KEEPING YOUR APPLICATION SANE USING STATECHARTS

MY NAME IS JOE

BLOG: JOEGAUDET.TUMBLR.COM
TWITTER: @JOGAUDET
GITHUB: GITHUB.COM/JOEGAUDET

Who am I and why should you listen to me? Well I am joe, this is where you find me on the internet.

CTO AT LEARNDOT

Wednesday, 17 October, 12

I’m a founder at learndot, I’ve been working as the CTO for 2 years now.

BSCENG & (.95)MASC

I’m a computer engineer

GNAR ENTHUSIAST

Wednesday, 17 October, 12

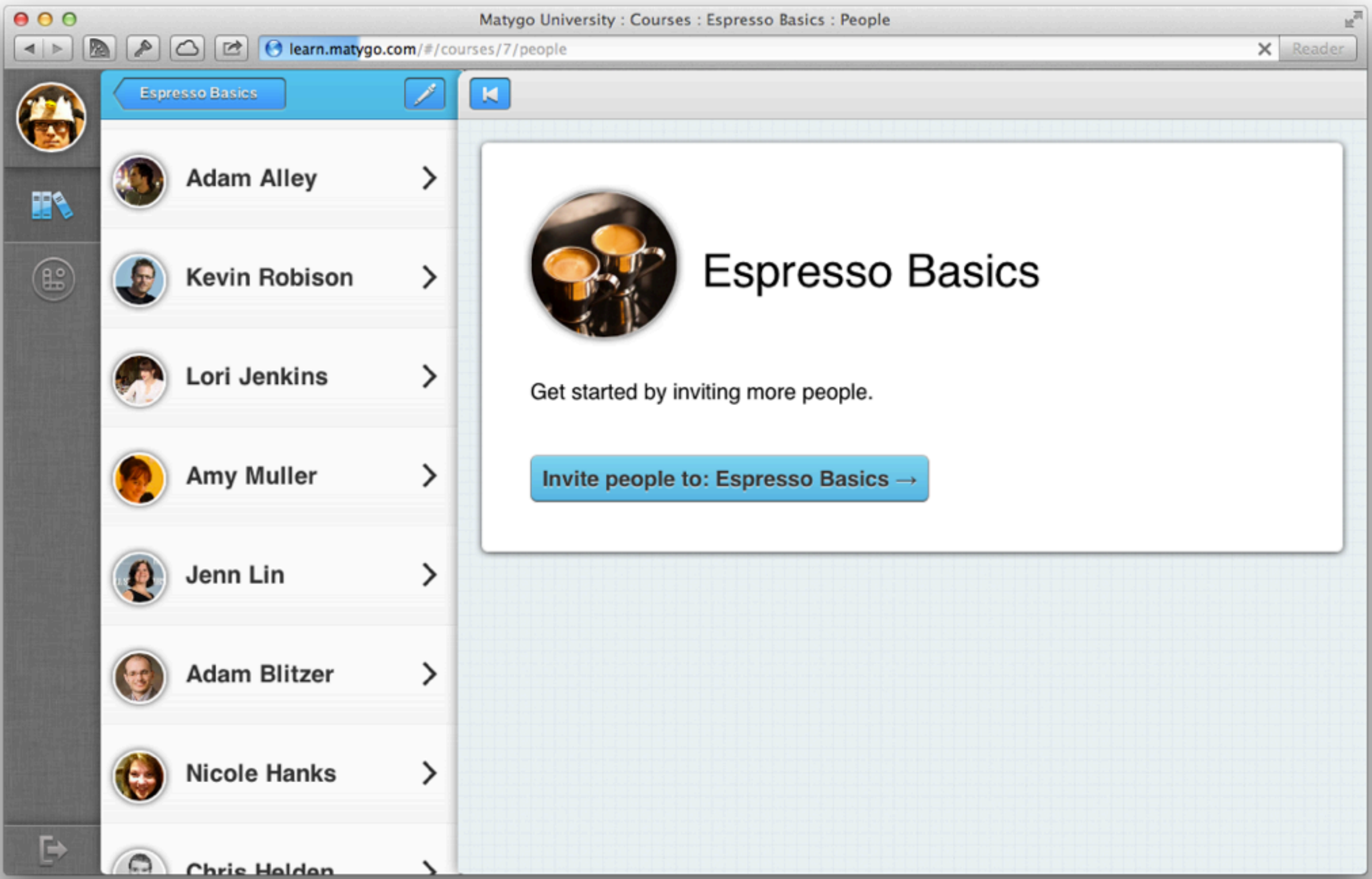
I love skiing.

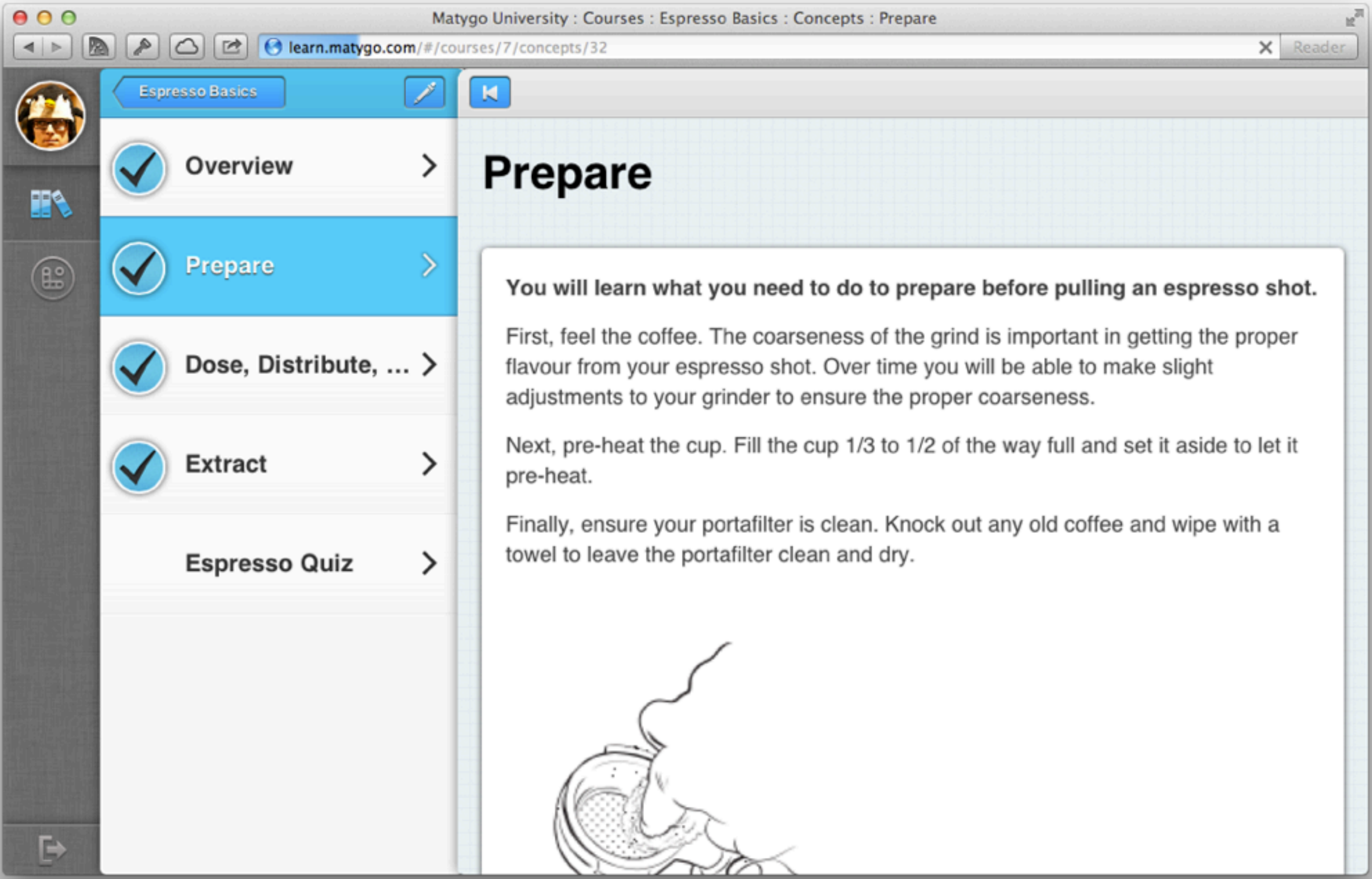
Wednesday, 17 October, 12

And making beer.

PROGRAMMER OF SPROUTCORE AND SCALA

But most importantly, I'm a programmer. At learndot we use SproutCore and Scala to deliver a corporate learning platform.


Matygo University : Courses : Espresso Basics : People
learn.matygo.com/#/courses/7/people
Reader
Espresso Basics
Adam Alley
Kevin Robison
Lori Jenkins
Amy Muller
Jenn Lin
Adam Blitzer
Nicole Hanks
Chris Holden
Espresso Basics
Get started by inviting more people.
Invite people to: Espresso Basics →


Matygo University : Courses : Espresso Basics : Concepts : Prepare
learn.matygo.com/#/courses/7/concepts/32
Reader
Espresso Basics
Overview
Prepare
Dose, Distribute, …
Extract
Espresso Quiz

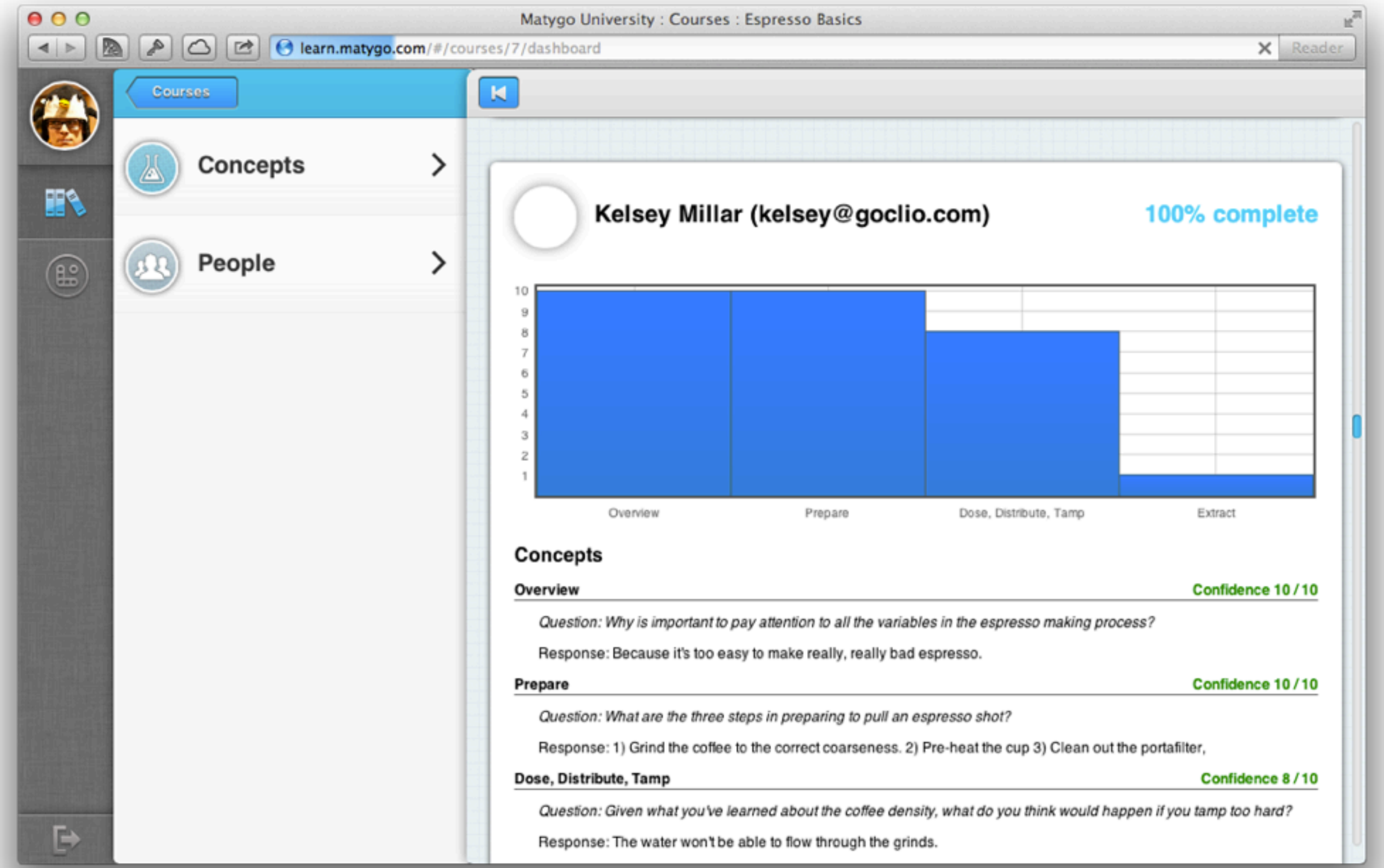
Prepare

You will learn what you need to do to prepare before pulling an espresso shot.

First, feel the coffee. The coarseness of the grind is important in getting the proper flavour from your espresso shot. Over time you will be able to make slight adjustments to your grinder to ensure the proper coarseness.

Next, pre-heat the cup. Fill the cup 1/3 to 1/2 of the way full and set it aside to let it pre-heat.

Finally, ensure your portafilter is clean. Knock out any old coffee and wipe with a towel to leave the portafilter clean and dry.


Matygo University : Courses : Espresso Basics
learn.matygo.com/#/courses/7/dashboard
Reader
Courses
Concepts
People
10
9
8
7
6
5
4
3
2
1
Overview
Prepare
Dose, Distribute, Tamp
Extract

Kelsey Millar (kelsey@goclio.com)

100% complete

Concepts

Overview Confidence 10 / 10

Question: Why is important to pay attention to all the variables in the espresso making process?

Response: Because it's too easy to make really, really bad espresso.

Prepare Confidence 10 / 10

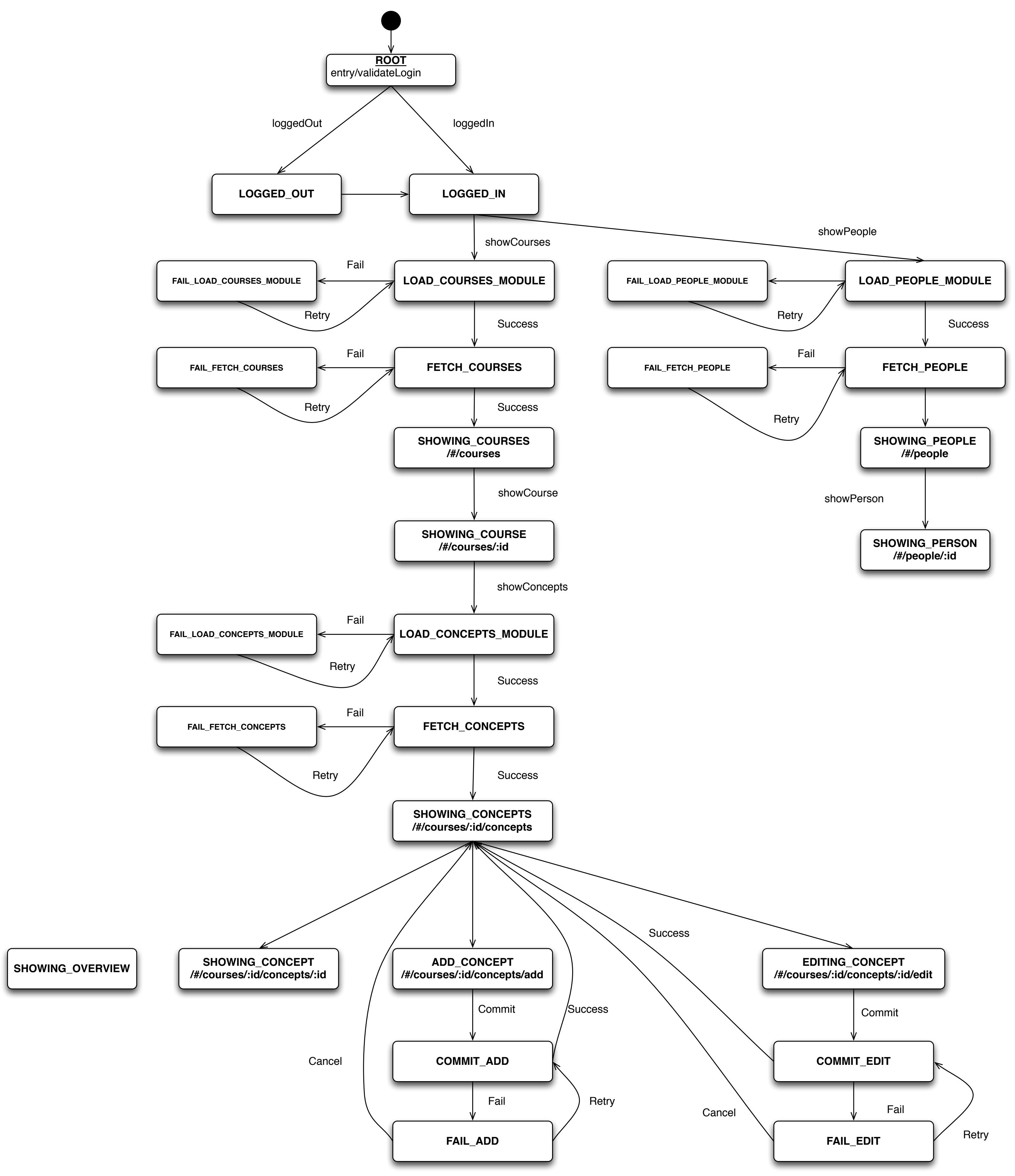
Question: What are the three steps in preparing to pull an espresso shot?

Response: 1) Grind the coffee to the correct coarseness. 2) Pre-heat the cup 3) Clean out the portafilter,

Dose, Distribute, Tamp Confidence 8 / 10

Question: Given what you've learned about the coffee density, what do you think would happen if you tamp too hard?

Response: The water won't be able to flow through the grinds.



And just cause this is a talk about statecharts, here’s a quick look at ours (this is a bit old, but you get the idea)

WHY STATECHARTS

MVC...

Wednesday, 17 October, 12

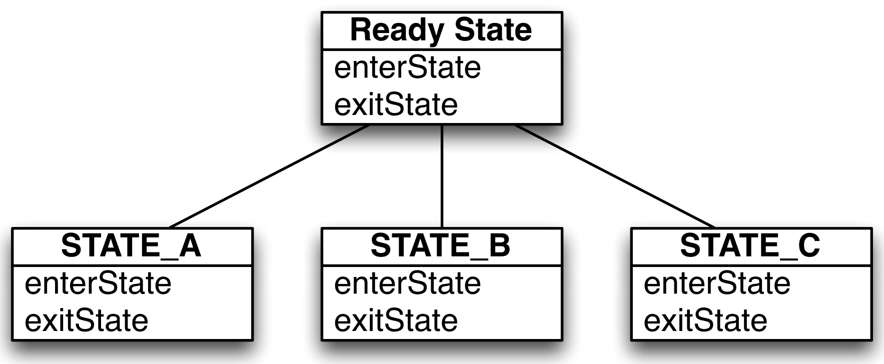
MVC is a great pattern that is very useful for organizing application logic. Each component has a well defined responsibility.

M-DATA
V-REPRESENTATION
C-INTERACTION

Those roles in brief....

This presents a problem where by application state is not covered by any of these components. It often ends up getting pushing into the controller layer, or managed in the main routine, or some other difficult to maintain construct.

STATECHARTS
MVCS



Statecharts solve this problem.

The statechart framework included in SC was written by Mike Cohen @frozencanuck. It organizes your application state into a DAG, which responds to events either sent from the SC view layer, or from your application. The statechart will then route the events to the current state (allowing any concurrent states to also answer the action).

Your application transitions from state to state, by calling gotoState with the name of the target state (in some circumstances you need a more explicit path).

When a state is entered or exited the enterState/exitState methods are called – usually this is where you append a view, or pane, fetch a record etc.

The Statechart framework is essentially the union of the State, Composite, and Chain of Responsibility patterns – and allows us to model our application as a well defined finite state machine.

GETTING STARTED

```
joe$ sc-gen app --statechart
```

- Generates a SproutCore application with the default responder set to the application Statechart

SproutCore supports statecharts out of the box, the sproutcore application generator has a flag to generate an application with a statechart.

You can also take an existing application and easily modify it to use a statechart.

MAIN.JS

```
var statechart = Example1.statechart;
SC.RootResponder.responder.set('defaultResponder', statechart);
statechart.initStatechart();
```

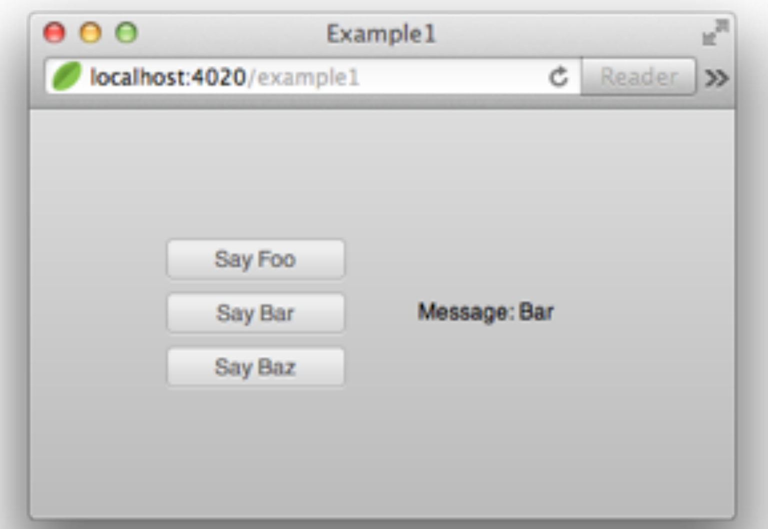
The relevant bit in main.js is shown here, setting the statechart to defaultResponder.

This makes the statechart the default target, preventing from having to set it explicitly on each view.

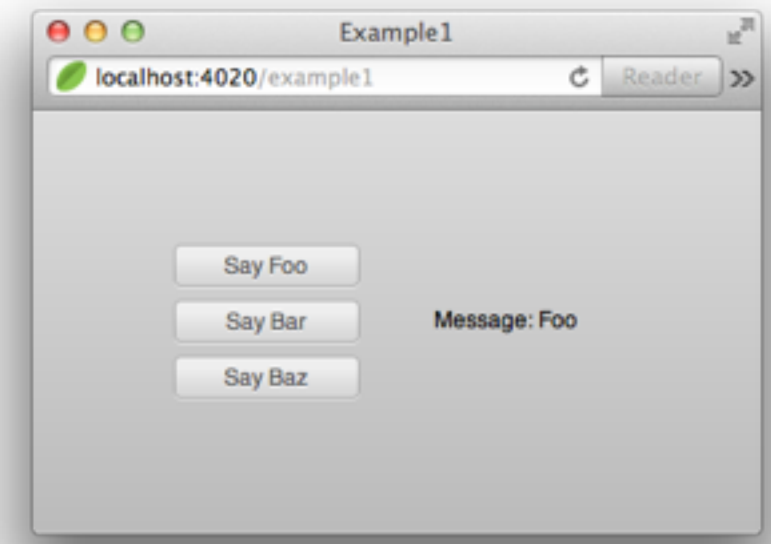
CODE EXAMPLES

EXAMPLE 1

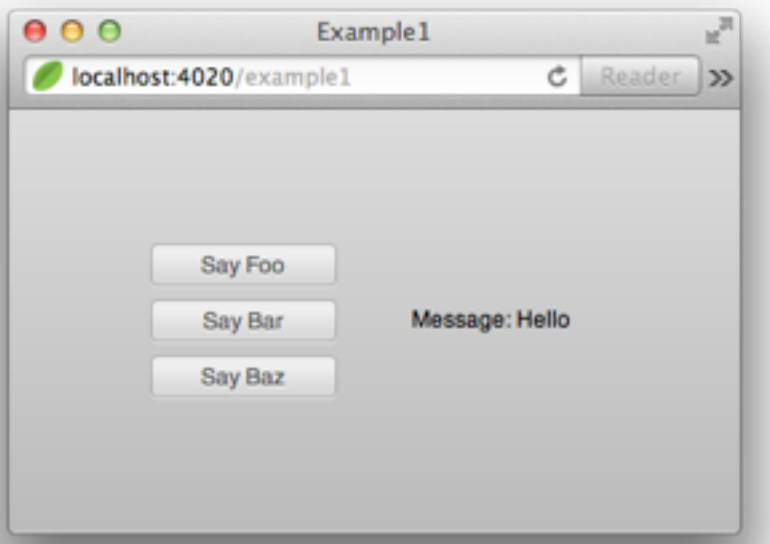
- Simple app with 1 state



Message: Bar



Message: Foo



Message: Hello

```
...
button1: SC.ButtonView.design({
        title: 'Say Foo',
        action: 'sayFoo',
        layout: {
              width: 100,
              height: 24,
              centerX: -70,
              centerY: -30
        }
}),
...
```

Wednesday, 17 October, 12

Simple app with one view, three buttons. Hello World... :P

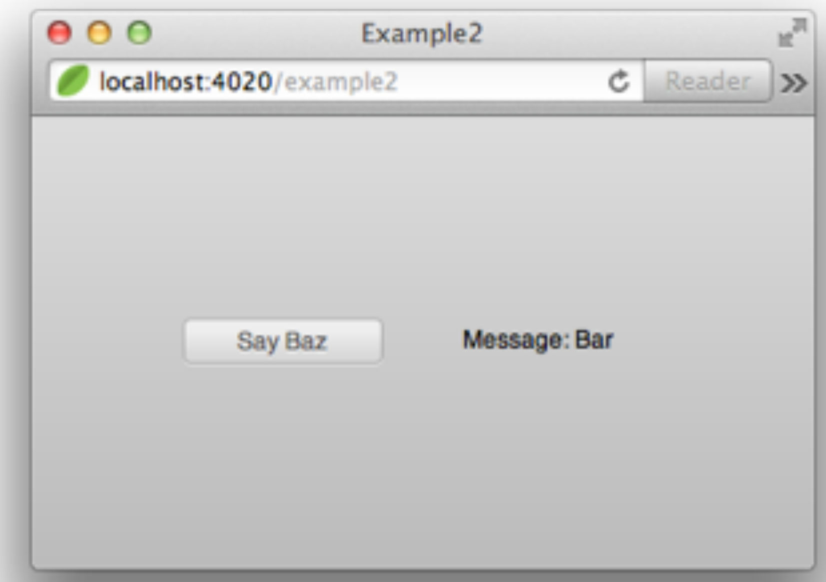
This example demonstrates the decoupling of the view from the application state, as shown in main_page.js the buttons have no explicit knowledge of what objects will execute their actions.

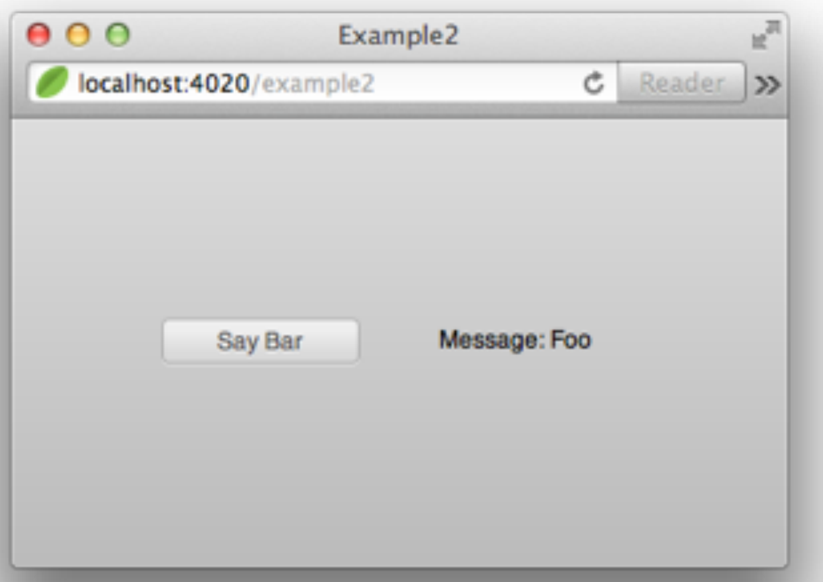
READY STATE

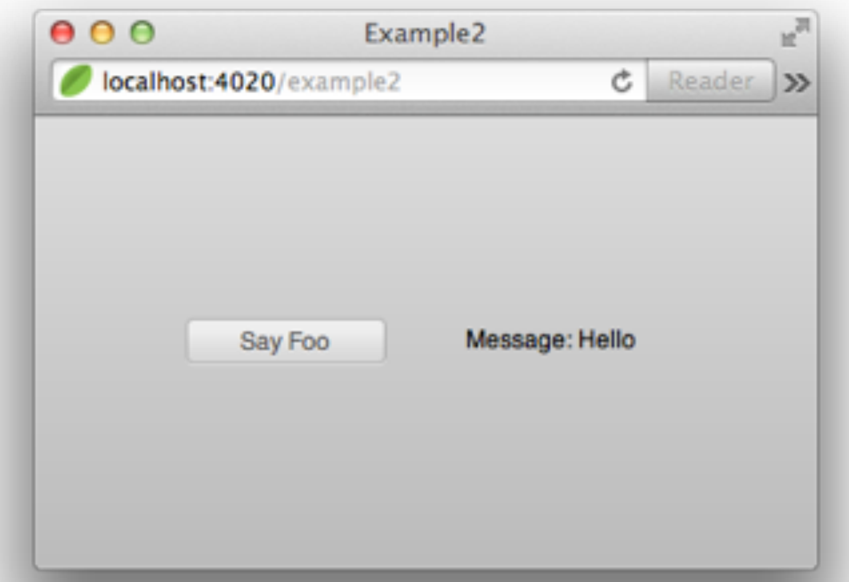
```
Example1.ReadyState = SC.State.extend({

     enterState: function() {
          Example1.getPath('mainPage.mainPane').append();
     },

     exitState: function() {
          Example1.getPath('mainPage.mainPane').remove();
     },

     sayFoo: function() {
          Example1.messageController.set('say', 'Foo');
     },

     sayBar: function() {
          Example1.messageController.set('say', 'Bar');
     },

     sayBaz: function() {
          Example1.messageController.set('say', 'Baz');
     }

});
```

EXAMPLE2

- Simple app with 3 states
- Collaborates with messageController to change text and button label







Wednesday, 17 October, 12

Each state answers the speak event, and transitions to the “next” state. When the “next” state is entered it updates the message controller, changing our view.

READY STATE

```
....

FOO_STATE: SC.State.extend({
      enterState: function() {
            Example2.messageController.set('say', 'Foo');
            Example2.messageController.set('title', 'Say Bar');
      },

      speak: function() {
            this.gotoState('BAR_STATE');
      }
}),

BAR_STATE: SC.State.extend({
      enterState: function() {
            Example2.messageController.set('say', 'Bar');
            Example2.messageController.set('title', 'Say Baz');
      },

      speak: function() {
            this.gotoState('BAZ_STATE');
      }
}),

BAZ_STATE: SC.State.extend({
      enterState: function() {
            Example2.messageController.set('say', 'Baz');
            Example2.messageController.set('title', 'Say Foo');
      },

      speak: function() {
            this.gotoState('FOO_STATE');
      }
})

 ...
```

CONSIDER

```
Example2.messageController = SC.ObjectController.create(
/** @scope Example2.messageController.prototype */
{

      title: 'Say Foo',

      say: "Hello World",

      message: function() {
            return "Message: " + this.get('say');
      }.property('say'),

      // Pre statechart method is gross

      states: {
            SAY_FOO: 1,
            SAY_BAR: 2,
            SAY_BAZ: 3
      },

      currentState: 1,
      speak: function() {
            if (this.currentState === this.states.SAY_FOO) {
                  this.currentState = this.states.SAY_BAR;
                  this.set('say', 'foo');
                  this.set('title', 'Say Bar');
            } else if (this.currentState === this.states.SAY_BAR) {
                  this.currentState = this.states.SAY_BAZ;
                  this.set('say', 'bar');
                  this.set('title', 'Say Baz');
            } else {
                  this.currentState = this.states.SAY_FOO;
                  this.set('say', 'baz');
                  this.set('title', 'Say Foo');
            }
      }

});
```

Consider a non statechart way to solve this problem.

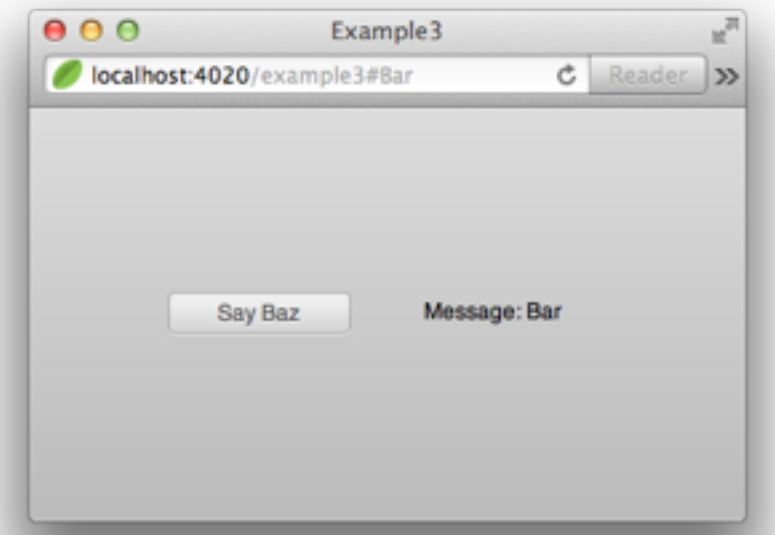
While in this trivial example it appears to be less code, the implementation is very brittle.

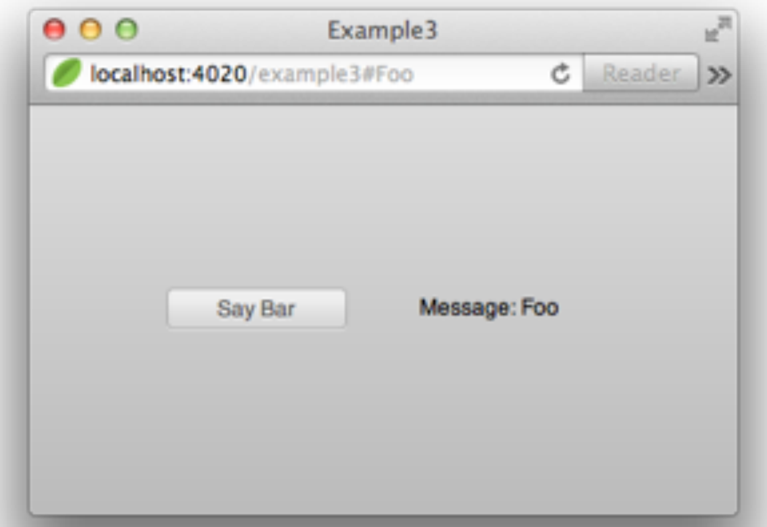
If we target the controller directly, it must resolve which action to take, which requires us to implement a state variable within the controller. There are a variety of problems with this approach, but the most disastrous is the distribution of application state logic through all over the place.

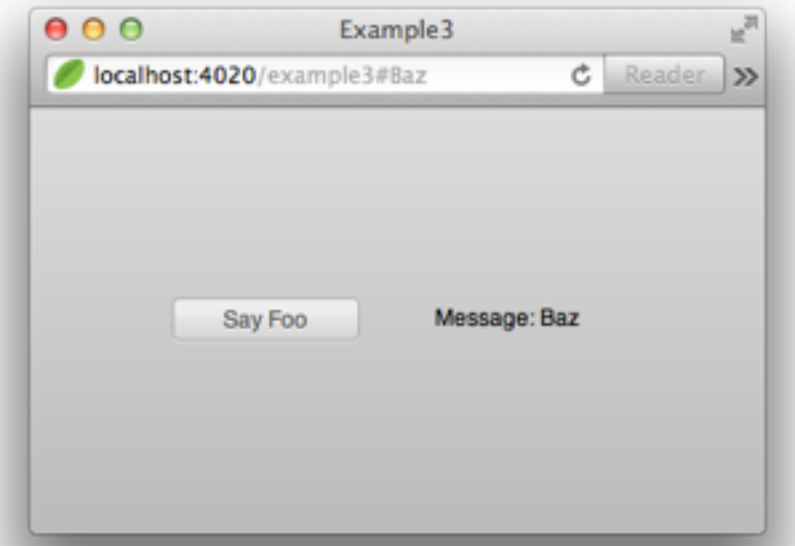
This makes testing difficult, and tracking down the source of problems, even more difficult

EXAMPLE3

- Same as example2 refactored, and with routing support.







MESSAGE STATE

```
Example3.MessageState = SC.State.extend({
     next: '',
     say: '',
     title: '',

     enterState: function(context){
       this._updateController();
       SC.routes.set('location', this.get('say'));
     },

     enterStateByRoute: function(context){
       this._updateController();
     },

     _updateController: function(){
       Example3.messageController.set('say', this.get('say'));
       Example3.messageController.set('title', this.get('title'));
     },

     speak: function(){
         this.gotoState(this.get('next'));
     }
});
```

Wednesday, 17 October, 12

In this example we make a class to handle some of the repeated code, setting controller values, and updating the location when the state is entered by way of gotoState.

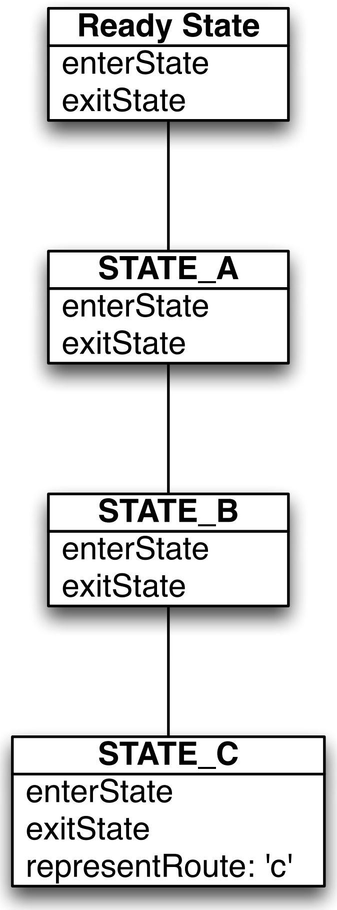
READY STATE

```
speak: function(){
  this.gotoState('FOO_STATE');
},

FOO_STATE: Example3.MessageState.extend({
  representRoute: 'Foo',
  say: 'Foo',
  title: 'Say Bar',
  next: 'BAR_STATE'
}),

BAR_STATE: Example3.MessageState.extend({
  representRoute: 'Bar',
  say: 'Bar',
  title: 'Say Baz',
  next: 'BAZ_STATE'
}),

BAZ_STATE: Example3.MessageState.extend({
  representRoute: 'Baz',
  say: 'Baz',
  title: 'Say Foo',
  next: 'FOO_STATE'
})
```

This example is somewhat trivial, but it demonstrates the way in which routes can be assigned to specific states, allowing for easy synchronization of the URL and the application state. Which gets us bookmarking + history.

EXAMPLE 4
ROUTES + GOTO

```
STATE_A: SC.State.extend({

      enterState: function() {
            console.log("Entered STATE_A");
            return this.performAsync('_someDelayedFunction');
      },

      _someDelayedFunction: function() {
            this.invokeLater(function() {
                  this._resume();
            }, 5000);
      },

      _resume: function() {
            var statechart = this.get('statechart');
            if (statechart.get('gotoStateSuspended')) {
                  statechart.resumeGotoState();
            }
      },

      STATE_B: SC.State.extend({
            enterState: function() {
                  console.log("Entered STATE_B");
            },

            STATE_C: SC.State.extend({
                  representRoute: 'c',

                  enterState: function() {
                        console.log("Entered STATE_C");
                        SC.routes.set('location', 'c');
                  },

                  enterStateByRoute: function() {
                        console.log("Entered STATE_C byRoute")
                  }

            })
      })
})
```



Simple 4 state app, the forth state is represented by route c (localhost:4020/example4#c). State a performs some asynchronous function, and returns 5 seconds later. The purpose of this example is to demonstrate how one can use many states to accomplish small tasks, and not forfeit the convenience of simple states.

EXAMPLE 4
ROUTES + GOTO

```
> Example4.statechart.gotoState('STATE_C');
Entered STATE_A                 ready_state.js:15
Entered STATE_B                 ready_state.js:34
Entered STATE_C                 ready_state.js:41
```

```
> // browser refresh
Entered STATE_A                 ready_state.js:15
Entered STATE_B                 ready_state.js:34
Entered STATE_C                 ready_state.js:41
```

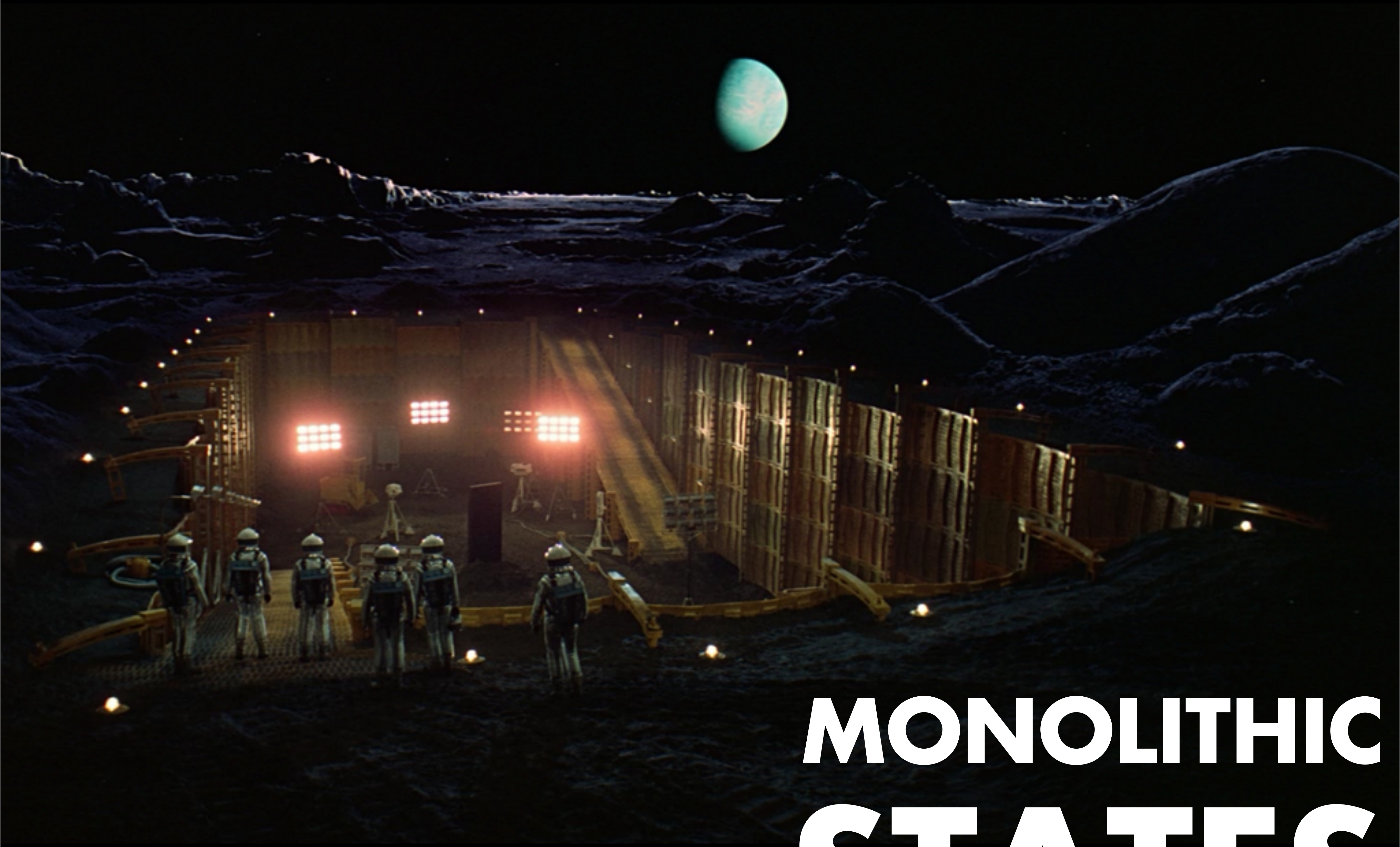
Wednesday, 17 October, 12

State C is the target in this case, which can be accessed by either changing the url to #c or by invoking gotoState, in both cases all intermediate states are entered, perform some task and continue. This pattern is very powerful and can increase code reuse all over the place, and improved testability.

WATCH OUT!

CONCURRENT STATES

If you have two concurrent states, that respond to the same action. You may be surprised when both methods are called. This is less of a pitfall, and more of something to just keep an eye out for.



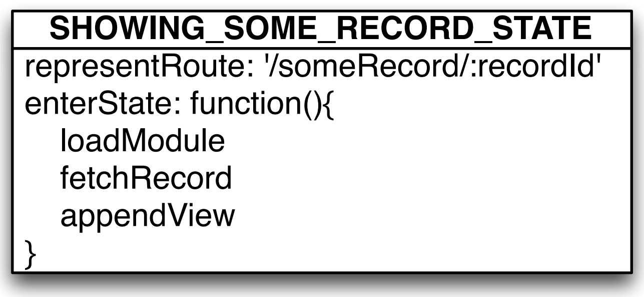
MONOLITHIC STATES

MONOLITHIC STATES

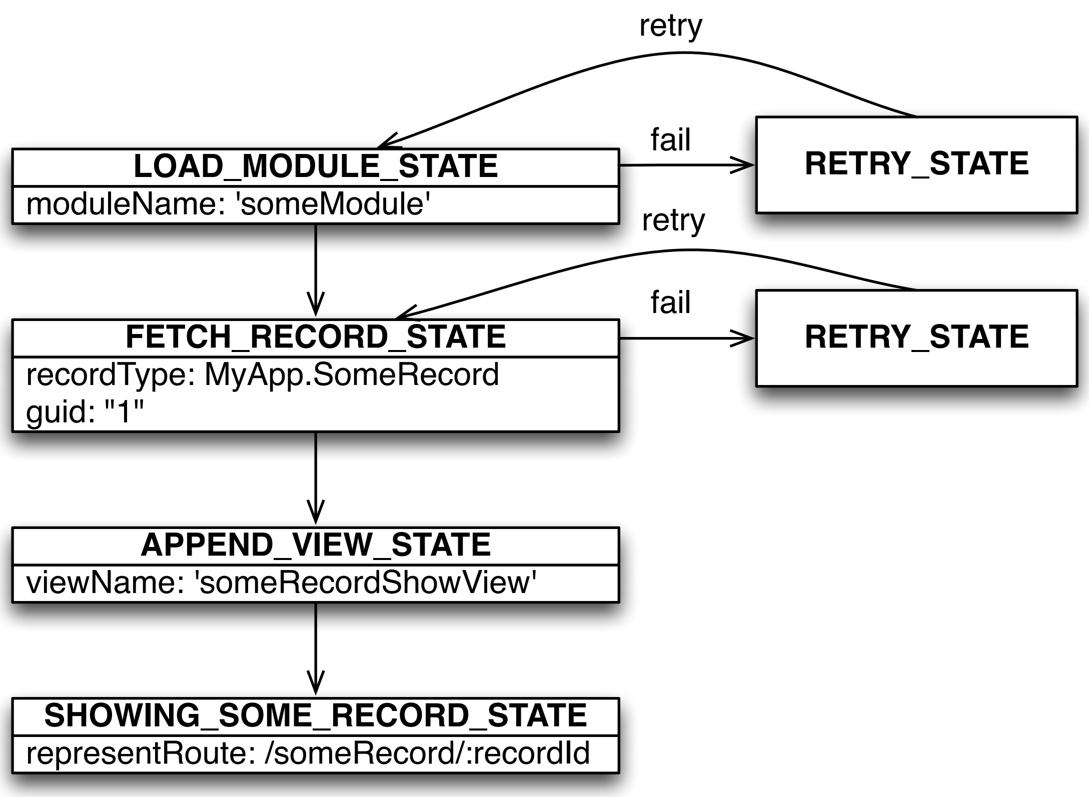
- As with all things OO, favor composition over inheritance.
- Use small re-usable / configurable states.

When you first get going with statecharts your inclination is to create large states that manage a lot of complexity. This can quickly become untenable - a better approach is to use very small single purpose states.

REFACTOR



SHOWING_SOME_RECORD_STATE
representRoute: '/someRecord/:recordId'
enterState: function(){
loadModule
fetchRecord
appendView
}



Wednesday, 17 October, 12

Here is a naive example from a refactoring I did earlier in the year. Originally the SHOWING_SOME_RECORD_STATE was responsible for loading the module that was associated with showing that record, fetching the record, appending the appropriate view to the app, and handling the routing.

The refactored solution breaks the monolithic state into what appears to be a more complex solution (in terms of moving parts). However the small states are configurable and reused throughout the system, keeping our code dry – additionally the small single responsibility objects can be unit tested in isolation.

WANT A JOB?

JOE@LEARNDOT.COM
PAUL@LEARNDOT.COM