

# Single Page App Day

## Brought to you by Chariot Solutions

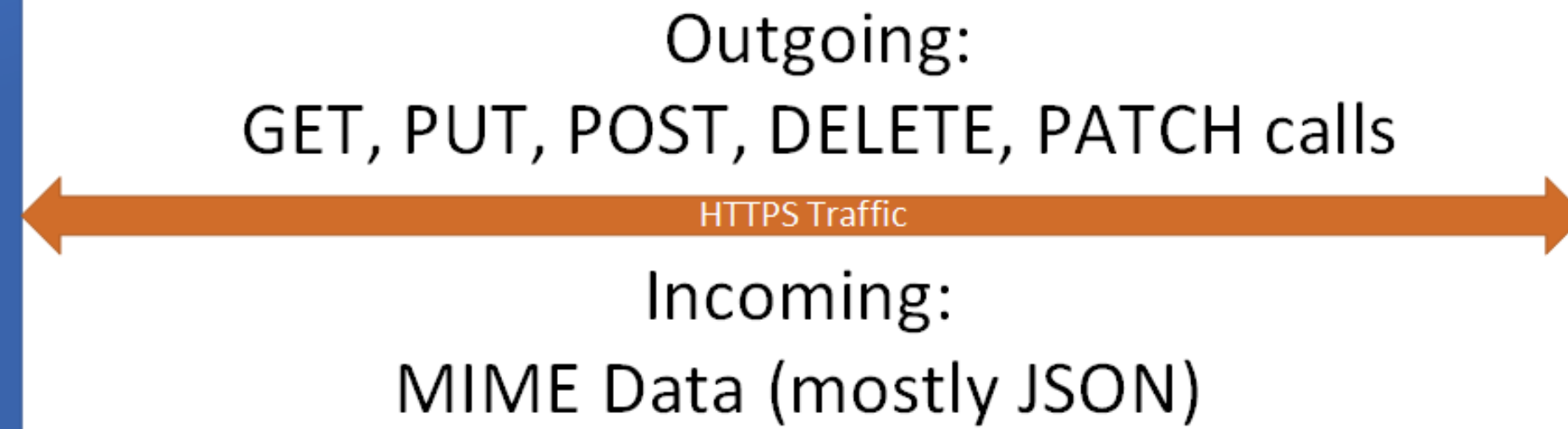
---

# What are Single Page Applications?



# Single Page Application Division of Labor

Single Page  
Application



“Server”

- Rich user interface
- Client-side logic
- Hold state, cache data
- Make requests to server(s)

- Respond to client requests
- Handle business logic
- Generally stateless
- Provide security

# Three popular SPAs

---

- **Angular** - Google-created *Framework* that includes APIs for many chores
- **React** - A Component library written from the ground up for a single purpose, has many "friends"
- **VueJS** - A "better, simpler Angular?"

# Why an SPA?

---

# Why an SPA?

---

# Why an SPA?

---



# Why an SPA?

---

# Why an SPA?

---

# Why an SPA?

---

# Why an SPA?

---

# Why an SPA?

---

# Why an SPA?

---

# Mad SPA Skills...

# "Modern" JavaScript

---



# "Modern" JavaScript

---

- A moving target...

# "Modern" JavaScript

---

- A moving target...
- **JavaScript** is actually officially ECMAScript

# "Modern" JavaScript

---

- A moving target...
- **JavaScript** is actually officially ECMAScript
- **ECMAScript 2018** is the current version

# "Modern" JavaScript

---

- A moving target...
- **JavaScript** is actually officially ECMAScript
- **ECMAScript 2018** is the current version
- Most developers develop in **ECMAScript 2015** or higher and translate it to an older version

Say it: ECMAScript = ES =  
JavaScript



# Babel - [babeljs.io](https://babeljs.io)

---

**Babel** "transpiles" ECMAScript 2018 → ECMAScript 5 (which every browser interprets) or higher...

# Babel - [babeljs.io](https://babeljs.io)

---

**Babel** "transpiles" ECMAScript 2018 → ECMAScript 5 (which every browser interprets) or higher...

- Compiles down to ECMAScript 5



# Babel - [babeljs.io](https://babeljs.io)

---

**Babel** "transpiles" ECMAScript 2018 → ECMAScript 5 (which every browser interprets) or higher...

- Compiles down to ECMAScript 5
- Can compile to newer versions of ECMAScript

# Design and UI Skills

---

- User Experience Design
- HTML and Cascading Style Sheets

# JavaScript Skills

---

- Shed the jQuery knowledge, it's obsolete
- JavaScript is NOT Java
- The event loop controls performance

Key JavaScript Skills (this IS a  
JavaScript day...)



# Prototypes ARE Inheritance Model

```
1  var prototype = {  
2    sayHello: function() {  
3      console.log(this.name);  
4    }  
5  };  
6  
7  var obj = Object.create(prototype);  
8  obj.name = 'Ken';  
9  
10 console.log(obj.sayHello());  
11  
12 // output: 'Ken'
```

# Class definitions (i.e. constructor functions)

```
1  class Person {  
2      constructor(first, last) {  
3          this.first = first;  
4          this.last = last;  
5      }  
6  }
```

Turns into...

```
1  function Person(first, last) {  
2      this.first = first;  
3      this.last = last;  
4  }
```



Classes create constructor functions!

# Classes and Inheritance

```
1  class Customer extends Person {  
2  
3      constructor(first, last, orders) {  
4          super(first, last);  
5          this.orders = orders;  
6      }  
7  
8      reportOrders() {  
9          return `Customer ${this.first} ${this.las  
10             has ${orders.length} orders`;  
11      }  
12 }
```



# The Customer Prototype

```
> Customer.prototype
< ◀ Person {constructor: f, reportOrders: f} ⓘ
  ▶ constructor: class Customer
  ▶ reportOrders: f reportOrders()
  ▼ __proto__:
    ▶ constructor: class Person
    ▶ sayIt: f sayIt()
    ▶ __proto__: Object
>
```

Figure 1. Prototypes are objects...

💡 JavaScript is *function driven* and supports objects...

this is a mess...

# Without arrow functions

This will not work... will set `window.customer`

```
1 // bad
2 const customerWrapper = {
3   customer: [],
4   getCustomer: function(key) {
5     setTimeout(function() {
6       this.customer = { first: 'Ken', ... };
7     }, 4000);
8   }
9 }
```

# Arrow functions remember the **this** keyword from the outer function

Inherits "this" from the outer calling code, *not* the code that initiated it

```
1  const customerWrapper = {  
2    customer: [],  
3    getCustomer: function(key) {  
4      setTimeout(() => {  
5        this.customer = { first: 'Ken', ... };  
6      }, 4000);  
7    }  
8  }
```

# Promises, async and await

```
1  function getLocation() {  
2      return new Promise((resolve, reject) => {  
3          try {  
4              navigator.geolocation.watchPosition(  
5                  function (geoposition) {  
6                      navigator.geolocation.clearWatch(wa  
7                      resolve({ coords: geoposition.coord  
8                  }  
9              );  
10         catch (e) { reject(e); }  
11     });
```

# Calling a function with a promise

Original way - use the **.then** callback

```
1 getLocation()  
2 .then(  
3   location => {  
4     this.coords = location.coords;  
5     this.error = undefined;  
6   },  
7   error => {  
8     console.error(rejectPayload);  
9     this.coords = undefined;  
10    this.error = error;  
11  });  
12 )
```

# Newer way: Use **async** and **await**

```
1  class MyComponent {  
2    async getLocation = () => {  
3      const location = await this.getLocation();  
4      this.coords = location.coords;  
5    }  
6  }
```

# Use **try** and **catch** to deal with errors

```
1  class MyComponent {  
2    async getLocation = () => {  
3      try {  
4        const location = await this.getLocation  
5        this.coords = location.coords;  
6        this.error = undefined;  
7      } catch (e) {  
8        this.coords = undefined;  
9        this.error = e.message;  
10     }  
11   }  
12 }
```



# Higher order functions in JavaScript

---

- Functions that
  - Take other functions as input arguments
  - Return other functions as results

# Sample higher-order functions

map - transform an object or array

```
1 let items = [1, 2, 3, 4, 5];  
2 items.map(item => item * item);  
3 // result: 1, 4, 9, 16, 25
```

reduce - produce a single output from a collection

```
1 let items = [1, 2, 3, 4, 5];  
2 items.reduce((accum, item) => accum + item, 0);  
3 // result: 15
```

chain two higher-order functions

```
1 let items = [1, 2, 3, 4, 5];  
2 items.map(item => item * item)  
3   .reduce((i, v) => v+i, 0);  
4 // 55
```

# Observables

---

# Observables

---

- A future ECMAScript feature candidate

# Observables

---

- A future ECMAScript feature candidate
- Currently implemented by RxJS

# Observables

---

- A future ECMAScript feature candidate
- Currently implemented by **RxJS**
- Similar to promises, but
  - Observables provide a stream of events
  - Provide message delivery, error, and complete callbacks

# Simple Observable Examples

## Observable events "from" a collection

```
1 import {from} from 'rxjs';  
2  
3 .from([1, 2, 3, 4, 5])  
4 .subscribe(x => { console.log(x); });
```

## Observable events from mouse movement

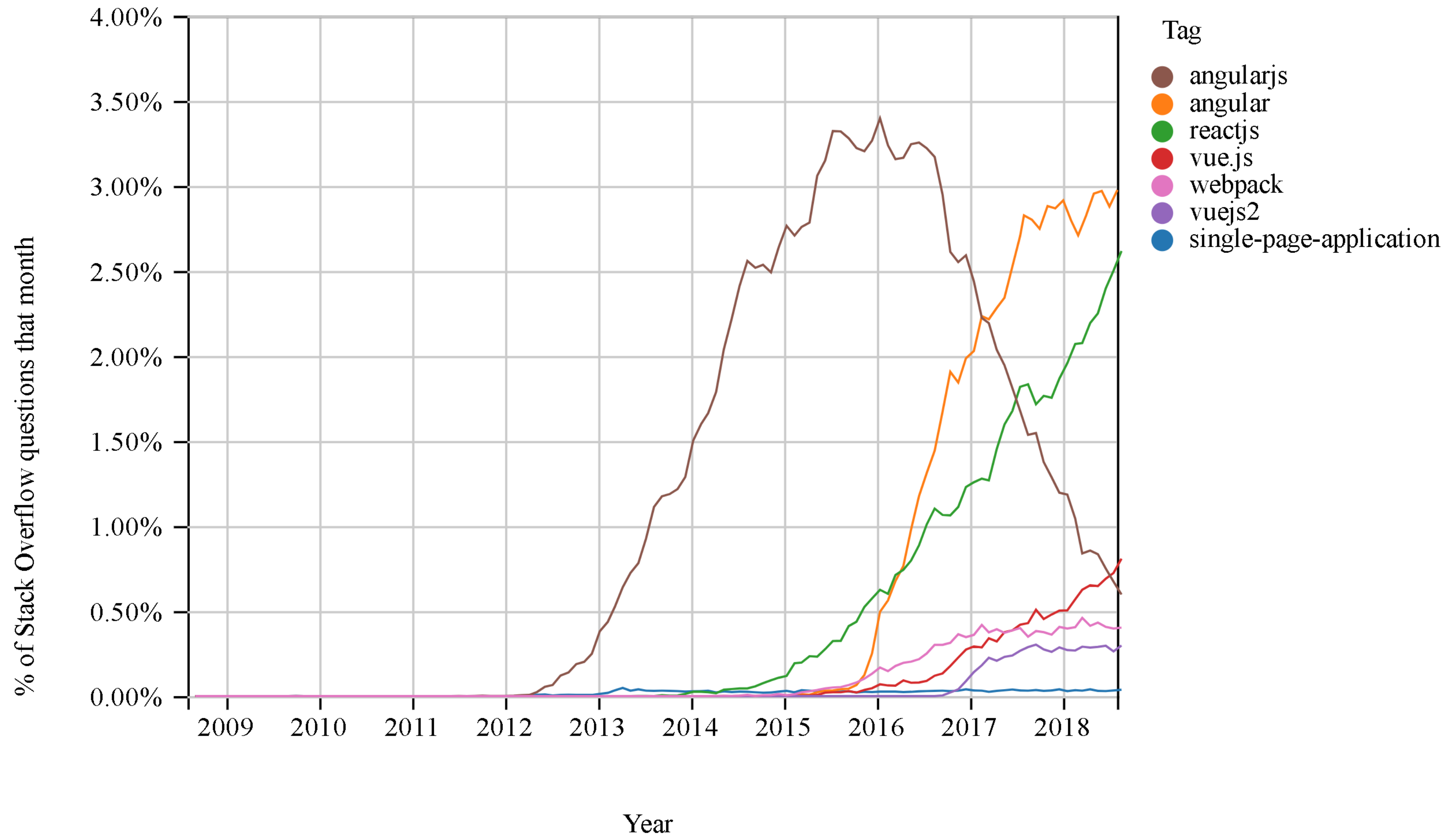
```
1 import {fromEvent} from 'rxjs';  
2  
3 fromEvent(document, 'mousemove')  
4 .subscribe(event =>  
5     console.log(event.clientX, event.clientY))
```

 Lots of interesting visualizations at <http://rxmarbles.com>

# The Growth of SPA

---





# Rest of Day : Agenda (lunch provided!)

# Rest of Day : Agenda (lunch provided!)

# Rest of Day : Agenda (lunch provided!)

# Rest of Day : Agenda (lunch provided!)

# Rest of Day : Agenda (lunch provided!)

# Rest of Day : Agenda (lunch provided!)

# Rest of Day : Agenda (lunch provided!)



# Rest of Day : Agenda (lunch provided!)

# Rest of Day : Agenda (lunch provided!)