Introduction to Knockout

Client Side MVVM

Knockout is a great tool for building client-heavy UIs.

Where is Knockout:

http://knockoutjs.com/

http://knockoutjs.com/downloads/knockout-2.2.1.js

Included in ASP.NET MVC4 Template

It's baked into ASP.NET MVC4, but it has no server-side components.

Best way to learn is to do.

Our Application:

Something Enterprise-y… say… a CMDB?

Our Application:

CMDB: The Cupcake Management Database

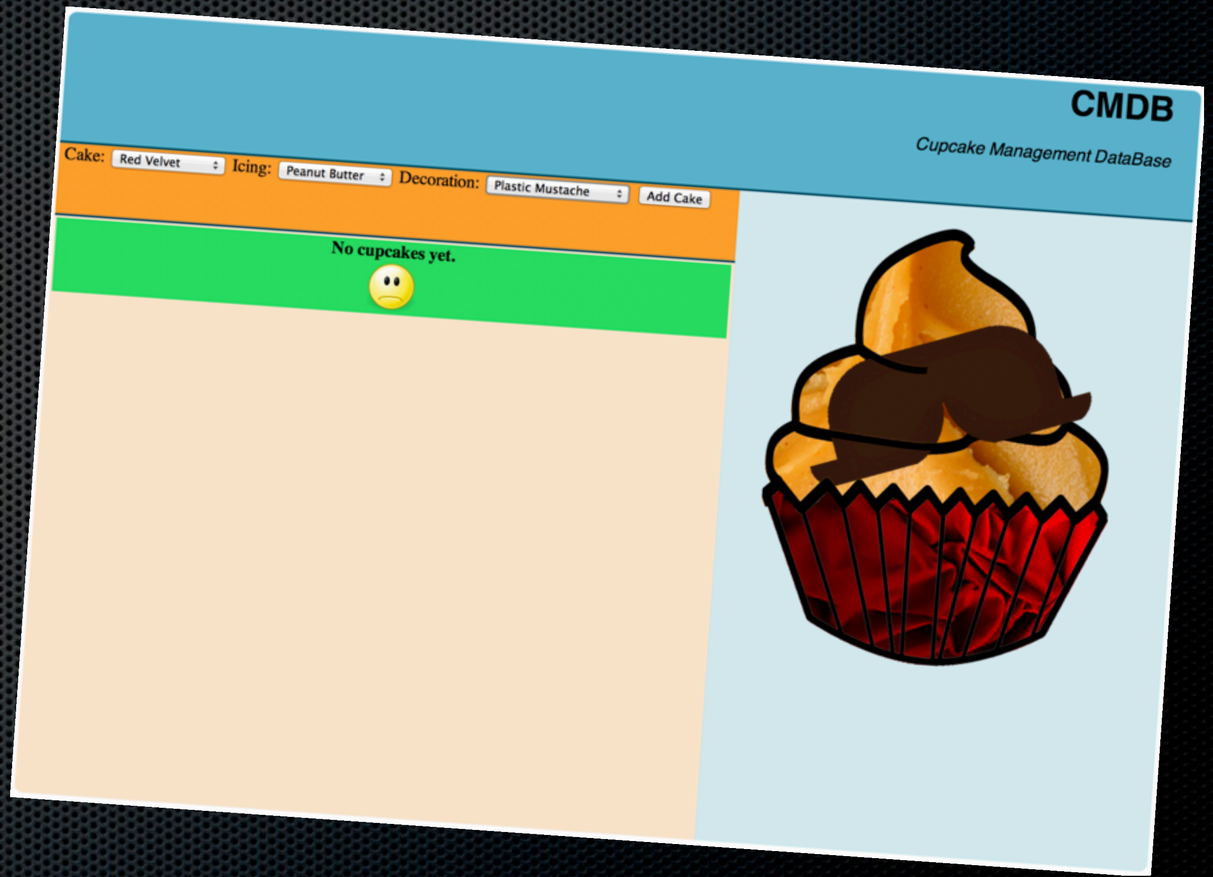
Let’s start by showing…

100% Client Code

Knockout,
HTML, CSS

No other libraries

Approx. 115 lines of JS



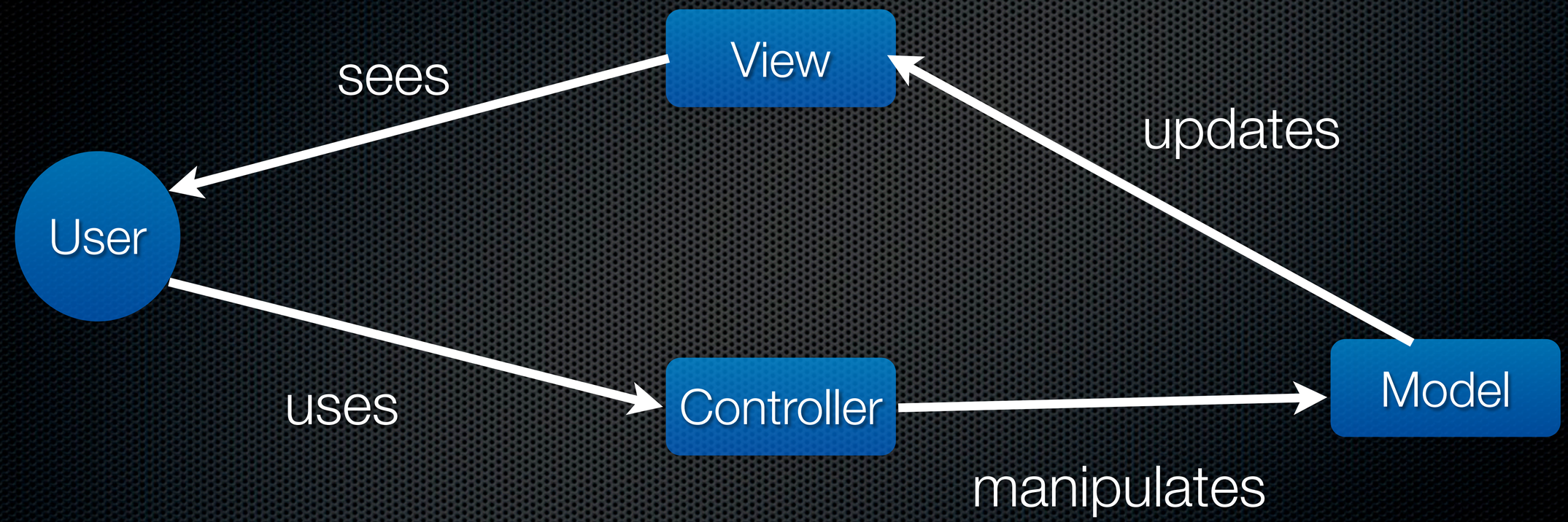
Knockout makes client rendering easy

For the ASP.NET MVC programmers in the audience, KO is Razor syntax on the client side.

How do we build UIs?

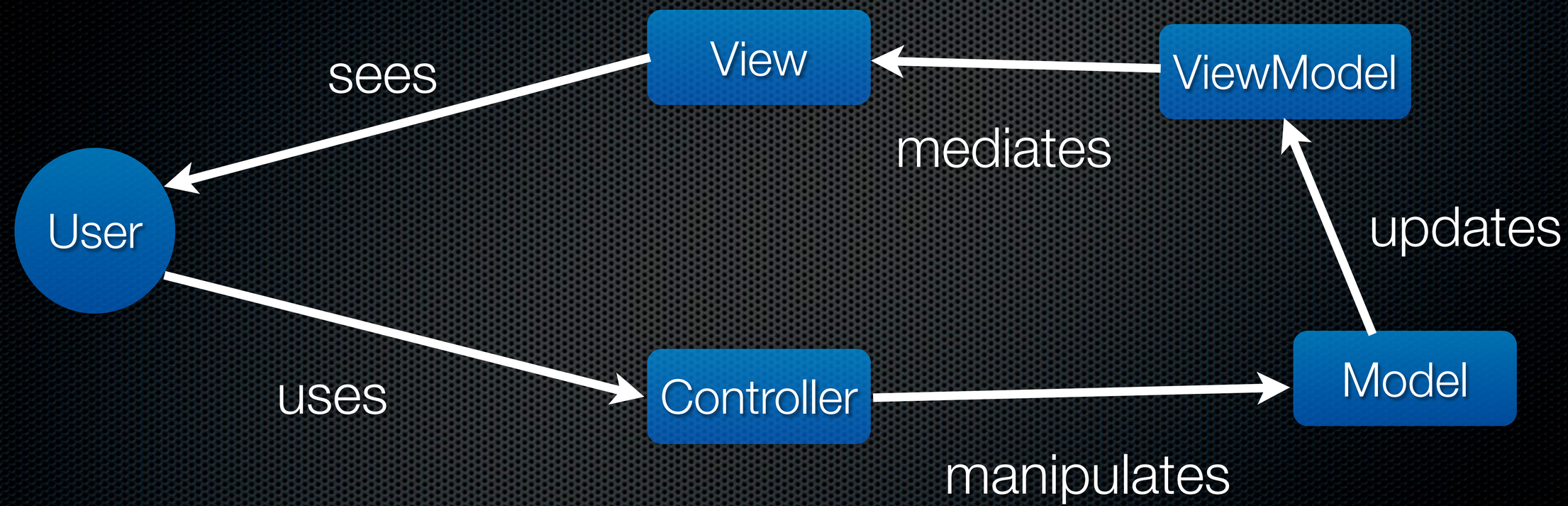
We use Model-View-Controller

Review: Model-View-Controller



The controller contains the operations we can perform on the model. The model is a “dumb” data object. The view renders based on the current state of the model.

Review: Model-View-ViewModel



The ViewModel is “glue”. It’s a “model of the view”. The ViewModel tracks UI state and Model state, but does not actually know anything about how things actually get rendered. The View still does the display.

The Big Deal

Design Views

(mostly) Pure HTML

Design View Models

Pure Javascript

Let Knockout bind the JS to the HTML

makes UIs **easy**

It does what it says on the tin

Knockout doesn’t care…

… where your objects come from…

It works with your JavaScript objects. They can come from Ajax calls, or be built by your application

… or other ways you manipulate the DOM…

It doesn’t require jQuery. You might not even want to use jQuery. This demo uses NO OTHER LIBRARIES.

… or what your server side is…

It has no back-end dependencies.

Define a Cupcake (Model)

```
var BasicCupcake = {
  CakeFlavor :"Chocolate",
  IcingFlavor: "Vanilla",
  Decoration :"Plain"
}
```


This is a standard, basic, JavaScript object. Nothing exciting.

Define a Cupcake (View)

```
<div id="Cupcake" class="PreDemo">
 Cake: <span>Cake Flavor Goes Here</span><br />
 Icing: <span>Icing Flavor Goes Here</span><br />
 Decoration: <span>Decoration Info Goes Here</span><br />
</div>
```


This is basic HTML. Now, we need some way to turn this into a usable view.

Define a Cupcake (View)

```
<div id="KnockoutCupcake" class="PostDemo">
 Cake: <span data-bind="text: CakeFlavor"></span><br />
 Icing: <span data-bind="text: IcingFlavor"></span><br />
 Decoration: <span data-bind="text: Decoration"></span><br />
</div>
```


This is basic HTML. Now, we need some way to turn this into a usable view.

Define a Cupcake (View)

```
<div id="KnockoutCupcake" class="PostDemo">
 Cake: <span data-bind="text: CakeFlavor"></span><br />
 Icing: <span data-bind="text: IcingFlavor"></span><br />
 Decoration: <span data-bind="text: Decoration"></span><br />
</div>
```

magic happens

This is basic HTML. Now, we need some way to turn this into a usable view.

One attribute to rule them all…

```
data-bind="text: Property"
```

The core of Knockout

You can apply the data-bind attribute to *any* HTML element. There are many possible bindings.

Also, the data-* attributes have been in HTML for awhile, but were standardized as part of HTML5. They are likely the *best* feature for developers, as they allow you to add arbitrary properties to DOM elements.

…and in the darkness bind them

```
ko.applyBindings(ViewModel, DOMElement);
```


You will usually invoke this method as part of your OnLoad event. The DOMElements need to have been created before you can invoke it.

STOP

The big idea: Knockout binds JavaScript objects to HTML UI elements

Observables

Okay, this is actually the real core of Knockout…

Observables are the key to Knockout, as they allow two-way binding

Observables

```
object.property = ko.observable(value)
```

returns a wrapper function

Observable objects are wrapped

Observables

```
object.property();  //get
object.property(NewValue);  //set
```


By wrapping values as Observables, Knockout knows when the values change.

Knockout magically informs the View when an Observable changes

demo: Observables

Observable Arrays

```
this.ListProperty = ko.observableArray([]);
```


Same thing. Observable arrays *do* offer much of the same functionality of arrays- you can push, pop, shift, etc., but you do have to “dereference” the array to index it.

The foreach binding

```
<div data-bind="foreach: ListProperty">
  <span data-bind="text: SomeFieldOnListItem">
  </span>
</div>
```

This bit is what's repeated

Foreach changes your binding context

A more thorough version…

demo: ObservableArrays

Stop: Let's Review Our Bindings

We've seen a lot of different bindings we can use in the data-bind attribute. Let's recap.

text:SomeProperty	Output the value of the property
value:SomeProperty	Link the “value” attribute of the form field to the property
click:SomeMethod	Invoke the method on a click event
foreach:SomeList	Repeat the content of this element once for each element in the array. Changes the binding context to the current element.

One more thing about click:

what's `this`?

This is actually a really cool feature of Knockout, but you have to keep it in mind. “this” is going to be *the object from your binding*, not an element, and not the object which defined the function.

Binding Contexts

Your Knockout bindings have a binding context. This context represents WHICH OBJECT IS BEING BOUND RIGHT NOW.

This feeds into how we handle events, and we'll often need to bind things to different levels of the context.

The Binding Context is the object you’re manipulating

`ko.applyBindings` sets the initial context to the object passed in

`foreach` and `with` change the binding context

Cheating the Context

`$parent` and `$root` go back up to the higher contexts

$parent goes back one level, $root goes back to the original binding context.

Better Forms

Drop Down List

Drop Down Bindings

```
options: someArray

optionsValue: 'someProperty'

optionsCaption: 'Text prompt'
```

`value` still maps to the actual store property

Appearance Bindings

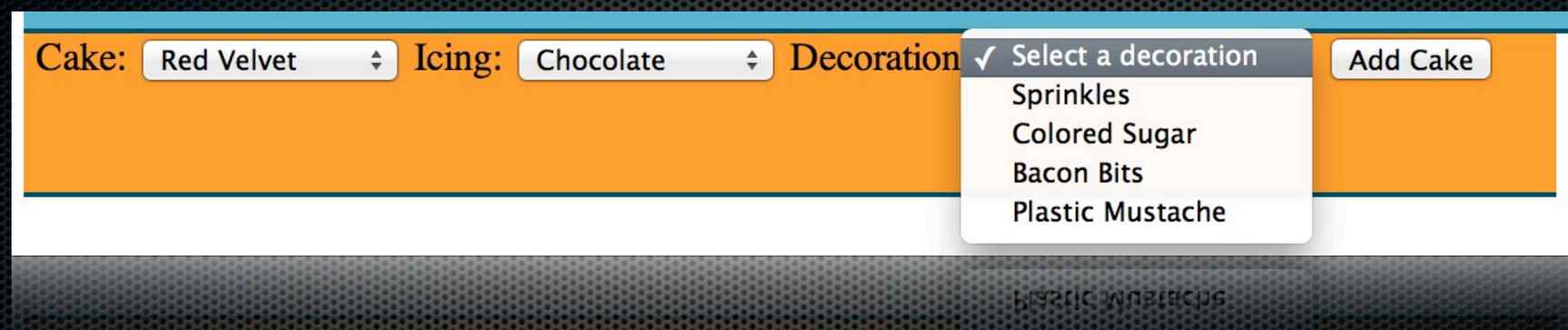
- `enable: someBoolean`
- `visible: someBoolean`
- `css: someNameOfACSSClass`
- `style: {‘css-property’:someValue}`
- `if: someBoolean`

Binding with expressions

```
visible: someObservable() > 10
```

Note that observables used must be invoked.

Cake Maker Form



Demo 05

Computed Observables

```
ko.computed(function() {});
```

Demo 06

ko.computed analyzes the function and determines what observables it depends on.

Notes on Computeds

by default, read only

by default, ALWAYS computed when a dependency changes

Computed Options

```
ko.computed({
    write: function(p) {},
    read: function() {},
    owner: what_this_shouldBe,
    defer: dontComputeUntilIAccessTheValue
})
```

Block Bindings

```
<!-- ko if: someExpression -->
    <span>This will render if the expression is true.</span>
<!-- /ko -->
<!-- ko ifnot: someExpression -->
    <span>Otherwise, this renders.</span>
<!-- /ko -->
```

Demo 07

Custom Bindings

These are so shockingly easy you aren't going to believe it when I show it to you.

Custom Bindings

```
ko.bindingHandlers.myCustomBinding =
{
    init: function(element,
valueAccessor, allBindingsAccessor) {
        /* Initial setup */
    },
    update: function(element,
valueAccessor, allBindingsAccessor) {
        /* when the VM changes */
    }
}
```

Demo 08

Binding other things

- `attr`
- `event`
- `style`
- `CSS`

- html attributes
- any other events
- style properties
- CSS classes

Final Result: CMDB