# React **Fun**damentals

In addition to some full-stack development!

# $whoami

> Jarrod Servilla
> CSSC Tech Director

> Daniel Laufer
> GDSC Workshop lead

> Milind Vishnoi
> GDSC Workshop lead

# What you'll learn

- What is React?
- What is JSX?
- Creating reusable react components
- Dynamic rendering
- Component lifecycle
- React hooks & why we use them
- Full stack development fundamentals
- Networking protocol basics (http)

and most importantly...

# you'll create your own full stack app!

*(kind of)*

# important resources

source code: https://github.com/utm-cssc/full-stack-react-workshop

gdsc workshops: https://github.com/Daniel-Laufer/GDSC-UTM-Workshops

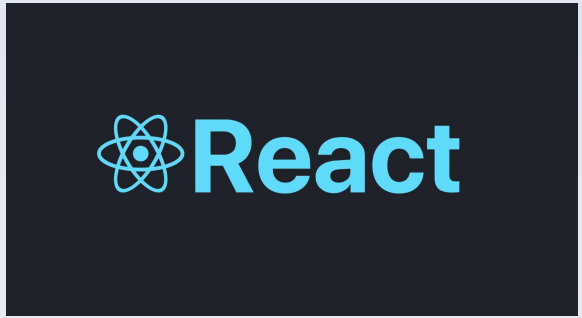
cssc site: https://cssc.utm.utoronto.ca/

if you're coding along with us:

- an ide (vscode)
- node.js
- docker

# What is React?

- a declarative, component-based **front-end javascript library** developed by Facebook
- enables developers to build modern, sleek, fast applications with a modular infrastructure
- react is by far the most popular front-end javascript library/framework!

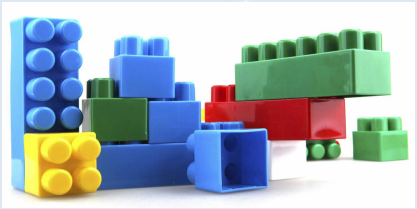



# Who uses React?




And soooooooo many more!

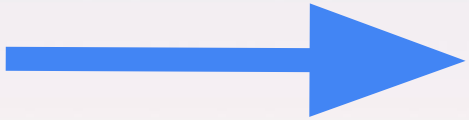
# Why should you use react?



- can organize parts of the UI into ***components*** that can be easily reused
- users can interact with the website without refreshing it
  - *Ex. dynamically rendering search results from a search query*
- you want to make use of its **rich ecosystem and large community** support
  - *if you search “how do I do X with react”, odds are there will be many relevant search results*
  - *there are tons of react libraries for you to use. For example: react-spring allows you to add sophisticated, good-looking animations into your apps*
  - *users can share their own components with the community by creating packages*

# What are React Apps made of?

- Primary **Javascript** and **JSX** (a '*syntax extension*' for **Javascript**).
    - Note that you can use plain Javascript to write React code but it's much more difficult/messy
- JSX **looks a lot like** standard **HTML**

**Let's take a look at an example!** 

# Say we want to create this beautiful component ------ >


this is the **Javascript** and **JSX** code needed to create this component

```jsx
import pog from "./pog.png";
import "./App.css";

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <h1>heyo</h1>
        <img src={pog} className="pog" alt="pog" />
      </header>
    </div>
  );
}

export default App;
```

the **raw html** generated by this code. Looks extremely similar right?

```
<noscript>You need to enable JavaScript to run this app.</noscript>
▼<div id="root">
  ▼<div class="App">
    ▼<header class="App-header">  flex  == $0
      <h1>heyo</h1>
      <img src="/static/media/pog.456a7ac2.png" class="pog" alt="pog">
    </header>
  </div>
</div>
<!--
```

```
import pog from "./pog.png";
import "./App.css";

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <h1>heyo</h1>
        <img src={pog} className="pog" alt="pog" />
      </header>
    </div>
  );
}

export default App;
```

# Components

A React **component** is a JavaScript **function** that optionally takes in inputs (props) and **returns <u>ONE</u> JSX element** *(this one element can have many children)*.

**our simple TodoItem component**

```jsx
function TodoItem(props) {
    return (
        <div>
            <h1>Message: {props.message}</h1>
            <h1>Colour: {props.colour}</h1>
        </div>
    )
};

export default TodoItem;
```

**Using our component and <u>passing data (props) into it</u>**

```jsx
function TodoList() {
    const todo = {
        message: "create kickass react workshop",
        colour: "blue",
        isChecked: false
    }
    return (
        <div>
            <TodoItem message={todo.message} colour={todo.colour} />
        </div>
    )
}
```

** you can also create components using classes but we won't discuss that in this workshop :)*

# Using components

```
function Navbar() {
  return (
    <div>
      <div>
        <h1>Dialog</h1>
      </div>
      <div>
        <button>
          Sign Up
        </button>
        <button>
          Login
        </button>
      </div>
    </div>
  );
};
```

components can be rendered in two ways: with and without children.

```
return (
    <Navbar />

    <PageWrapper>
        <Navbar />
    </PageWrapper>
);
```

*Here **Navbar is a child** of **PageWrapper***

# Using components

- You can continue nesting components as much you'd like!
- For example...

```
<PageWrapper>
    <Navbar />
    <div>
        <PageWrapper>
            <Navbar />
            <Navbar />
            <Navbar />
        </PageWrapper>
    </div>
</PageWrapper>
```

# JavaScript inside JSX components

you may have seen us wrap some portions of code in curly braces i.e {...}. Why is that?

```
<h1>
  {
    messages.reduce((prev, curr) => prev.concat(curr), "")
  }
</h1>
```

- here everything outside the '{..}' is JSX, and everything inside is javascript.
- if we didn't have the curly branches there, our javascript code would be interpreted as a string and NOT code (ie "messages.reduce((prev, curr) ⇒ prev.concat(curr), "")")

# Before we get into coding, let’s take a look at some interesting JavaScript syntax you will see Jarrod use

# two ways of writing functions in Javascript

```
function someFunc(num1, num2) {
  if (num1 > num2)
    console.log(num1);

  return num1 + num2;
}
```

```
const someFunc = (num1, num2) => {
  if (num1 > num2)
    console.log(num1);

  return num1 + num2;
};
```

think of these as being equivalent function definitions for this workshop. (There are some more technical differences between these two functions but don't worry about them for now 😉)

# using components: destructuring objects

```
<Card
  url="http://github.com/white-van/discussion-board"
  title="Contribute"
  msg="View the project directory here"
/>
```

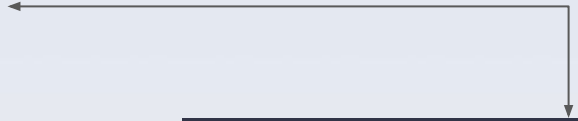
here is a component that takes in multiple props.

```
export const Card = ({url, title, msg}) => {
  return (
    <a href={url}>
      <h2>{title}</h2>
      <p>{msg}</p>
    </a>
  );
};
```

the { ... } is called **object destructuring**, which pulls out the values from the props object and exposes it to us as url, title, and msg respectively.

# destructuring objects: continued

```
export const Card = ({url, title, msg}) => {
  return (
    <a href={url}>
      <h2>{title}</h2>
      <p>{msg}</p>
    </a>
  );
};
```



```
export const Card = (props) => {
  let url = props.url;
  let title = props.title;
  let msg = props.msg;
  return (
    <a href={url}>
      <h2>{title}</h2>
      <p>{msg}</p>
    </a>
  );
};
```

- just think of javascript doing this behind the scenes automatically for you.
- **a lot less coding for us**!

# Time to Code!

# Dynamic rendering

how do we render based on input?

# Dynamic rendering

Suppose you wanted to create a component like this that contains an arbitrary amount of children

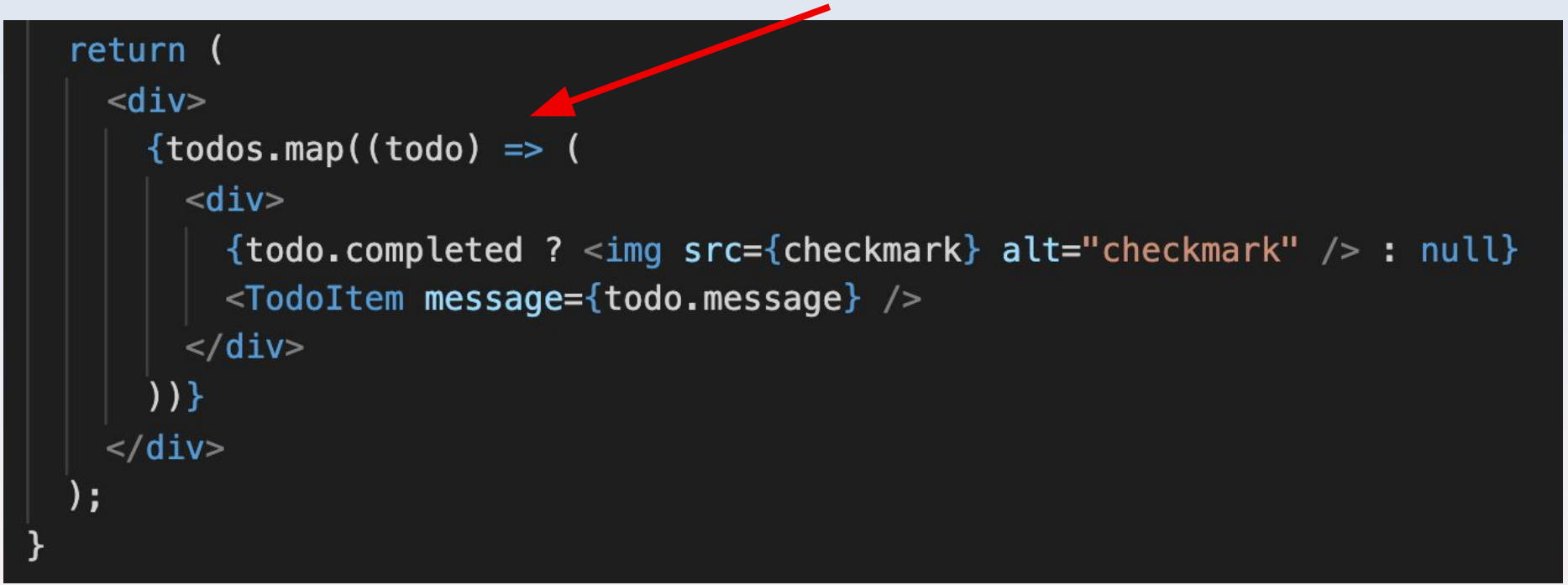
```
return (
  <div>
    <div>
      {todos[0].completed ? <img src={checkmark} alt="checkmark" /> : null}
      <TodoItem message={todos[0].message} />
    </div>
    <div>
      {todos[1].completed ? <img src={checkmark} alt="checkmark" /> : null}
      <TodoItem message={todos[1].message} />
    </div>
    <div>
      {todos[2].completed ? <img src={checkmark} alt="checkmark" /> : null}
      <TodoItem message={todos[2].message} />
    </div>
    <div>
      {todos[3].completed ? <img src={checkmark} alt="checkmark" /> : null}
      <TodoItem message={todos[3].message} />
    </div>
  </div>
);
```

one of the strengths of react is that we can use javascript to render React elements dynamically!

# instead of doing this....

```
return (
  <div>
    <div>
      {todos[0].completed ? <img src={checkmark} alt="checkmark" /> : null}
      <TodoItem message={todos[0].message} />
    </div>
    <div>
      {todos[1].completed ? <img src={checkmark} alt="checkmark" /> : null}
      <TodoItem message={todos[1].message} />
    </div>
    <div>
      {todos[2].completed ? <img src={checkmark} alt="checkmark" /> : null}
      <TodoItem message={todos[2].message} />
    </div>
    <div>
      {todos[3].completed ? <img src={checkmark} alt="checkmark" /> : null}
      <TodoItem message={todos[3].message} />
    </div>
  </div>
);
```

# … do this!



```
  return (
    <div>
      {todos.map((todo) => (
        <div>
          {todo.completed ? <img src={checkmark} alt="checkmark" /> : null}
          <TodoItem message={todo.message} />
        </div>
      ))}
    </div>
  );
}
```

It’s basically just a fancy for loop that generates a list of react elements!

# Time to Code!

# lifecycle + hooks

updating the view after modifications occur

# component lifecycle

🌱 mounting: component initialization, and addition to dom

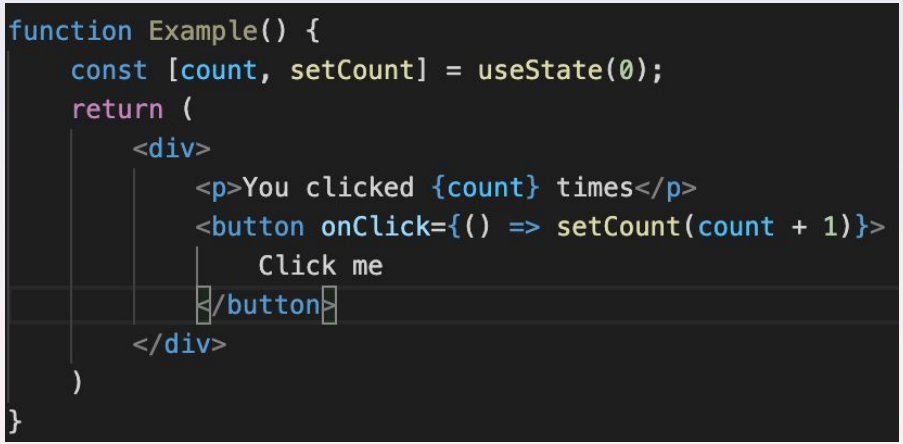
🔄 updating: when props/state of a component changes, rerender!

⚰️ unmount: cleanup component resources, remove from dom

# hooks

Hooks was introduced in React 16.8

Hooks let you use state and other React features without writing a class.



```
function Example() {
    const [count, setCount] = useState(0);
    return (
        <div>
            <p>You clicked {count} times</p>
            <button onClick={() => setCount(count + 1)}>
                Click me
            </button>
        </div>
    )
}
```

# why hooks?

- no one knows how “this” works.
- organizing our components by lifecycle methods forces us to sprinkle related logic throughout our components.
- Makes testing easier

```
class ReposGrid extends React.Component {
  state = {
    repos: [],
    loading: true
  }
  componentDidMount () {
    this.updateRepos(this.props.id)
  }
  componentDidUpdate (prevProps) {
    if (prevProps.id !== this.props.id) {
      this.updateRepos(this.props.id)
    }
  }
  updateRepos = (id) => {
    this.setState({ loading: true })

    fetchRepos(id)
      .then((repos) => this.setState({
        repos,
        loading: false
      }))
  }
```

# useState

this react hook allows us to **persist data across re-renders** (and forces a re-render when our state changes) in that state!

In the example function you can create a count state for the component using useState as shown.

You can use count to access count within the component.

```
function Example() {
    const [count, setCount] = useState(0);
    return (
        <div>
            <p>You clicked {count} times</p>
            <button onClick={() => setCount(count + 1)}>
                Click me
            </button>
        </div>
    )
}
```

# useEffect

this react hook allows us to **run code** (fetching data, updating state) when changes occur in the states specified.

```
// Similar to componentDidMount and componentDidUpdate:
useEffect(() => {
  // Update the document title using the browser API
  document.title = `You clicked ${count} times`;
});
```

```
useEffect(() => {
  document.title = `You clicked ${count} times`;
}, [count]); // Only re-run the effect if count changes
```

# useEffect as componentDidMount

We can use ‘useEffect’ to implement ‘componentDidMount’  function.

```
const [loading, setloading] = useState(true);

useEffect(() => {
    setLoading(false);
}, []); // [] makes the code take effect only once
```

# useEffect as componentWillUnmount

We can use ‘useEffect’ to implement ‘componentWillUnmount’  function.

```
useEffect(() => {
    // When we return a function in useEffect you will
    // be able to use useEffect as ‘componentWillUnmount’
    // function.
    return () => {
        console.log('Component is being unmounted');
    }
}, []);
```

# create custom hooks

When we need to use function logic in more than one component we can extract that logic into another function (hook).

A custom hook is a JavaScript function whose name starts with ”use” and that may call other hooks.

**for example:** using a custom hook to fetch data for different URL.

# creating custom hooks

## Custom hook

```
import { useState, useEffect } from "react";

const useFetch = (url) => {
  const [data, setData] = useState(null);

  useEffect(() => {
    fetch(url)
      .then((res) => res.json())
      .then((data) => setData(data));
  }, [url]);

  return [data];
};

export default useFetch;
```

## Using the custom hook

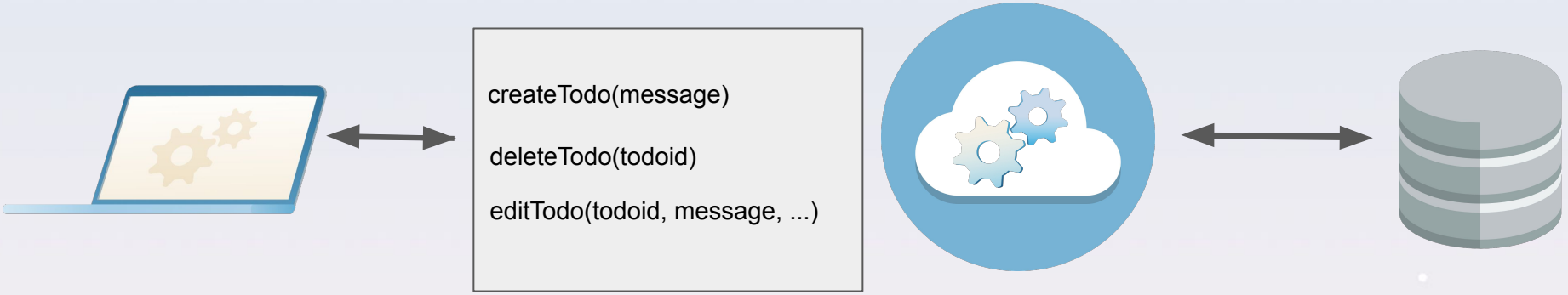
```
const Home = () => {
  const [data] = useFetch("https://apicall.com");
```

# Time to Code!

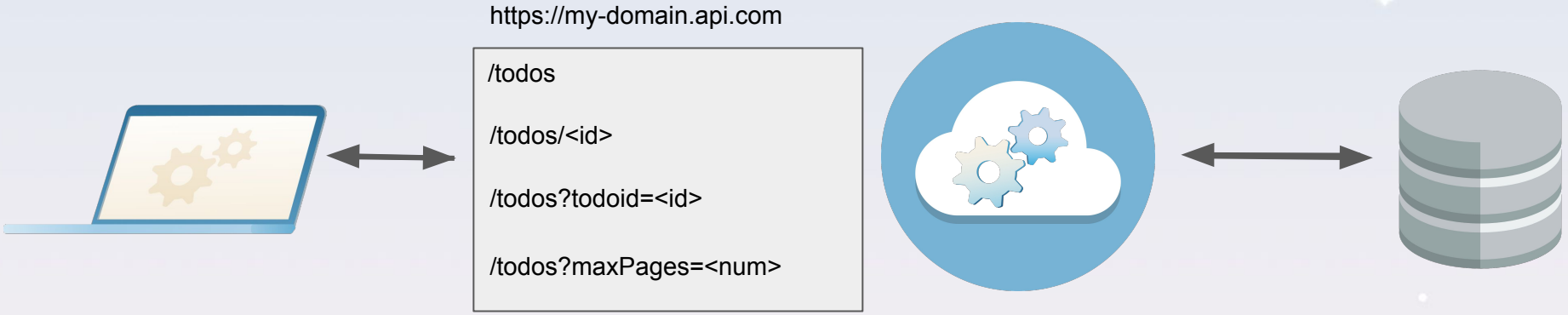
# full stack apps

how do we persist data?


Client
Server
Database

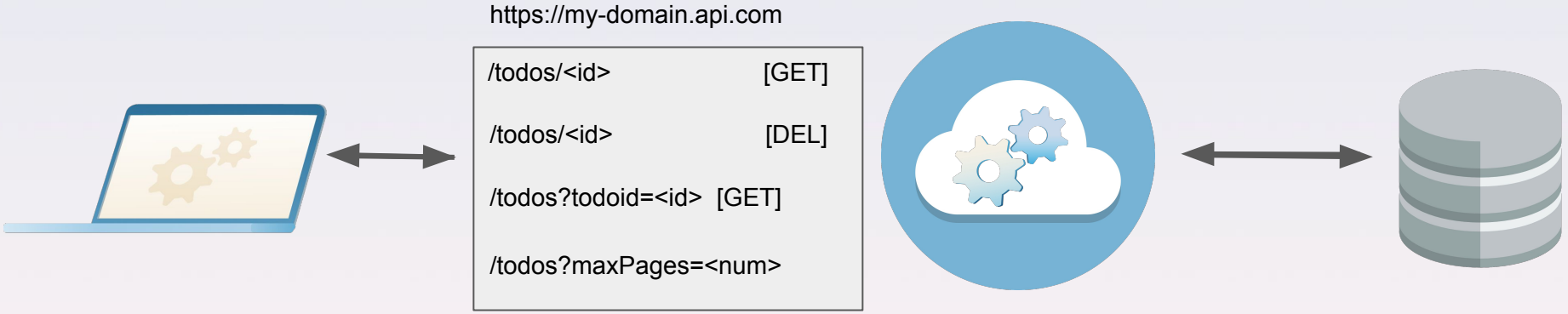



API: Application Programming Interface




URL: Universal Resource Locator

| | |
|---|---|
| **GET** | retrieve information |
| **HEAD** | retrieve resource headers |
| **POST** | submit data to the server. |
| **PUT** | save an object at the location |
| **DELETE** | delete the object at the location |




HTTP: Hypertext Transfer Protocol

# http statuses

after issuing an http request, we expect to receive a status code and response body (typically JSON). http statuses describe what the server did in response to the request.

**200 OK:** The response has succeeded!

**201 Created:** The request has succeeded, and the resource has been created (usually for POST)

**400 Bad Request:** The server could not understand the request due to invalid syntax

**401 Unauthorized:** The client is not allowed to get the requested response

**404 Not Found:** The server cannot find the requested resource

**418 I'm a teapot:** The server refuses to brew coffee because it is, permanently, a teapot.

**500 Internal Server Error:** The server has encountered an issue while processing your request

# api integration

connecting our frontend to our backend

# Time to Code!

# Thank you!

## Any questions?