ColdFusion + ReactJS

Expanding Horizons through Interactivity Programming

"I'm a second-time tour guide!" - Minh Vo

Adobe ColdFusion Summit East 2019,
12:10PM
Washington DC

Preparing for the Trip

~ Overview, Build, Peruse ~

Don’t blink, you’re going to see unicorns…

Why are we exploring? Why expand horizons?

“Getting to the moon is not an extension of climbing a tree.”

Solutions to problems you want to solve may not exist in the realm of knowledge you currently possess.

Technology ~ ground beneath us is ever-changing and cannibalizing!

What we think is right and true today, might not be tomorrow.
What is impossible today, might not be impossible tomorrow.

Exploring will keep us more aware so we can fail quickly and remain flexible.

What did we sign up for?

We're going to explore a light-hearted example of Interactivity Programming

GOAL: Prove to you the ColdFusion + ReactJS combination is worth your time!

http://draftstudios.com/demo/

What is ReactJS?

JavaScript library in charge of front-end/client-side views.

- Sits where Flex used to sit.
- Does not handle back-end logic or databases = ColdFusion to the rescue.
- Composable - JSX (custom syntax) is tag-based (similar to CF Custom Tags)
  - gets “transpiled” back to vanilla JavaScript

```
<ComponentName prop1={value} prop2={function()} onWheel={wheelHandler()}
onClick={clickHandler()}>
	<Child … /> {/* reference these children as this.props.children */}
</ComponentName>
```

Why is ReactJS being adopted?

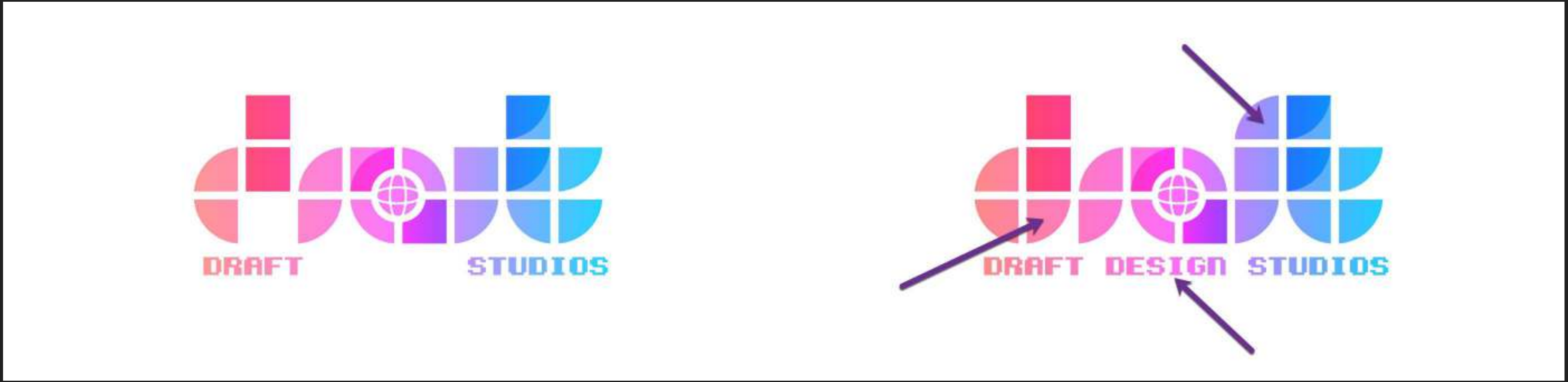
1. Fundamental change in the way components (and web apps) should be built.
    a. Introduced unidirectional data flow through props and state. View (what you see on screen) is a pure function of the data!
        i. If the *data* (properties and state) is *consistent*, the resulting view will always be the same.
        ii. Easier to reason about and debug!

2. Removes mundane, boilerplate, and configuration to allow you to focus on rapid prototyping… thanks to create-react-app.

3. Extremely performant (fast) due to VirtualDOM.

VirtualDOM

Immediate-mode-rendering brought to the DOM. React keeps track of what's in the DOM for you and only re-renders what has changed since the last render cycle.



What we’ll forego (*that are actually important*)?

- ReactJS vs. Vue vs. Angular vs. …
- Lifecycle Methods (we’ll only briefly cover constructor(), render(), and componentDidMount() ~ similar to .ready())
- Service Workers (progressive apps that fail gracefully)
- React Router
- Redux / Flux / MobX / RxJS / ... (we’ll keep state-management Vanilla)
- Advanced Styling: SASS/LESS/CSS Modules
- Higher Order Components
- Server-Side Rendering

Now some more fun dry stuff...

Must be aware… background

npm

- Node.js Package Manager
  - Used for dependency management
  - Driven by /package.json file
  - Installs external libraries into project’s /node_modules directory

Must be aware...

create-react-app

- An npm package
- Wizard, pre-rigged, pre-configured starter app
  - Webpack (bundler)
  - Babel (compiler / “transpiler”)
  - Pre-selected /node_modules (utility libraries)
  - Dev server with live-reloading (also look into HMR)
  - Linter
  - Build process

- Sole reason we can forego so much technical detail

Must be aware...

directory structure

- /
  - Note package.json
- /node_modules
- /src
  - Where .js/.jsx sits
  - App.css for now will go here as well
- /public ~ specifically CF code and assets will go here
  - Notice that static .html and .css files here by default. Will be copied
- /build

Must be aware… styling

css

- Absolute positioning, so we can abuse LEFT | TOP | RIGHT | BOTTOM style assignments
- vh and vw units, I like them better than percentage units (% of what)
- display: flex
  - Mainly for auto-margins (centering Person) and flush alignment of assets
- In React, inline styling uses camel-case style={{backgroundColor: “skyblue”}}
  - Otherwise it’s probably in /src/App.css where we reference class with className=“style-class”

Must be aware...

React/JS-specific Stuff:

- Props and State: You can think of them as scopes, kinda.
  - Props ~ cfarguments
  - State ~ session or even application scope (if every application is a new component)

```
<ComponentName prop1={value} prop2={function()} onWheel={wheelHandler()}
onClick={clickHandler()}></ComponentName>
```

```
this.state = {
	wealthy: false,
}
```
(set in the constructor() function of a class)

Must be aware… animations

css animations

- transform(ations): translateX(), translateY(), rotate()
- transition(s): change properties over time - opacity, color
- animation: shorthand to peg a @keyframe to an element
  - Keyframes - like pages on a flipbook, at 1% {change opacity to .1}, 2% {change to .2…}
  - Steps(): modifier to set how many keyframes will render in the animation timeframe. Will use to death for our vegas sprite-sheets.

Must be aware...

React/JS-specific Stuff:

- Updating state object since you probably want to modify it

```
this.setState({ wealthy: true });
```

- Events: Specifically onWheel() event, returning e.deltaY (-100 to 100)

```
<ComponentName arg1={value} arg2={eventHandler()}
onWheel={wheelHandler()} onClick={clickHandler()}></ComponentName>
```

Must be aware...

React/JS-specific Stuff:

- Ternary operators
  - && -> shorthand for if/then… { this.state.wealthy && beHappy(); }
  - condition ? true : false shorthand for if/else… { this.state.wealthy ? beHappy() : beSad() }
- Comments in JSX: {/* comments go here */}
- What are the curly brackets? {} = ## in CF
  - Evaluate your variables
- setTimeout(), setInterval(), requestAnimationFrame()
- ES6 (ECMA Script 2015 spec), specifically arrow function definitions
  - facing = (direction) => { return direction ? "forward" : "backward" }
  - defines and **binds** function to component's scope (attach to class prototype)

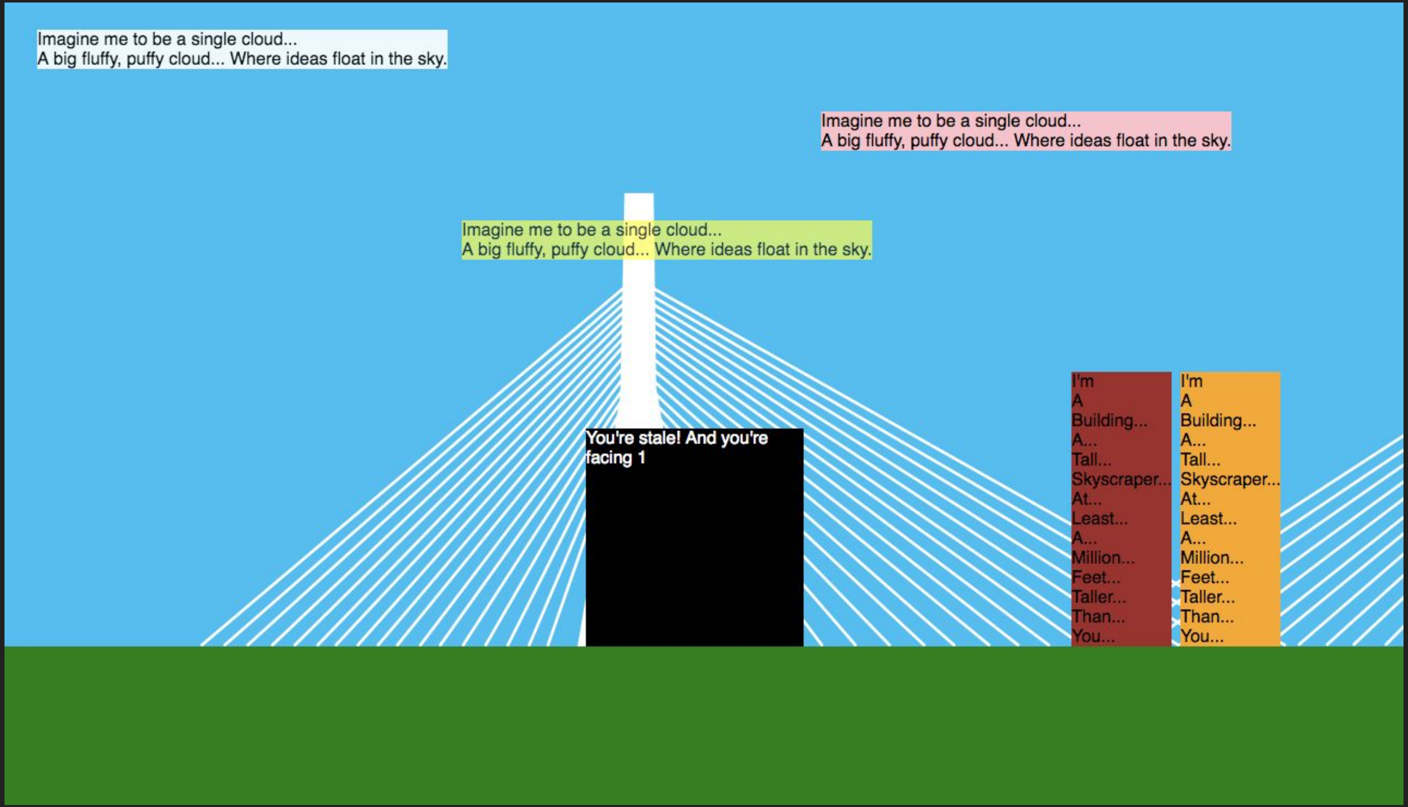
Technical assumptions

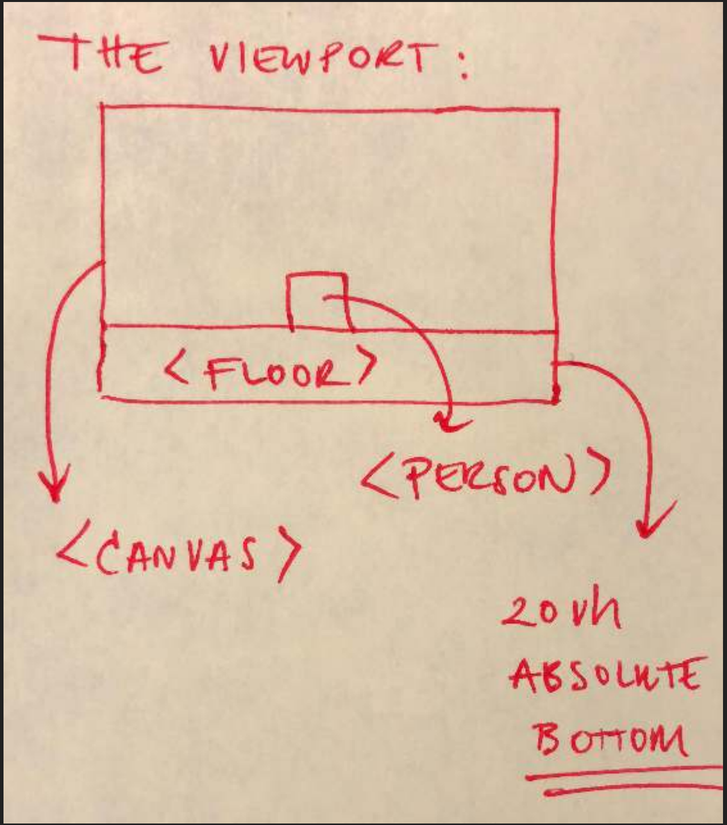
General knowledge of how websites are. CF, JS, JSON, CSS familiarity will help!

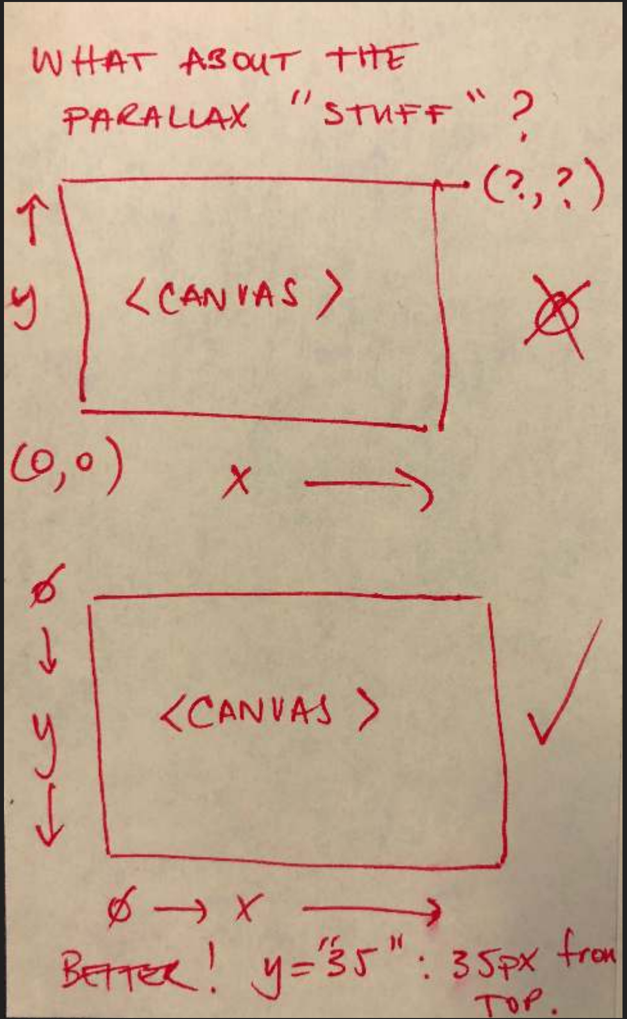
75% ReactJS / 25% CF

- npm installed (v5.2+)
- create-react-app available | npx create-react-app
- React Developer Tools installed
- Commandbox is running CF2018 on :8080 serving /draft-studios-website/build/* (`npm run build` inside /draft-studios-website first)
    - Component Cache: off
    - WebSockets on :8581 (default)
- create-react-app is running Webpack dev server on :3000 (default)
    - `npm <run> start`

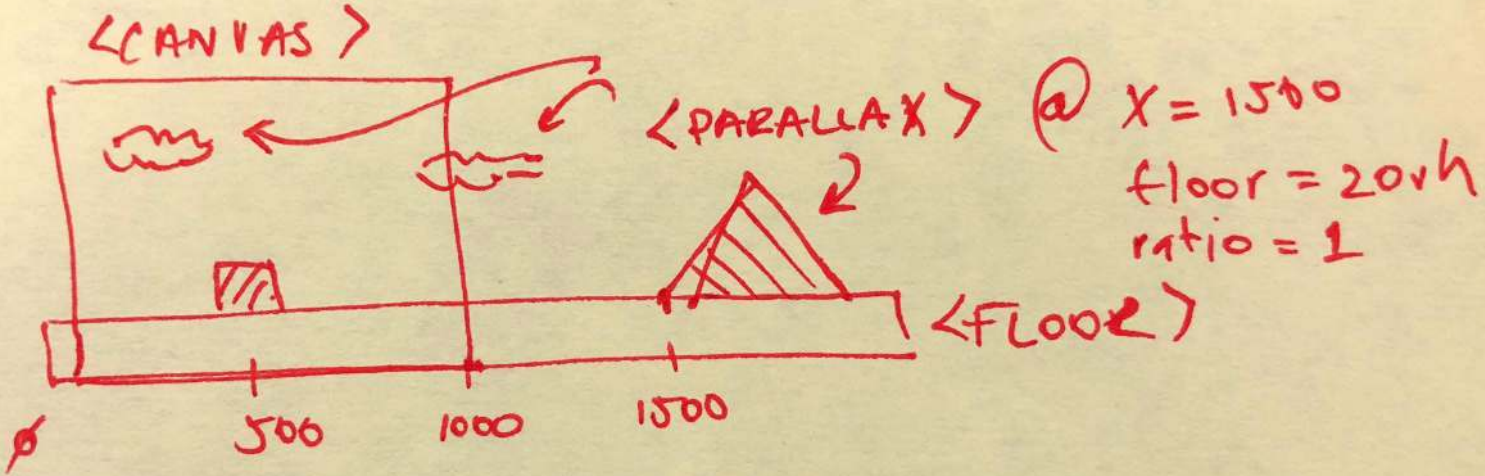
Buildout (equivalent to draft-studios-rig repository)


Imagine me to be a single cloud...
A big fluffy, puffy cloud... Where ideas float in the sky.
Imagine me to be a single cloud...
A big fluffy, puffy cloud... Where ideas float in the sky.
Imagine me to be a single cloud...
A big fluffy, puffy cloud... Where ideas float in the sky.
You're stale! And you're facing 1
I'm A Building... A... Tall... Skyscraper... At... Least... A... Million... Feet... Taller... Than... You...
I'm A Building... A... Tall... Skyscraper... At... Least... A... Million... Feet... Taller... Than... You...


THE VIEWPORT:


WHAT ABOUT THE PARALLAX "STUFF"?

BETTER! y="35": 35px from TOP.



★ ABUSING ABSOLUTE POSITIONING

LEFT | TOP | RIGHT | BOTTOM style

ASSIGNMENTS.

DISPLAY : FLEX !

Buildout:

```
$ npx create-react-app <app-name>
$ cd <app-name>
$ npm run start
```

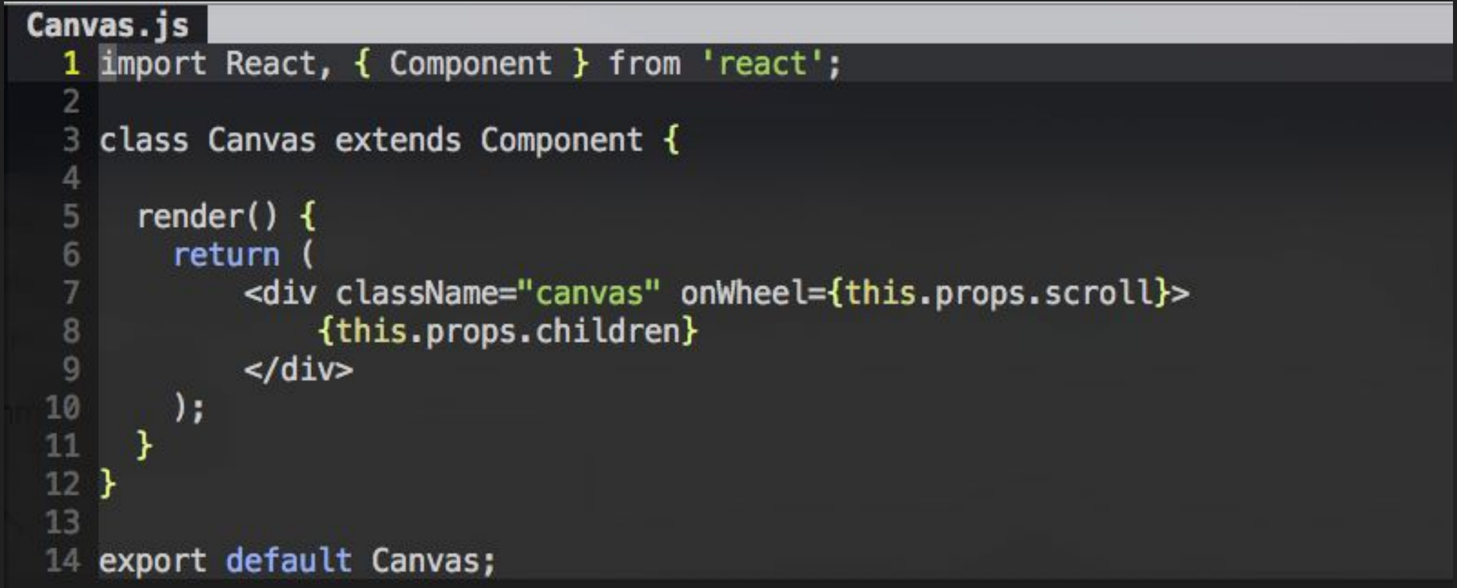
Note: I realized in my presentation buildout I didn’t get to cover breaking our individual <div> elements into their own separate components. This is an important concept and can be confusing if you don’t know what to lookout for. Totally helps as you’re navigating either draft-studios-rig or draft-studios-website repos.

To illuminate this, I’m adding the next three slides to supplement! - Minh

Breaking out components

/draft-studios-rig/src/Canvas.js is probably easiest example… note the import on line 1, class Canvas extends Component and export default Canvas;

{this.props.children} also important--putting all nested <Parallax/> inside Canvas.



Canvas.js

```
 1 import React, { Component } from 'react';
 2
 3 class Canvas extends Component {
 4
 5   render() {
 6     return (
 7         <div className="canvas" onWheel={this.props.scroll}>
 8             {this.props.children}
 9         </div>
10     );
11   }
12 }
13
14 export default Canvas;
```

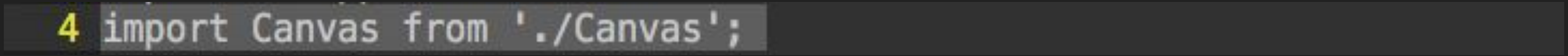
Breaking out components

/draft-studios-rig/src/Canvas.js could also be written like below, using functional components (since you don't need State, class assignment is unnecessary)! You call these "pure" or "stateless" functions/components. Read this to understand.

```
import React from 'react';
const Canvas = (props) => {
 return (
   <div className="Canvas" onWheel= {props.scroll}>
      {props.children} {/* no this scope, we're not in a class! */}
   </div>
 );
}
export default Canvas;  // this code is functionally identical to last slide!
```

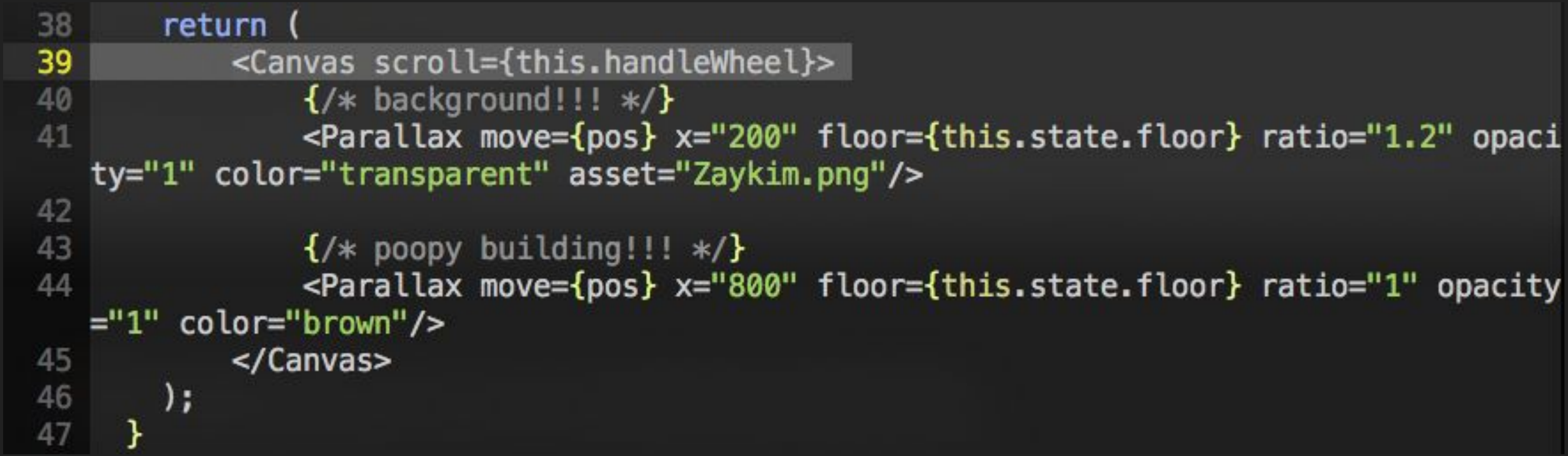
Breaking out components

/draft-studios-rig/src/App.js would then import Canvas (the default export):



```
import Canvas from './Canvas';
```

And then use them like declarative tags (notice again <Parallax/> children):

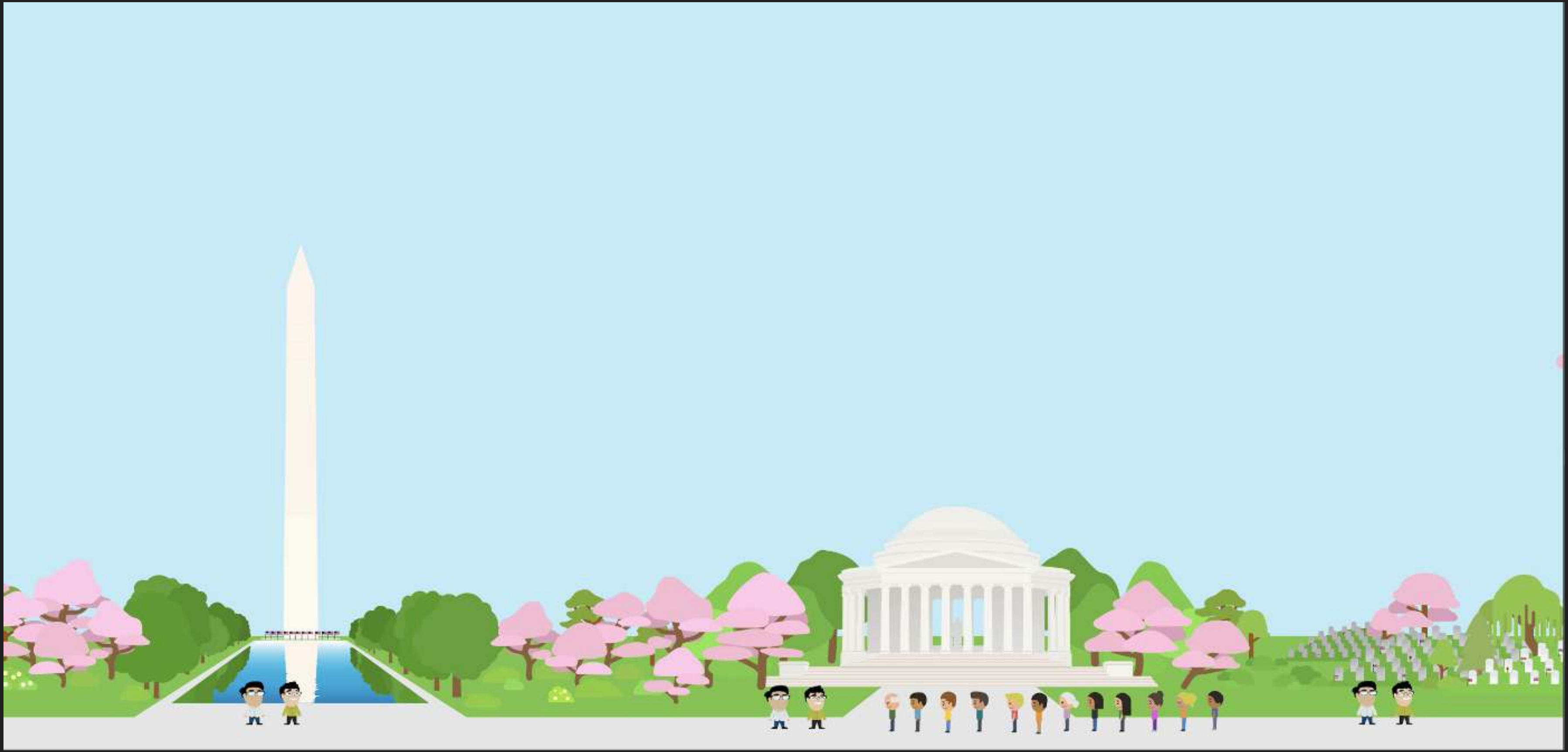


```
    return (
        <Canvas scroll={this.handleWheel}>
            {/* background!!! */}
            <Parallax move={pos} x="200" floor={this.state.floor} ratio="1.2" opacity="1" color="transparent" asset="Zaykim.png"/>

            {/* poopy building!!! */}
            <Parallax move={pos} x="800" floor={this.state.floor} ratio="1" opacity="1" color="brown"/>
        </Canvas>
    );
  }
```

Jump to draft-studios-rig code

- Add some buildings and clouds
- Change the floor height and color
- Go over handleWheel() event handler

Where do we go from here? Jump to the full repo!



You just showed me all JS, where's CF?

Done - Parallax (clouds, buildings, trees)

Done - Scroll Handling (translating movement)

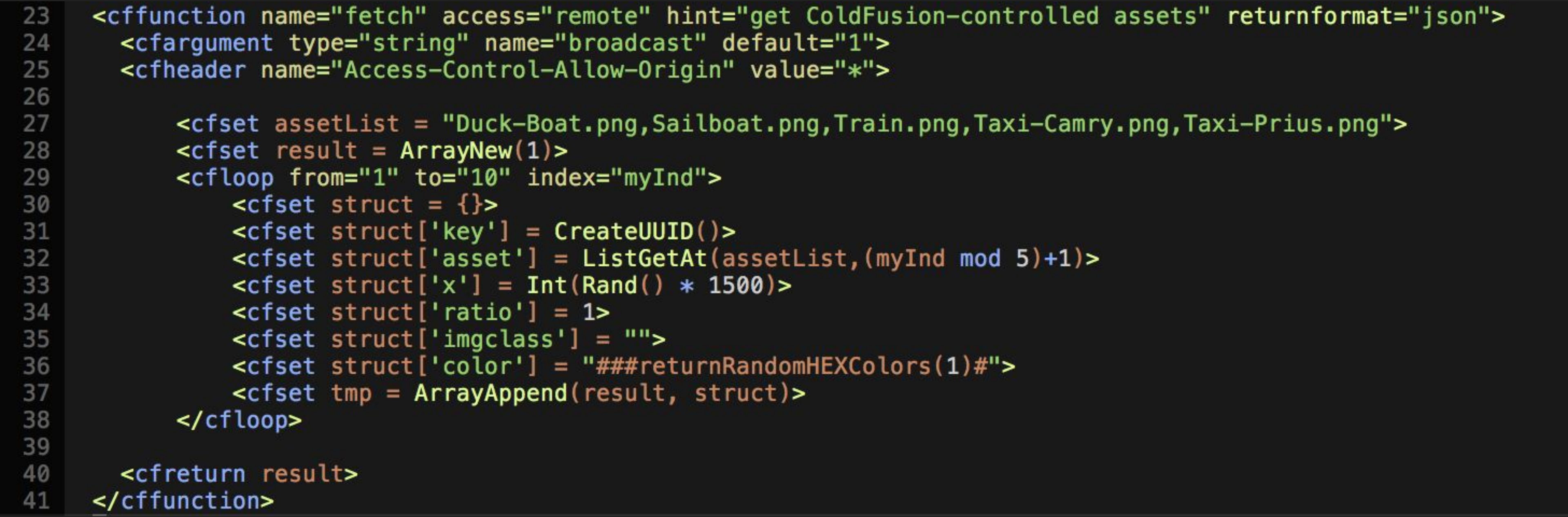
- Fetching JSON data from a ColdFusion CFC the regular way
- ColdFusion Web Sockets for real-time interactivity (world boss!)

Simple Animations (running, etc)

In the draft-studios-website repository you'll find the code referenced in the next chunk of slides. Please follow the directions on the repository's README.md.
For prod deployment, run `npm run build` (look at package.json if you're curious).
It will generate a /build directory which is what you'll upload to your CF app directory.

Fetching data via CFC

CFC Setup: /draft-studios-website/public/services/ds.cfc

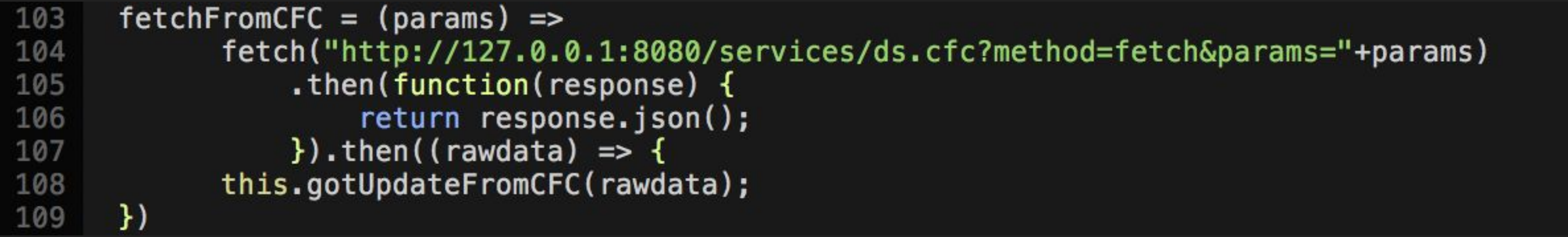


```
<cffunction name="fetch" access="remote" hint="get ColdFusion-controlled assets" returnformat="json">
  <cfargument type="string" name="broadcast" default="1">
  <cfheader name="Access-Control-Allow-Origin" value="*">

      <cfset assetList = "Duck-Boat.png,Sailboat.png,Train.png,Taxi-Camry.png,Taxi-Prius.png">
      <cfset result = ArrayNew(1)>
      <cfloop from="1" to="10" index="myInd">
          <cfset struct = {}>
          <cfset struct['key'] = CreateUUID()>
          <cfset struct['asset'] = ListGetAt(assetList,(myInd mod 5)+1)>
          <cfset struct['x'] = Int(Rand() * 1500)>
          <cfset struct['ratio'] = 1>
          <cfset struct['imgclass'] = "">
          <cfset struct['color'] = "###returnRandomHEXColors(1)#">
          <cfset tmp = ArrayAppend(result, struct)>
      </cfloop>

  <cfreturn result>
</cffunction>
```

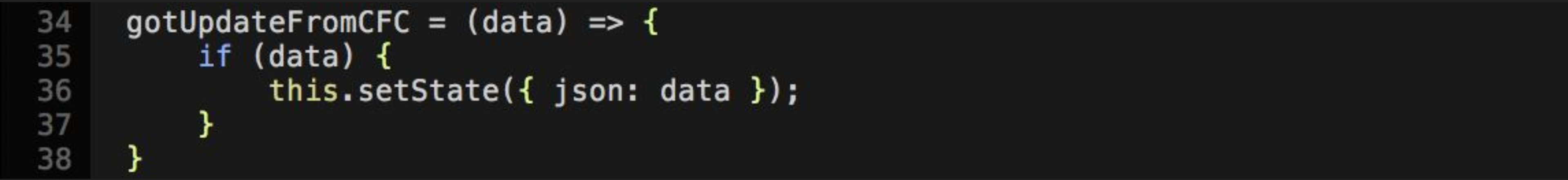
NOTE use array notation here instead of dot for keeping case,
older CF upper-cases struct keys which can be an issue during SerializeJSON step

Fetching data via CFC - JS side

CFC Setup: /draft-studios-website/src/FetchColdFusionAssets.js

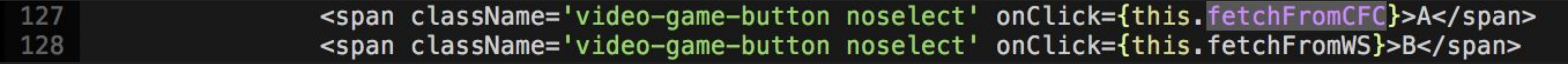


```
fetchFromCFC = (params) =>
      fetch("http://127.0.0.1:8080/services/ds.cfc?method=fetch&params="+params)
          .then(function(response) {
              return response.json();
          }).then((rawdata) => {
      this.gotUpdateFromCFC(rawdata);
})
```



```
gotUpdateFromCFC = (data) => {
    if (data) {
        this.setState({ json: data });
    }
}
```

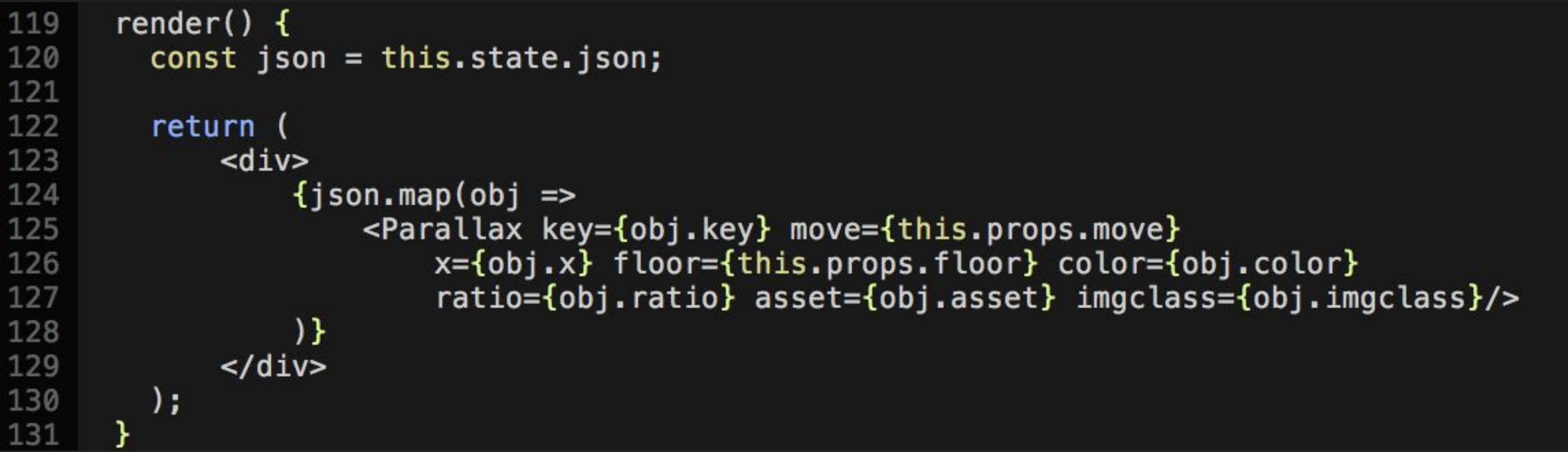
Trigger the fetch this way:



```
<span className='video-game-button noselect' onClick={this.fetchFromCFC}>A</span>
<span className='video-game-button noselect' onClick={this.fetchFromWS}>B</span>
```

Fetching data via CFC - JS side

render() should then look like this, so as to react to this.state.json change:

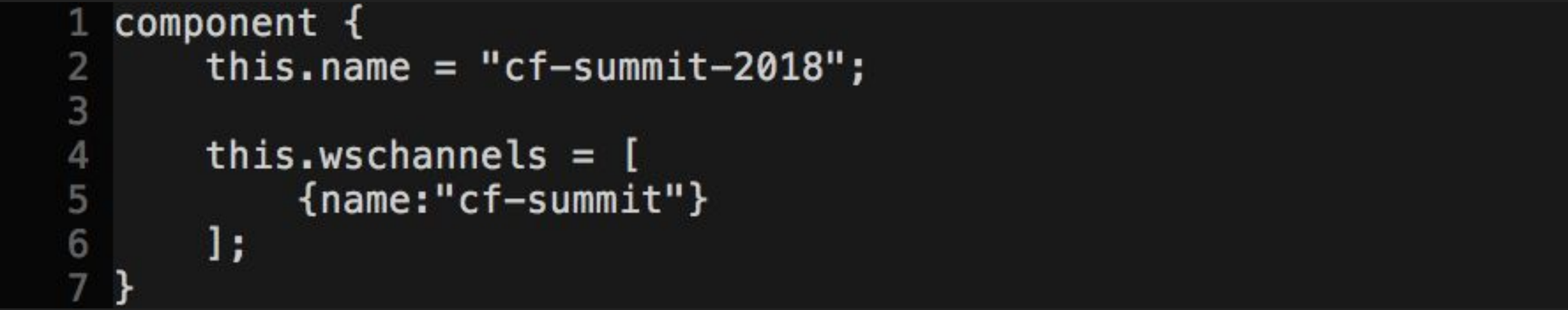


```
render() {
  const json = this.state.json;

  return (
      <div>
          {json.map(obj =>
              <Parallax key={obj.key} move={this.props.move}
                  x={obj.x} floor={this.props.floor} color={obj.color}
                  ratio={obj.ratio} asset={obj.asset} imgclass={obj.imgclass}/>
          )}
      </div>
  );
}
```

It’s like looping through a collection/array of structs...

Fetching data via WebSockets - Channel Setup

WS Setup: /draft-studios-website/public/services/Application.cfc



```
component {
    this.name = "cf-summit-2018";

    this.wschannels = [
        {name:"cf-summit"}
    ];
}
```

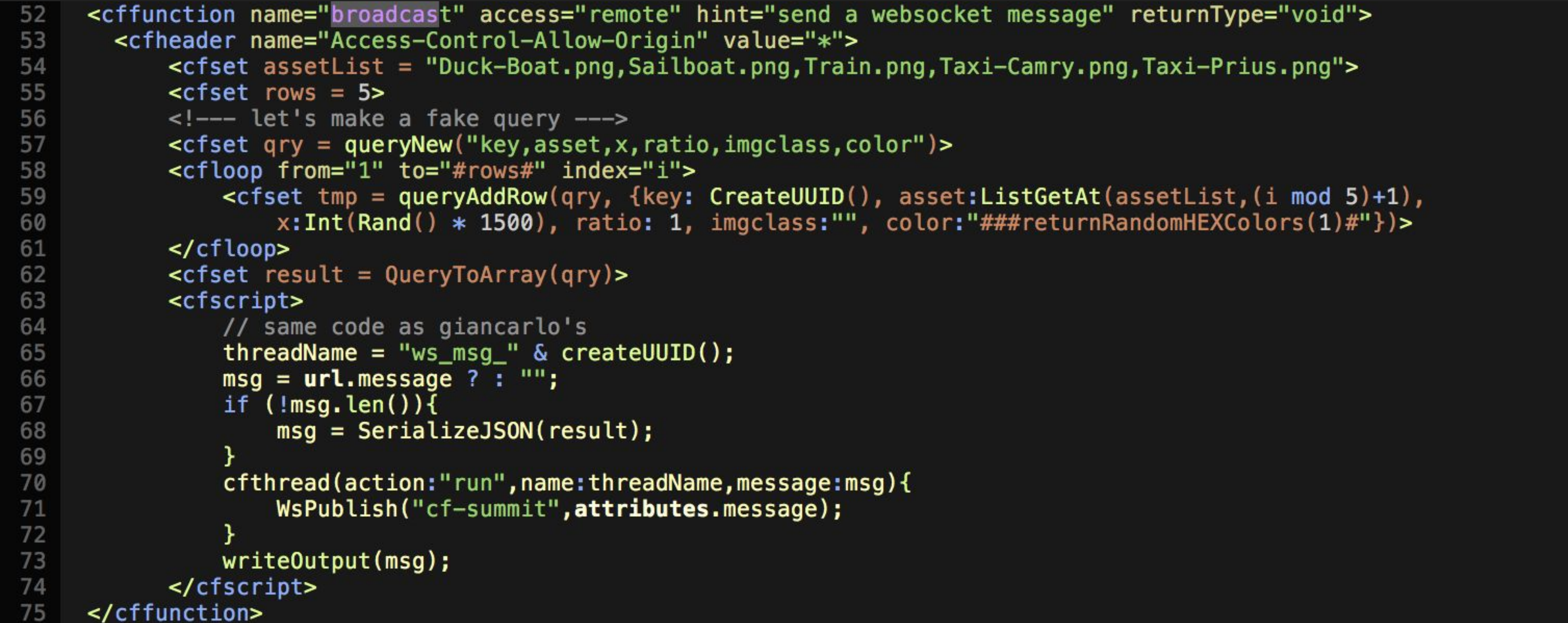
NOTE This registers an app name and a channel name so we can begin sending messages through it.

See Giancario Gomez’s presentation. He’s the master of this!

Make sure ColdFusion WebSockets are enabled and verify the port it’s running on.

Fetching data via WebSockets - CF side

WS Setup: /draft-studios-website/public/services/ds.cfc

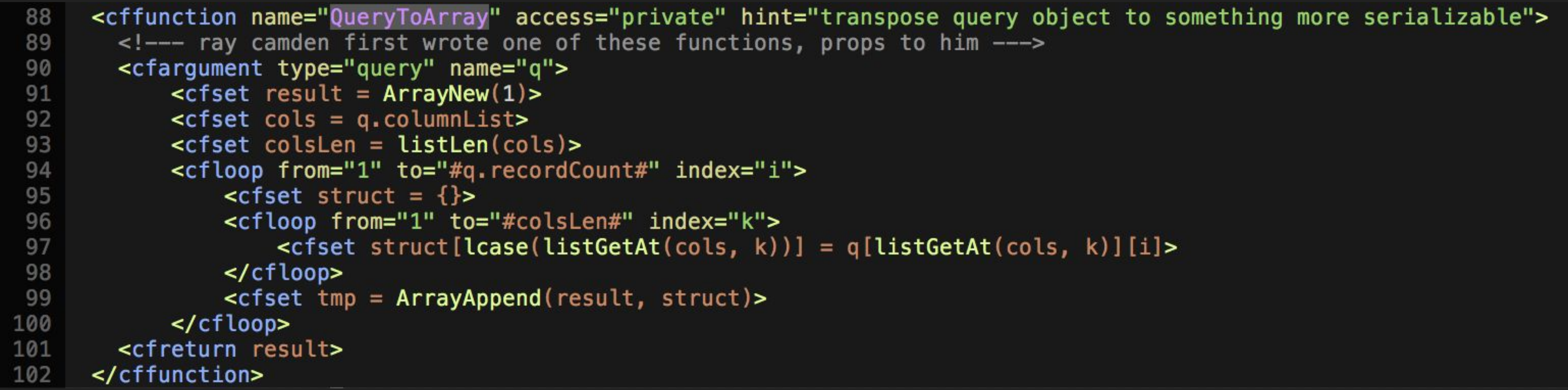


```
<cffunction name="broadcast" access="remote" hint="send a websocket message" returnType="void">
 <cfheader name="Access-Control-Allow-Origin" value="*">
   <cfset assetList = "Duck-Boat.png,Sailboat.png,Train.png,Taxi-Camry.png,Taxi-Prius.png">
   <cfset rows = 5>
   <!--- let's make a fake query --->
   <cfset qry = queryNew("key,asset,x,ratio,imgclass,color")>
   <cfloop from="1" to="#rows#" index="i">
     <cfset tmp = queryAddRow(qry, {key: CreateUUID(), asset:ListGetAt(assetList,(i mod 5)+1),
       x:Int(Rand() * 1500), ratio: 1, imgclass:"", color:"###returnRandomHEXColors(1)#"})>
   </cfloop>
   <cfset result = QueryToArray(qry)>
   <cfscript>
     // same code as giancarlo's
     threadName = "ws_msg_" & createUUID();
     msg = url.message ? : "";
     if (!msg.len()){
       msg = SerializeJSON(result);
     }
     cfthread(action:"run",name:threadName,message:msg){
       WsPublish("cf-summit",attributes.message);
     }
     writeOutput(msg);
   </cfscript>
</cffunction>
```

Fetching data via WebSockets - CF side

WS Setup: /draft-studios-website/public/services/ds.cfc

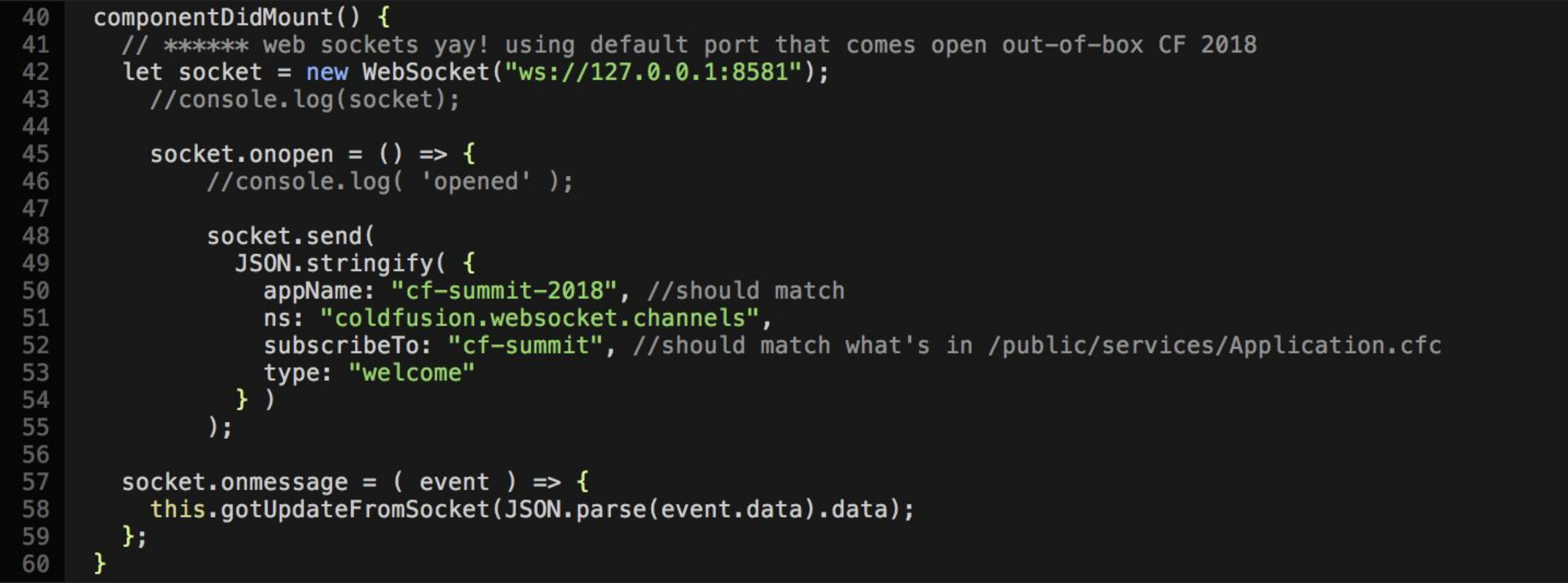
Working with Queries!



```
<cffunction name="QueryToArray" access="private" hint="transpose query object to something more serializable">
 <!--- ray camden first wrote one of these functions, props to him --->
 <cfargument type="query" name="q">
   <cfset result = ArrayNew(1)>
   <cfset cols = q.columnList>
   <cfset colsLen = listLen(cols)>
   <cfloop from="1" to="#q.recordCount#" index="i">
     <cfset struct = {}>
     <cfloop from="1" to="#colsLen#" index="k">
       <cfset struct[lcase(listGetAt(cols, k))] = q[listGetAt(cols, k)][i]>
     </cfloop>
     <cfset tmp = ArrayAppend(result, struct)>
   </cfloop>
 <cfreturn result>
</cffunction>
```

Fetching data via WebSockets - JS side

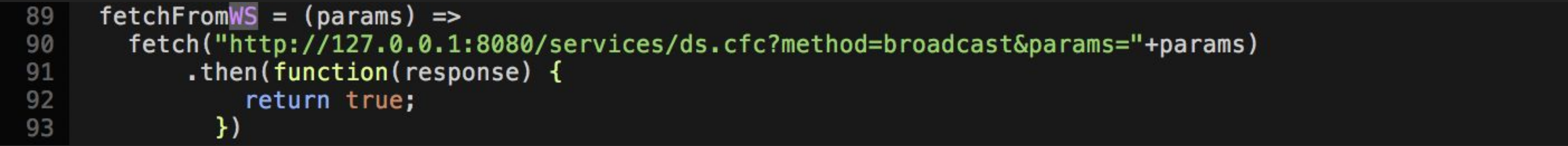
WS Setup: /draft-studios-website/src/FetchColdFusionAssets.js



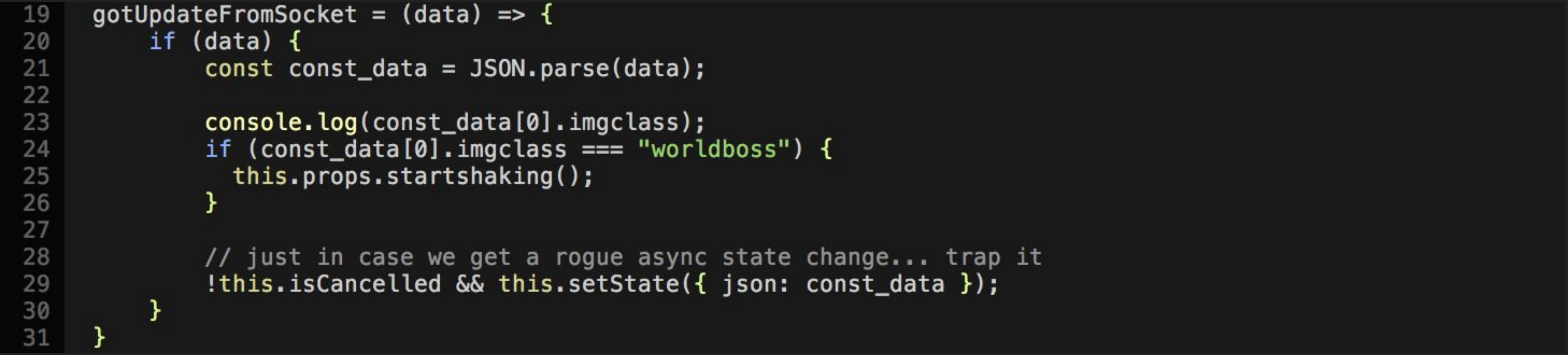
```
componentDidMount() {
 // ****** web sockets yay! using default port that comes open out-of-box CF 2018
 let socket = new WebSocket("ws://127.0.0.1:8581");
  //console.log(socket);

  socket.onopen = () => {
    //console.log( 'opened' );

    socket.send(
     JSON.stringify( {
      appName: "cf-summit-2018", //should match
      ns: "coldfusion.websocket.channels",
      subscribeTo: "cf-summit", //should match what's in /public/services/Application.cfc
      type: "welcome"
     } )
    );

 socket.onmessage = ( event ) => {
  this.gotUpdateFromSocket(JSON.parse(event.data).data);
 };
}
```

Fetching data via WebSockets - JS side

WS Setup: /draft-studios-website/src/FetchColdFusionAssets.js

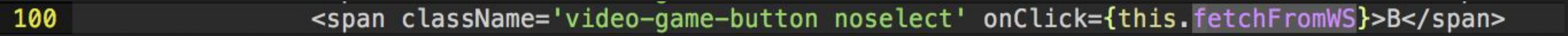


```
fetchFromWS = (params) =>
  fetch("http://127.0.0.1:8080/services/ds.cfc?method=broadcast&params="+params)
      .then(function(response) {
          return true;
        })
```



```
gotUpdateFromSocket = (data) => {
    if (data) {
        const const_data = JSON.parse(data);

        console.log(const_data[0].imgclass);
        if (const_data[0].imgclass === "worldboss") {
          this.props.startshaking();
        }

        // just in case we get a rogue async state change... trap it
        !this.isCancelled && this.setState({ json: const_data });
    }
}
```

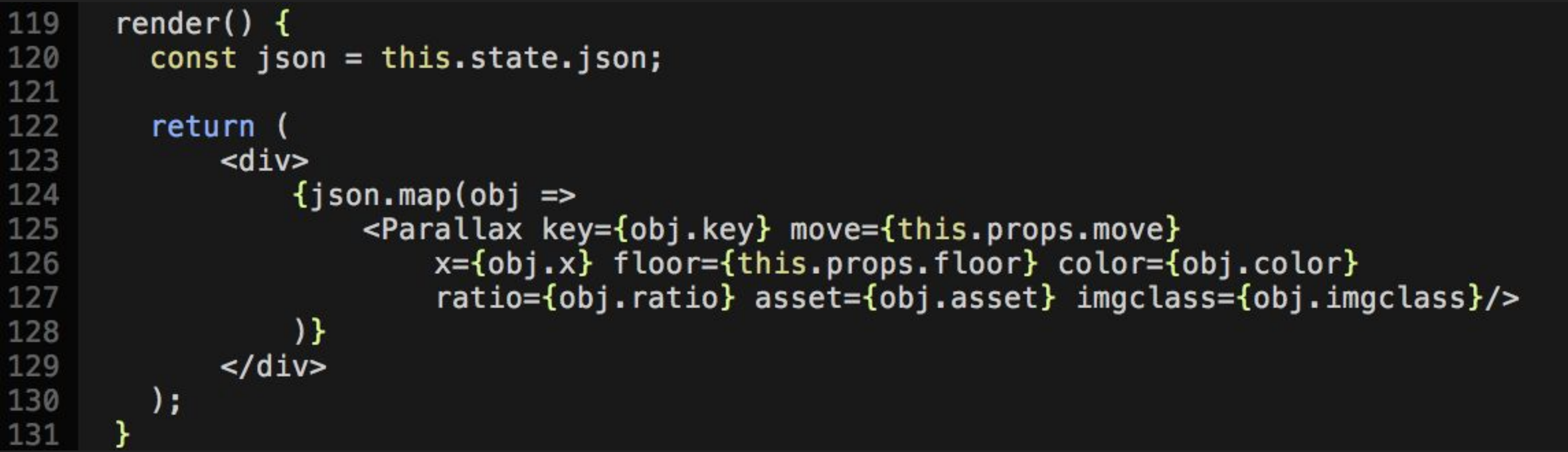
Trigger the broadcast similar way:



```
                <span className='video-game-button noselect' onClick={this.fetchFromWS}>B</span>
```

Fetching data via WebSockets - JS side

render() should is actually the same, so as to react to this.state.json change:

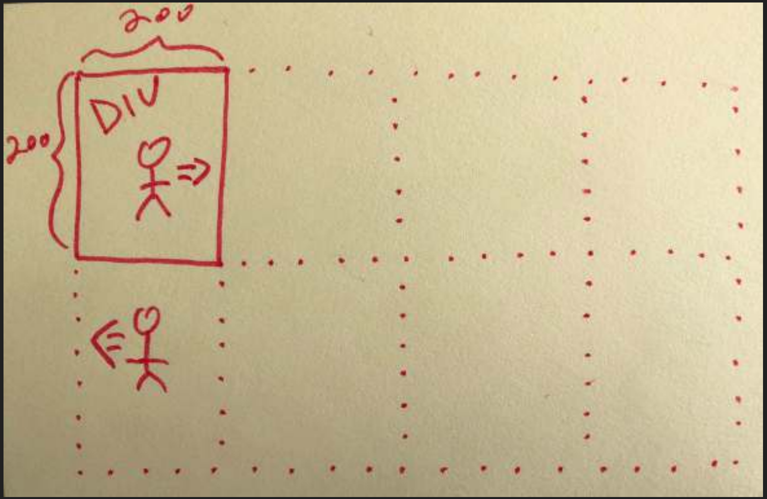


```
render() {
  const json = this.state.json;

  return (
      <div>
          {json.map(obj =>
              <Parallax key={obj.key} move={this.props.move}
                  x={obj.x} floor={this.props.floor} color={obj.color}
                  ratio={obj.ratio} asset={obj.asset} imgclass={obj.imgclass}/>
          )}
      </div>
  );
}
```

It's again like looping through a collection/array of structs...

ANIMATIONS!!! Hard one first.

/draft-studios-website/src/Person.js and /draft-studios-website/src/App.css



2x8 Grid a.k.a. Sprite Sheet

First row signifies animation frames going right
Second row going left

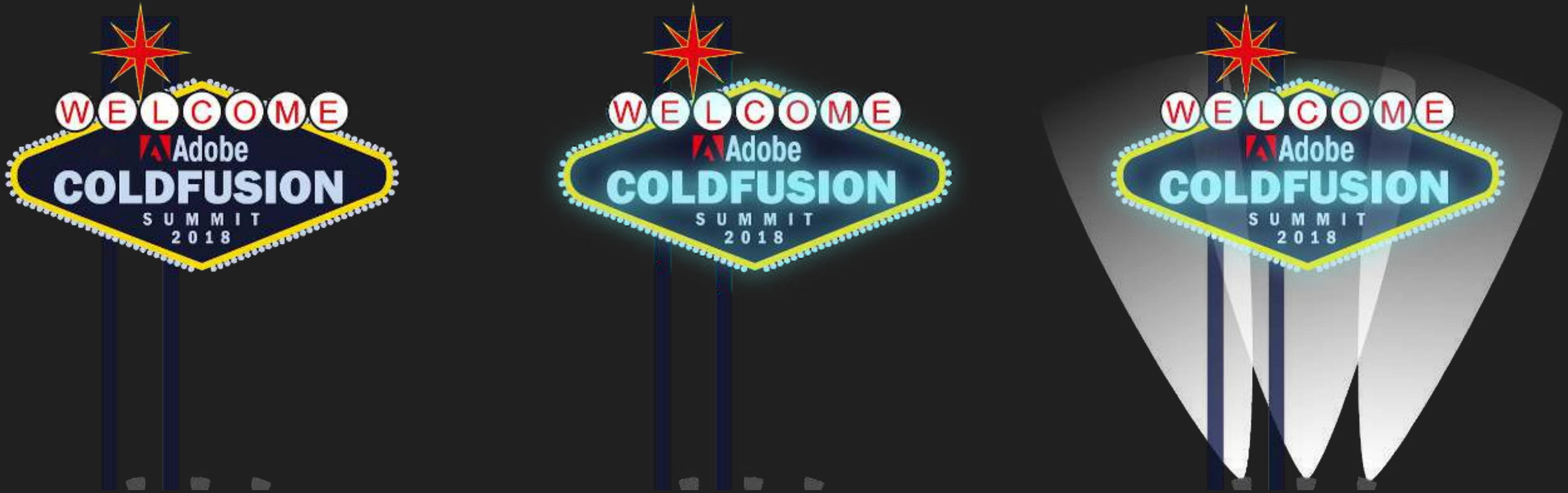
Two <div>'s

- one at the frame height and width with overflow:hidden.
- bigger one is the entire sheet (ie: 400x800)!

Since Person.js' animation needed to be state aware, css alone won't cut it!

ANIMATIONS!!! Easy one.

/draft-studios-website/src/App.js and /draft-studios-website/src/App.css

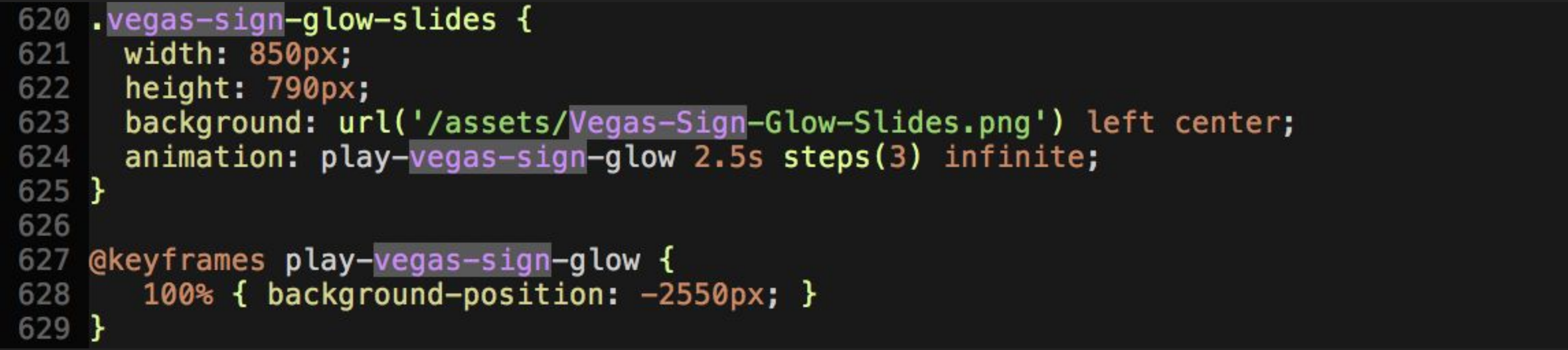


1x3 Sprite Sheet, One <div>

- Use step(3) css animation <see next slide>

ANIMATIONS!!! Easy one.

/draft-studios-website/src/App.js and /draft-studios-website/src/App.css



```css
.vegas-sign-glow-slides {
  width: 850px;
  height: 790px;
  background: url('/assets/Vegas-Sign-Glow-Slides.png') left center;
  animation: play-vegas-sign-glow 2.5s steps(3) infinite;
}

@keyframes play-vegas-sign-glow {
   100% { background-position: -2550px; }
}
```

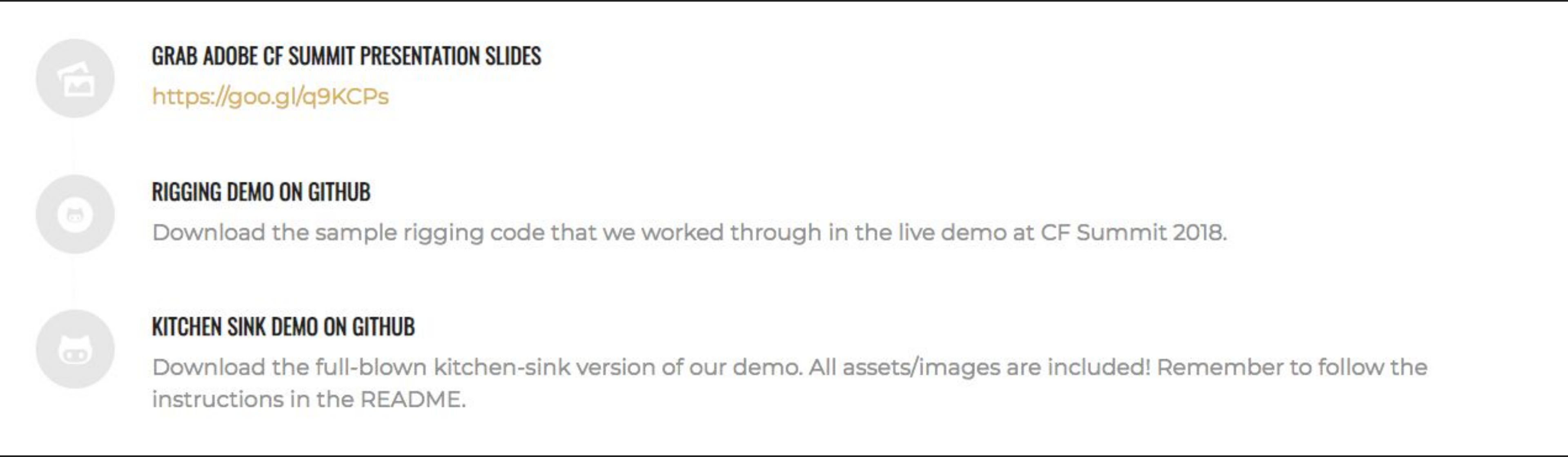
- Explicitly set className to div. Explicitly set width and height to single frame. Set background to big image. Animation shorthand property set to @keyframe name. Easy peasey!

Let’s take a step back...

Hey, stop reading and pay attention!

Resources from this session

All found on http://draftstudios.com right at the top of the screen!



GRAB ADOBE CF SUMMIT PRESENTATION SLIDES

https://goo.gl/q9KCPs

RIGGING DEMO ON GITHUB

Download the sample rigging code that we worked through in the live demo at CF Summit 2018.

KITCHEN SINK DEMO ON GITHUB

Download the full-blown kitchen-sink version of our demo. All assets/images are included! Remember to follow the instructions in the README.

“the world is vast and you can’t possibly learn everything…”

Minh Vo

Lead Engineer @ **Draft Studios**
Head of Development, Security @ **Advantage Data**

Minh is a full-stack developer, Oracle certified whiz, and pixel-perfect graphics artist who's worked on the bleeding edge of Technology for almost twenty years. He's created two award-winning FinTech products, written financial models, developed phonetic and sentiment analysis algorithms, built highly interactive charting libraries and countless other business apps.

His work has primarily impacted the Finance and Education industries for which he's groomed CEOs, VPs, and Directors. His awe for Technology is contagious. His thirst for knowledge insatiable. Not to mention he's a pretty nice guy!

On his off-time, he racks his brain over happiness and the meaning of life. He thinks it has something to do with playing guitar, shooting a bow and arrow, mixing drinks, and karaoking to old rock jams with family.

Thanks!

programmer
minh@draftstudios.com
www.draftstudios.com

illustrator
jimmy@draftstudios.com

HUGE THANKS to everyone I've personally met at the conference and the Adobe Staff!

You're all so supportive!!! <3

DRAFT DESIGN STUDIOS

Special thanks to Kishore and Elishia!
(´·ω·`) Sheila, Molly, and Jimmy!!