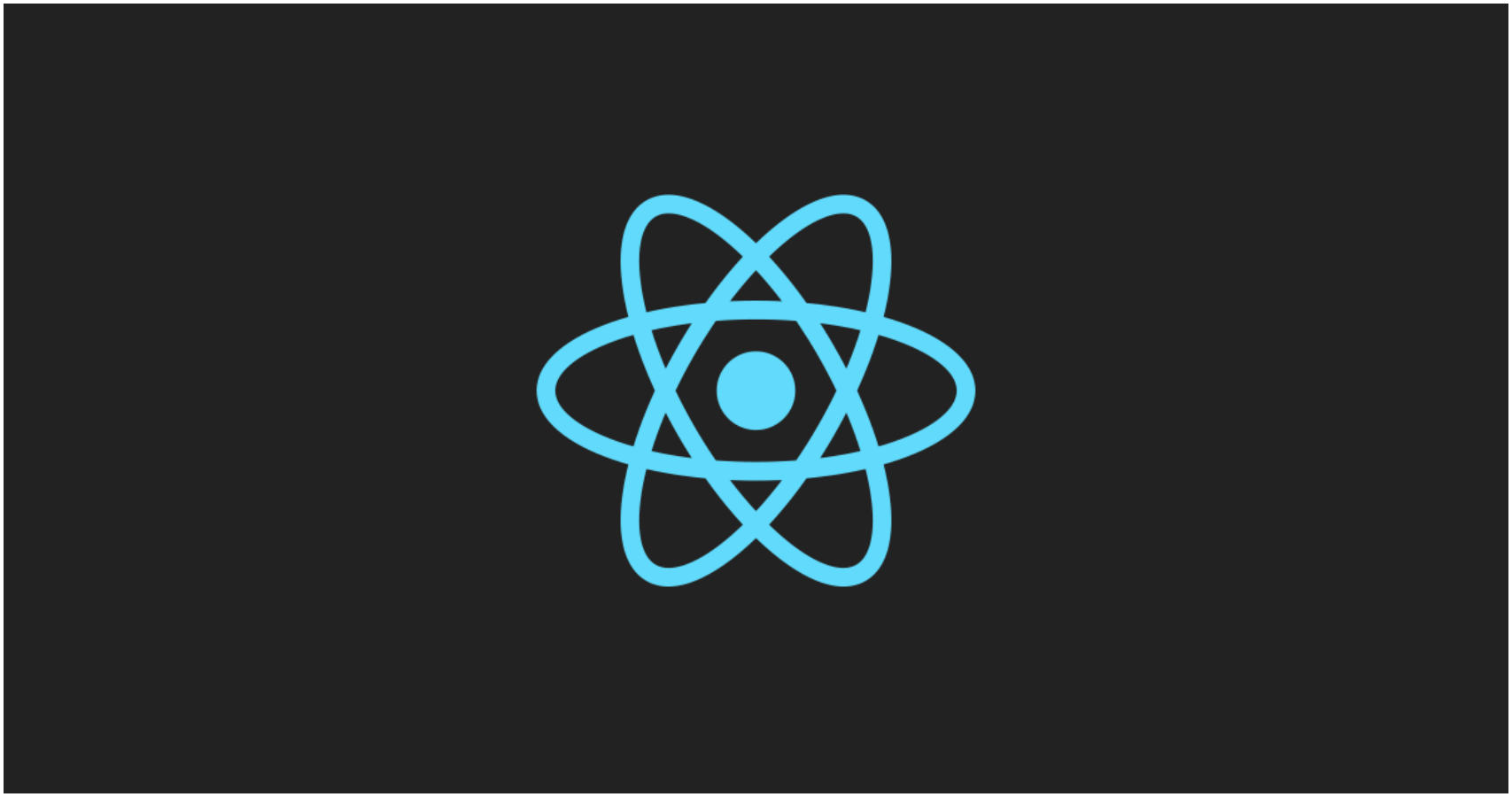
Client-side applications



Andrew DeOrio 2017

Agenda

- Review
- Client-side applications: JavaScript + REST APIs
  - Fetch API
- Closures
- Anonymous functions
- Frameworks and React

Review: client-side dynamic pages

- Client-side dynamic pages: JavaScript running in the client's web browser modifies the DOM. The rendered page changes.

1. Client executes JavaScript
2. JavaScript code modifies the DOM
3. Rendered page changes

Review: client-side dynamic pages

```
<html><body>
  <button onClick="hello()" type="button">
    Click Me!
  </button>
  <div id="JSEntry"></div>
  <script>
  function hello() {
    const n = document.getElementById('JSEntry');
    n.innerHTML = 'Hello World!';
  }
  </script>
</body></html>
```

Before click

After click

Review: event-driven programming

- In event-driven programming, the flow of the program is determined by *events*
  - Example of event built into the browser: `onclick`: user clicks a button
- Example: `onClick` is an event

```
<button onClick="hello()" type="button">
  Click Me!
</button>
```

- A main loop listens for events and triggers a *callback function*
- Example: `hello()` is a callback
  - A callback function is just a normal function, waiting to be executed

```
function hello() {
  n = document.getElementById('JSEntry');
  n.innerHTML = 'Hello World!;
}
```

Review: event queue and DOM

```
<html><body>
  <button onClick="hello()" type="button">
    Click Me!
  </button>
  <div id="JSEntry"></div>
  <script>
  function hello() {
    const n = document.getElementById('JSEntry');
    n.innerHTML = 'Hello World!';
  }
  </script>
</body></html>
```

Server-side vs. client-side dynamic pages

Server-side dynamic pages

- Server response is the output of a function
- Good for database access
  - Server function runs SQL query
- Bad for user interaction
  - Refresh required

Client-side dynamic pages

- JavaScript running in the client's web browser modifies the DOM
- Bad for database access
  - JS runs on client, not server
- Good for user interaction
  - Modify the DOM without refresh

Can we have the best of both worlds?

Client-side *with* server-side dynamic pages

- Goal: link client-side dynamic pages to server-side dynamic pages via a REST API
- JavaScript running in the client's web browser *makes a REST API request* and then modifies the DOM *using data from the response*
- Server response is the output of a function *which runs an SQL query and returns the result in JSON format*

- A client-side application usually uses both client-side dynamic pages and server-side dynamic pages
- Sometimes called "Full stack" development

Agenda

- Review
- **Client-side applications: JavaScript + REST APIs**
  - Fetch API
- Closures
- Anonymous functions
- Frameworks and React

Where we are going

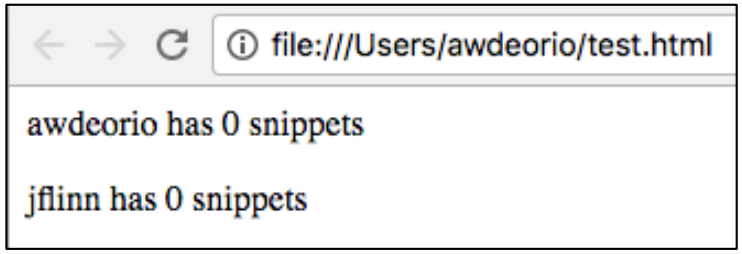
- Goal: modify the DOM using data from the REST API

1. Client requests root (index) page
   - Server responds with static HTML
2. Client loads HTML into DOM
3. Static HTML includes a `<script>` tag with file path to JavaScript
4. Client requests JavaScript source code
   - Server responds with static JavaScript file
5. Client executes JavaScript
6. JavaScript makes request to REST API
   - Server responds with JSON
7. JavaScript parses JSON into an object
8. JavaScript uses data in object to modify the DOM
9. Page updates

Where we are going

- Turn JSON data into a webpage that looks like this

```
<html>
  <head></head>
  <body>
    <div>
      <p>awdeorio has 0 snippets</p>
      <p>jflinn has 0 snippets</p>
    </div>
  </body>
</html>
```




```
{
  "snippets": [],
  "url": "/api/v1/users/100/",
  "username": "awdeorio"
},
{
  "snippets": [],
  "url": "/api/v1/users/200/",
  "username": "jflinn"
}
```

Separate HTML and JavaScript

- We'll separate the HTML from the JavaScript

```
<!-- index.html -->
<html>
  <head></head>
  <body>
    <div id="JSEntry"></div>
    <script src="/static/users.js"></script>
  </body>
</html>
```

```
//users.js
function showUsers() {
  // ...
}
showUsers();
```

Exercise

```
<!-- index.html -->
<html>
  <head></head>
  <body>
    <div id="JSEntry"></div>
    <script src="/static/users.js"></script>
  </body>
</html>
```

```
//users.js
function showUsers() {
  const entry = document.getElementById('JSEntry');

  // HINT: create paragraph node:
  // n = document.createElement('p');

  // HINT: create text node:
  // t = document.createTextNode('hello');

  // HINT: connect nodes:
  // n.appendChild(t);
}
showUsers();
```

When `showUsers()` is done, the DOM will look like this HTML ->

```
<!-- Final DOM looks like this -->
<html>
  <head></head>
  <body>
    <div id="JSEntry">
      <p>awdeorio has 0 snippets</p>
      <p>jflinn has 0 snippets</p>
    </div>
  </body>
</html>
```

Solution

```html
<!-- index.html -->
<html>
  <head></head>
  <body>
    <div id="JSEntry"></div>
    <script src="/static/users.js"></script>
  </body>
</html>
```

```javascript
//users.js
function showUsers() {
  const entry = document.getElementById('JSEntry');

  const n1 = document.createElement('p');
  const t1 = document.createTextNode(
    'awdeorio has 0 snippets');
  n1.appendChild(t1);
  entry.appendChild(n1);

  const n2 = document.createElement('p');
  const t2 = document.createTextNode(
    'jflinn has 0 snippets');
  n2.appendChild(t2);
  entry.appendChild(n2);
}
showUsers();
```

Adding nodes to the DOM

- DOM now looks just like static HTML mock-up

```
Elements   Console   Sources   Netwo
<html>
  <head></head>
...▼<body> == $0
  ▼<div>
      <p>awdeorio has 0 snippets</p>
      <p>jflinn has 0 snippets</p>
    </div>
    <script src="/static/users.js"></script>
  </body>
</html>
```

- Rendered page looks just like static HTML mock-up

file:///Users/awdeorio/test.html

awdeorio has 0 snippets

jflinn has 0 snippets

Mock up

- Next, create mock-up data
- Same data that is returned by the REST API

```
function showUsers() {
  const entry = document.getElementById('JSEntry');
  const users = [
    {
      "snippets": [],
      "url": "/api/v1/users/100/",
      "username": "awdeorio"
    },
    {
      "snippets": [],
      "url": "/api/v1/users/200/",
      "username": "jflinn"
    }
  ]
// ...
```

Mock up

- Read data structure and insert elements into the DOM

```
function showUsers() {
  const entry = document.getElementById('JSEntry');

  const users = /* ... */ ;

  users.forEach((user) => {
    const n = document.createElement('p');
    const s = `${user.username} has ${user.snippets.length} snippets`;
    const t = document.createTextNode(s);
    n.appendChild(t);
    entry.appendChild(n);
  });
}
```

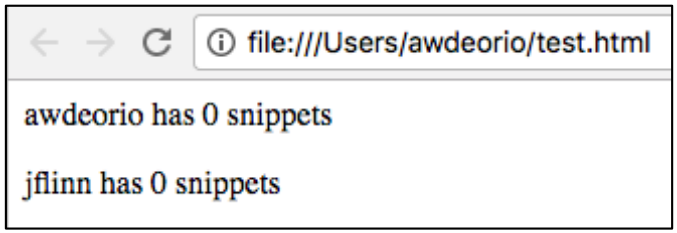
```
function showUsers() {
  const entry = document.getElementById('JSEntry');

  const users = [
    {
      "snippets": [],
      "url": "/api/v1/users/100/",
      "username": "awdeorio"
    },
    {
      "snippets": [],
      "url": "/api/v1/users/200/",
      "username": "jflinn"
    }
  ];

  users.forEach((user) => {
    const n = document.createElement('p');
    const s = `${user.username} has ${user.snippets.length} snippets`;
    const t = document.createTextNode(s);
    n.appendChild(t);
    entry.appendChild(n);
  });
}
```

Adding nodes to the DOM

- Again, page now looks just like our static HTML mock-up



- Next: fetch the data from a REST API

Agenda

- Review
- Client-side applications: JavaScript + REST APIs
  - **Fetch API**
- Closures
- Anonymous functions
- Frameworks and React

Fetching data from the REST API

- The Fetch API provides an interface for HTTP requests
- Call a function when the response arrives
  - Parse JSON into JavaScript object
- Call another function when JSON parsing is finished
  - Add DOM nodes using JavaScript object
- Why call functions? *Asynchronous programming*, which we'll cover in detail next time.

```
function showUsers() {
  const entry = document.getElementById('JSEntry');
  //...
  fetch("/api/v1/users/")
    .then(/* handle response and parse JSON */)
    .then(/* handle data and add DOM nodes */)
}
```

Server's perspective

```
<!-- index.html -->
<html>
  <head></head>
  <body>
    <div id="JSEntry"></div>
    <script src="/static/users.js">
    </script>
  </body>
</html>
```

- Notice requests for JS source code and REST API in server logs

```
$ python3 api.py
 * Running on /
127.0.0.1 - - [28/Sep/2017 18:28:30]
  "GET / HTTP/1.1" 200 -
127.0.0.1 - - [28/Sep/2017 18:28:30]
  "GET /static/users.js HTTP/1.1" 200 -
127.0.0.1 - - [28/Sep/2017 18:28:30]
  "GET /api/v1/users/ HTTP/1.1" 200 -
```

```
function showUsers() {
  const entry = document.getElementById('JSEntry');
  //...
  fetch("/api/v1/users/")
    .then(/* handle response and parse JSON */)
    .then(/* handle data and add DOM nodes */)
```

Fetching data from the REST API

- Function to parse JSON from HTTP response
- This function is called later when the response arrives

```
function showUsers() {
  const entry = document.getElementById('JSEntry');

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

  fetch("/api/v1/users/")
    .then(handleResponse)
    .then(/* handle data and add DOM nodes */)
}
```

Inner functions

- Notice that JavaScript allows inner functions
- Functions are first class objects in JavaScript
  - You can create them at run time

```
function showUsers() {
  const entry = document.getElementById('JSEntry');

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

  fetch("/api/v1/users/")
    .then(handleResponse)
    .then(/* handle data and add DOM nodes */)
}
```

Fetch and error handling

- `handleResponse` might throw an exception
- Add error handling

```
function showUsers() {
  const entry = document.getElementById('JSEntry');
  //...

  function handleResponse(response) {
    if (!response.ok) throw Error(response.statusText);
    return response.json();
  }

  function handleError(error) {
    console.log(error)
  }

  fetch("/api/v1/users/")
    .then(handleResponse)
    .then(/* handle data and add DOM nodes */)
    .catch(handleError);
}
```

Handle the data

- Add a function to process the data parsed from the JSON response

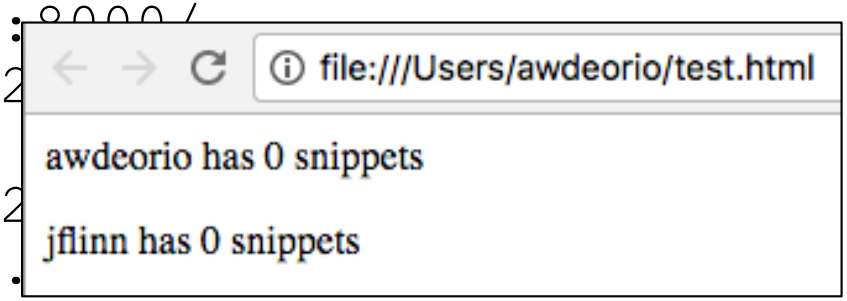
```
function showUsers() {
  //...
  function handleData(data) {
    const users = data.results;
    users.forEach((user) => {
      const node = document.createElement('p');
      const text = `${user.username} has ${user.snippets.length} snippets`;
      const textnode = document.createTextNode(text);
      node.appendChild(textnode);
      entry.appendChild(node);
    });
  }
  fetch("/api/v1/users/")
    .then(handleResponse)
    .then(handleData)
    .catch(handleError);
}
```

Front end and back end work together

```
<!-- index.html -->
<html>
  <head></head>
  <body>
    <div id="JSEntry"></div>
    <script src="/static/users.js">
    </script>
  </body>
</html>
```

- Notice requests for JS source code and REST API in server logs

```
$ python3 api.py
 * Running on http://localhost:8000/
127.0.0.1 - - [28/Sep/2017 18:2
  "GET / HTTP/1.1" 200 -
127.0.0.1 - - [28/Sep/2017 18:2
  "GET /static/users.js HTTP/1.
127.0.0.1 - - [28/Sep/2017 18:28:30]
  "GET /api/v1/users/ HTTP/1.1" 200 -
```




```
function showUsers() {
  const entry = document.getElementById('JSEntry');
  //...
  fetch("/api/v1/users/")
    .then(handleResponse)
    .then(handleData)
    .catch(handleError);
}
```

DOM nodes created

- No HTML for our paragraphs

view-source:localhost:8000

```
1 <html>
2   <head></head>
3   <body>
4     <div id="JSEntry"></div>
5     <script src="/static/users.js">
6     </script>
7   </body>
8 </html>
9
```

- But you can see them in the DOM

Elements Console Sources Netwo

```
<html>
  <head></head>
...▼<body> == $0
  ▼<div>
      <p>awdeorio has 0 snippets</p>
      <p>jflinn has 0 snippets</p>
    </div>
    <script src="/static/users.js"></script>
  </body>
</html>
```

Agenda

- Review
- Client-side applications: JavaScript + REST APIs
  - Fetch API
- **Closures**
- Anonymous functions
- Frameworks and React

A "closure" look

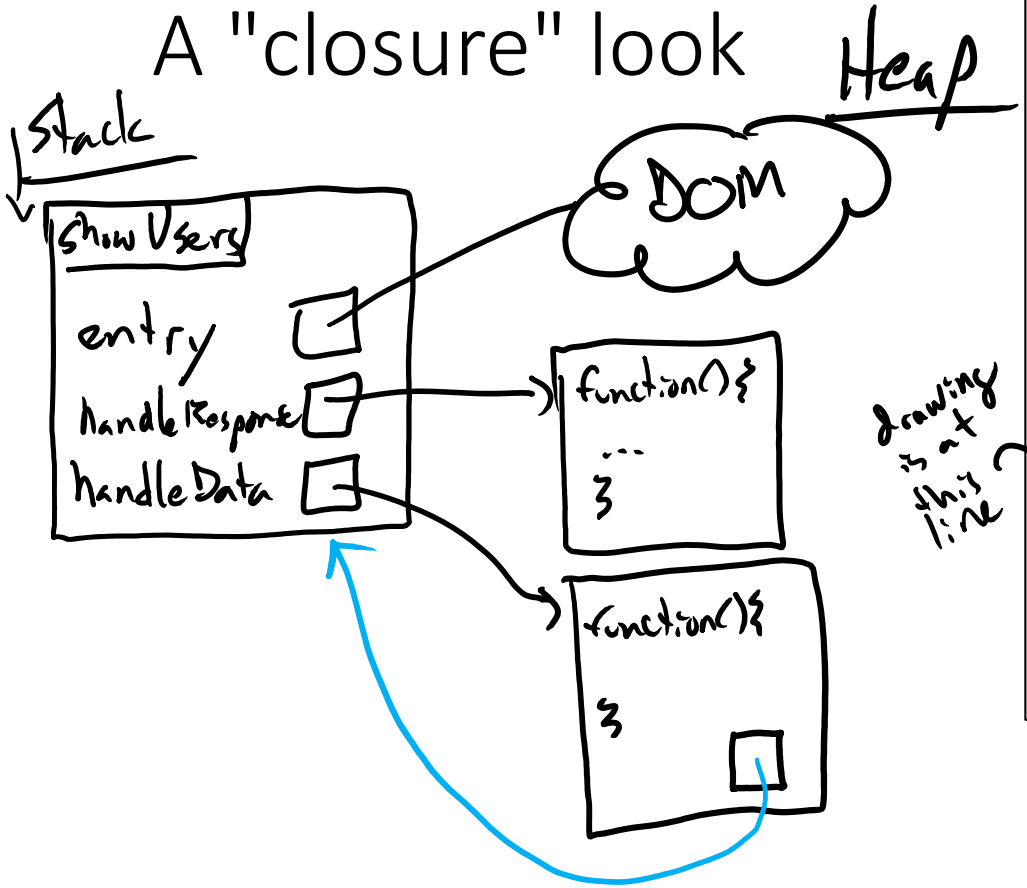
```
function showUsers() {
  const entry =
    document.getElementById(...);

  function handleResponse(response)
  {/*...*/}

  function handleData(data) {
    // ...
    entry.appendChild(node);
  }

  fetch(/*...*/)
    .then(handleResponse)
    .then(handleData)
}
```

A "closure" look




```
function showUsers() {
  const entry = const entry =
    document.getElementById(...);

  function handleResponse(response)
  {/*...*/}

  function handleData(data) {
    // ...
    entry.appendChild(node);
  }
  fetch(/*...*/)
    .then(handleResponse)
    .then(handleData)
}
```

drawing is at this line

inner function accesses outer variable. Closure!

Event Table

(empty)

Queue

(nothing yet)

Closures

- Notice that the inner function has access to outer function's variables
- Lexically scoped name binding
- This is called a closure

```
function showUsers() {
  const entry = document.getElementById('JSEntry');
  //...
  function handleData(data) {
    //...
    entry.appendChild(node);
  }
  fetch(/* ... */) // ...
}
```

Closures

- The inner function has a longer lifetime than the outer function
- `handleData()` has access to `entry` even though `showUsers()` has already returned!

```
function showUsers() {
  const entry = document.getElementById('JSEntry');

  function handleResponse(response) { /*... */ }

  function handleData(data) {
    // ...
    entry.appendChild(node);
  }

  fetch(/*...*/)
    .then(handleResponse)
    .then(handleData)
    .catch(error => console.log(error));
}
```

Closures

- The `entry` variable used by the inner function `handleData()` is not a copy, it is a reference to the original object created by the outer function `showUsers()`
- This is possible because the inner function `handleData()` has access to the context in which it was created, the scope of the outer function `showUsers()`
  - *This is a closure*

Closures and classes

- A *class* is data with functions attached
- A *closure* is a function with data attached

Closures in the interpreter

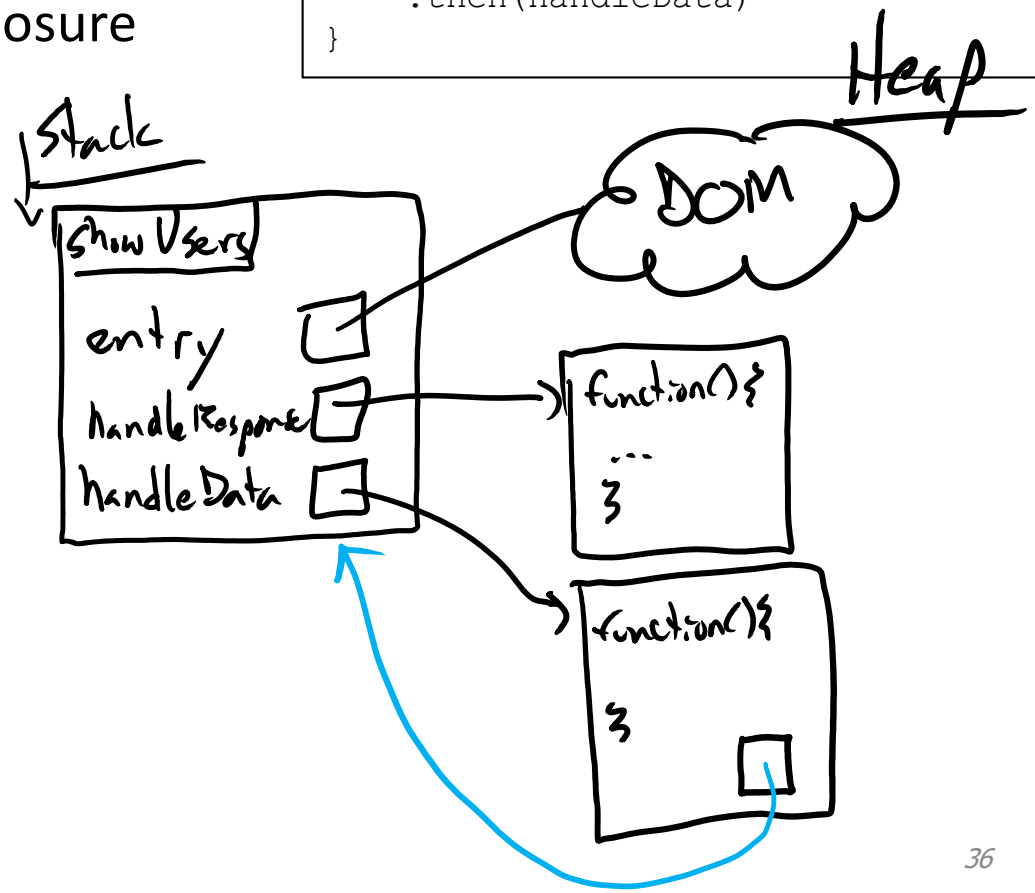
- Activation records live on the heap
- *Act* like a stack
- Stay when referenced by a closure

```
function showUsers() {
  const entry = /*...*/;

  function handleResponse(response)
  { /*...*/ }

  function handleData(data) {
    // ...
    entry.appendChild(node);
  }

  fetch(/*...*/)
    .then(handleResponse)
    .then(handleData)
}
```



Closures in the interpreter

```
function showUsers() {
  const entry = /*...*/;

  function handleResponse(response)
  { /*...*/ }

  function handleData(data) {
    // ...
    entry.appendChild(node);
  }

  fetch(/*...*/)
    .then(handleResponse)
    .then(handleData)
}
```

1. Objects created for `entry`, `handleResponse`, `handleData`
2. `fetch` function executes
   1. Enqueue callbacks `handleResponse`, `handleData`
   2. `fetch` returns before response arrives
3. Stack pointer points to null
4. .... wait for response
5. Later, response arrives and `handleResponse` executes
6. Later, JSON data is ready and `handleData` executes

Why closures are important in web dev

- Callback Functions are everywhere
- These functions need to remember their context
  - Remember that execution keeps going past a `fetch()` call

Agenda

- Review
- Client-side applications: JavaScript + REST APIs
  - Fetch API
- Closures
- **Anonymous functions**
- Frameworks and React

Anonymous functions

- These callback functions are used only once

```
function showUsers() {
  const entry = document.getElementById('JSEntry');

  function handleResponse(response) {/*... */}

  function handleData(data) {/*...*/}

  fetch('/api/v1/users/')
    .then(handleResponse)
    .then(handleData);
}
```

Anonymous functions

- Refactor to use *anonymous functions*

```
function showUsers() {
  const entry = document.getElementById('JSEntry');

  function handleResponse(response) {/*... */}

  function handleData(data) {/*...*/}

  fetch('/api/v1/users/')
    .then(function(response) {
      //...
    })
    .then(function(data) {
      //...
    })
}
```

Anonymous functions

- Same as when the functions had names
- Also called a lambda function or function literal

```
function showUsers() {
  const entry = document.getElementById('JSEntry');

  fetch('/api/v1/users/')
    .then(function(response) {
      //...
    })
    .then(function(data) {
      //...
    })
}
```

Anonymous functions in ES6

- ES6 provides a convenient syntax for anonymous functions
- "Arrow functions"

```
function showUsers() {
  const entry = document.getElementById('JSEntry');

  fetch('/api/v1/users/')
    .then((response) => {
      //...
    })
    .then((data) => {
      //...
    })
}
```

Anonymous functions in ES6

- Anatomy of an anonymous function
  - Inputs
  - Body
  - Arrow
- Creates a function object on the heap
  - Just like "regular" functions
- Long format

```
(INPUTS) => {
  // BODY
}
```

- Short cut for body with one function call

```
INPUT => my_function(INPUT)
```

Anonymous functions in ES6

- What does this do?

```
$ node
> let f = (x) => { return x + 1; }
[Function: f]
> f(1)
```

Anonymous functions in ES6

- What does this do?
- Another way to create a "normal" function!

```
$ node
> let f = (x) => { return x + 1; }
[Function: f]
> f(1)
2
```

Agenda

- Review
- Client-side applications: JavaScript + REST APIs
  - Fetch API
- Closures
- Anonymous functions
- **Frameworks and React**

A problem with raw JavaScript

- Large JavaScript applications quickly become unwieldly
- All functions act on the DOM, DOM acts like a giant global variable
- Difficult to decompose program into abstractions

```
function showUser() {
  fetch()
  .then(function(data) {
      const users = data.results;
      users.forEach((user) => {
        const node = document.createElement('p');
        //...
        node.appendChild(textnode);
        entry.appendChild(node);
      });
    })
//...
```

jQuery

- jQuery library helps
  - Convenience functions for DOM manipulation
- Same fundamental problem: DOM acts like a giant global variable

```
function showUser() {
  fetch()
  .then(function(data) {
      const users = data.results;
      users.forEach((user) => {
        let node = $("<p></p>").text("...");
        $("JSentry").append(node);
      });
    })
//...
```

A problem with jQuery

- Operations on the DOM are very slow compared to other operations
- The performance of large JavaScript applications suffers

```
function showUser() {
  fetch()
  .then(function(data) {
      const users = data.results;
      users.forEach((user) => {
        let node = $("<p></p>").text("...");
        $("JSentry").append(node);
      });
    })
//...
```

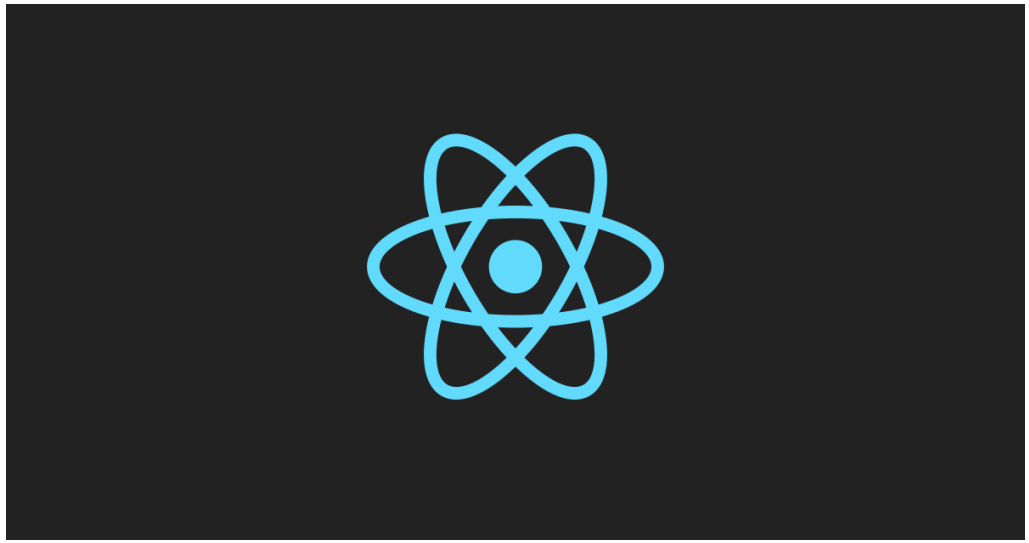
Enter frameworks

- JavaScript frameworks offer a tool for abstraction
  - Manage complexity, hide details
  - Encapsulation and information hiding
  - Performance enhancements implemented under the hood
- Difference between a library and a framework
  - You call a library
  - A framework calls you
- Two large, popular frameworks
  - React (created by Facebook)
  - Angular (created by Google)
  - (Yes, others exist too)

React

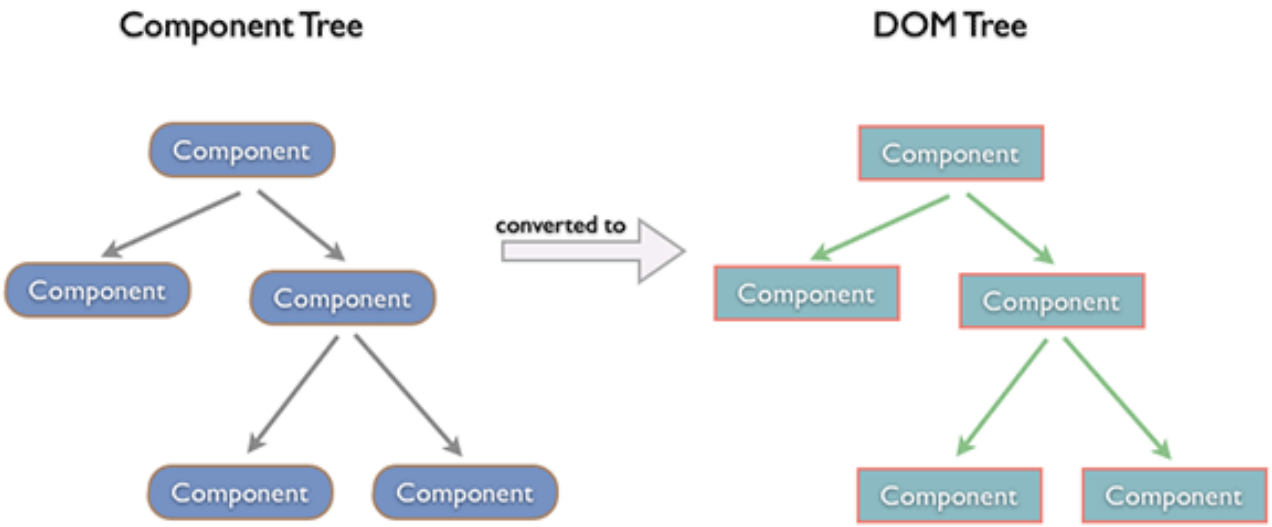
- React is a framework built by Facebook
- Build encapsulated components that manage their own state
  - Compose them to make complex UIs
- Efficient updates to the DOM

- https://reactjs.org/



Refactor example into React

- Refactor our raw JavaScript to use React
- Components
  - Functional: usually stateless
  - Class-type: usually stateful
- Tree of composable components -> DOM



Virtual DOM

- Components rendering cause other components to render
- This would cause lots of DOM updates, which are slow
  - Because the actual screen changes
- Solution: a Virtual DOM

- Periodically reconcile virtual DOM with real DOM
  - Avoids unnecessary changes

- For lots of details: https://reactjs.org/docs/reconciliation.html

Recap

- Goal: modify the DOM using data from the REST API

1. Client requests root (index) page
   - Server responds with static HTML
2. Client loads HTML into DOM
3. Static HTML includes a <script> tag with file path to JavaScript
4. Client requests JavaScript source code
   - Server responds with static JavaScript file
   - *Now it uses the React framework*
5. Client executes JavaScript
6. JavaScript makes request to REST API
   - Server responds with JSON
7. JavaScript parses JSON into an object
8. JavaScript uses data in object to modify the DOM
   - *DOM manipulation provided by React framework*
9. Page updates

React documentation

- Required reading for project 3: https://reactjs.org/docs/hello-world.html

- Live example https://codepen.io/awdeorio/pen/yzXjzZ?editors=1010