cwd(), "public");  
app.get("/profile/:id", (req, res) => {  
  res.sendFile("profile.html", { root: PUBLIC_PATH });  
});
```

res.sendFile(path, options)

Send the file as a response

path must be absolute, or options.root must be set

For security, probably best to always use root

Careful: if path is coming from user, could introduce security issues

E.g. "send me the file ../../../../secret.txt"

Using root helps, but still should be careful

Frontend: history API

window.history: interact with browser's URL bar and history

.pushState(state, title, url)

- Change URL bar without loading page

- No one uses title (pass "")

- Can pass "state" that isn't shown in URL

 - Accessed through `history.state`

.replaceState(state, title, url)

- Replace history entry with new one (no load, can't Back to current page)

.back, .forward, .go

- Programmatically move through browser history

Frontend: history API

popstate event on window

Fired when user (or program) moves through history (back, forward, etc.)

NOT fired when pushState or replaceState

Example

```
window.addEventListener("popstate", (event) => {  
  console.log(`Location: ${location.pathname}`);  
  loadProfile();  
});
```

Saving state across pages

window.sessionStorage

Allows you to save data on one page and read it on another

Or save state while refreshing the page

Reminder: frontend only, backend is stateless

Example

```
window.sessionStorage.setItem("user", "mchang");
```

```
let user = window.sessionStorage.getItem("user");
```

```
window.sessionStorage.removeItem("user");
```

Summary

Today

Assorted things that may be relevant to your project

Before next time

assign3.2 ongoing

Next time: auth and users

Tracking who's logged in, API tokens

Using 3rd-party auth (like Google), OAuth

Security considerations