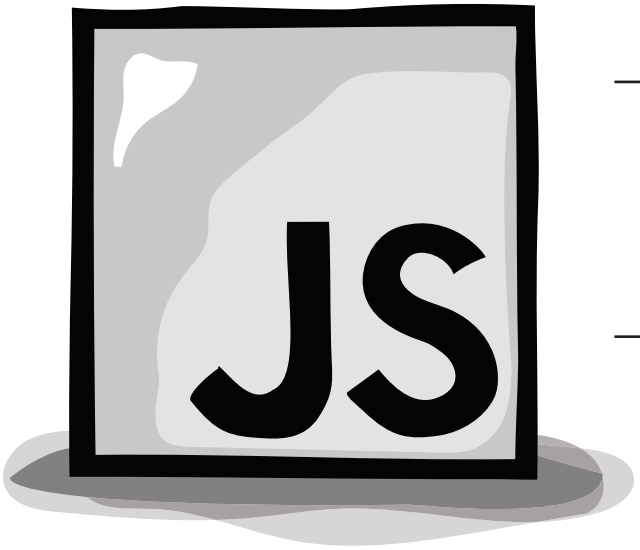


GYMNASIUM

JAVASCRIPT FOUNDATIONS

Lesson 6

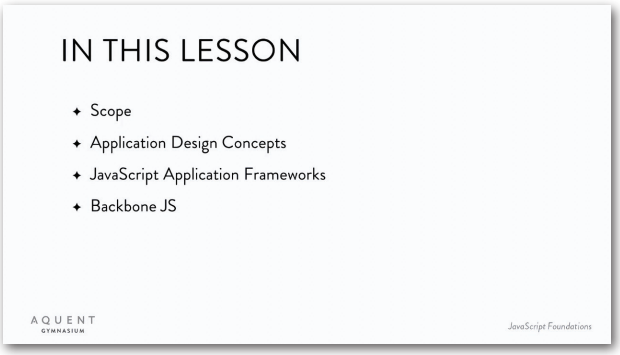
Advanced Concepts

ABOUT THIS DOCUMENT

This handout is an edited transcript of the JavaScript Foundations lecture videos. There's nothing in this handout that isn't also in the videos, and vice versa. Some students work better with written material than by watching videos alone, so we're offering this handout to you as an optional, helpful resource. Some elements of the instruction, like live coding, can't be recreated in a document like this one. We encourage you to use this handout alongside the videos, rather than as a replacement of them.

INTRODUCTION

Well, here we are at the last lesson of the course. I wanted to wrap up by covering the subject of scope. Then, go into some basic concepts of application design. And finally, wrap up by looking at some application frameworks, and doing a bit of hands-on work with one of those frameworks, Backbone JS.



First, let's talk about scope. This is something we covered very briefly in the early part of the course. But, there's a lot more to be said about it. To refresh your memory, scope is the part of the code where a variable is visible, and accessible. I talked about scope, when I mentioned local variables.

Here, you can see that we're declaring a local variable, message, inside the sayHello function. If we try to access the message variable outside of that function, either before, or after it, message does not exist. It only exists inside the function, when that function is running.

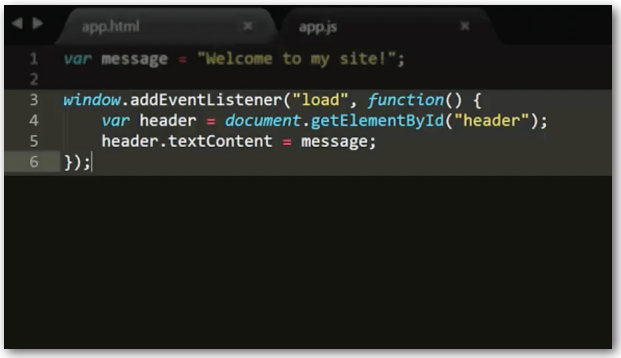
This may seem like a minor, or somewhat interesting point, but it winds up being one of the most powerful, and important features of JavaScript. It's especially important when you start working on larger projects that have different developers, or whole different teams of developers working on them.

If you don't pay attention to scope, you run the risk of your code interfering with, or being interfered by, someone else's code. This can cause all kinds of crazy bugs that are tough to track down, because the bug may not be in your code, but in someone else's. So, keep all that in mind, while we take off in a slightly different direction for a moment.

Now, what about variables that are not local variables? What is their scope? To answer that question, I've set up a sample application. There's an HTML file called app.html that loads a JavaScript file called app.js. The HTML also defines an empty H1 tag with an ID of header. You should download these files from the course's website, and follow along on your own.

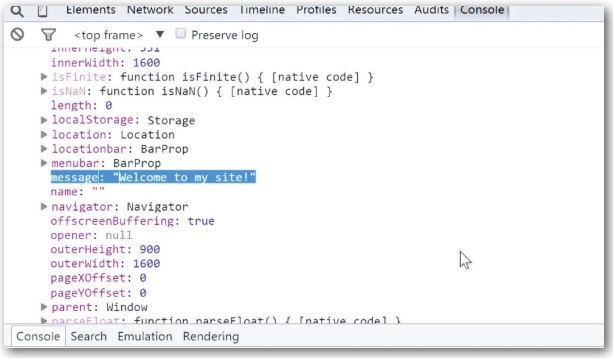
In app js, I'm defining message variable, not in any function. Then, I listen for the window load event, so, I know that the H1 tag is available. When it is, I set that H1 tag's text content to the message variable. We can

run that, and see that it creates the header, with the message. So, my question is, what is the scope of that message variable? Obviously, it's available anywhere in the app.js script file. But, is that all?



Back in the browser, let's open the console. Remember when we were first talking about the DOM, or document object model? We looked at the window object. We saw that, not only was that the top level of the DOM but it also contained any JavaScript objects, functions, and values that were needed, in order for JavaScript to run. Here, we could see all kinds of things that are accessible by any JavaScript program: CSS stuff, HTML5 stuff, math, media, string, text, WebGL, et cetera.

But, if we go down far enough, look at that. It's our message variable. It's now a property of the window object. Now, remember that there's so much stuff on the window object. And, it's also important that you can access any of it without typing window. Anything on the window object is what we call, global scope.



So, those are your two scope choices. If a variable's in a function, it's a local variable, only available from within that function. If it's not in a function, then it's global, attached to the window object and available from anywhere. That, available from anywhere part, is important, because it means that a global variable is even accessible from other scripts running on the same page. This may seem like a plus point at first, and indeed, it is in many ways.

This is how many, if not most, JavaScript libraries work. For example, when you use jQuery, the jQuery file is loaded, and creates a jQuery variable, as well as, the shortcut dollar sign variable in the global scope. Now, jQuery's available to use from any script on that page, likewise with Mustache.js. It creates a Mustache object in the global scope. So, you can just call mustache.render from any script.

But, this global behavior can cause some problems, as well. Say, my app was getting some decent traffic, and I decided to put some ads on it. An ad company provides some code that I can add to my site, that will serve up ads on my page. All I need to do is add a script tag that will load this ad.js to my HTML page. And, this is going to create a div with the ad text in it, and add that div to the page.

app.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Scope</title>
    <script type="text/javascript" src="app.js"></script>
    <script type="text/javascript" src="ad.js"></script>
</head>
<body>
    <h1 id="header"/>
</body>
</html>
```

So, we run that and, okay, the ad is there, but wait, my header's all messed up, now. My header has been turned into an ad. But, I didn't touch any of my app js code. So, how could that possibly happen? We better look at that ad code. So, here's the problem. This code also defines a message variable that it uses to create the ad.

ad.js

```
var message = "Buy Bob's Widgets";


window.addEventListener("load", function() {
    var div = document.createElement("div");
    div.style.width = "200px";
    div.style.backgroundColor = "#ffffcc";
    div.style.fontSize = "20px";
    div.style.border = "1px solid black";
    div.style.position = "absolute";
    div.style.right = 0;
    div.style.top = 0;
    div.innerHTML = "<em>Advertisement</em><br/>" + message;

    document.body.appendChild(div);
});
```

What happens is, our app code loads, defines our message, and then, waits for the window load event. But, before that window load event happens, the ad code loads, and also defines a message variable. Because both these message variables are in the global scope, the ad's message simply overwrites the original value of my message. When the window.load event fires, it uses the current value of message to set the header text content.

app.js

```
var message = "Welcome to my site!";

window.addEventListener("load", function() {
    var header = document.getElementById("header");
    header.textContent = message;
});
```

Now, you see the problem with global scope. On a large commercial site, there could be dozens of scripts running on a single page: application code, ads, analytics, video players, slideshows shopping carts, and many, many others. Most of these will have been created by different developers in different groups, likely in different companies, altogether. So, there's often no possible coordination between these teams. No way to say, hey, I'm going to use the global variable called, message in my code. So, don't touch that.

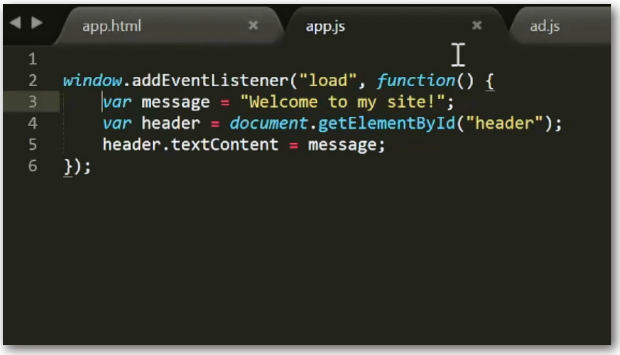
So, what's the solution? Simple solution is don't ever use global variables, unless you're specifically creating something that you want other scripts to be able to use, and then, limit it to a single, very unique variable. It's unlikely that anyone else has defined Mustache, or jQuery as global variables.

But, in fact, other libraries do use the dollar sign. And, jQuery had to come up with a special, no conflict mode to allow for the alternate use of the dollar sign symbol, without causing any problems. This doesn't only apply to variables, either. If you create a named function like, do stuff, this function will be created on the window object as window.dostuff, and could potentially be interfered with by another script, creating a global variable, or a function with the same name.

Creating properties in the global scope is called, polluting the global namespace. Here, namespace basically means the same thing as scope. This is generally a bad thing and will make other developers want to do bad things to you, so avoid it. Of course, if you're a single developer creating your own site or page, this doesn't necessarily apply. But, as soon as you get into other code running on the page, you could be in trouble.

So, if you want to avoid global scope, the only alternative is to use local scope. You simply make sure that any code you run is wrapped inside a function, and all the variables, and functions are created within that function. Depending on how you're structuring your code, you might already be doing this.

For example, if you're using jQuery's ready method, like so, then, you're pretty much all set. Any variables, or inner functions you create will have local scope. The same if you're using the window load event, or the DOM content loaded event. So, in the case of our application, and add problem, we could simply move this message variable inside the window load handler.

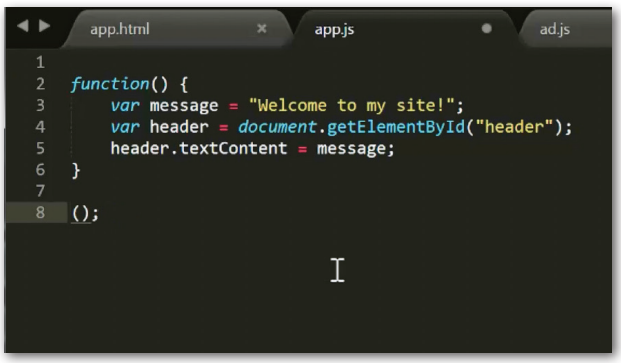


Whoever wrote that ad code should do the same thing. But, you often don't have control over other people's code. So, the best you can do is to make sure your own code is bulletproof. Now, that message is inside the load handler function, it's a local variable, not a global one. So, even though the ad code is still polluting the global namespace, it's not affecting us, and our header is back to normal.

But, what if you weren't using any of the ready, or load handling functions, but had just put your script at the end of the body tag, like we've done with most of the examples in the course, up to now? I'll just move the scripts down here, and now, I'll remove the window load handling function.

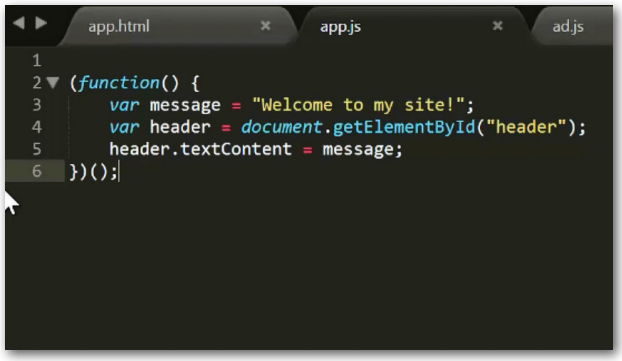
Now, we're back with everything in global scope. To get this back into local scope, we just need to wrap this in a function. I'll create a function called, in it, and then, I'll call that in it function. So, everything is local now, right? Well, not everything. This in it function we just created is now in the global scope. Oops. What if someone else creates an in it function there, too?

What we need to do is create an anonymous function. An anonymous function, as you may recall, is simply a function without a name. Because it has no name, it won't be in the global namespace. So, we just remove the name, in it, right? But, if the function has no name, how do you call it? The answer is called, an immediately invoked function expression, also abbreviated IIFE.



That's a very big name for a very simple concept. You simply take an anonymous function, and wrap it in parentheses. Then, you can just call that function, like any other function, by putting the open and close parentheses, right after it. So, it's a function that gets immediately invoked, or called. And, because it's anonymous with no name, it never goes into the global scope.

So, let's try that in our app. We'll just wrap this code in an anonymous function, and then, wrap the function in parentheses. And, then, call that anonymous function by putting a pair of parentheses after it. Now, the problem is handled with a 100 percent pollution-free solution. Our code runs, and everything is local to that anonymous function.



Now, obviously, this point has been ignored in the course, up until now, as we're doing simple stand-alone examples. But, once you start working on a larger project, interacting with other developers and teams, and using third party code, this point will become a vital necessity. Next up, we'll begin to look at some of the key concepts of application architecture, starting with software design patterns.

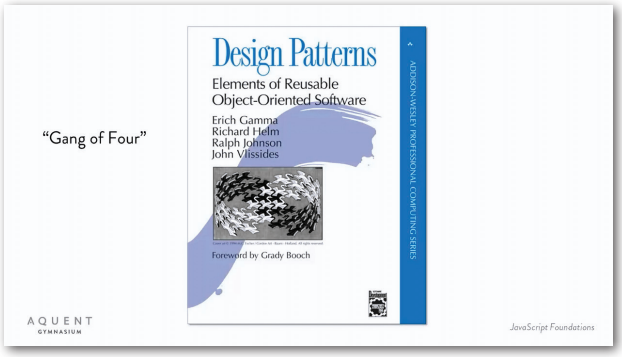
SOFTWARE DESIGN PATTERNS

In a couple of the larger projects in this course, I've acknowledged that it can be tough to know where to start, when facing a blank page, and the concept of an application you want to build. I gave a few small tips on how to get started. But, when faced with a more complex application, you want to get a bit more organized, and plan out the different parts of your app, and what each piece will do. This whole subject of how to structure an application is known as, software architecture.

Now, not surprisingly, during the history of computer programming, developers have noticed that they're solving some of the same basic problems over, and over, again. Often these issues come up on completely

different platforms, in different languages. But, they essentially boil down to the same basic problems. And, the solutions that developers come up with for these problems wind up being very similar, in spite of the differences in language, and platform.

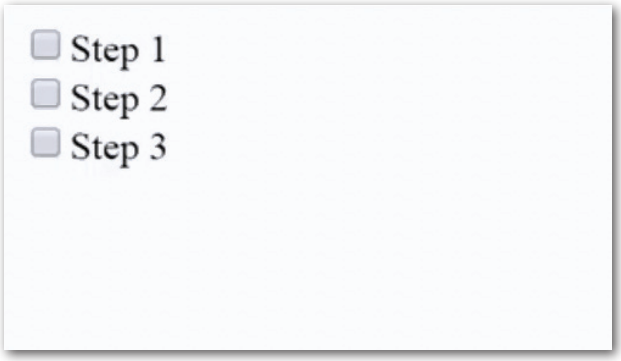
These timeworn, developer-proven solutions have come to be known as, software design patterns. Software design patterns were popularized in a book called, Design Patterns: Elements of Reusable Object-Oriented Software, written by four authors, who came to be known as, the Gang of Four. You'll sometimes hear this book being referred to as, the Gang of Four Book, or even abbreviated GOF.

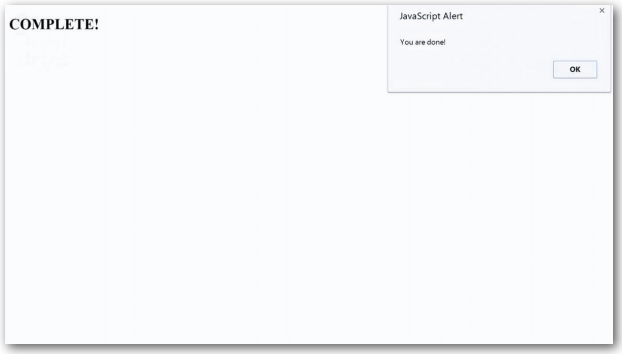


The book catalogued 23 design patterns. But, many other patterns have been described, and popularized, since then. There are even several books on design patterns, specifically written for JavaScript. I'll include some links to them in this lesson's materials.

What I'd like to do in this chapter is take a look at a single, commonly used pattern, the problem that it is designed to solve, and take a crack at coding, and implementation of that pattern in JavaScript. Then, we'll look at some already existing implementations of the same pattern.

The pattern we'll look at is called, the observer pattern. It's purpose is to allow certain objects to observe changes in another object, and react somehow, when those changes occur. To demonstrate this in action, we'll make this simple application. It has three checkboxes. When all three are checked, that triggers a change, and a few different functions are then called in reaction to that change; removing the checkboxes; adding some other text to the page; and calling up an alert box.

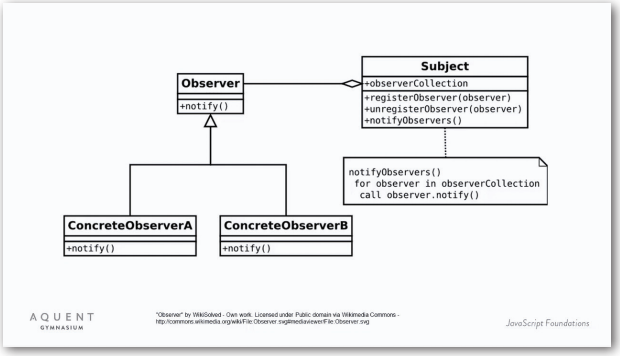




Now, here's a Wikipedia page that describes the observer pattern. We can see that it says that there's an object called, the subject that has a list of observers, and it notifies those observers of any changes by calling a method on those observers. Don't worry, that'll become clearer as we work through this.

Here, we see a diagram of the pattern. This is called, a UML diagram. UML stands for unified modeling language, which is just a way of visually representing how objects are made up, and how they relate to each other in a program. We can see that the subject has a property called, observer collection. A collection is a

generic word for any kind of list. In JavaScript that would be an array.



It also has a few methods. The registerObserver method is used to add an observer to the observer list. And, the unregisterObserver would remove an observer from the list. The notifyObserver method would loop through all the observers in the list, and call notify on each one of them.

On the left, we can see that the only requirement for an observer is that it has a notify method. Well, that's not too complicated. Let's see if we can create an example of this in JavaScript.

I've already created the HTML file. It has a div containing three checkboxes, and a script tag. What I want to do is create a subject that will keep track of the state of the checkboxes, and a few observer objects. When all three checkboxes are checked, the subject will notify its observers of this fact by calling their notify methods. For simplicity's sake, I'll just write the code directly, here. But, remember, that in a live app, you'd probably want to use an IIFE, or some other function to prevent putting stuff into the global namespace.

observer.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Observer Demo</title>
</head>
<body>

<div id="form">
    <input type="checkbox" id="cb1">Step 1</input><br/>
    <input type="checkbox" id="cb2">Step 2</input><br/>
    <input type="checkbox" id="cb3">Step 3</input><br/>
</div>

<script type="text/javascript" src="observer.js"></script>
</body>
</html>
```

First, we'll create an object called, subject. We'll follow that UML diagram, and create an array called, observers, then a method called, registerObserver. This will get a parameter called observer, which we can just push onto the array.

Next, an unregisterObserver method. This isn't so important for the demo, here. So, I'll leave that to you, to implement on your own. Finally, the notifyObservers method will loop through the observers array. For each observer it calls, notify. All pretty straightforward, so far.

observer.js

```
var subject = {
    observers: [],

    registerObserver: function(observer) {
        this.observers.push(observer);
    },

    unregisterObserver: function(observer) {
        //  you can do this!
    },

    notifyObservers: function() {
        for(var i = 0; i < this.observers.length; i++) {
            this.observers[i].notify();
        }
    }
};
```

But, now, we need to deal with the checkboxes, to know when to notify the observers. There are lots of ways to do this. But, I'll keep it simple.

I'll add a listener for the change event of each checkbox, and assign a function called, onCheckboxChanged. In that, I'll check if checkbox one is checked, and checkbox two is checked, and checkbox three is checked. That double ampersand, as you should recall, is the logical 'and' operator, which can combine multiple Boolean values. Here, if all three checkboxes are checked, then, we call subject.notifyObservers.

Now, ideally, all of this logic would probably be part of the subject, itself. But, I've left it out here in the open, just to leave the core functionality of the observer pattern in the subject.

```
        for(var i = 0; i < this.observers.length; i++) {
            this.observers[i].notify();
        }
    }

};


document.getElementById("cb1").addEventListener("change", onCheckboxChanged);
document.getElementById("cb2").addEventListener("change", onCheckboxChanged);
document.getElementById("cb3").addEventListener("change", onCheckboxChanged);


function onCheckboxChanged(event) {
    if(document.getElementById("cb1").checked &&
        document.getElementById("cb2").checked &&
        document.getElementById("cb3").checked) {
        subject.notifyObservers();
    }
}

```

Now, let's create an observer. This is simply an object with a notify method. We'll call it observer1. Now, in that notify method, let's find the div that's holding the checkboxes, and remove it from the document body.

Now, we register this observer with the subject, and we're ready to run this. We check one checkbox, then another, uncheck the first, check the third, and finally, check all three of them. And, bang, the form is gone. Perfect.

```
function onCheckboxChanged(event) {
    if(document.getElementById("cb1").checked &&
        document.getElementById("cb2").checked &&
        document.getElementById("cb3").checked) {
        subject.notifyObservers();
    }
}

var observer1 = {
    notify: function() {
        var form = document.getElementById("form");
        document.body.removeChild(form);
    }
};

subject.registerObserver(observer1);
```

So, let's create another observer. In this one, we'll create an h2 element, fill it with some text that says, COMPLETE! and add that to the document body, and we'll register that.

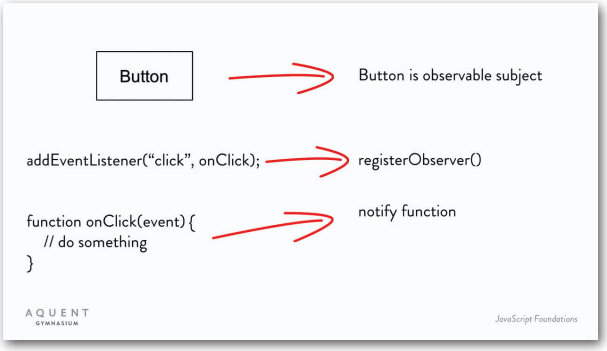
And, while we're here, let's create a third observer. This one we'll create and register, all in one shot. This observer is just going to call up an alert box. Let's test it. Click all of the checkboxes, and all three actions happen.

Now, an important takeaway to this pattern is that each part of the code is very independent. The subject doesn't know what the observers are doing, or anything else about them, other than the fact that they have a notify method. The observers don't know anything about checkboxes, or the subject, or that any other observers exist. All they know is that they have a notify method, and it does what it's supposed to do, when it's called.

Code like this is called, loosely coupled code, and it's very robust, and easy to maintain. A program where every piece of code is highly dependent on every other piece is called, tightly coupled code. It can be very fragile, and bug prone, as any minor change can break things all over the place.

Now, while learning about the observer pattern, you may have felt that it was a little bit familiar. If so, you may have been thinking about the event system that's already in use in JavaScript. If we listen for, say, a click event on a button, that button is essentially an observable subject. And, the fact of it being clicked is a change.

Now, instead of a registerObserver, and unregisterObserver method, we have methods called, addEventListener and removeEventListener. And, instead of objects with a notify method, we can pass a function, directly. And, finally, because a button can have several types of events, (click, mouse down, mouse up, mouse move, et cetera), we can specify a specific event that we want to be notified about. But, other than those details, the event system is very definitely an example of the observer pattern. So, you can see that a design pattern isn't a strict set of rules, but a guideline to be adapted to individual cases.



So, there's one design pattern. There are a lot more to learn. I'll leave a list of resources in the material for this lesson. And, I'll leave you with one warning about software design patterns. Often, when developers first discover them, they get very excited about them, and go into a kind of design pattern madness; trying to implement every possible design pattern they learn about, in every single project they do.

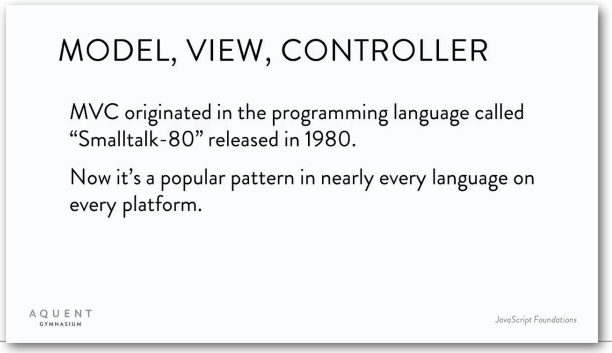
This sometimes winds up making their code over-complicated, for a while. Eventually, they recognize that each pattern solves a specific problem. And, then, recognize when to use a particular pattern, when they run across that problem. Of course, you may only get that experience by over-using the patterns, in the first place. But, it's something to be aware of.

Next up, we'll look at an even higher-level concept of application structure called, the model view controller.

MODEL, VIEW, CONTROLLER

In this video, we'll discuss a very common pattern for creating applications, known as, the model view controller pattern. This is a higher-level pattern than regular software design patterns, which are designed to handle a specific problem.

The Model View Controller pattern, or MVC, is an overall pattern for architecting an entire application. As its name implies, MVC breaks the application down into three main parts: the model, the view, and the



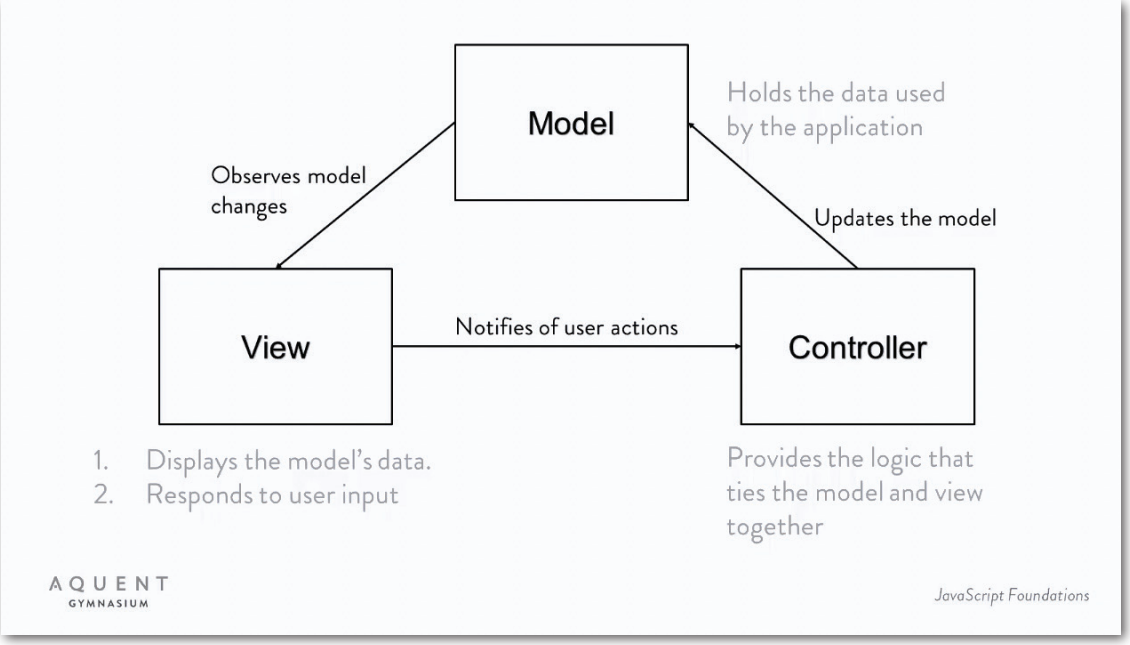
controller. MVC originated with a programming language called Smalltalk-80, but has evolved, and adapted to become the most popular way of structuring all kinds of applications, across almost every language being used today.

Let's look at the three component parts of MVC, a bit more closely. The model holds the data that's being used by the application. In a word processor, for example, the model would hold the text of the document, and maybe any formatting. In a spreadsheet, it would hold all the numbers and the formulas. In an application, where data was loaded from a server, that data would probably be stored in the model.

Now, the view has two functions. First, it's sort of a window on the model showing the data, or some portion of the data, from the model formatted in some way. Secondly, it allows for user feedback. The view may have buttons, text inputs, or other user controls that allows the user to take actions, within the application.

The view captures these actions, and forwards them to the third part of the trio, the controller. The controller ties the model and view together, and implements the functionality of the application. When the view forwards user actions to the controller, the controller uses that data to make some changes to the model. When the model changes, it notifies the view, which updates itself to reflect those changes.

Now, you may have noticed the observer pattern there, between the model, and the view. The model is the subject, and a view is an observer of the model.



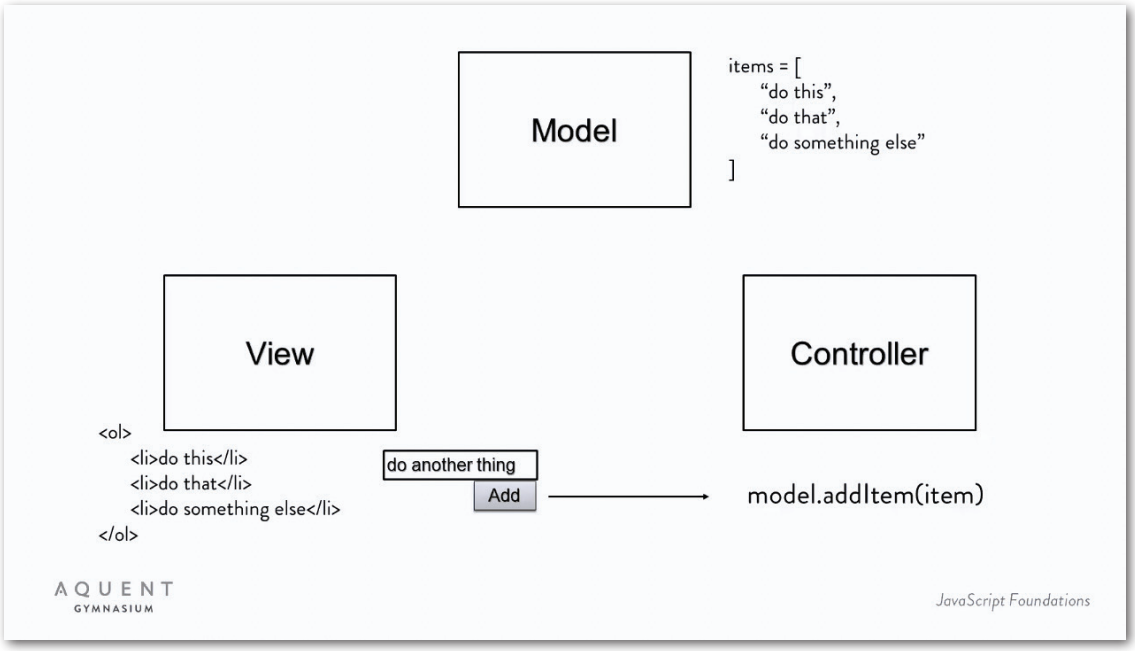
As with other patterns, MVC is a guide that can be adapted for specific applications. In fact, there are many variations in MVC, such as, Hierarchical MVC, or HMVC, Model View Adapter, or MVA, Model View Presenter, or MVP, and Model View View Model, or MVVM.

We won't get into all the variations, here. But, you should notice that they generally include the concept of a model, and a view. But, they vary in how that third part binds them together. Collectively, these patterns are

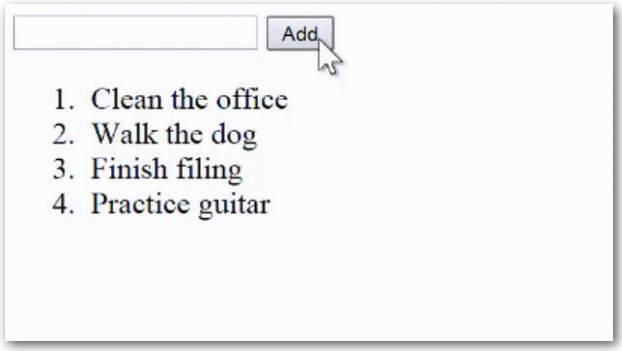
often called MV*, with the star, or asterisk, being a wild card, for whatever that third component may be.

Now, it can take a while to fully understand, and be comfortable with MVC, and all its variants. But, I wanted to give you a little taste of it here, with a very simple example. This example is going to be a simple to-do list.

The model will hold a list of to-do items. The view will display these items, and provide a text input, and a button to add a new item. When the button is clicked, the view will notify the controller, which will take care of adding the new item to the model. The model will then notify its observers, which in this case, is just the view. And, then, the view will update to display the current list of items.



Now, here's what the app will look like in action. Initially the view is just the input field, and a button. Typing in a to-do, and clicking on the button causes the controller to add that item to the model. Then, the view renders what's in the model to a list. So, let's start coding.



Now, first, I want to show you a quick preview of the finished code. The thing to note here, is that there are three distinct sections of code. This part of the code holds all the model logic. This part is the view, and here's the controller. So, this code is far more organized than most of what we've done, so far.

We'll start with this HTML file, mvc.html. It has an input text element, a button, an empty list, and a script tag. Very simple. In the script, we'll start by creating our three MVC objects: the model, the view, and the controller.

The model will be an observable subject. So, we'll give it an observers array, an addObserver, and notifyObservers as functions, just like the example in the last video. It will also have an items array, which is the data that this model is responsible for. Finally, it also gets an addItem method, so that the controller can add items to it. AddItem will just push the new item onto the items array, and then call notifyObservers. And, that's our whole model.

mvc.html mvc.js

```
var model = {
    observers: [],
    items: [],

    addObserver: function(observer) {
        this.observers.push(observer);
    },

    notifyObservers: function() {
        for(var i = 0; i < this.observers.length; i++) {
            this.observers[i].notify();
        }
    },

    addItem: function(item) {
        this.items.push(item);
        this.notifyObservers();
    }
};

var view = {
```

The view will be the object most intimately connected to the HTML, itself. So, it will have properties for the input text, the button, and the list based on those elements' IDs. The view will be an observer of the model. So, it will need to have a notify function. In that notify function, we'll loop through the model's items, and create a list item for each one in a string that we can set as innerHTML.

Now, this could also have been done by creating element objects, or with a document fragment, or even a template. But, we'll keep it really simple, here.

Finally, we'll need some code that will hook up the button click event, and add the view as a listener to the model. We'll put that in a function called init. This is a common name given to functions that get run a single time to initialize, and set up an object.

```
var view = {
    input: document.getElementById("input"),
    list: document.getElementById("list"),
    button: document.getElementById("button"),

    notify: function() {
        var html = "";
        for(var i = 0; i < model.items.length; i++) {
            html += "<li>" + model.items[i] + "</li>";
        }
        this.list.innerHTML = html;
    }
};

var controller = {

};
```

```
var view = {
    input: document.getElementById("input"),
    list: document.getElementById("list"),
    button: document.getElementById("button"),

    init: function() {
        model.addObserver(this);
        this.button.addEventListener("click", function() {
            controller.addItem();
        });
    },

    notify: function() {
        var html = "";
        for(var i = 0; i < model.items.length; i++) {
            html += "<li>" + model.items[i] + "</li>";
```

Inside of init, we'll register the view as an observe of the model. Then, we'll listen for a click event of the button. And, that handler we'll call, controller.addItem, which we'll define in a moment. And, that's our view. We'll just call view.init, right below that to make sure that the view gets initialized.

```
36              html += "<li>" + model.items[i] + "</l
37          }
38          this.list.innerHTML = html;
39      }
40  };
41
42  view.init();
43
44  var controller = {
45      addItem: function() {
46          var item = view.input.value;
47          model.addItem(item);
48          view.input.value = "";
49      }
```

Then, on to the controller. Now, as you've just seen, the controller will need to have an addItem method. That will get the value of the views input element, and use that to call model.addItem. It will then, clear out the input text, so that it's ready for another item to be typed, and that's the last piece. Let's see it in action.

At first, we just see the empty input, and button. Let's add a to-do, and click the button. This notifies the controller, which updates the model, which is observed by the view, which rerenders the list.

So, not exactly rocket science, here. But look at the code. It's nicely compartmented into three separate objects. The model just keeps track of the items, and notifies the observers when it changes. The view updates the display when the model changes, and tells the controller when the user has clicked the button. And, the controller ties it together by getting the item, and adding it to the model.

In other words, each part of the application has a specific purpose, and functionality. When you get into larger, and more complex applications, keeping your code organized into separate sections becomes vital for your own sanity.

Also, as I mentioned earlier, there are many variations on the MVC theme. And, even in a simple application like this, there are lots of different ways you could have arranged this. Maybe the view could have gotten the value of the item, and passed it on to the controller. For that matter, it could pass the item to the model directly, eliminating the need for the controller in a simple application, like this.

As you'll see in the next video, different MVC frameworks emphasize different parts of the MVC pattern. In some, the controller is all-important, and the view is just left to HTML. In others, the model, and view do most of the work, and the controller is an afterthought.

I like to think of MVC as more of a guideline, than some set of rules that are carved in stone, although you'll find plenty of developers engaged in heated arguments about the correct, proper, and right way to do MVC. It's up to you, how deep down that rabbit hole, you want to go.

In the next video, we'll take a quick look at a few of the more popular application frameworks that exist, out there. Many of these use some variation of MVC, although you'll certainly see some differences between them.

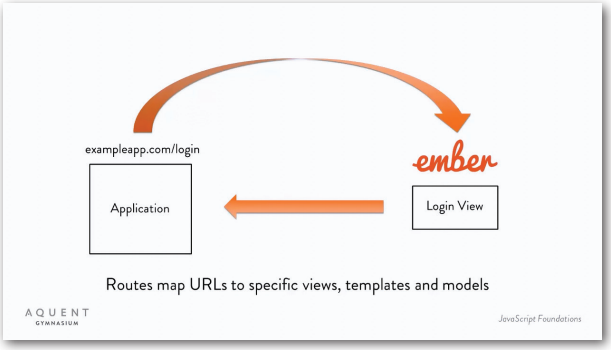
JAVASCRIPT APPLICATION FRAMEWORK

The last few years has seen an explosion in the number of JavaScript libraries being released, and used. A few years ago, you could get by as a JavaScript developer knowing nothing more than jQuery, and maybe one or two others. Now, it's pretty much impossible to keep up with all the various libraries being pumped out.

A number of these libraries are actually application frameworks, usually based on some variation of MVC. Many libraries, like jQuery for example, are what I call, toolkits. They contain a collection of objects, and functions that you can use, as you see fit.

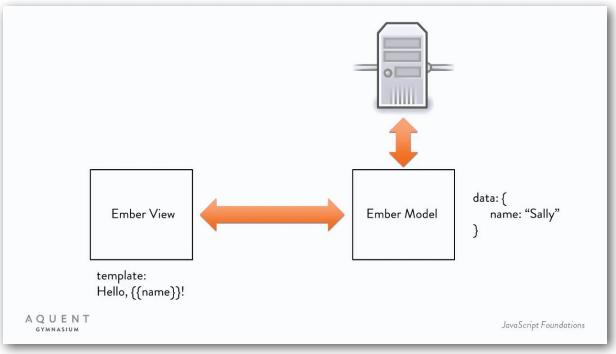
In lesson five, we used the AJAX functionality of jQuery, but we ignored the rest of it. Frameworks however, generally dictate the entire structure of your application. They help you to organize your code into logical sections, each having its own purpose. We'll take a very quick look at a couple of popular frameworks, ember and Angular, and then, go a bit deeper into another framework called, Backbone.

First, let's take a brief look at the ember framework. This framework was designed for making very large, complex web applications. Here are some of the sites and apps using ember. As you can see, it's pretty popular.



One thing ember emphasizes heavily is the URL. Normally, you would expect that a specific URL, such as exampleapp.com/login, would point to a specific HTML document on the server. But ember intercepts changes to the URL, and without loading a new page, it changes the state of the application.

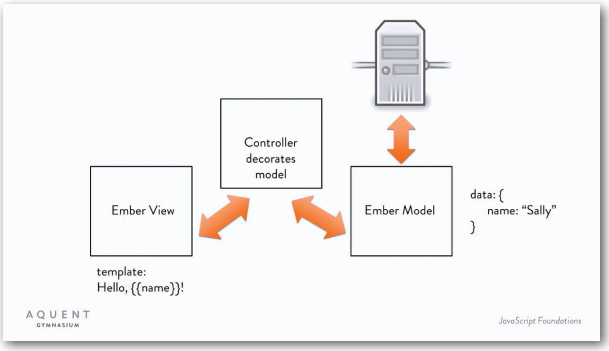
It does this through objects called, routes. Routes generally map a specific URL to a specific template that will be used to build a specific view. Thus, the URL exampleapp.com/login might activate a login route that would render a login template.



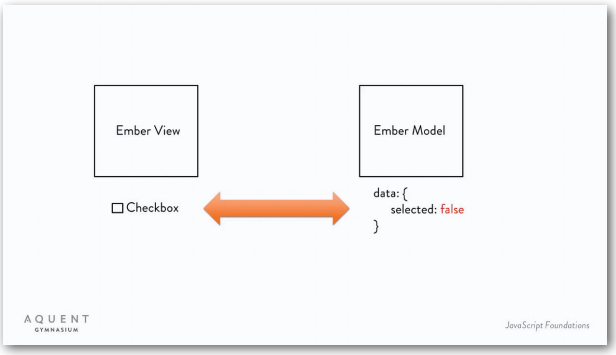
Templates in ember are essentially the same as mustache and handlebars, with perhaps a few more advanced features. So, a URL, route, and template will comprise a specific view. Each view has its own model. That model often consists of data that's pulled down from a web server, somewhere.

For example, you could have an article view that would display the text of a specific article loaded from a server. That model's data would be used to render the template in the view. Ember also has the concept of a controller, but it's a bit different than you would expect in traditional MVC.

Ember controllers sit between the model and the view, and are said to decorate the model. That means that they may add, or change some of the model's data, in some way. For example, the model might contain the full text of an article, but the controller might shorten that text to display a summary, or intro.



Ember also makes use of binding. Binding is a piece of functionality that allows a visual item, such as an HTML element, to be bound to, or permanently connected to, a source of data in the model. For example, you could have a Boolean value in the model called, selected, and a check box in the HTML. That check box could be bound to that value, so that if the value changes, the check box will automatically be updated.

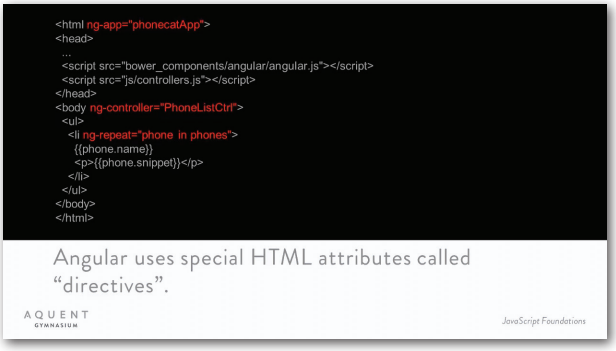


There's also two-way binding, which means that if the user changes the check box, checking or unchecking it, it would automatically change the value in the model. The fact of binding, connecting the model more directly with the view is probably a big reason why the controller takes on a lesser roll in ember. In traditional MVC, a lot of that communication between the model and view would go through the controller.

Ember is a popular, and very powerful framework for creating large sites and apps. But, all that power means that it can have a pretty, steep, learning curve. Be ready to buckle down, and do some reading and experimenting, in order to learn it well.

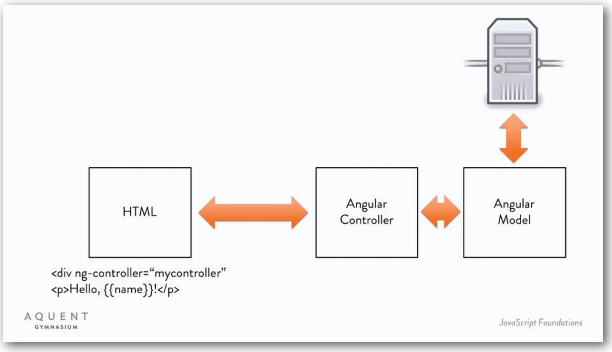
Another popular framework right now is AngularJS. Again, here's a list of different projects that are using Angular. Angular derives its name from the angle brackets that define HTML pages. This is because it has a heavy involvement in the HTML, itself.

Rather than using specific templates, and accessing the HTML, from code alone, Angular relies on special attributes, within the HTML text. These are called Angular Directives. It also allows for template-like formatting, right inside the HTML; in a sense turning the HTML page itself into a template. These template-like



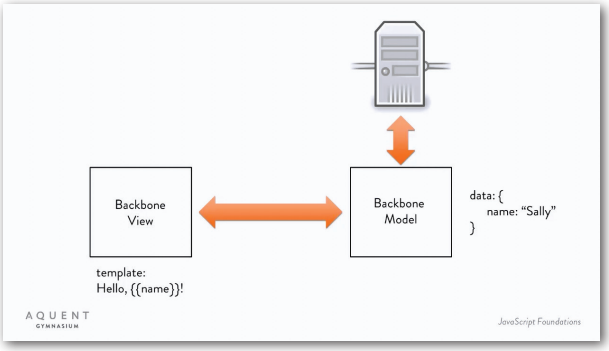
structures are actually binding statements. Angular relies heavily on its powerful two-way binding.

Because of the directives and bindings, the HTML, itself, functions as the view. Angular emphasizes the controllers, which provide the logic for the application, and model objects for the view to bind to. Angular is another very powerful MVC framework, suitable for making large applications. However, because it doesn't impose a lot of specific structure on your project, you could just use pieces of it in a smaller application.



Now, the final framework we'll look at a bit more in-depth is called, Backbone.JS. As MVC frameworks go, Backbone is pretty straightforward, and probably has the lowest learning curve. It's an excellent choice for smaller applications. Not that it couldn't be used for larger applications, but something like ember might be overkill for simpler projects.

Not to be outdone, Backbone also has a page listing projects, and companies using it. Backbone emphasizes models and views, and also uses routes for mapping URLs to specific parts of an application. In that way, it's very much like ember in structure.

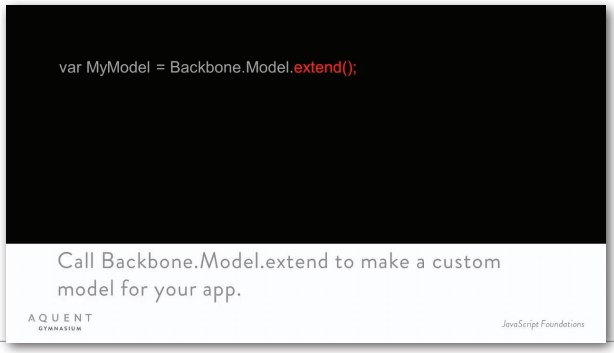


It doesn't have any built-in templating, or much in the way of binding, but you can use any templating system, alongside it. Backbone views are tied to a specific element on the HTML page. They listen for changes in a model, and then render themselves based on that model's data. They can also listen for actions in the view, and deal with those.

Models are often tied to web services, getting the data from the service, and providing it to a view. There's no built-in controller object in Backbone. It's up to you to determine how you want to tie your models, and views together.

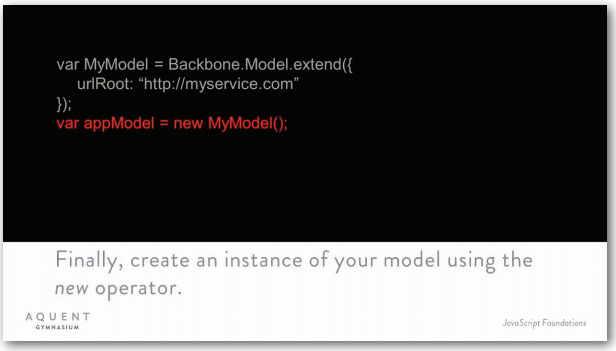
You could have the models and views share the logic that would normally go into a controller, or you could write your own separate controller code. Backbone doesn't enforce much structure on you. Now, let's take a deeper look at views and models in Backbone.

There's a built object in Backbone called, backbone.model. This is a generic object that you need to extend, to create your own model. By extending it, we mean that you're creating a copy of it, with properties, and functions that are specific to your application. You do this by calling Backbone.Model.extend, and you



pass in an object with those properties, and functions that you want to add to it.

If you're going to be using the model to load, and store data from a web service, then you can extend it with a property called, urlRoot, which would be the base URL of the service you are contacting. Once you extend your model, you need to make an instance of it. You're basically making another copy of your specific extended model.

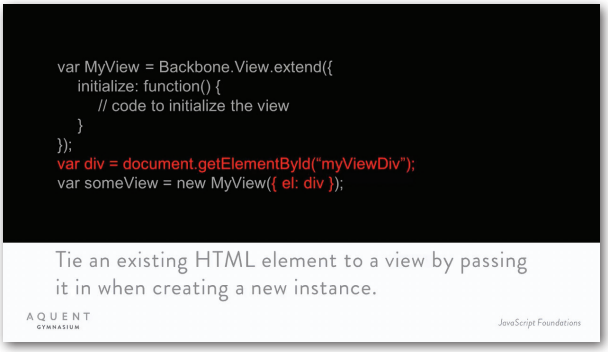


You do this with the new operator. Now, you have a model ready to use in your application. We'll see more of exactly what this model will do, in the next video.

Likewise, there's a backbone.view object which you can extend. You would, again pass in an object with specific properties, and/or functions that would be used to customize your view. And, again, create a new instance of that view with the new operator.

One function you might find useful to extend your views with is initialize. This is exactly like the init function we created in our last video. But, in Backbone, that initialize method will be automatically called, when you create the view instance.

I mentioned that Backbone views are tied to HTML elements. By default, a view will create a div element to act as a view. But, you could also tie it to an existing element, if you want. You do that by passing an object in, when you create the new instance. That object can have a property named EL, the letters E, L, that points to an existing HTML element that you want to tie the view to.



There are other parts of Backbone, of course, but the model and view will be enough for us to do the final example app in this course. Now, that may be a lot to take in right now, but you'll have plenty of hands-on experience creating views, and models in the next video. So, when you're ready, we'll see you there.

MOVIE FINDER, REVISITED

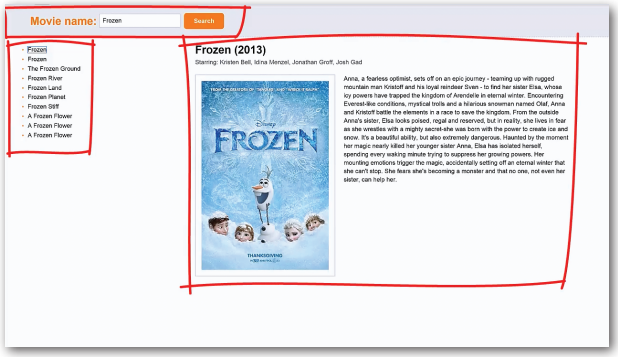
For the final example in this course, we'll be rewriting the movie finder application from lesson five, but using Backbone models, and views. The main goal here, is for you to see how using an application framework can provide structure for your code, which becomes extremely important for your applications, when they get large and complex.

Let's first take a look at the application's code, as it stands. We have a mixed bag of variables up at the top here, and a list of functions, down below. Now, this isn't too bad, because it's really quite a simple app. But,

in a larger app, you might have dozens, or even hundreds, of variables, and the same number of functions. Scrolling through page, after page of code, looking for that one function that does something you want, could be painful. Backbone's really, just going to help us organize this code better.

Now, bear in mind that using a framework tends to impart some overhead on a project, and in a relatively, small project like this, that overhead can wind up making the project look more complex than it already is. In general, the larger, and more complex an application already is, the more it's going to benefit from a framework.

Now, to get an idea of what we'll be doing, let's look at the app itself. You can easily break this down into three different views. First is the search view, which captures the search item that the user enters, and the button click. Then, there's the results view, showing the list of results from that first server call. This view will have its own model containing those results. It'll also take note when one of those items is clicked. And, finally, there's the details view, again, backed by a model, containing a specific movie's details. So, it looks like we'll be making three views, and two models.



Now, we won't need to change anything in the HTML or CSS, but we will need to add a script tag for Backbone. Also, Backbone has a dependency on another library called, Underscore. You don't need to worry about that much. It's just used internally in Backbone, but we will need to add it in, here. You can find download links for both of these libraries on the Backbone site. Backbone uses jQuery behind the scenes, as well, for server communication, so we'll leave that in there. And we'll continue to use Mustache for our templates.

```
18      <img id="poster" src="{{Poster}}">
19      <p>{{Plot}}</p>
20      </script>
21
22  </head>
23  <body>
24      <div id="search">
25          <label>Movie name:</label> <input id="search_text" type="text">
26          <button id="search_button">Search</button>
27      </div>
28      <div id="list"></div>
29      <div id="details"></div>
30
31      <script type="text/javascript" src="mustache.js"></script>
32      <script type="text/javascript" src="jquery-1.11.0.min.js"></script>
33      <script type="text/javascript" src="underscore-min.js"></script>
34      <script type="text/javascript" src="backbone-min.js"></script>
35      <script type="text/javascript" src="movies.js"></script>
36  </body>
37  </html>
```

Now, in the main JS file, we'll just get rid of all this code here, so we can start fresh. First, we'll extend Backbone.Model, giving a URL root of http://omdbapi.com. Then, we'll make two instances of this model. One we'll call, search model, and this one will be used for making the initial search, and getting a list of results. The other one will be called, details model, and as you can guess, this will be used for getting a

movies.html × movies.js

```
1  var MovieModel = Backbone.Model.extend({
2      urlRoot: "http://omdbapi.com"
3  });
4
5
6
7
8
9
10
11 var searchModel = new MovieModel(),
12     detailsModel = new MovieModel();
13
```

particular movie's details. Although these models are basically identical, right now, we'll use one for each purpose, and connect a different view to each one, so that when the search results come in, in the search model, the results view will be updated. And, when the details results come in, in the details model, the details view will be updated.

Next, we'll move on to the views. We'll make three views, one for the search form, one for the search results, and one for the movie details. The search view will listen for a click on the search button, get the search term entered in the search text field, and tell the search model to fetch the results. So, first, we extend Backbone view. We'll give it a searchText property that will point to that searchText input element. Then, we have to listen for click event on the search button.

Now, Backbone has an interesting way of assigning event handlers. We'll make a property called, events, which will be an object, and that object will have a property for each event we want to listen to, and the function that should be called, when that event occurs. The property name is a string containing the name of the event, plus a space, plus a CSS selector that will match the element, or elements that we want to listen to.

So, here, we want to listen for a click event on the element with an ID of search button. When that event happens, we'll call a function named, doSearch. Now, realize that this is all Backbone-specific syntax. Internally, at some level, the same old addEventListener code that you're used to will be used to do the actual work of adding the listener to the proper element.

And, finally, we need to create this doSearch function. That's another property in here. And, that, we'll call, searchModel.fetch. This tells the model to contact its URL root, and request data from it. But, remember that, that URL root doesn't have any query string parameters on it. So, how do we add those? Well, when we call fetch, we can pass in an object. That object can have a property called, data. Any properties in that data object will get added as query string parameters.

movies.html movies.js

```
1  var MovieModel = Backbone.Model.extend({
2      urlRoot: "http://omdbapi.com"
3  });
4
5  var SearchView = Backbone.View.extend({
6      searchText: document.getElementById("search_text"),
7
8      events: {
9          "click #search_button": "doSearch"
10     },
11
12     doSearch: function() {
13         searchModel.fetch({
14
15         })
16     }
17 });
18
19
20
21
```

So, we can give data an s property set to this searchText.value. If the value of searchText was Star Wars, for example, this would translate into a query string of s equals Star Wars. Now, it's important to know that when the model successfully fetches its data, it will dispatch a change event. We'll use that in the next view.

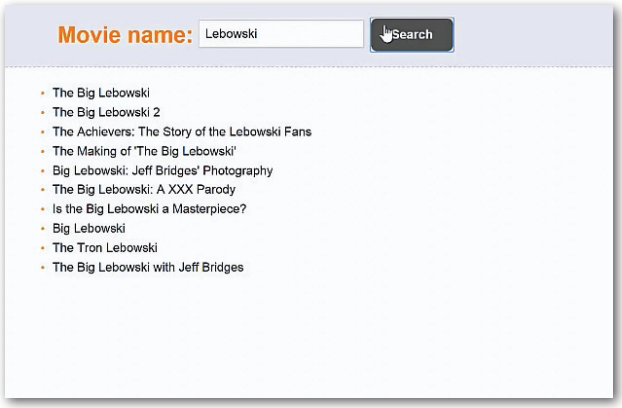
Finally, for the search view, we'll need to create an instance. We'll set the element of this search view to the element with the ID, search, which is our search form. The next few will be the search results view. I'll just call this, results view. This has two functions. First, it needs to listen to search model, and when search model changes, it needs to render the search template, using search model's updated data. It also needs to listen for clicks on those list items, and tell the details model to fetch details for a selected movie.

So, we extend Backbone.View. And, we'll need a reference to the template, so I'll add that, here. And, we'll create an initialize function. In the initialize function, we'll listen for changes on the model. Now, in the search view, you saw how we created an events property with special syntax to listen for a click event. That works for user events on HTML elements within the view, but we'll need to do something a bit different to listen for events on the model. We'll use a function called, listenTo, saying, this, listenTo, searchModel, change, this.render. In other words, when searchModel dispatches a change event, it will run the render method of this view.

```
20
21  var ResultsView = Backbone.View.extend({
22      template: document.getElementById("list_template").innerHTML,
23
24      initialize: function() {
25          this.listenTo(searchModel, "change", this.render);
26      }
27  })
28
29
30
31  var searchModel = new MovieModel(),
32      detailsModel = new MovieModel(),
33      searchView = new SearchView({ el: document.getElementById("sear
34
35
```

Now, that render method. This will get the data from the model, and use it with the template to render the search results. This view will be tied to the empty list div in the HTML document. So, we can say, this.el.innerHTML equals Mustache.render. Now, we already have the template, so all we need, now, is a way to get the data out of the model. We do that by saying, searchModel to JSON. What this returns is, a JavaScript object based on the JSON that was loaded from the OMDB service, and that's exactly what we need to render the template.

With this, we should be able to test the search. First, we'll make a new instance of the results view, passing in the list div, as the views element. Then, we fire this up in the browser. I'll enter, Lebowski. And, the search button click is picked up by the search view, which calls fetch on the searchModel. The searchModel gets the data, and fires a change event, and the results view hears this change event, and renders the JSON, fetched by the model, into the template. Perfect.



Next, we need to listen for clicks on that list of results. To do that, back in the results view, I'll add an events object. Here, we'll listen to the click event on any anchor tag elements by specifying click space a, and this will call a function called, getDetails. So, we'll create that function.

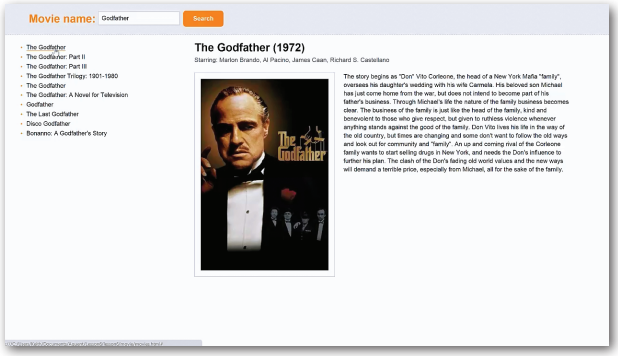
This will be very similar to the doSearch method of the search view. We'll call fetch on the details model this time, passing in an object with a data property, and setting i to the ID of the clicked element. Remember that, the ID would be the IMDB ID for that particular movie that was clicked.

```
    initialize: function() {
        this.listenTo(searchModel, "change", this.
    },

    events: {
        "click a": "getDetails"
    },

    getDetails: function(event) {
        detailsModel.fetch({
            data: {
                i: event.target.id,
                plot: "full"
            }
        });
    },
```

We'll also set plot to full for a more detailed plot. Again, both of these properties will get added as query string parameters. All we need now, is a details view. We extend Backbone view again, get a reference to the template we'll use for this view, and create an initialize function. And that, we'll listen to changes on the details model, again, calling render. And, in render, we do much the same thing we did in results view. Set this.el.innerHTML to the results of Mustache.render, passing in the template and the model's to JSON method.

```
});

var DetailsView = Backbone.View.extend({
    template: document.getElementById("details_template").innerHTML,

    initialize: function() {
        this.listenTo(detailsModel, "change", this.render);
    },

    render: function() {
        this.el.innerHTML = Mustache.render(this.template, detailsModel.toJSON());
    }
});



var searchModel = new MovieModel(),
    detailsModel = new MovieModel(),
    searchView = new SearchView({ el: document.getElementById("search")}),
    resultsView = new ResultsView({ el: document.getElementById("list")}),
    detailsView = new DetailsView({ el: document.getElementById("details")});
```

Last, but not least, we'll create an instance of the details view, passing in the empty details div as an element, and that's that. Now, the search works the same, but now, when we click on an item, the results views tells the details model to fetch its data, passing in the IMDB ID, and plot equals full. The details model dispatches change when it's fetched its data, and the details view renders this data in its template.

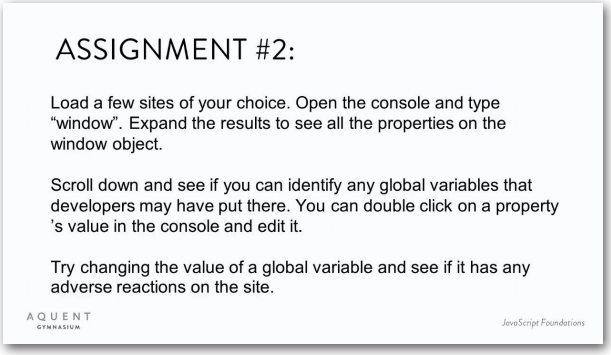


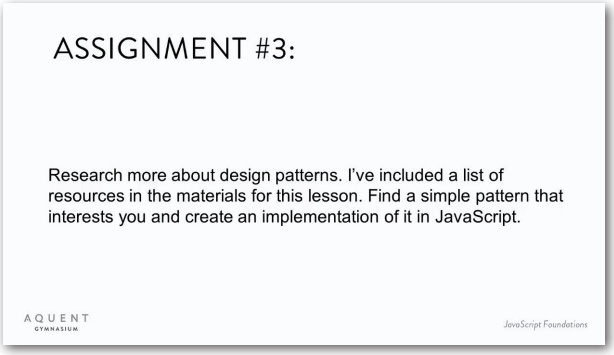
Now, you may be thinking that this example winds up with a lot more, and more complex code than the original. So, how can that be better? Again, you're right, and that's the problem with creating simple examples for a course, like this. The fact is, that application frameworks really do simplify, and help complex applications. But in very simple applications, like we have here, they can actually make things look more complex.

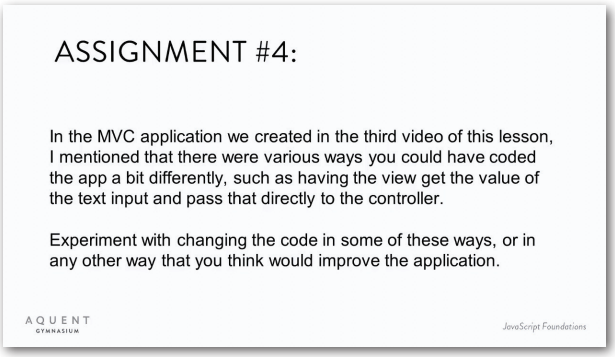
And last, but not least, you have your assignments. First of all, is the final quiz. Second, load a few sites of your choice. Open the console and type, window. Expand the results to see all the properties on the window object. Scroll down, and see if you can identify any global variables that developers may have put there. You can double click on a property's value in the console, and edit it. Try changing the value of some global variable, and see if this has any adverse reactions on the site.

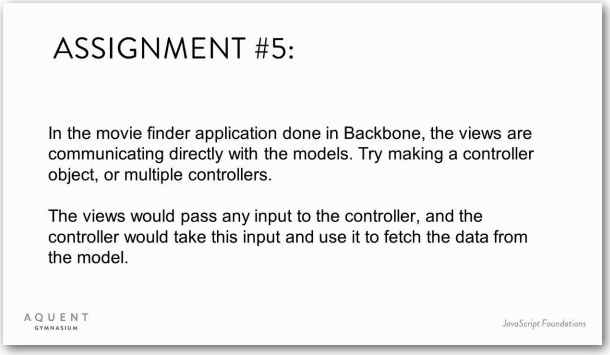
Three, research more about design patterns. I've included a list of resources, below. Find a simple pattern that interests you, and try creating an implementation of it, in JavaScript. Four, in the MVC application we created in the third video of this lesson, I mentioned that there were various ways you could have coded the app a bit differently, such as having the view get the value of the text input, and pass that directly to the controller. Experiment with changing the code in some of these ways, or in some other way you think that would improve the application.

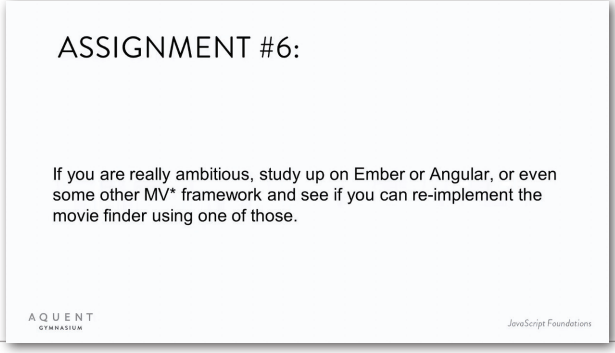
Five, in the movie finder application done in Backbone, the views are communicating directly with the models. Try making a controller object, or multiple controllers. The views would pass any input to the controller, and the controller would take this input, and use it to fetch the data from the model. And, finally, six, if you're really ambitious, study up on ember, or Angular, or even some other MV* framework, and see if you can re-implement the movie finder using one of those.



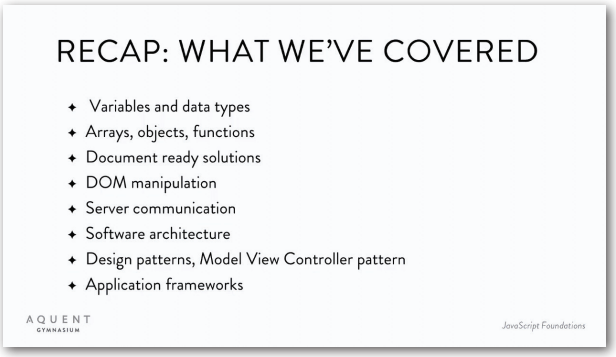








SUMMARY

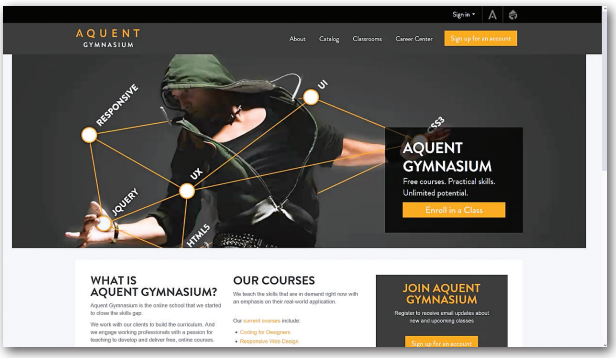


Well, assuming you've watched all the videos, worked through the examples with me, and completed the assignments, you should be in a very different place from when you started this course. You now, have a good understanding of the basics of JavaScript. You know the different types of variables, and how to use them, including the composite data types, objects, and arrays, as well as, how to create reusable bits of code, and functions. You know where to put your code, so that it loads, and initializes when the rest of the page is ready. And when the page is ready, you know how to access the HTML elements on it, change their properties and styles, and even remove, or reposition them on the page.

You can create new HTML elements directly with document fragments, or with templates, and you can use those elements, just like any other element that already existed on the page. We've also covered the various ins, and outs, of server communication, how to contact a server, send data to it, and get specific data back from it, and how to parse, and use that data in your programs.

Finally, we looked at application architecture, including software design patterns in the Model View Controller pattern, and even delved into some of the more popular application frameworks in use, currently. You've made several simple applications, and the final Movie Finder application is nearly full-featured enough to be a professional application that you could release, and have people find useful. And, with what you know, you can definitely customize it, and fill it out, or even use a different web service to create some other useful application.

In short, take this course as a launching pad. There's a lot more to learn about the language, and there are many avenues in specialization. You might want to learn more about application frameworks, or server communication, for example. Or perhaps, you are more interested in user interface design, or templating via JavaScript. Or maybe, you want to look into audio, video, animation, or game programming; all areas which we haven't even touched in this course, but you are now in a position to learn, and understand.



I hope that you've learned some useful stuff in this course, and I really hope that you continue what you've started here by learning more, and putting it all to use. I highly suggest you check out some of the other courses here, on the Aquent Gymnasium site. There are probably a few that would be a perfect fit for the level of knowledge, you now have.

Thanks for going through this course with me. Good luck, and happy coding.