Single Page Applications

what is a single page application?

"*A web app that loads a single HTML page and dynamically updates that page as the user interacts with the app. SPA's use AJAX and HTML5 to create fluid and responsive Web apps, without constant page reloads. This means much of the work happens on the client side, in JavaScript*".

- MSDN

history of single page applications

The concepts of the single page application go back to as early as 2003, and the term itself is thought to have been originally used circa 2005. The first well known implementation was Gmail, released in 2004. It's big differentiating feature being the use of AJAX to update the status of the app without the full page refreshes being used in Yahoo! Mail and Hotmail.

history of single page application tools

The tools we're familiar with today, Angular, Ember, Backbone, React, were preceded by some of the older libraries, that have been doing this for many years, such as Dojo toolkit, Google Web Toolkit, Prototype & Scriptaculous. They set the table and established patterns for a lot of the tools and frameworks that followed.

single page applictions today

SPA's have become the default standard for many of the things people use on a day-to-day basis. Applications such as Spotify, Twitter, Facebook, AirBnB, Gmail, mapping applications, such as store locators. These are applications that have fast, seamless experiences, but with book-markable url's. The experience in the SPA should mirror the experience a user would get with a document to document page load experience.

building single page applications

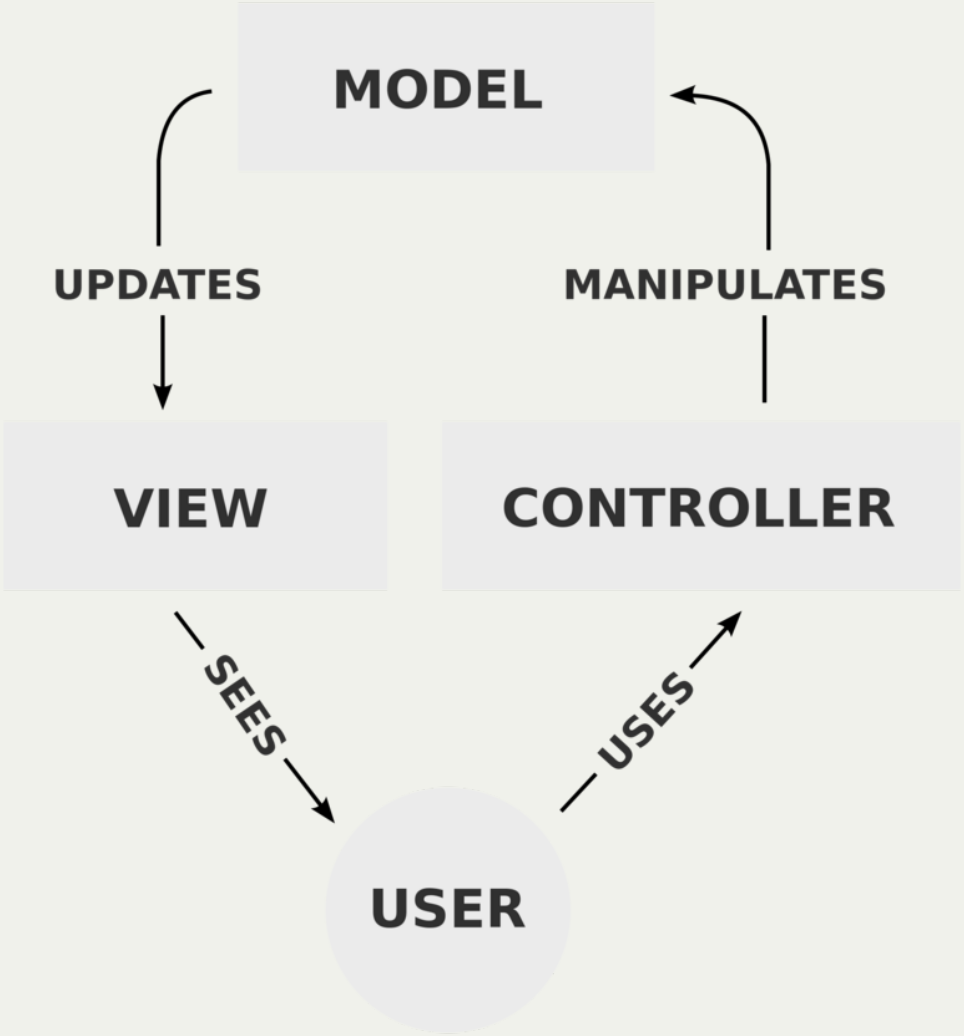
Making fast user experiences, that behave like native applications, is why you want to build single page applications. Avoiding the tight coupling and entangled code that inevitably results in projects where you're only using tools like jQuery is why you want to use more advanced tools and even frameworks.

jQuery specifically ties our code's logic to our DOM structure, and doesn't provide a way to persist data in our client-code: as soon as this executes there is no record of having done this.

```
$(document).ready(function() {

    // attach a click event handler
    $('.my-thing').click(function () {

        // send a message to the server
        $.post('/do-stuff', {
            message: 'Hello world'
        }, function (response) {

            // render the response on the page
            $('.my-thing').html(response);
        });

    });

});
```

jQuery, because it is DOM-centric, encourages you to tie your application logic to the DOM, where you're forced to interact with the DOM any time you want to get access to any of your data

This leads us into talking about some of the object-oriented patterns that you can put in place to counteract these problems, one such pattern being MVC. MVC, or model view controller, is a very commonly used pattern that separates concerns, specifically the representation of data from the user's interaction with it.

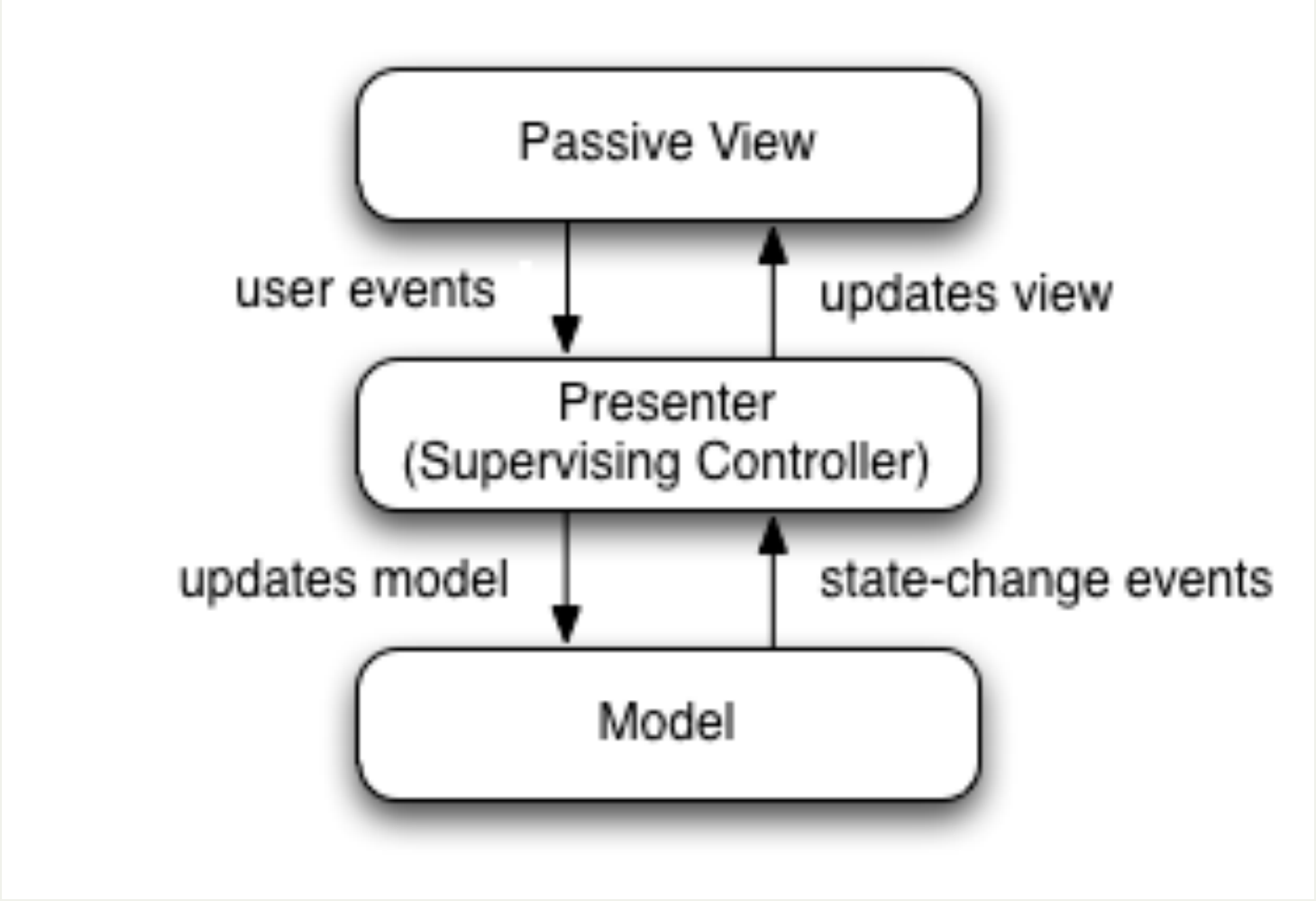

MODEL
UPDATES
MANIPULATES
VIEW
CONTROLLER
SEES
USES
USER

server side MVC

- **Model**: defines and manages the application's data structure. Has no knowledge of any user interface.
- **View**: the user interface. Displays data to the user and captures the interactions.
- **Controller**: communicates between the view and model.
- View rendering is completely separate from the app's business logic.
- Frameworks like Ruby on Rails, Django, .NET MVC

client side MVC - a little bit different

- **Model**: interfaces with a server to get data using AJAX
- **View**: displays model data on the page
- **Controller**: accepts user interaction handles communication between the model and view
- functions more like model-view presenter in the browser where the view layer is more representative of the DOM


Passive View
user events
updates view
Presenter
(Supervising Controller)
updates model
state-change events
Model

We'll be using a library that provides some of these low level components, without giving us a full framework that directs us how to build it.

Backbone.js

What is Backbone.js?

A JavaScript library that provides helpful types for building and organizing rich JavaScript interfaces.

As stated previously, Backbone is not a framework. The common distinction between libraries and frameworks is that frameworks call your code, and you call libraries. Another way to think about it is that frameworks design your application. They specify things like where you put what files, what you can do, and how you do it.

Backbone, as a library, provides help with a number of common user interface features, but it doesn't help with how you design your application. It's feature set is fairly minimal, essentially providing enough abstractions to organize code which previously would have been a messy nest of callbacks and tightly coupled code.

Backbone is described as an MVC (Model, View, Controller) framework, but it isn't. It's also not MVVM (Model, View, ViewModel) Instead it's often described as MV* or MV whatever. In reality, Views in Backbone, which have very little default implementation, function more as controllers, and the templates used to render the UI are more comparable to the traditional notion of View.

from the Backbone documentation:

Backbone.js gives structure to web applications by providing models with key-value binding and custom events, collections with a rich API of enumerable functions, views with declarative event handling, and connects it all to your existing API over a RESTful JSON interface.

Backbone encourages good object-oriented program design, but it still leaves a lot of the work up to you. It does not give you hard and fast opinions about how to use things. This makes Backbone both frustrating for some, and advantageous to others who need the flexibility to address the specific needs of their apps.

Anatomy of a Backbone application:

Backbone involves moving state away from the server and into the client. The client-side application consists of routers, for handling page transitions, views, for rendering models, models, for representing the data and state in your application, and collections, for managing many models.

Anatomy of a Backbone application (cont):

The server's responsibility is to serve the initial application, then provide the RESTful web services for the client-side models. The server and client exchange serialized JSON data over the http protocol.

Anatomy of a Backbone application (cont):

Pros:

fast - there is an initial cost to download the application up front, but this can be cached for subsequent visits. After that, the traffic is just the JSON models and even that happens asynchronously so the user never has to wait for the network.

The user's experience is greatly improved over traditional network applications because they receive immediate feedback in response to their actions, allowing the application to be highly interactive. There's no round-trip involved for every interaction so the perception of speed is much faster.

Anatomy of a Backbone application (cont):

Cons:

The application cannot be indexed easily by search engines (without extra work.) Google-bot has gotten more advanced in terms of executing scripts, but all of the routes your application may support won't necessarily be discoverable by the bot.

Client-side applications are relatively difficult to test. Anything that interacts with the DOM requires operating in the browser. Breaking the complex interactions down into discrete units is challenging.

Client-side code has some security implications. Users of your application have your source code and they can change it and execute it. Security must still be enforced by the server.

Why Backbone? (or any other framework or library)

Building a client-side application is challenging without a framework or library. Often it degenerates into a mess of nested callbacks and event handlers. While frameworks provide solutions to this, Backbone retains all the flexibility of writing pure JavaScript, without the sizable footprint of other frameworks. It shines when used with a progressive enhancement development approach.

What Backbone provides:

Models: models represent the data required by your application. They hold your data and raise events when the data changes.

Collections: collections group models. They also forward events from the models they contain as well as raise events of their own. Collections depend on models

Views: views handle events from models and collections. They are responsible for rendering any markup. Views are the only Backbone component that interacts with the DOM, so views can also handle events from the DOM.

Routers: Backbone routers are used to simulate page changes, and to provide support for page history and bookmarking. While a router can depend on views, it can optionally also depend on models and collections.

Backbone dependencies:

underscore.js or lodash.js

jQuery/zepto (optional as of 1.2.0)

a minimal Backbone.js environment:

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>Backbone boilerplate</title>
</head>
<body>
    <!-- these script src values presume hosting the files locally -->
    <script src="jquery.js"></script>
    <script src="lodash.js"></script>
    <script src="backbone.js"></script>
</body>
</html>
```

Backbone models don't store their data directly on the model, which is why we use methods like 'get':

```
var book = new Backbone.Model({ title: 'White Tiger', author: 'Aravind Adiga'

// to retrieve a value set on the model
book.get('title');
```

Accessing all of a model's data:

```
var book = new Backbone.Model({ title: 'White Tiger', author: 'Aravind Adiga'

// to see all the properties stored on the model
book.toJSON();

// or alternatively
book.attributes
```

Backbone models:

Backbone is a collection of loosely coupled components, so it is possible to build an application without using Backbone models, but you probably won't.

The purpose of models:

Models form the core of your application. They contain your application's state as well as logic and behavior. Models are the single point of truth for data. Even though JavaScript already has built-in objects for storing data, Backbone builds on top of that by providing models a lifecycle: objects are created, they go through a series of changes, they are validated, and synchronized to some data source.

The purpose of models (cont):

They communicate changes to the rest of the application via events. Through this mechanism, changes to a model can ripple throughout the interface without any direct coupling or redundancy.

Defining new model types:

We can create new models directly from the Backbone.Model constructor, but most of the time you'll want to define your own model types, which is done by extending Backbone.Model.extend({});

Defining new model types:

```
/*
 * In this example, User is a new model type that inherits from Backbone.Model
 * User is capitalized because this is a convention we use for constructors. U
 * is now a constructor for User model instances.
 */
var User = Backbone.Model.extend({});
```

The extend method:

```
/*
 * the extend method is a helper function shared by Model, Collection, Router,
 * and View that establishes an inheritance relationship between two objects.
 * A similar function is found in underscore and jQuery
 */
var User = Backbone.Model.extend({
    role: 'student'
});

var user1 = new User();
var user2 = new User();
user1.role = 'instructor';

console.log('user1: ', user1.role);
console.log('user2: ', user2.role);
```

The extend method (cont):

```
/*
 * passing a second argument to extend creates a "class"
 * property or static property on the Constructor object itself
 */
var ExtendedModel = Backbone.Model.extend({}, {
    summary: function () {
        return 'this is a user';
    }
});

console.log(ExtendedModel.summary());
```

Backbone model types are JavaScript constructor functions, therefore to create a model instance, call its constructor with the new operator.

```
var model = new Backbone.Model();
```

Usually you'll be instantiating custom types:

```
// define the new type constructor first by extending Backbone.Model
var User = Backbone.Model.extend({});
var user = new User();
```

It's common to provide a model's data as an argument to the constructor:

```
// setting the name and age properties for the newly created model instance
var model = new Backbone.Model({
    name: 'James',
    age: 46
});
```

Initialize: if a model has an 'initialize' method defined, it will be called when the model is instantiated.

```
// implement the initialize method
var User = Backbone.Model.extend({
    initialize: function () {
        console.log('user created');
    }
});
var user = new User();
```

Models can inherit from other models. We can further specialize by extending any model type:

```
// define a custom type
var User = Backbone.Model.extend({
    initialize: function () {
        console.log('user created');
    },
    asString: function () {
        return JSON.stringify(this.toJSON());
    }
});
var user = new User({
    name: 'James',
    age: 46
});
console.log(user.asString());

// then extend the custom type
var Student = User.extend({});

var student = new Student({
    status: 'busy'
});

console.log(student.asString());
```

Check types after this inheritance:

```
console.log(student instanceof Student);
console.log(student instanceof User);
console.log(student instanceof Backbone.Model);

// just to prove the inheritance behaves as expected
console.log(user instanceof Student);
```

Attributes can be set by passing an object to a model type's constructor, or by using the 'set' method.

```
// setting attributes using the set method
var student = new Student();
student.set('name', 'Abby');

// or set many attributes at once
student.set({
    name: 'Jonathan',
    age: 34
});
```

Attributes are accessed by using the 'get' method:

```
// accessing attributes with 'get'
student.get('name');
```

Escaping html content when accessing properties: useful for preventing cross-site scripting attacks

```
// setting a model with some potentially malicious markup
student.set('description', '<script>alert("script injection")</script>');
student.escape('description');
```

example of how this attack vector works:

```
// setting a model with some potentially malicious markup
student.set('description', '<script>alert("script injection")</script>');
$('body').append(student.get('description'));

// instead use escape when accessing content not set by your application
$('body').append(student.escape('description'))'
```

Testing for an attribute:

```
// use 'has' to test if an attribute has been defined
var student = new User();
student.set('type', 'first-year');
student.has('type'); // true
student.has('year'); // false
```

Model Events:

Models raise events when their state changes. This is one of the most valuable features of Backbone models. Events are the reason get and set methods are required when working with a model's attributes. By wrapping attributes in get and set, Backbone has a chance to detect changes and raise events.

To detect a change to a model, listen for the 'change' event:

```
// listening for the 'change' event
user.on('change', function () {});
```

The second parameter is a function to execute when the event is triggered. It's also possible to listen to changes to a single property:

```
/*
 * event namespacing is a convention that should be used
 * when you need to organize your own custom events
 */
user.on('change:type', function () {});
```

event handling example:

```
// setting up a change handler
var person = new Backbone.Model({
    name: 'James',
    age: 45
});
person.on('change', function () {
    console.log('something changed');
});
```

triggering the event:

```
// to trigger the change event, change the data on the model
person.set('name', 'Roger');
```

triggering the event:

```
// to listen for a change event on a specific property
person.on('change:name', function () {
    console.log('name changed');
});

// trigger the appropriate change event
person.set('age', 46); // nothing happens
person.set('name', 'Skip');
// name changed
// something changed
```

Custom model events

It's possible to define, trigger, and observe custom model events. Events are identified by string identifiers. Use the 'on' method to bind to an event. The second argument to the on method is a callback that will be invoked when the event is triggered. Use the trigger method to trigger and event. Subsequent arguments to the trigger method will be forwarded on to the event handler.

custom event example:

```
// an example of extending a plain JavaScript object with Backbone.Events
var volcano = _.extend({}, Backbone.Events);
volcano.on('disaster:eruption', function () {
    console.log('duck and cover');
});

// triggering the event
volcano.trigger('disaster: eruption');

// passing data to the handler:
volcano.on('disaster:eruption', function (options) {
    console.log('duck, cover, and ', options.plan);
});
volcano.trigger('disaster:eruption', { plan: 'run' });
```

To stop handling events, use the off method

```
// after calling off for this event, nothing will happen when it triggers
volcano.off('disaster:eruption');
volcano.trigger('disaster:eruption', { plan: 'run' });
```

Model Identity

When you have a lot of objects, you need a way to distinguish between them. Backbone will do this using an id property. The 'id' property represents the model' persistent identity. It is undefined until the model has been saved.

checking the id for a new model instance:

```
var user = new User();
user.id; // undefined
```

When the model is saved the id property is set to the server's identity for the model object. The 'cid' property is a termporary identifier used until a model is assigned it's 'id'. It is assigned as soon as the model is created.

```
var user = new User();
user.cid; //c1
```

The cid property is no longer required once a model is saved and acquires an id property. Model objects have an isNew() method that can be used to test if the model has been saved to the server and has an id property.

```
// checking if a model has been saved and assigned an id
var newUser = new User();
console.log(newUser.isNew()); // true
```

Defaults

When you define a new Backbone Model, you have the opportunity to define default attributes. The 'defaults' property specifies default values for attributes that are not set in the constructor.

defining defaults on a model:

```
// default values on a Student model
var Student = Backbone.Model.extend({
    defaults: {
        enrolled: true,
        type: 'first-year'
    }
});

var student = new Student();
student.get('enrolled'); // true
student.get('type'); // 'first-year'
```

Validation

Backbone exposes model validity through two methods:

`validate`

tests the validity of the model and returns any errors found

`isValid`

returns a boolean indicating the current validity of the model using the validate method

Validate is called by Backbone during set, if { validate: true } is also passed, or save operations. Save is a method used to persist a model. If the model is found to be invalid, the operation is cancelled and an error is triggered on the model.

To add validation to a model, we provide an implementation of the validate method:

```
// add an implementation of the validate method to the model definition
var User = Backbone.Model.extend({

    // validate function takes the model attributes as an argument
    validate: function (attrs) {
        var validDays = ['Monday', 'Tuesday', 'Wednesday', 'Thursday'];
        var validAppointment = false;

        if (attrs.appointment && !_.contains(validDays, attrs.appointment)) {
            return 'Invalid appointment scheduled: only allowed on ' +
                    validDays.join(', ');
        }
    }
});

var user = new User();
user.on('invalid', function (model, error) {
    console.log(error);
});
```

Triggering the validation on set:

```
// to trigger validation on set, pass { validate: true } as the final paramete
user.set('appointment', 'Monday', { validate: true }); // validation passes
user.set('appointment', 'Saturday', { validate: true }); // invalid event fire

// check that the operation was cancelled
user.get('appointment');
```

toJSON

The toJSON method converts a model's attributes to a JavaScript object. This method can be a little confusing because you may expect it to return a string representation of the model's attributes, but it doesn't. It simply returns a copy of the model's attributes object. This method is useful as a first step in serializing a model or when you just want to inspect a model's state.

The object returned from toJSON can be passed to JSON.stringify to get a string representation of a model:

```
var User = Backbone.Model.extend({});
var user = new User();
user.set('name', 'James');
var attrs = user.toJSON();
console.log(attrs);

// compare this to logging the model instance itself:
console.log(user);

// to get a string representation of the model:
console.log(JSON.stringify(user.toJSON()));
```

save, fetch, destroy

Models have save, fetch, and destroy methods for synchronizing with the server.

Save performs insert and update operations depending on the state of the model. If the model is new, and has never been saved, to the server before, then Backbone will perform a save operation using **POST**. If the model has previously been inserted and has an id property, then Backbone will perform an update using **PUT**.

Fetch updates the model with the current server-side state using **GET**.

Destroy uses a **DELETE** operation to delete the model from the server.