Octane

A Lightweight App Development Framework

Why Octane

Octane was developed to provide a stepping stone into the world of frontend MVC (model-view-controller) frameworks and two-way data binding, without requiring developers to learn extensive and API-specific jargon. While there are methods unique to Octane, the framework was built to keep those to a minimum, while at the same time using generic and descriptive language that should give the developer a good idea what a method does simply by its name. For the most part, Octane strives to stay out of the way, allowing you to access its model and controller structures when you need them, and write code like you're used to everywhere else

Be aware: you'll probably write a little more code with Octane than with some other frameworks, but that's part of the idea. We don't obfuscate everything to the point it feels like magic when you use Octane's models and methods. We want developers to still write javascript the way they want to, with the freedom to lean on Octane but not rely on it. At the same time, Octane's open ended-interface allows developers to add modules as they see fit, incorporating the core model/view/controller/view-model system to accomplish their end goals. Ultimately, you're always writing your applications *your way*, not the Octane way.

Overview

At its core, Octane is a model/view/controller/view-model (MVCVM) framework. The premise of any MV* framework is to compartmentalize different roles of an application code. Each component of an MV* have a specialized role, much like HTML, CSS, and javascript have separate functions on the DOM. We'll break down each of Octane's **model**, **view**, **controller**, and **view-model** components in more detail further on, but here's a brief overview for those new to MVC/MVCVM or who want to learn how Octane implements a the MV* design pattern.

To get a good grasp on MV* and other Javascript design patterns, check out the free e-book Learning Javascript Design Patterns by Addy Osmani.

The Octane Circuit

Octane's components are held together by the Circuit, a continuous loop of state monitoring, comparing, and updating that's triggered with any change to a data key bound between the **View** (DOM layer) and the **Model**. This connection between the View and the Model is known as two-way data binding, and it's the most powerful weapon in an MV*'s arsenal. In Octane, this loop can be triggered in two ways, each from opposite ends of the Circuit: one is in the **View Model**, whenever bound data is changed in the DOM by the user. This starts the Circuit's uptake channel, where data makes its way to the Model via the **Controller** and its methods. The other trigger is inside the Model itself, fired when its data is altered. This initiates the Circuit's outflow channel as the updated data makes its way back to the DOM for the user to see.

Models

Traditionally, a Model holds the data users see in your application. Static content is still held in HTML, but content that changes in the view depending on user interaction should be held in a Model. It provides a centralized location for all data related to your application, or to a certain part of your application.

Octane's Models are unique in that they are partitioned into REST-style resource data that you set when you initiate the Model, and active data that changes depending on the user's interaction with your application. Resource data can be accessed throughout your application in response to changes on input fields, select boxes, etc, but it cannot be edited. Active data, on the other hand, is entirely fluid and changes as users progress through the flow of the application. The Resource data is by its nature stateless, while the active data is entirely mutable and comprises the state of your application. Throughout this documentation, **active data** and **state data** may be used interchangeably to describe this dynamic data object in the Model.

using Models

View Models

A View Model in Octane is actually initialized in tandem with each Model instance, but its purpose is different enough to warrant its own explanation. The View Model has one job and one job only: to keep the data in the view in sync with the Model's state at all times. This means when a user enters or selects information in the UI, the Model is updated. Conversely, if user-entered data is manipulated by the Controller in any way or there's a change to any data point in the Model's scope, the view reflects that newly altered data syncronously.

You won't interact with the View Model directly. It serves as an interface layer, performing its job silently in cooperation to your Controllers, Models, and Views.

View Models

Views

Views are the presentation layer, written in HTML and styled by CSS. In classic MV* implementaions, a view has no logic within it. In Octane, this is mostly true, however Octane creates is a **View Object** for every DOM element you create with the `<o-view>` tag.

The View Object is has a very few methods, mostly to deal with animating itself in and out of the viewport. While Octane's Router module handles the logic for tasks like loading, unloading, and updating the history, the View object holds information about how each particular view should adhere to those tasks. Think of the View as the HTML, and the View Object as reference to the HTML stored in the Router.

writing Views

Controllers

A Controller is the logic center of an MV* application. Whereas a Model holds application data, the Controller decides how that data is presented to the user. In Octane, the Controller also determines how data a user enters or chooses is related to other data. For example, a user's input in the application can trigger changes in the Model state data beyond it's own key. Or dragging a slider on one part of the screen could prompt a change in another part, without the two being explicitly tied to one another in the traditional manner. The funtionality implemented inside the controller is completely up to the developer, but we'll go into greater detail on where certain types of functionality fit into the uptake and outflow channels of the Circuit.

using Controllers

Octane API

.trip (element)

- @param `element` [HTMLDocumentElement]

Artificially start Octane's uptake channel by dispatching an event from the element, if the element has an `o-bind` attribute defined. Use in your code when you need to trigger the uptake circuit programmatically.

Try it

Favorite Movie Jurassic Park

Titanic-ize

HTML

```
    <label>Favorite Movie</label>
    <input id="input1" o-bind="myMovies.favorite"/>
    <span o-update="myMovies.favorite">Jurassic Park</span>
```

Javascript

```
    var
    input1 = document.getElementById('input1'),
    myModel = octane.model('myMovies');

    input1.value = 'Titanic';
    octane.trigger(input1);

    myModel.get('favorite'); // 'Titanic'
```

.goose (model,dirty)

- @param `model` [string] the model to update with your data
- @param `dirty` [object] the data to update the model with

Similar to trip, but doesn't fire an event on an o-binded element to trigger the Circuit's uptake channel.

.model (name[,config])

- @param `name` [string] name for your model`
- @param `config` [object] `{ default:[object], db:[object] }`

Creates an Octane model instance, or return the named model. Pass the model's default state data with the optional config's `default` property. Pass the model's stateless resource data as config's `db` property. Passing config params to an already instanced model will be ignored.

```
    var
    startsWith = {
        favorite : 'Jurassic Park',
        genre    :   'awesome'
    },
    getMovies = function(){
        // run some async code
    },
    movies = getMovies();

    movies.then(function(result){
        octane.model('myMovies',{
            default : startsWith,
            db      : result
        });
    });
```

model instance methods

.controller(model)

- @param `model` [string] model name your controller interacts with

Creates an Octane controller instance, or return a controller by the model name passed.

```
octane.controller('myMovies');
```

controller methods

Modules

Octane Your Way

Modules are how you'll primarily interact with Octane. While you *could* cook up all sorts of spaghetti on the global scope, it will be a pain to debug, your team will hate you, and your mother won't invite you for Christmas anymore. Solution? Use a module.

```
/* simple module */

    octane.module('myModule',function(config){
        // your excellent code
    });
```

Octane modules are encouraged to be written in the **revealing module pattern**, discussed here by Javascript pioneer Douglas Crockford. A basic implementation:

```
/* module implementing revealing module pattern */

    octane.module('myModule',function(config){

        var private : 'private property';

        function getPrivate(){
            console.log(private);
        }

        this.publicMethod = getPrivate;
    });

octane.myModule.publicMethod(); // "private property"
octane.myModule.getPrivate(); // Typeerror
octane.myModule.private // undefined
```

Alternatively, using .extend()

```
octane.module('myModule',function(config){

    var private : 'private property';

    function getPrivate(){
        console.log(private);
    }

    this.extend({
        publicMethod : getPrivate
    });
});
```

Writing the module in this way allows you to present a public-facing API that is uncluttered by helper methods and variables. It also helps other developers using/extending your module from inadvertently deleting or overwriting a necessary part of your code.

As well as creating your own methods, Octane modules inherit the .model and .controller methods from the octane object itself. Why not just use `octane.model()` or `octane.controller()` ? Binding your model and controllers to `this` inside your module namespaces them, which is effective as your application grows in size.

```
/* module implementing .model and .controller */

    octane.model('notNamspaced');

    octane.module('myModule',function(config){

        var
        $model = this.model('namespaced'),
        $cntrl = this.controller('namespace');

        $cntrl.extend({
            //some new methods
        });
    });

octane.model('namespaced')      // > Model{name:"namespaced",c
ontext:"myModule module",...}
octane.model('notNamespaced')   // > Model{name:"notNamespaced
",context:"Application",...}
```

Defining Module Dependencies

Any module or library that your module depends on must be listed as a dependency upon declaration, which ensures they will be loaded in the right order when Octane initializes.

```
/* dependent module */

    octane.module(
        'dependentModule',{
            modules: ['dependency1','dependency2'],
            libraries: 'some library'
        },
        function(config){
            // your code
    });
```

Notice the loose syntax for passing dependencies. If you have only one dependency, you may declare it as a string. Multiple dependencies should be declared as an array.

Note: Octane's module dependency manager is strictly for modules and libraries written in the context of the Octane application using `octane.module()` and `octane.addLibrary()`. If your module is dependent on other libraries (ex. jQuery Mobile), add those to your project's bower.json, load it via AMD/Require.js, or include it via script tag ahead of octane.js. * **The AMD pattern will likely be implemented in a subsequent version of Octane.**