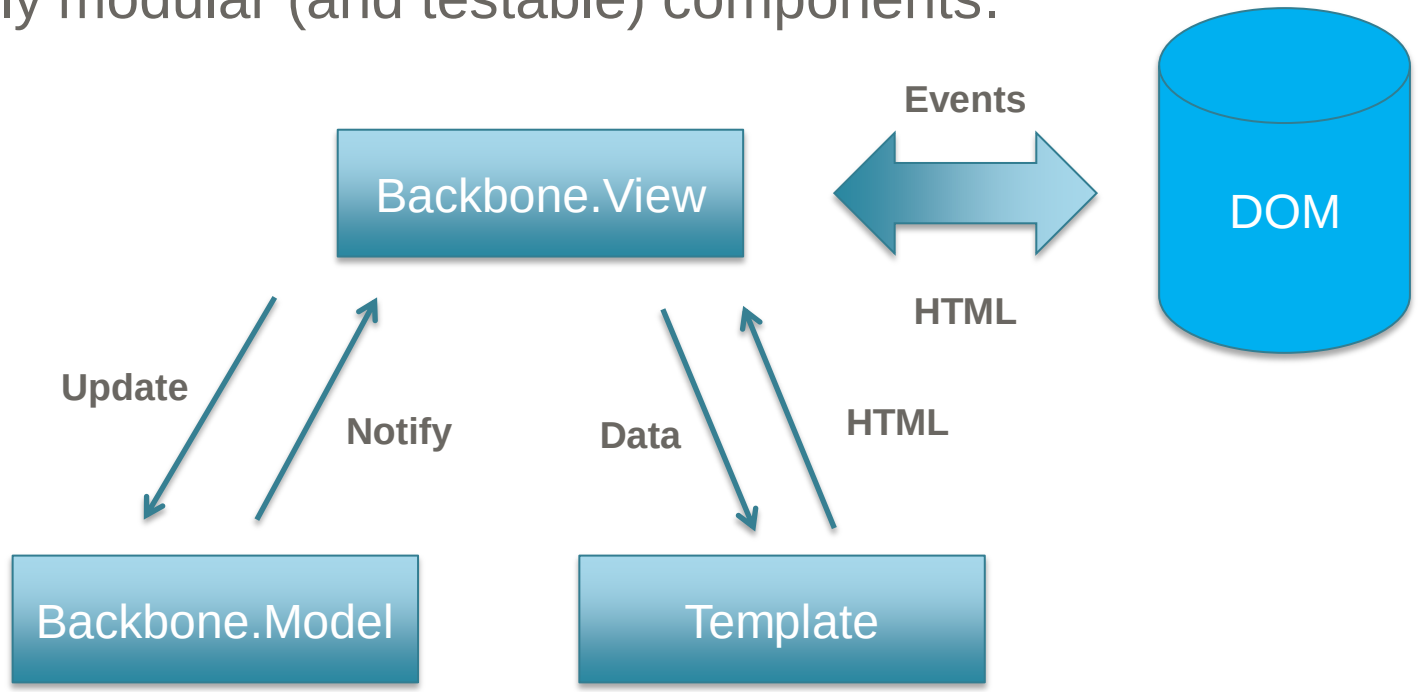
ZocDoc

Get well sooner.

- **Writing Maintainable JavaScript Using Backbone, Mustache & Jasmine**
- Highly Configurable Synchronization

JavaScript Is Risky Business

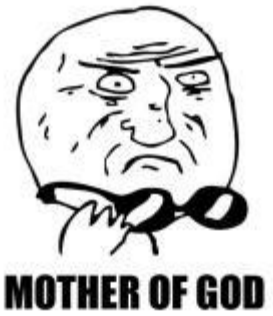
- Dynamically typed language that is interpreted at run time by your browser, aka, very fragile.
- Tends to be tightly coupled to the DOM using JQuery
- Backbone is essentially client side MVC that lets you create incredibly modular (and testable) components.



Some Bad JavaScript

/* Imagine there is a webpage with an element with class showMeTheInfo that when you click it must show some data that was passed in from the server on page load. */

```
$('.showMeTheInfo').click(function(event){
    var doctorHtml =
        '<div id="' + doctorId + '" class="' + doctorClass + '">' +
            (function() {
                var paras = '';
                for (var i = 0; i < docInfo.length; i++) {
                    paras += '<h3>' + docInfo [i].Title + '</h3>';
                    paras += '<p>' + docInfo [i].Info + '</p>';
                }
                return paras;
            })() + '</div>';
    $('.showMeTheInfoWrapper').append(doctorHtml ) //Psh, I don't even need semicolons
    DoFancyCalculation();
    alert('Done');
}
```



Part 1: A Mustache Template

Mustache provides very basic control flow such as conditionals and loops for traversing objects in a list.

```
<div id='{{DoctorId}}' class='{{DoctorStyles}}'>
    {{#DoctorInfo}}
        <h3>{{Title}}</h3>
        <p>{{Info}}</p>
    {{/DoctorInfo}}
</div>
```

Backbone Models

Models manage data/state and perform CPU intensive operations. Traditionally they are passed into the template to provide the variables in the template.

```
var Doctor = Backbone.Model.extend({
    defaults: {
        DoctorId: 1337,
        DoctorStyles: 'doctor-wrapper',
        State: 'noResult'
    }

    doFancyCalculation: function () {
        // Heavy lifting in progress...
        this.set({ state: 'Done' });
    }
});
```

Backbone Views

```
var DoctorView = Backbone.View.extend({
    events: {
        'click .showMeTheInfo': ‘render‘
    },

    initialize: function () {
        _.bindAll(this, 'render');
        this.template = templates['showMeTheInfo'];
        this.model.bind('change:state', alert(this.model.get(‘state’));
    },

    render: function () {
        this.$el.append(this.template.render(this.model.toJSON()));
        this.model.doFancyCalculation();
        return this;
    }
});
```

Cache container, and delegate events to it, not elements

Bind to changes in the model

Restrict DOM manipulation to the render method

New Code

```
var docModel = new Doctor({
        DoctorId: doctorId,
        DoctorClass: doctorClass,
        DoctorInfo: docInfo
});
var view = new DoctorView({
        el: $('.showMeTheInfoWrapper'),
        model: docModel
});
```

This is an incredibly trivial example but show's the modularity of Views and Models and the performance benefits.

Now What?

So we've written some super schweet modular backbone application. Great! Now let's test it. Why you ask?

1. How do we know it works correctly in all cases?

2. How do we stop other people from breaking it accidentally?

- Tradition test framework that can assert conditions and mock out function calls.
- When testing Backbone, traditionally we test models because they maintain state and perform the computationally intensive tasks.
- When the layout is especially important, testing the view is simple using some fun Jasmine plugins. The Jasmine Jquery plugin provides assertions to compare JQuery objects for equality/input state (is this checkbox checked), etc.

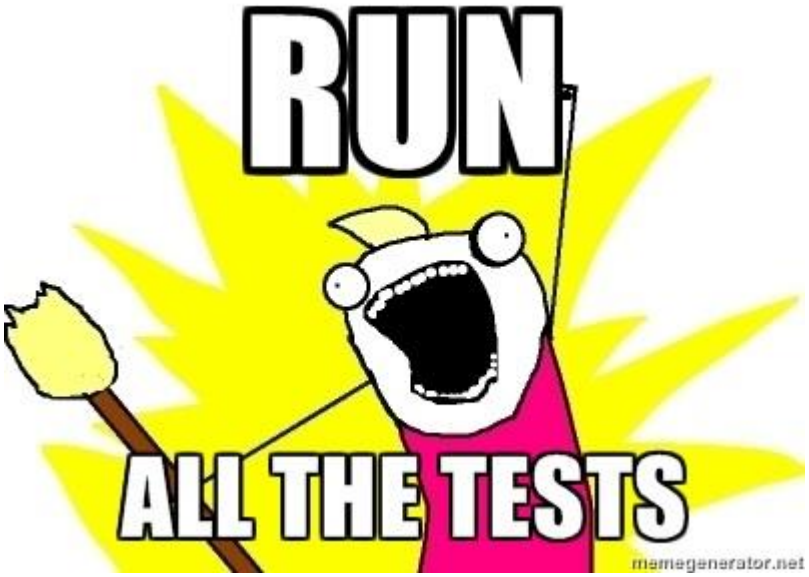
Jasmine

Jasmine 1.2.0 revision 1337005947

3 specs, 0 failures in 0.044s Finished at Fri Nov 30 2012 11:24:15 GMT-0500 (Eastern Standard Time)

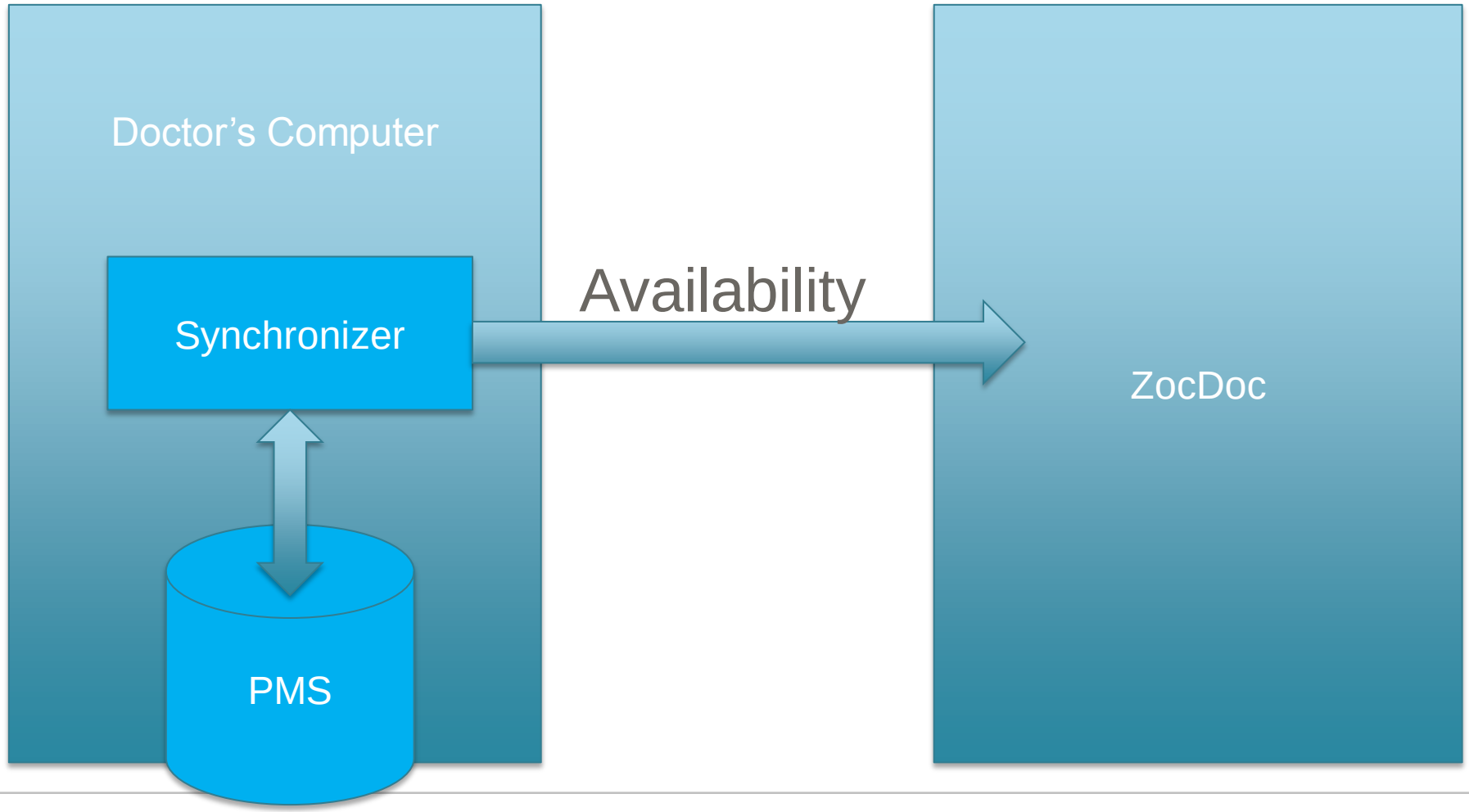
Example1 tests

- You can mock out simple fixtures
  - mess around with imaginary DOM elements
- You can import complex fixtures to test against
  - mess around with real DOM elements
- basic tests

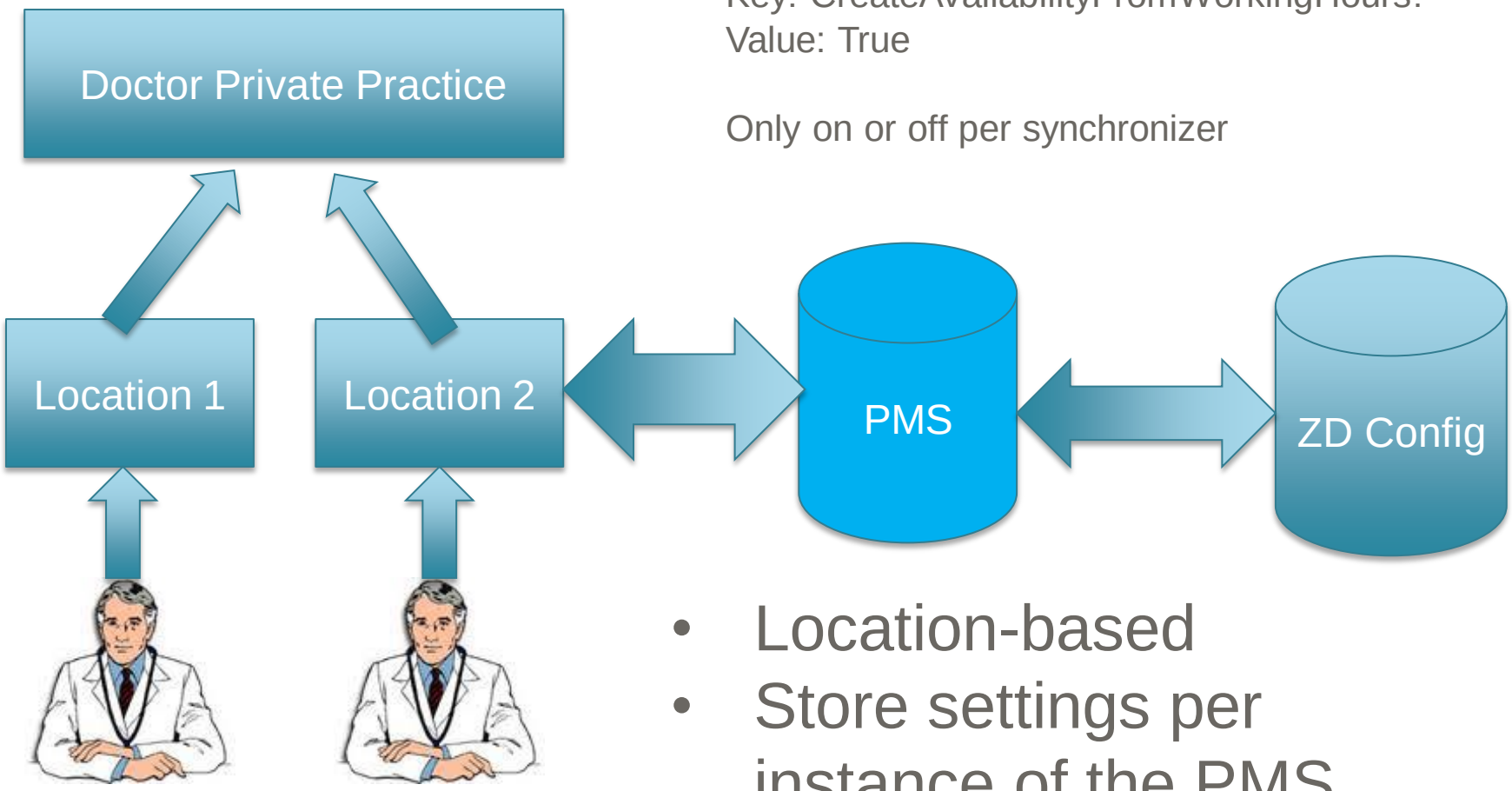


- Writing Maintainable JavaScript Using Backbone, Mustache & Jasmine
- **Highly Configurable Synchronization**

Synchronizing Schedules



Our initial solution



Key: CreateAvailabilityFromWorkingHours:
Value: True

Only on or off per synchronizer

- Location-based
- Store settings per instance of the PMS

Lessons Learned

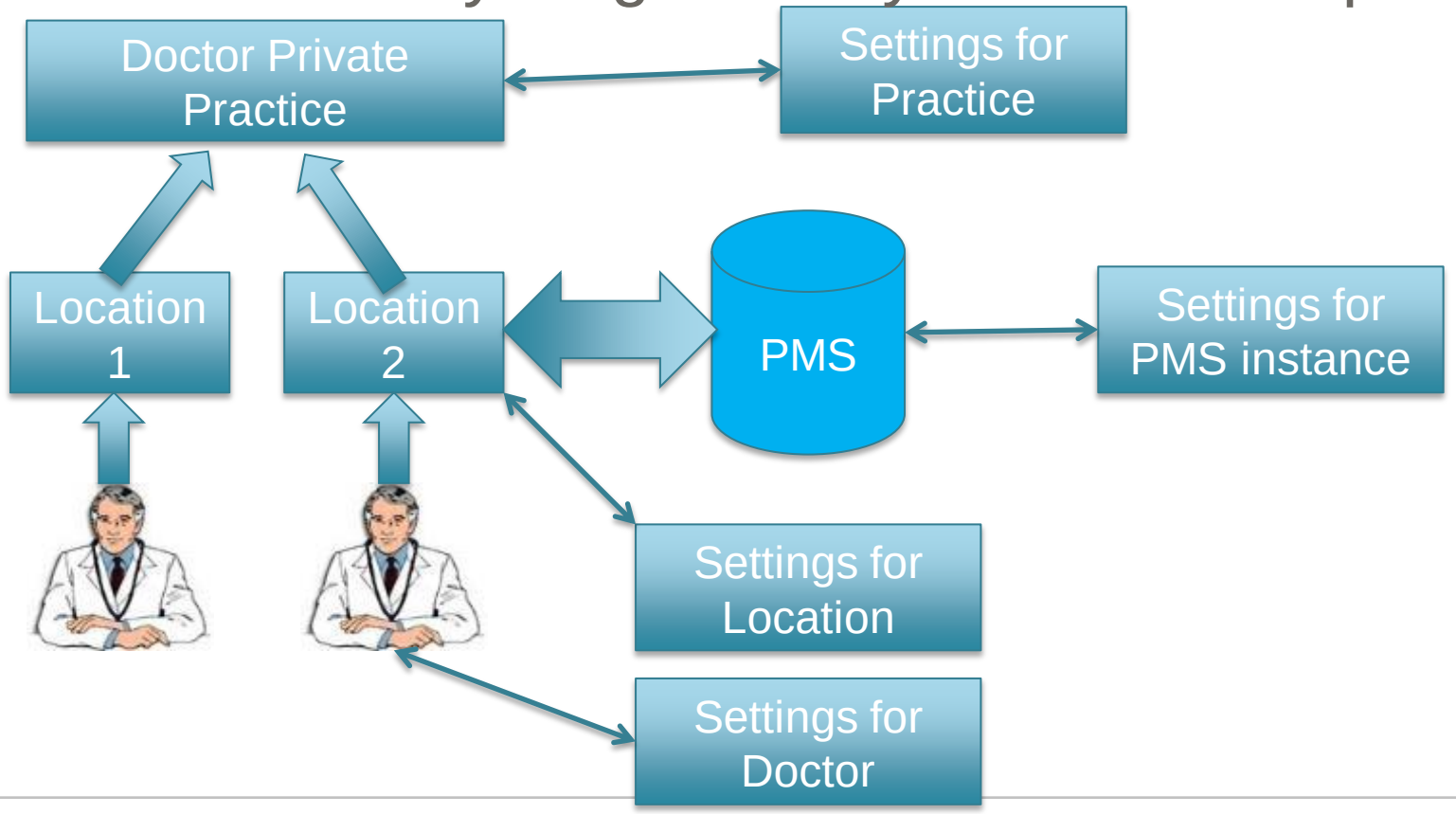
- Enterprise Clients (> 50 practices / locations / multiple PMS systems)
- Multiple practices on one PMS
- All doctors manage to use their system differently, even if it’s the same system

Lessons Learned

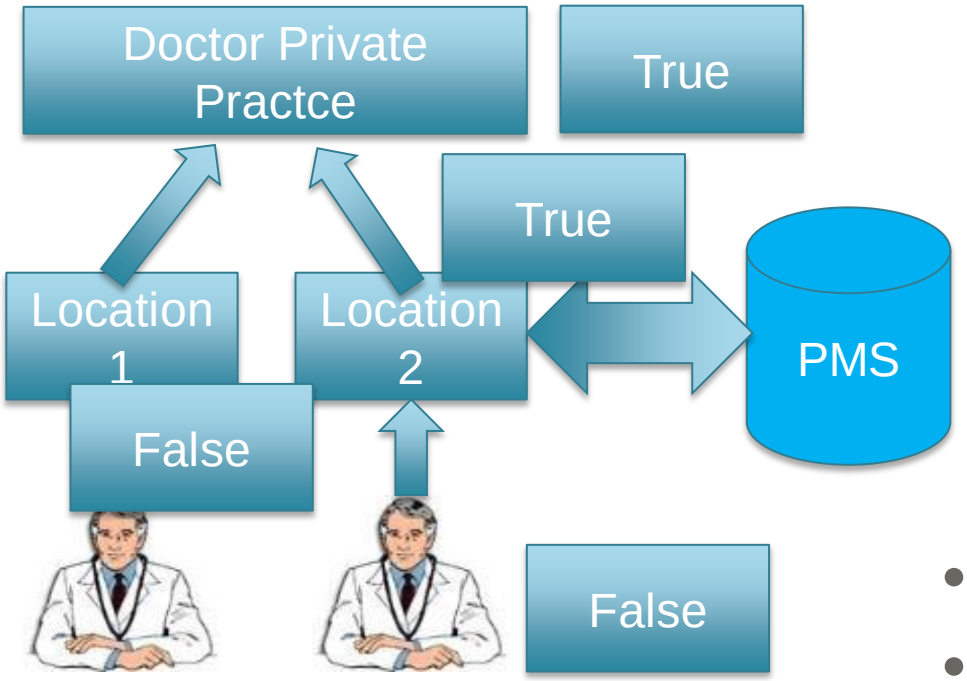
- Everyone does it wrong
- One-off settings for everyone
- People use the same data in different ways
- Short term fix => Key value pair of settings per ZD config

New Solution

- Create Graph of entity mappings
- Prioritize settings based on specificity
- Store everything in a key-value based pair



Example



- Key: CreateAvailabilityFrom WorkingHours

- For Practice: True
- For Location 2: True
- For Location 1: False
- For Doctor 2: False

What do we get?

- Different values for settings
- Adding a new setting is easy
- Set values for entire systems through inheritance