Schedule

JavaScript:

- Communicating between classes

- Loading data from files
  - Fetch API
  - Promises - High-level!
  - JSON

Communicating between classes

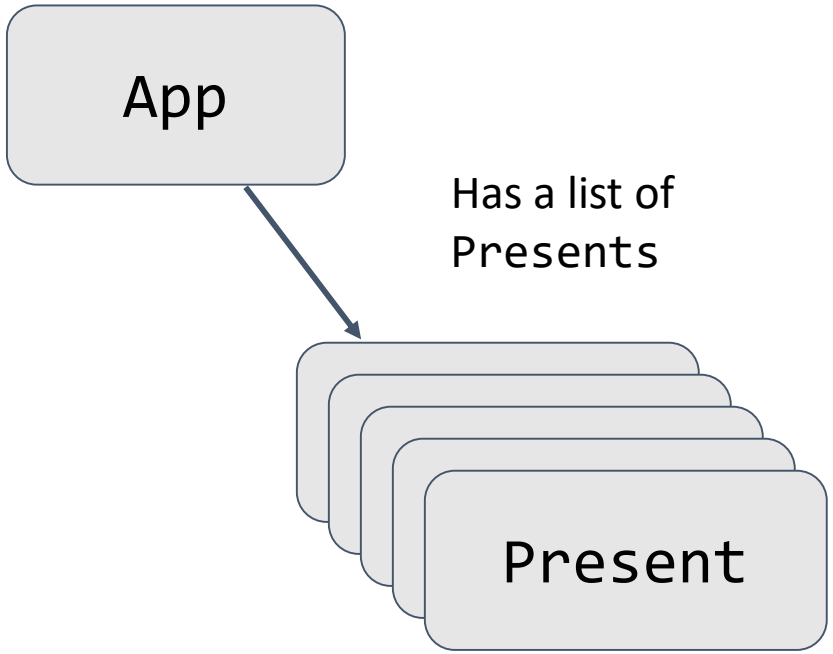
Multiple classes

Let's say that we have multiple presents now (CodePen):

Multiple classes

And we have implemented this with two classes:

- App: Represents the entire page
  - `Present`: Represents a single present



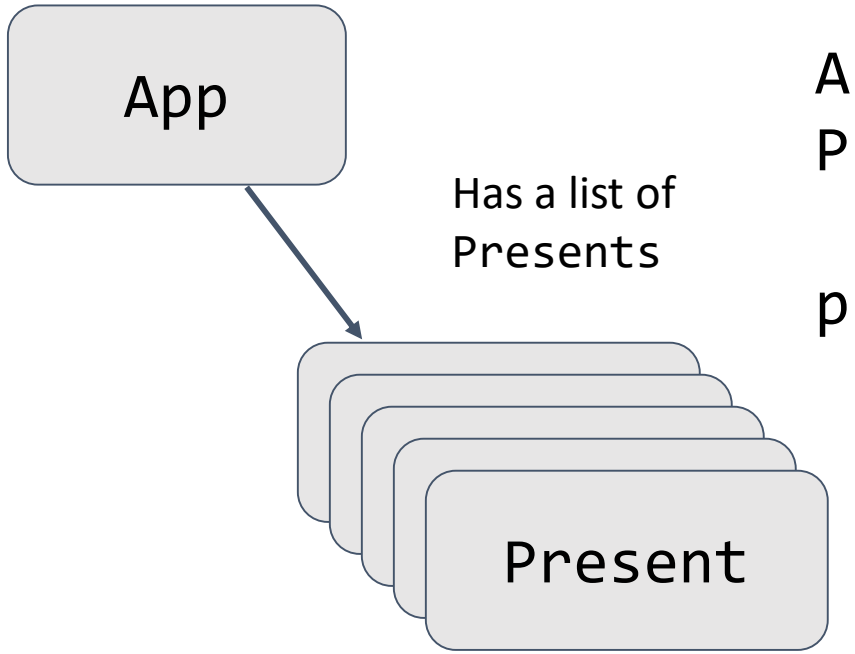
CodePen

Communicating btwn classes

What if we want to change the **title** when all present have been opened? (CodePen)

Communication btwn classes

Communicating from App → `Present` is easy, since App has a list of the `Present` objects.

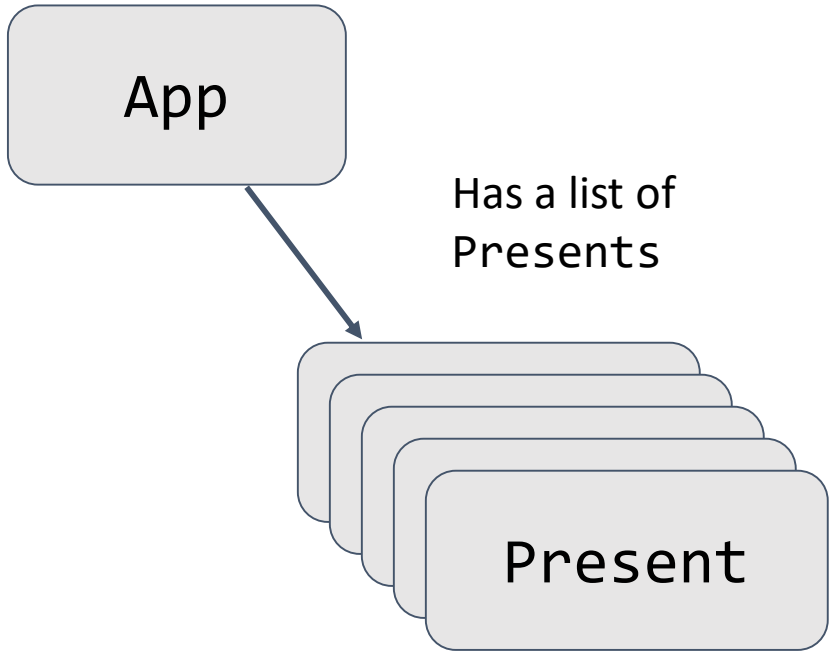


App can just call methods on `Present`:

```
present.doWhatever();
```

Communication btwn classes

However, communicating `Present` → App is not as easy, because `Presents` do not have a reference to App



Communicating btwn classes

You have three general approaches:

1. Add a reference to App in Photo

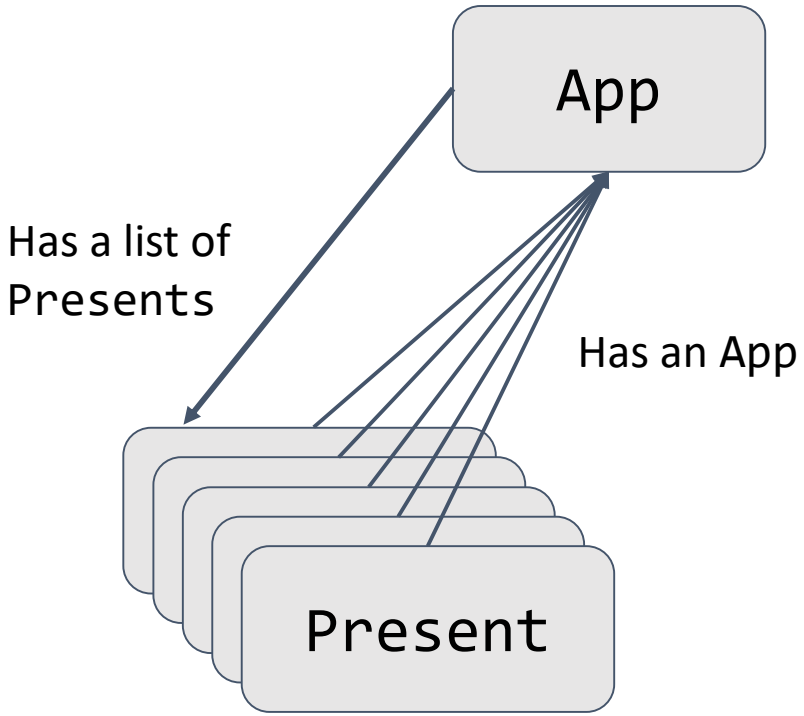
   This is poor software engineering, though we will allow it on the homework because this is not an OO design class
2. Fire a custom event

   OK (don't forget to bind)
3. Add onOpened "callback function" to Present

   Best option (don't forget to bind)

Terrible style: Presents own App

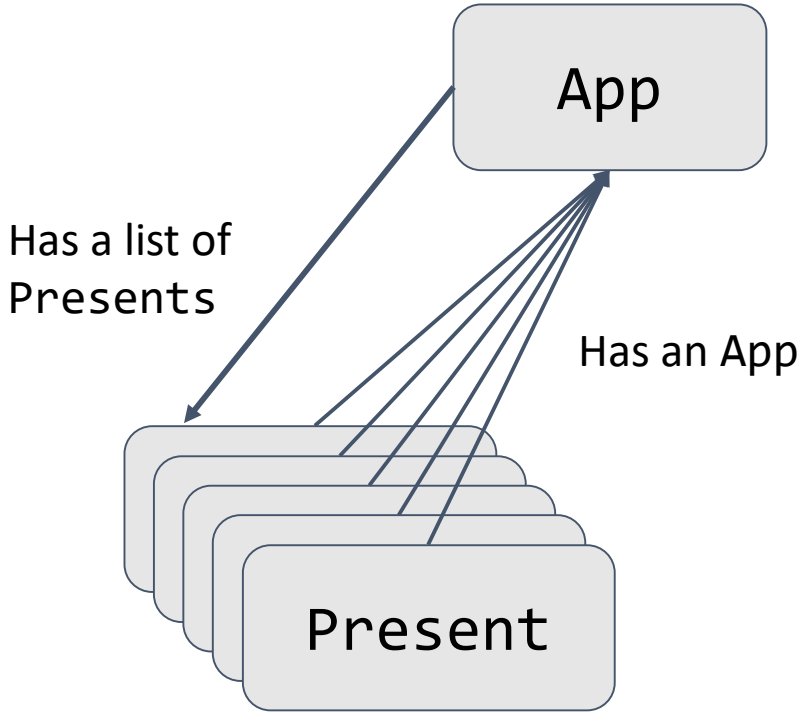
A naive fix is to just give `Present` a reference to App in its constructor: CodePen



(Please don't do this.)

Terrible style: Presents own App

This is the easiest workaround, but **it's terrible software engineering.**



- Logically doesn't make sense: a `Present` doesn't have an App
- Gives `Present` way too much access to App
- Especially bad in JS with no private fields/ methods yet

Custom events

Custom Events

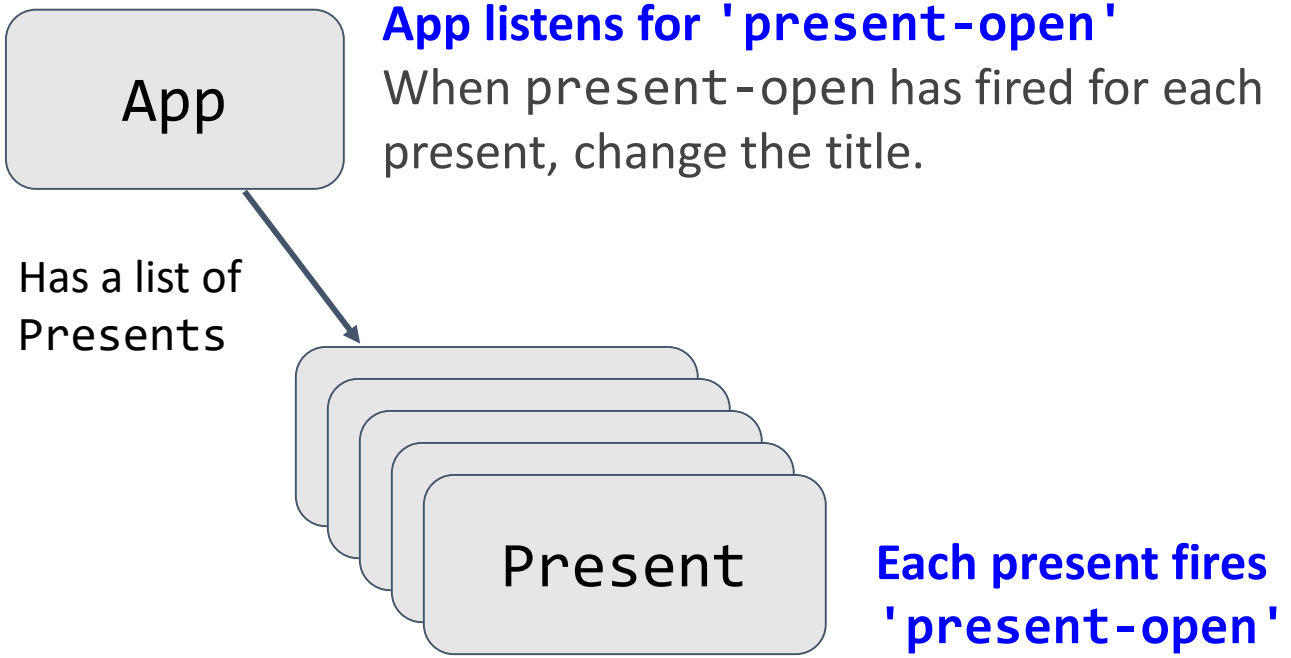
You can listen to and dispatch Custom Events to communicate between classes (mdn):

```
const event = new CustomEvent(
    eventNameString, optionalParameterObject);
element.addEventListener(eventNameString);
element.dispatchEvent(eventNameString);
```

However, CustomEvent **can only be listened to / dispatched on HTML elements,** and not on arbitrary class instances.

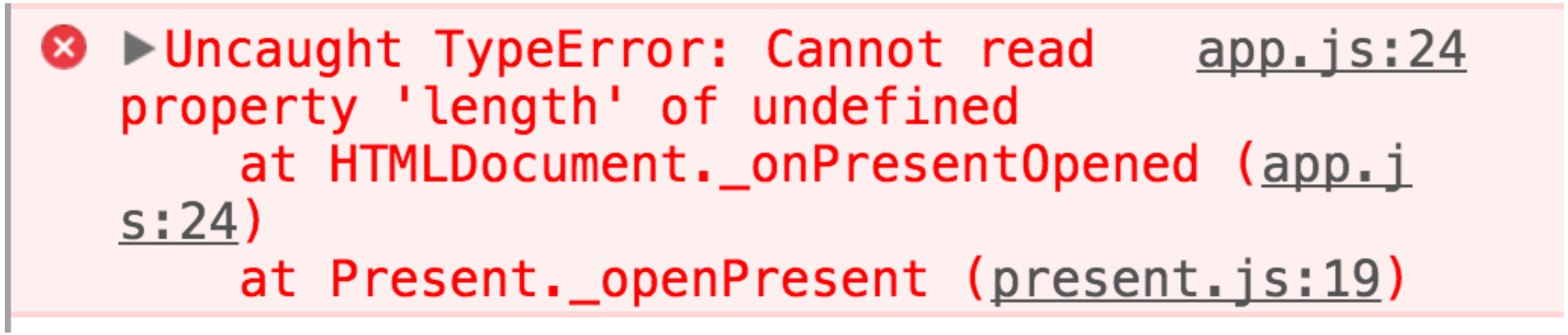
Custom Events: Present example

Let's have the App listen for the `'present-open'` event...



[CodePen attempt]

`this` in event handler



Our first attempt at solution results in errors again!
(CodePen attempt)

Solution: `bind`

To make `this` always refer to the instance object for a method in the class (i.e. to get `this` to behave as you'd expect), you can add the following line of code in the `constructor`:

```
this.methodName = this.methodName.bind(this);
```

```
this._onPresentOpened = this._onPresentOpened.bind(this);
```

[CodePen solution]

First-class functions

Recall: `addEventListener`

Over the last few weeks, we've been using **functions** as a parameter to `addEventListener`:

```
image.addEventListener(
    'click', this._openPresent);
```

First-class functions

JavaScript is a language that supports **first-class functions**, i.e. functions are treated like **variables of type Function**:

- Can be passed as parameters
- Can be saved in variables
- Can be defined without a name / indentifier
  - Also called an **anonymous function**
  - Also called a **lambda function**
  - Also called a **function literal value**

Function variables

You can declare a function in several ways:

```
function myFunction(params) {
}

const myFunction = function(params) {
};

const myFunction = (params) => {
};
```

Function variables

```
function myFunction(params) {
}
```

```
const myFunction = function(params) {
};
```

```
const myFunction = (params) => {
};
```

Functions are invoked in the same way, regardless of how they were declared:

```
myFunction();
```

Simple, contrived example

```
function greetings(greeterFunction) {
  greeterFunction();
}

const worldGreeting = function() {
  console.log('hello world');
};

const hawaiianGreeting = () => {
  console.log('aloha');
};

greetings(worldGreeting);
greetings(hawaiianGreeting);
```

CodePen

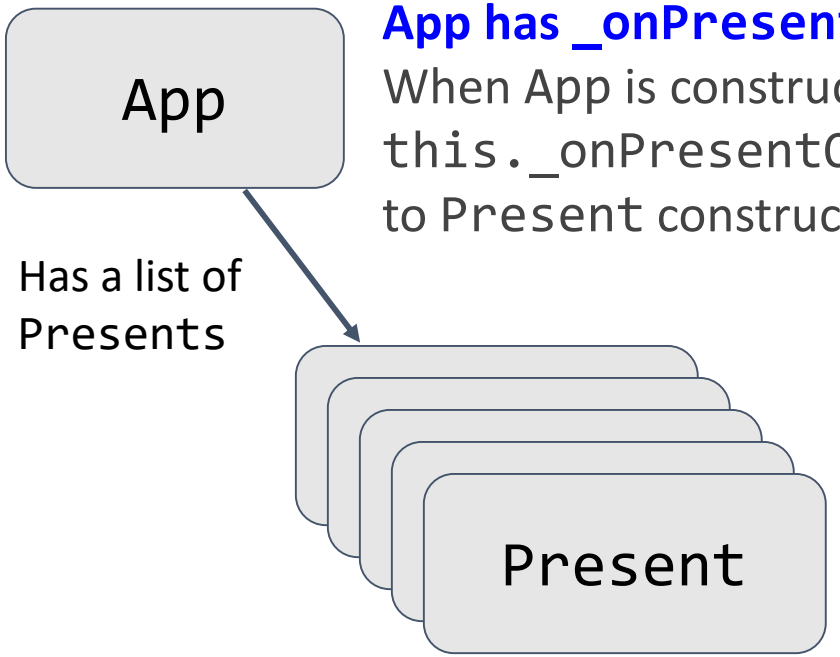
A real example: Callbacks

Another way we can communicate between classes is through callback functions:

- **Callback**: A function that's passed as a parameter to another function, usually in response to something.

Callback: Present example

Let's have `Presents` communicate with App via callback `parameter`: (CodePen attempt)



App has `_onPresentOpened` method

When App is constructing `Presents`, pass its `this._onPresentOpened` method as parameter to `Present` constructor

Each `Present` saves an `onOpenCallback` parameter in the constructor

When the present is opened, fire the callback

`this` in a method

`this` in different contexts

`this` in a constructor:

- `this` is set to the new object being created

`this` in a function firing in response to a DOM event:

- `this` is set to the DOM element to which the event handler was attached

`this` being called as a method on an object:

- `this` is set to the object that is calling the method, or the object on which the method is called.

(all values of `this`)

One more look at `bind`

Objects in JS

Objects in JavaScript are sets of property-value pairs:

```
const bear = {
  name: 'Ice Bear',
  hobbies: ['knitting', 'cooking', 'dancing']
};
```

Classes in JS

```
class Playlist {
  constructor(name) {
    this.playlistName = name;
    this.songs = [];
  }

  addSong(songName) {
    this.songs.push(songName);
  }
}

const playlist = new Playlist('More Life');
playlist.addSong('Passionfruit');
```

Classes in JavaScript produce **objects** through new.
(CodePen)

Classes in JS

```
class Playlist {
  constructor(name) {
    this.playlistName = name;
    this.songs = [];
  }

  addSong(songName) {
    this.songs.push(songName);
  }
}

const playlist = new Playlist('More Life');
playlist.addSong('Passionfruit');
```

Q: Are the objects created from classes also sets of property-value pairs?

Classes and objects

```
const playlist = new Playlist('More Life');
```

A: Yes. The `playlist` object created by the constructor essentially* looks like this:

```
{
  playlistName: 'More Life',
  songs: [],
  addSong: function(songName) {
    this.songs.push(songName);
  }
}
```

Technically addSong (and the constructor function) is defined in the prototype of the playlist object, but we haven't talked about prototypes and probably won't talk about prototypes until the end of the quarter.

Classes and objects

```
const playlist = new Playlist('More Life');
```

In JavaScript, a **method** of an object is just a **property** whose value is of `Function` type.

```
{
  playlistName: 'More Life',
  songs: [],
  addSong: function(songName) {
    this.songs.push(songName);
  }
}
```

Classes and objects

```
const playlist = new Playlist('More Life');
```

In JavaScript, a **method** of an object is just a **property** whose value is of `Function` type.

```
{
  playlistName: 'More Life',
  songs: [],
  addSong: function(songName) {
    this.songs.push(songName);
  }
}
```

And just like any other Object property, the value of that method can be changed.

Rewriting a function

```
class Playlist {
  constructor(name) {
    this.playlistName = name;
    this.songs = [];
  }

  addSong(songName) {
    this.songs.push(songName);
  }
}

const playlist = new Playlist('More Life');
playlist.addSong = function(songName) {
  console.log("Nah");
};

playlist.addSong('Passionfruit');
console.log(playlist);
```

Q: What is the output of this code?

CodePen

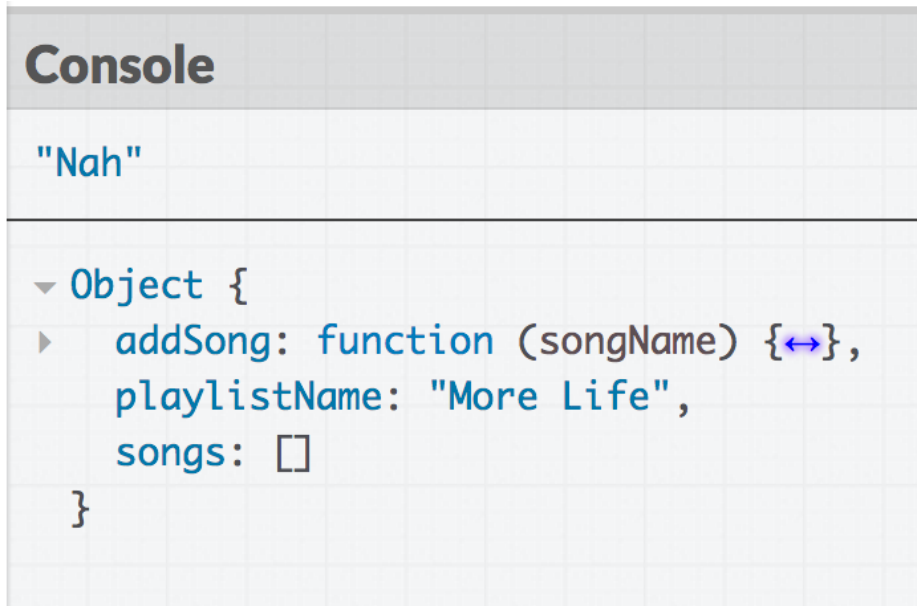
Rewriting a function

```
class Playlist {
  constructor(name) {
    this.playlistName = name;
    this.songs = [];
  }

  addSong(songName) {
    this.songs.push(songName);
  }
}

const playlist = new Playlist('More Life');
playlist.addSong = function(songName) {
  console.log("Nah");
};

playlist.addSong('Passionfruit');
console.log(playlist);
```



When would you ever want to rewrite the definition of a method?!

`bind` in classes

```
constructor() {
  const someValue = this;
  this.onClick = this.onClick.bind(someValue);
}
```

The code in purple is saying:

- Make a copy of `onClick`, which will be the exact same as `onClick` except `this` in `onClick` is always set to the `someValue`

`bind` in classes

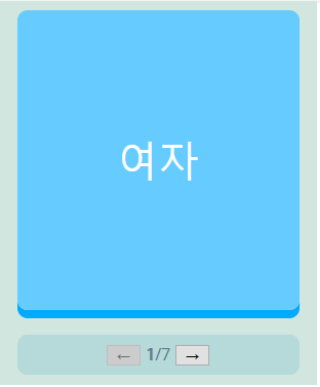
```
constructor() {
  const someValue = this;
  this.onClick = this.onClick.bind(someValue);
}
```

The code in purple is **rewriting the `onClick` property** of the object:

- Assign the value of the **`onClick`** property: set it to the new function returned by the call to `bind`

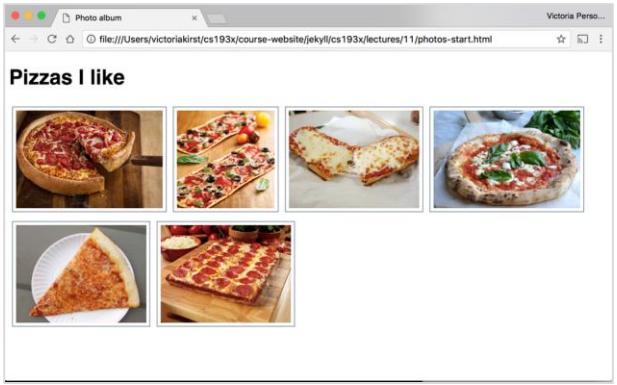
Loading data from file

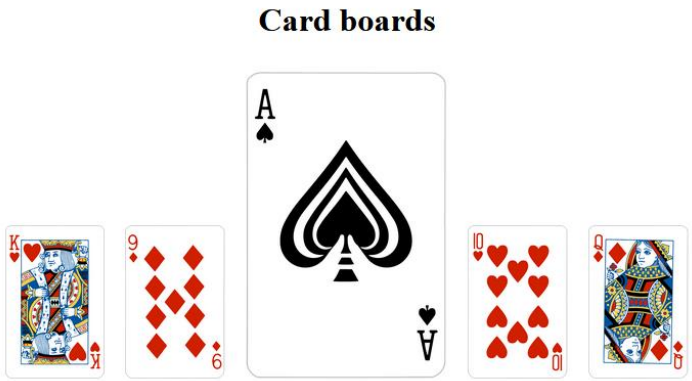
Our client applications



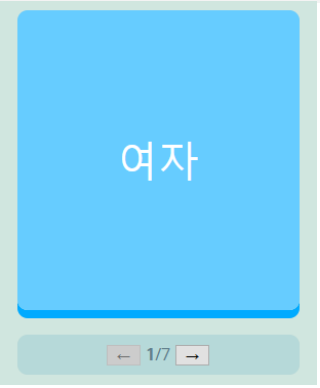
Lecture: Presents, Photo album (case study)
Tutorials: Playing Cards, Flash cards

WHERE IS THE DATA FROM?

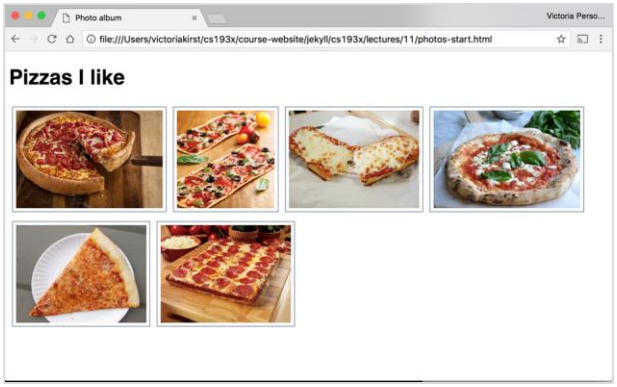




Our client applications



Lecture: Presents, Photo album (read more)
Tutorials: Playing Cards, Flash cards



WHERE IS THE DATA FROM?

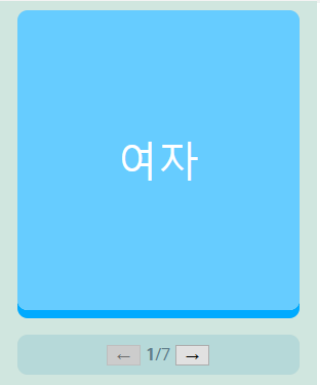
Files

photo-list.js,
present-sources.js,
data.js, …

Card boards

Enjoy your presents!

Our client applications

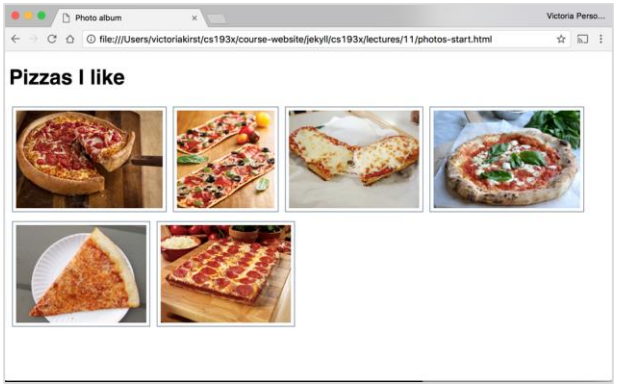


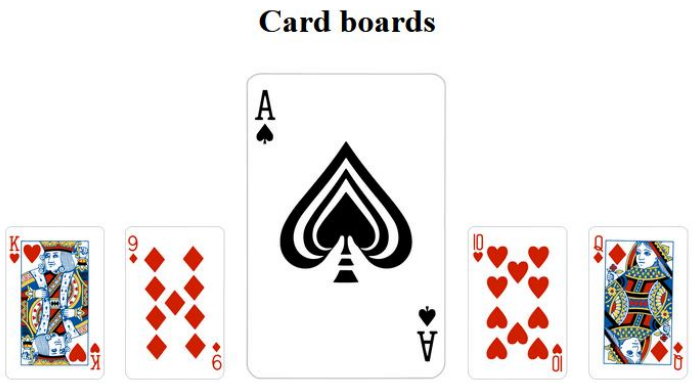
Lecture: Presents, Photo album (read more)
Tutorials: Playing Cards, Flash cards

WHERE IS THE DATA FROM?

Files

FIXED





Our client applications

BUT PRACTICALLY,
WHERE IS THE DATA FROM?

Our client applications

BUT PRACTICALLY,
WHERE IS THE DATA FROM?

Database, [3rd party] API

Our client applications

BUT PRACTICALLY,
WHERE IS THE DATA FROM?

Database*, [3rd party] API+

DYNAMIC

* Database = Files + Database Management System (DBMS)
+ Facebook API, Google API, Github API…

Loading data from files

Loading data from a file

What if you had a list of images in a text file that you wanted to load in your web page?

```
1  https://media1.giphy.com/media/xNT2CcLjhbI0U/200.gif
2  https://media2.giphy.com/media/3o7btM3VVVNtssGReo/200.gif
3  https://media1.giphy.com/media/l3q2uxEzLIE8cWMq4/200.gif
4  https://media2.giphy.com/media/LDwL3ao61wfHa/200.gif
5  https://media1.giphy.com/media/3o7TKMt1VVNkHV2PaE/200.gif
6  https://media3.giphy.com/media/DNQFjMJbbsNmU/200.gif
7  https://media1.giphy.com/media/26FKTsKMKtUSomuNq/200.gif
8  https://media1.giphy.com/media/xThuW5Hf2N8idJHFVS/200.gif
9  https://media1.giphy.com/media/XlFfSD0CiyGLC/200.gif
10 https://media3.giphy.com/media/ZaBHSbiLQTmFi/200.gif
11 https://media3.giphy.com/media/JPbZwjMcxJYic/200.gif
12 https://media1.giphy.com/media/FArgGzk7KO14k/200.gif
13 https://media1.giphy.com/media/UFoLN1EyKjLbi/200.gif
14 https://media1.giphy.com/media/11zXBCAb9soCQM/200.gif
15 https://media4.giphy.com/media/xUPGcHeIeZMmTcDQJy/200.gif
16 https://media2.giphy.com/media/apZwWJInOBvos/200.gif
17 https://media2.giphy.com/media/sB4nvt5xIiNig/200.gif
18 https://media0.giphy.com/media/Y8Bi9lCOzXRkY/200.gif
19 https://media1.giphy.com/media/12wUXjm6f8Hhcc/200.gif
20 https://media4.giphy.com/media/26gsuVyk5fKB1YAAE/200.gif
21 https://media3.giphy.com/media/l2SpMU9sWIvT2nrCo/200.gif
22 https://media2.giphy.com/media/kR1vWazNc7972/200.gif
23 https://media4.giphy.com/media/Tv3m2GAAl2Re8/200.gif
24 https://media2.giphy.com/media/9nujydsBLz2dq/200.gif
25 https://media3.giphy.com/media/AG39l0rHgkRLa/200.gif
```

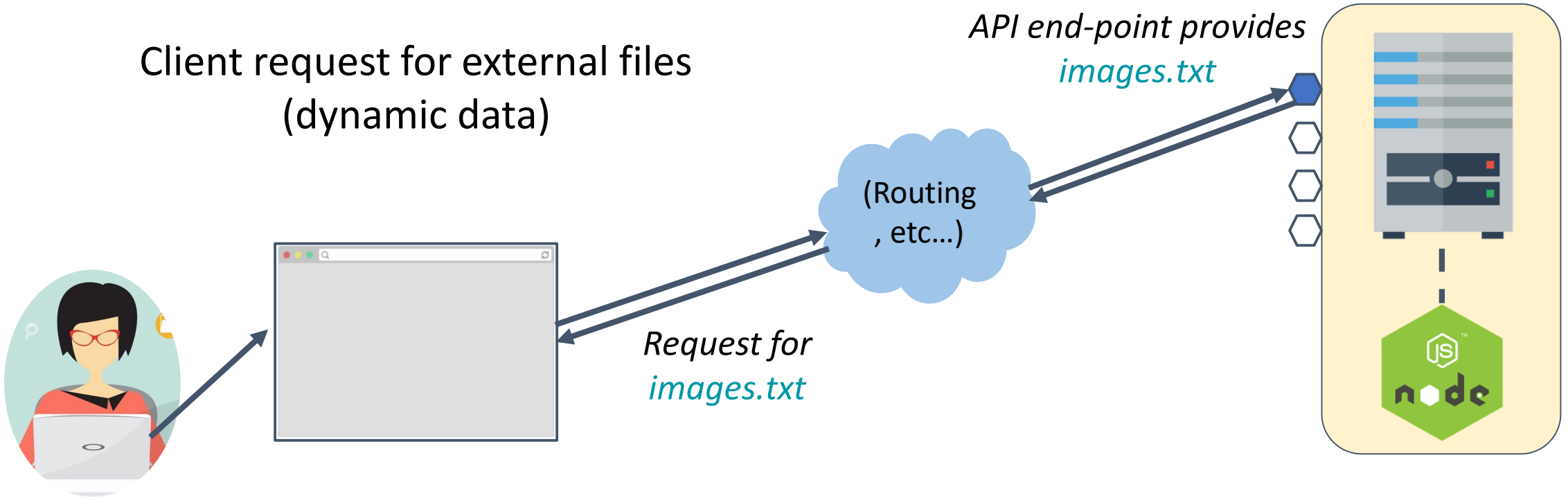
Intuition: `loadFromFile`

If we wanted to have an API to load **external** files in JavaScript, it might look something like this:

```
// FAKE HYPOTHETICAL API.
// This is not real a JavaScript function!
const contents = loadFromFile('images.txt');
```

Design artifacts

Client request for external files
(dynamic data)



Server provides **API end-points** to serve “data”

API endpoint example

Look at the URL for this Google slide deck:

`https://docs.google.com/presentation/d/1Rim3-IXt6yN7yny_SBv7B5NMBiYbaQEiRMUD5s66uN8`

- **`presentation`**: Tells the server that we are requesting a doc of type "presentation"
- **`d/1Rim3-IXt6yN7yny_SBv7B5NMBiYbaQEiRMUD5s66uN8`**: Tells the server to request a doc ("d") with the document id of "1Rim3-IXt6yN7yny_SBv7B5NMBiYbaQEiRMUD5s66uN8"

Example: Spotify

Spotify has a REST API that external developers (i.e. people who aren't Spotify employees) can query:

Our Web API endpoints give external applications access to Spotify catalog and user data.

Web API Base URL: `https://api.spotify.com` User Guide | Tutorial | Code Examples

Search:

METHOD	ENDPOINT	USAGE	RETURNS
GET	/v1/albums/{id}	Get an album	album
GET	/v1/albums?ids={ids}	Get several albums	albums
GET	/v1/albums/{id}/tracks	Get an album's tracks	tracks*
GET	/v1/artists/{id}	Get an artist	artist
GET	/v1/artists?ids={ids}	Get several artists	artists
GET	/v1/artists/{id}/albums	Get an artist's albums	albums*

Intuition: `loadFromFile`

```
// FAKE HYPOTHETICAL API.
// This is not real a JavaScript function!
const contents = loadFromFile('images.txt');
```

A few problems with this hypothetical fake API:

- We want to load the file **asynchronously**: the JavaScript should not block while we're loading the file
- There's no way to check the status of the request. What if the resource didn't exist? What if we're not allowed to access the resource?

Intuition: `loadFromFile`

An asynchronous version of this API might look like this:

```
// FAKE HYPOTHETICAL API.
// This is not real a JavaScript function!
function onSuccess(response) {
  const body = response.text;
  ...
}
loadFromFile('images.txt', onSuccess, onFail);
```

Where *`onSuccess`* and *`onFail`* are callback functions that should fire if the request succeeded or failed, respectively.

Fetch API

Fetch API: `fetch()`

The Fetch API is the API to use to load external resources (text, JSON, etc) in the browser.

The Fetch API is made up of one function, and its syntax is concise and easy to use:

```
fetch('images.txt');
```

Note: `XMLHttpRequest` ("XHR") is the old API for loading resources from the browser. XHR still works, but is clunky and harder to use.

Fetch API: `fetch()`

The Fetch API is the API to use to load external resources (text, JSON, etc) in the browser.

The Fetch API is made up of one function, and its syntax is concise and easy to use:

```
fetch('images.txt');
```

- The `fetch()` method takes the string path to the resource you want to fetch as a parameter
- It returns a `Promise`

Fetch API: `fetch()`

The Fetch API is the API to use to load external resources (text, JSON, etc) in the browser.

The Fetch API is made up of one function, and its syntax is concise and easy to use:

```
fetch('images.txt');
```

- The `fetch()` method takes the string path to the resource you want to fetch as a parameter
- It returns a `Promise`
  - **What the heck is a `Promise`?**

`Promise`s:
Another conceptual odyssey

`Promises` and `.then()`

A `Promise`:

- An object used to manage asynchronous results
- Has a *`then()`* method that lets you attach functions to execute *onSuccess* or *onError*
- Allows you to build **chains** of asynchronous results.

`Promises` are easier to **use** than to **define**...

Simple example: getUserMedia

There is an API called `getUserMedia` that allows you get the media stream from your webcam.

There are two versions of getUserMedia:

- `navigator.getUserMedia` **(deprecated)**
  - Uses callbacks

- `navigator.mediaDevices.getUserMedia`
  - Returns a `Promise`

getUserMedia with callbacks

```
const video = document.querySelector('video');

function onCameraOpen(stream) {
  video.srcObject = stream;
}

function onError(error) {
  console.log(error);
}

navigator.getUserMedia({ video: true },
  onCameraOpen, onError);
```

CodePen

getUserMedia with Promise

```
const video = document.querySelector('video');

function onCameraOpen(stream) {
  video.srcObject = stream;
}

function onError(error) {
  console.log(error);
}

navigator.mediaDevices.getUserMedia({ video: true })
  .then(onCameraOpen, onError);
```

CodePen

Hypothetical Fetch API

```
// FAKE HYPOTHETICAL API.
// This is not how fetch is called!
function onSuccess(response) {
  ...
}

function onFail(response) {
  ...
}

fetch('images.txt', onSuccess, onFail);
```

Real Fetch API

```
function onSuccess(response) {
  ...
}

function onFail(response) {
  ...
}

fetch('images.txt').then(onSuccess, onFail);
```

Promise syntax

Q: How does this syntax work?

```
fetch('images.txt').then(onSuccess, onFail);
```

Promise syntax

Q: How does this syntax work?

```
fetch('images.txt').then(onSuccess, onFail);
```

The syntax above is the same as:

```
const promise = fetch('images.txt');
promise.then(onSuccess, onFail);
```

Promise syntax

```
const promise = fetch('images.txt');
promise.then(onSuccess, onFail);
```

The object `fetch` returns is of type `Promise`.

A promise is in one of three states:

- **pending**: initial state, not fulfilled or rejected.
- **fulfilled**: the operation completed successfully.
- **rejected**: the operation failed.

You attach handlers to the promise via `.then()`

Promise syntax

```
const promise = fetch('images.txt');
promise.
```

The objec

A promis

- **pendi**
- **fulfill**
- **reject**

(Right now we will just use Promises.)

You attach handlers to the promise via `.then()`

Using Fetch

```
function onSuccess(response) {
  console.log(response.status);
}

fetch('images.txt').then(onSuccess);
```

The success function for Fetch gets a response parameter:

- `response.status`: Contains the status code for the request, e.g. 200 for HTTP success
  - HTTP status codes

Fetch attempt

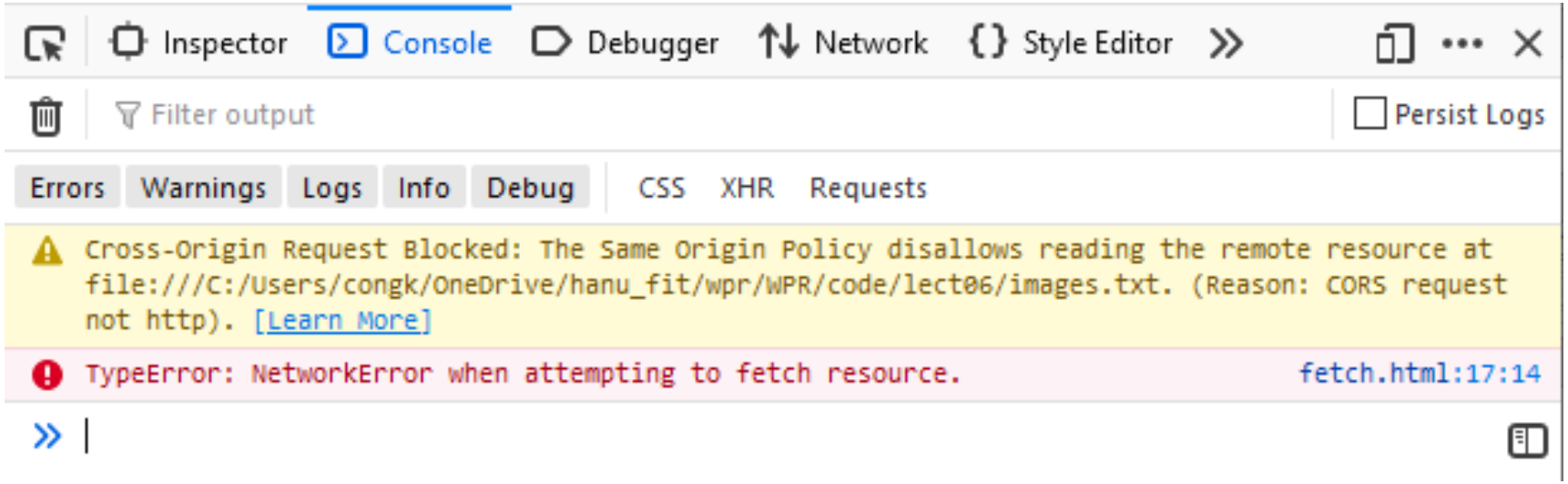
```
function onSuccess(response) {
  console.log(response.status);
}

function onError(error) {
  console.log('Error: ' + error);
}

fetch('images.txt')
    .then(onSuccess, onError);
```

Fetch error

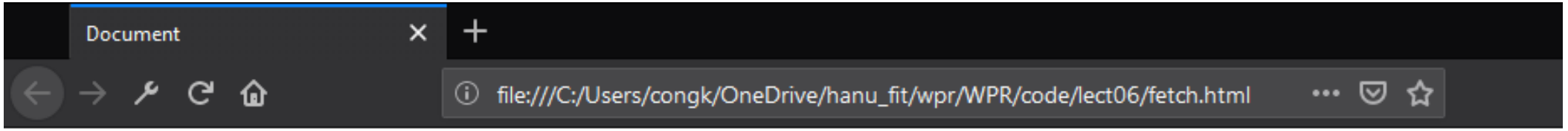
If we try to load this in the browser, we get the following JavaScript error:



Notice that our onError function was also called.

Local files

When we load a web page in the browser that is saved on our computer, it is served via `file://` protocol:



We are **not allowed** to load files in JavaScript from the `file://` protocol, which is why we got the error.

Serve local files over HTTP

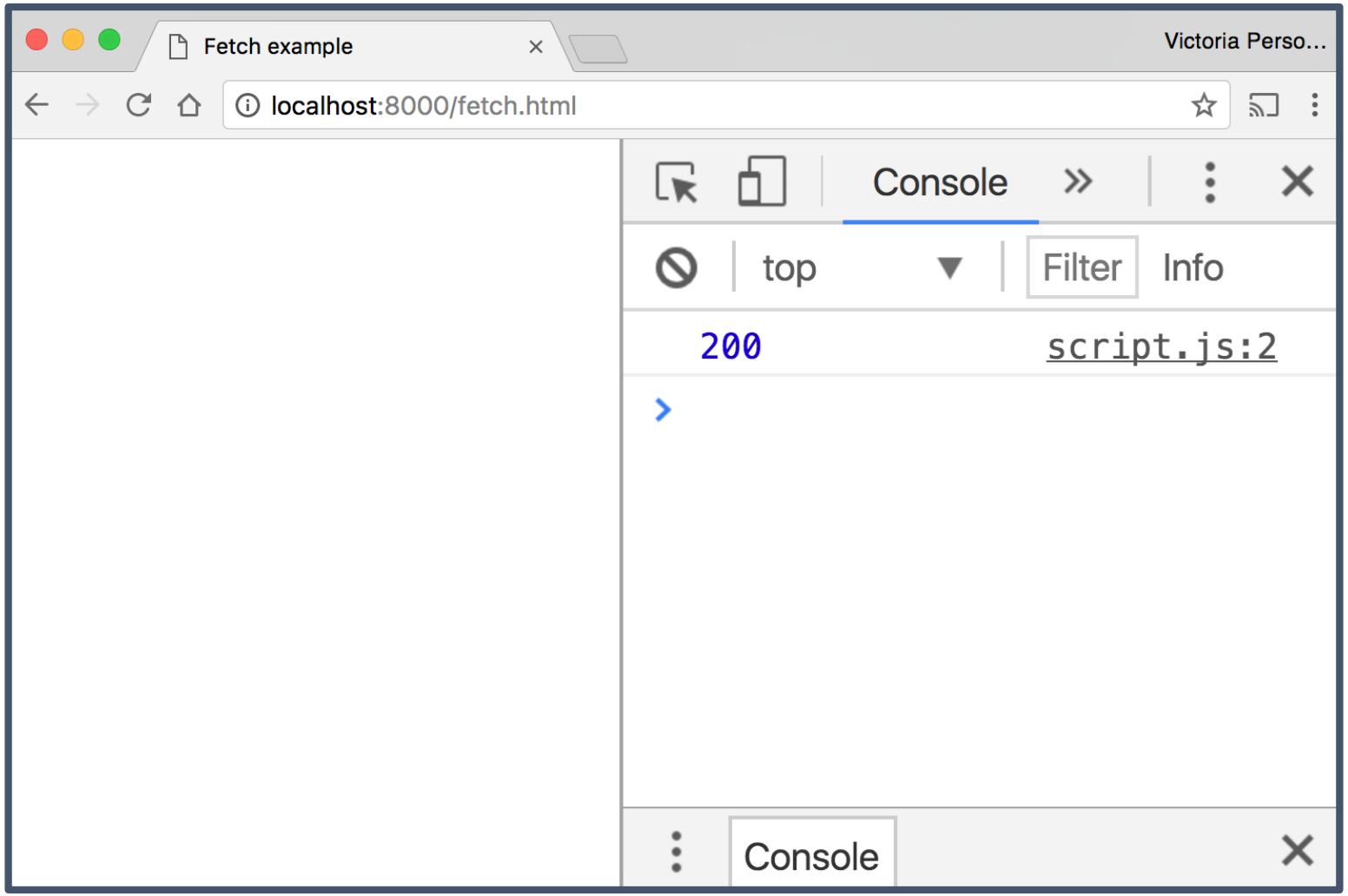
From Visual Studio Code search for extension named **Live Sever**

Open folder containing the source code

Click **Go Live** (bottom right corner)

This now starts up a **server** that can load the files in the current directory over HTTP.

- We can access this server by navigating to: `http://localhost:5000/fetch-demo/`



We got HTTP response 200, which is success! (codes)

How do we get the data from `fetch()`?

Using Fetch

```
function onSuccess(response) {
  ..
}
fetch('images.txt').then(onSuccess);
```

- `response.status`: Status code for the request
- `response.text()`:
  - **Asynchronously** reads the response stream
  - **Returns a `Promise`** that resolves with the string containing the response stream data.

text() Promise

Q: How do we change the following code to print out the response body?

```
function onSuccess(response) {
  console.log(response.status);
}

function onError(error) {
  console.log('Error: ' + error);
}

fetch('images.txt')
    .then(onSuccess, onError);
```

```
function onStreamProcessed(text) {
  console.log(text);
}

function onResponse(response) {
  console.log(response.status);
  response.text().then(onStreamProcessed);
}

function onError(error) {
  console.log('Error: ' + error);
}

fetch('images.txt').then(onResponse, onError);
```

Chaining Promises

We want the following asynchronous actions to be completed in this order:

1. When the `fetch` completes, run `onResponse`
2. When `response.text()` completes, run `onStreamProcessed`

```
function onStreamProcessed(text) { … }
function onResponse(response) {
  response.text().then(onStreamProcessed);
}
fetch('images.txt').then(onResponse, onError);
```

We can rewrite this:

```
function onStreamProcessed(text) {
  console.log(text);
}

function onResponse(response) {
  response.text().then(onStreamProcessed);
}

function onError(error) {
  console.log('Error: ' + error);
}

fetch('images.txt').then(onResponse,
onError);
```

We can rewrite this:

```
function onStreamProcessed(text) {
  console.log(text);
}

function onResponse(response) {
  return response.text();
}

function onError(error) {
  console.log('Error: ' + error);
}

fetch('images.txt')
    .then(onResponse, onError)
    .then(onStreamProcessed);
```

Chaining Promises

```
function onStreamProcessed(text) {
  console.log(text);
}

function onResponse(response) {
  return response.text();
}

fetch('images.txt')
    .then(onResponse, onError)
    .then(onStreamProcessed);
```

Chaining Promises

```
function onStreamProcessed(text) {
  console.log(text);
}

function onResponse(response) {
  return response.text();
}

const responsePromise = fetch('images.txt')
    .then(onResponse, onError)
responsePromise.then(onStreamProcessed);
```

The Promise returned by `onResponse` is effectively* the Promise returned by `fetch`. (*Not actually what's happening, but that's how we'll think about it for right now.)

Chaining Promises

```
function onStreamProcessed(text) {
  console.log(text);
}

function onResponse(response) {
  return response.text();
}

fetch('images.txt')
    .then(onResponse, onError)
    .then(onStreamProcessed);
```

If we don't think about it too hard, the syntax is fairly intuitive.

We'll think about this more deeply later!

Completed example

```
function onStreamProcessed(text) {
  const urls = text.split('\n');
  for (const url of urls) {
    const image = document.createElement('img');
    image.src = url;
    document.body.append(image);
  }
}
function onSuccess(response) {
  response.text().then(onStreamProcessed)
}
function onError(error) {
  console.log('Error: ' + error);
}

fetch('images.txt').then(onSuccess, onError);
```

JSON

JavaScript Object Notation

JSON: Stands for **J**ava**S**cript **O**bject **N**otation

- Created by Douglas Crockford
- Defines a way of **serializing** JavaScript objects
  - **to serialize**: to turn an object into a string that can be deserialized
  - **to deserialize**: to turn a serialized string into an object

JSON.stringify()

We can use the `JSON.stringify()` function to seralize a JavaScript object:

```
const bear = {
  name: 'Ice Bear',
  hobbies: ['knitting', 'cooking', 'dancing']
};

const serializedBear =
JSON.stringify(bear);
console.log(serializedBear);
```

CodePen

JSON.parse()

We can use the `JSON.parse()` function to deseralize a JavaScript object:

```
const bearString = '{"name":"Ice Bear","hobbies":["knitting","cooking","dancing"]}';

const bear = JSON.parse(bearString);
console.log(bear);
```

CodePen

Fetch API and JSON

The Fetch API also has built-in support for JSON:

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

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

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

Return `response.json()` instead of `response.text()` and Fetch will essentially call `JSON.parse()` on the response string.

Why JSON?

Let's say we had a file that contained a list of albums.

Each album has:
- Title
- Year
- URL to album image

We want to display each album in chronological order.

Text file?

We could create a text file formatted consistently in some format that we make up ourselves, e.g.:

```
The Emancipation Of Mimi
2005
https://i.scdn.co/image/dca82bd9c1ccae90b09972027a408068f7a4d700

Daydream
1995
https://i.scdn.co/image/0638f0ddf70003cb94b43aa5e4004d85da94f99c

E=MC²
2008
https://i.scdn.co/image/bca35d49f6033324d2518656531c9a89135c0ea3

Mariah Carey
1990
```

Text file processing

We would have to write all this custom file processing code:

- Must convert numbers from strings
- If you ever add another attribute to the album, we'd have to change our array indices

```
function onTextReady(text) {
  const lines = text.split('\n\n');
  const albums = [];
  for (let i = 0; i < lines.length; i++) {
    const infoText = lines[i];
    const infoStrings = infoText.split('\n');
    const name = infoStrings[0];
    const year = infoStrings[1];
    const url = infoStrings[2];
    albums.push({
      name: name,
      year: parseInt(year),
      url: url
    });
  }
  ...
}
```

Live example / GitHub

JSON file

It'd be much more convenient to store the file in JSON format:

```
{
  "albums": [
    {
      "name": "The Emancipation Of Mimi",
      "year": 2005,
      "url":
"https://i.scdn.co/image/dca82bd9c1ccae90b09972027a408068f7a4d700
"
    },
    {
      "name": "Daydream",
      "year": 1995,
      "url":
"https://i.scdn.co/image/0638f0ddf70003cb94b43aa5e4004d85da94f99c
"
    },
    {
```

JSON processing

Since we're using JSON, we don't have to manually convert the response strings to a JavaScript object:

- JavaScript has built-in support to convert a JSON string into a JavaScript object.

```
function onJsonReady(json) {
  const albums = json.albums;
  ...
}
```

Live example / GitHub

JavaScript Object Notation

JSON: Stands for **J**ava**S**cript **O**bject **N**otation

- Created by Douglas Crockford
- Defines a way of **serializing** JavaScript objects
  - **to serialize**: to turn an object into a string that can be deserialized
  - **to deserialize**: to turn a serialized string into an object
- `JSON.stringify(`***object***`)` returns a string representing ***object*** serialized in JSON format
- `JSON.parse(`***jsonString***`)` returns a JS object from the ***jsonString*** serialized in JSON format

JSON.stringify()

We can use the `JSON.stringify()` function to seralize a JavaScript object:

```
const bear = {
  name: 'Ice Bear',
  hobbies: ['knitting', 'cooking', 'dancing']
};

const serializedBear = JSON.stringify(bear);
console.log(serializedBear);
```

CodePen

JSON.parse()

We can use the `JSON.parse()` function to deseralize a JavaScript object:

```
const bearString = '{"name":"Ice
Bear","hobbies":["knitting","cooking","dancing"]}';

const bear = JSON.parse(bearString);
console.log(bear);
```

CodePen

Why JSON?

JSON is a useful format for storing data that we can load into a JavaScript API via fetch().

Let's say we had a list of Songs and Titles.

- If we stored it as a text file, we would have to know how we are separating song name vs title, etc
- If we stored it as a JSON file, we can just deserialize the object.

JSON

songs.json

```json
{
  "cranes": {
    "fileName": "solange-cranes-kaytranada.mp3",
    "artist": "Solange",
    "title": "Cranes in the Sky [KAYTRANADA Remix]"
  },
  "timeless": {
    "fileName": "james-blake-timeless.mp3",
    "artist": "James Blake",
    "title": "Timeless"
  },
  "knock": {
    "fileName": "knockknock.mp4",
    "artist": "Twice",
    "title": "Knock Knock"
  },
  "deep": {
    "fileName": "janet-jackson-go-deep.mp3",
    "artist": "Janet Jackson",
    "title": "Go Deep [Alesia Remix]"
  },
  "discretion": {
    "fileName": "mitis-innocent-discretion.mp3",
    "artist": "MitiS",
    "title": "Innocent Discretion"
  },
  "spear": {
    "fileName": "toby-fox-spear-of-justice.mp3",
    "artist": "Toby Fox",
    "title": "Spear of Justice"
  }
}
```

Fetch API and JSON

The Fetch API also has built-in support for json:

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

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

fetch('songs.json')
    .then(onResponse, onError)
    .then(onStreamProcessed);
```

Next week:
NodeJS Server