10 months with **Sproutcore**

the view from the inside

Thursday, March 10, 2011

Welcome Slide

THE PLAN

- Personal Intro
- Part 1: A tour of Sproutcore
- Part 2: Lessons Learned
- Recommendations
- Resources

Thursday, March 10, 2011

Part 1: No prior experience assumed aside from familiarity with javascript. Show some examples, give a quick intro to the major features

Part 2: Jump head first into the major hurdles we encountered with the framework and share our solutions.

Recommendations: When does Sproutcore really shine? When should you avoid it?

Resources: Links, blogs, etc..

- Paul Lambert, co-founder of Matygo
- Developing hosted E-Learning technology, Sproutcore App currently piloting with 4 UBC courses.
- Full time since May 2010. Client-side exclusively Sproutcore

Thursday, March 10, 2011

Paul: Graduated from UBC CS in May 2010

Joe: Currently MApSC student at UBC, graduating this Spring.

Both were primiarily Rails developers before this project.

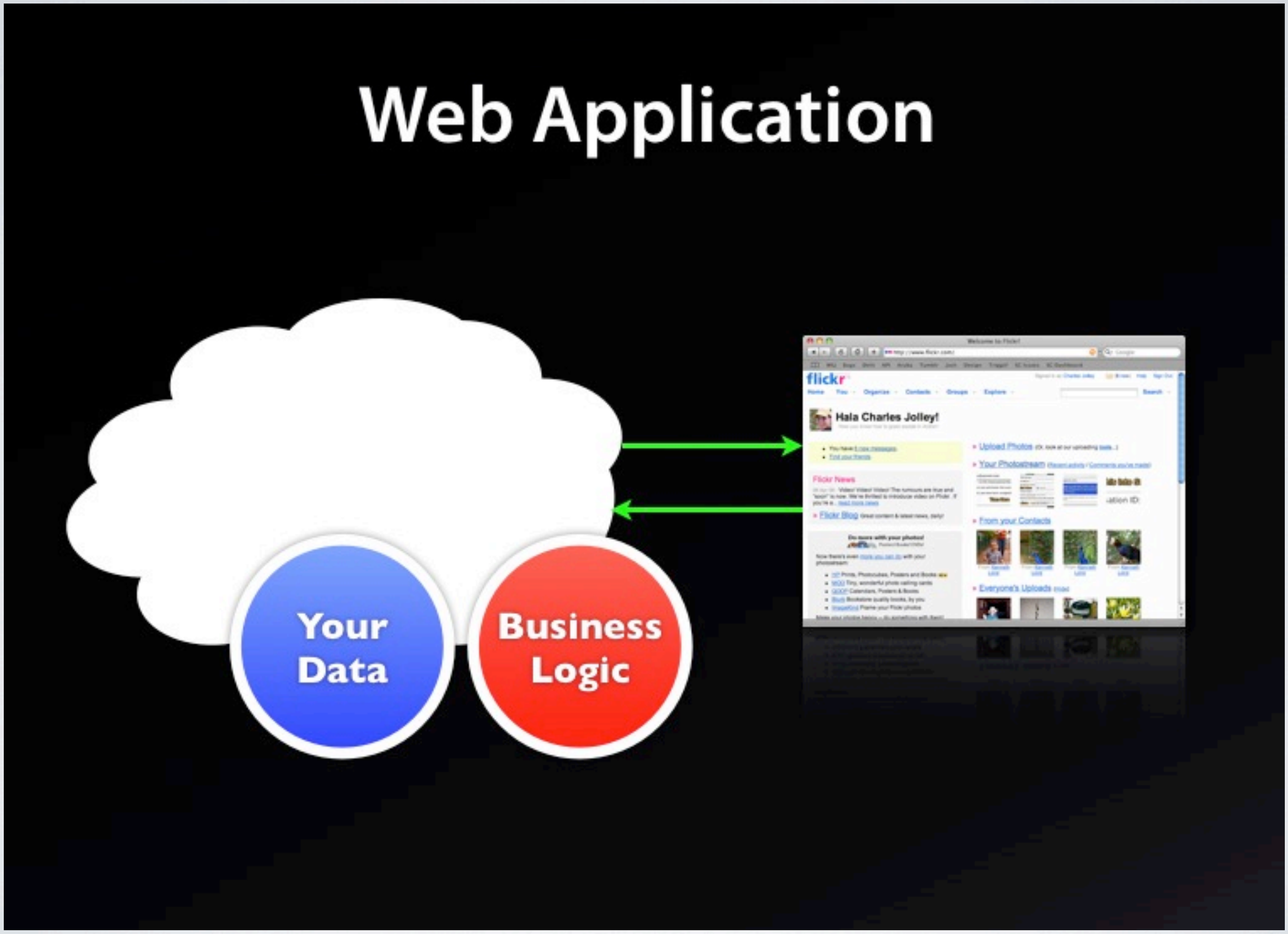
SPROUTCORE

- Thick Client Javascript Framework for "Cloud Apps"
- Classical (class-based) Object Model w/ Ruby-style Mixins
- Key-Value Observing (KVO)
- MVC + Datastore

Thursday, March 10, 2011

These are the main points we will be talking about in our 'tour of sproutcore' as part 1 of the talk

- "Cloud Apps" Desktop-style feature rich apps.



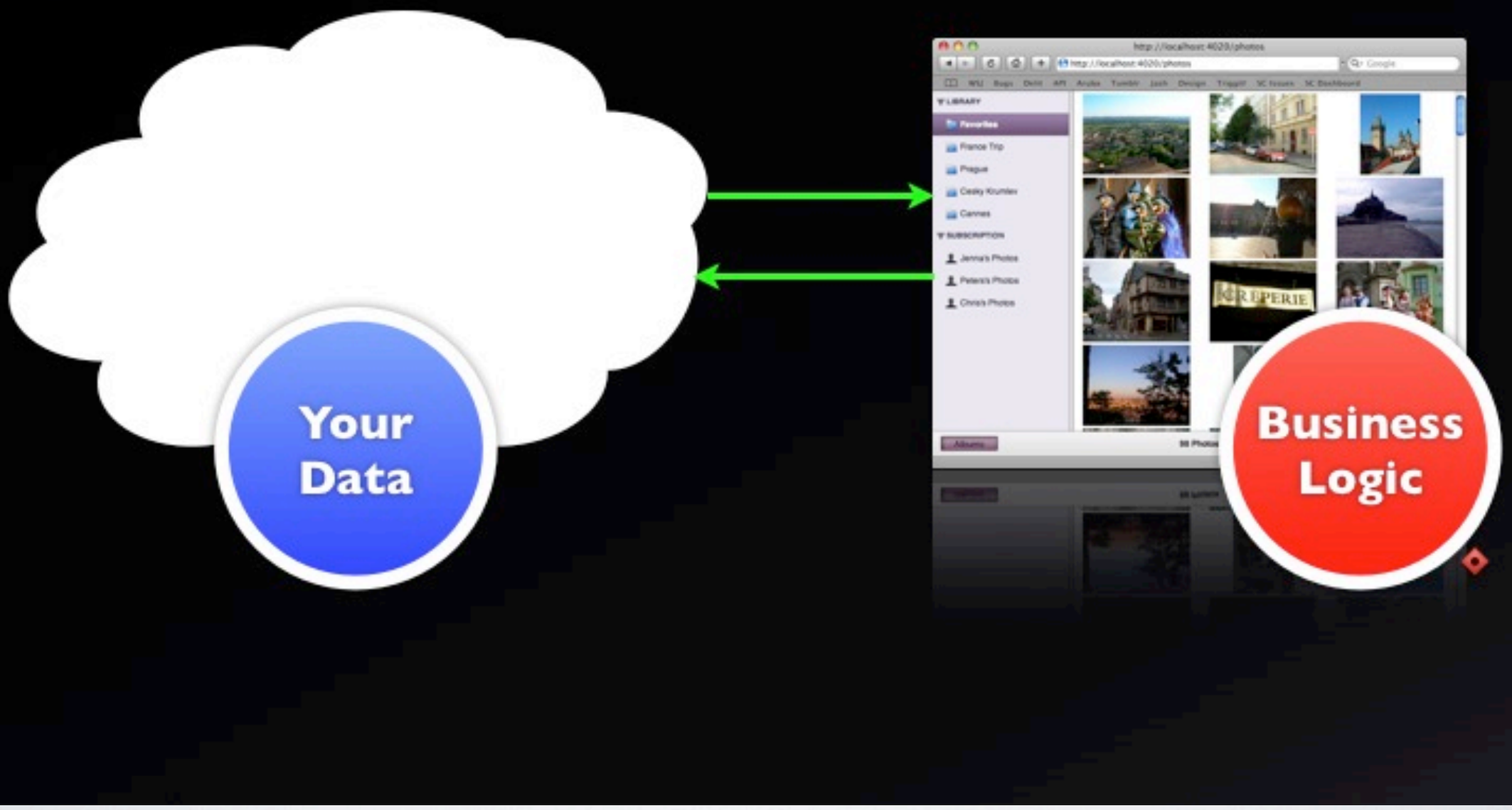
So what do they mean by ‘cloud app’

- In traditional web apps, data and logic lives on the server. Client is just presentation and pzzazz.

Desktop Application

Desktop apps a more immersive, responsive, and have all the logic and data locally.
Unfortunately they also share the downside on being non portable, difficult to upgrade, etc...

Cloud Application



Cloud Apps are the synthesis of these two approaches

SC EXAMPLES

- iWork.com (Apple)
- Eloqua (Marketing Management)
- MobileMe (Apple)
- NPR webapp in the Chrome Web Store
- The Matygo webapp (yours truly)

NPR was done by Strobe Inc., the main organization behind the development of Sproutcore. Founded by Sproutcore's original author Charles Jolley after he left Apple in mid 2010.

NPR: Show audio framwork, URLS routing

Matygo: Show windowing, multitasking, course controls

CLASSICAL OO

- Create classes with .extend. Single inheritance, SC.Object at base of hierarchy
- Instantiate with .create
- Behaviour can be easily composed using mixins

Thursday, March 10, 2011

- Sproutcore provides a class-based object orientation model that feels like language-level enhancements.

- Everything in Sproutcore inherits from SC.object which mixes in SC.Observable (note: verify this) and provides the core features like Key-Value observing, and this class model.

```
MyApp = {};
MyApp.Animal = SC.Object.extend({
    // properties
    name: "Annie",

    // constructor
    init: function(){
        sc_super();
        this.sayName();
    },

    // methods
    sayName: function(){
        console.log("An Animal named " +
                this.get('name'));
    }
});

// A Mixin
MyApp.Barkable = {
    bark: function(){
        console.log("Woof woof");
    }
};
```

```
// Create an Animal, mixin Barkable
// and override the sayName method
// so it takes advantage of newfound
// barking ability
var dog = MyApp.Animal.create
(MyApp.Barkable, {
    // override
    name: "Rover",

    // override
    sayName: function(){
        sc_super();
        this.bark();
    }
});

// > I am an Animal named Rover
// > Woof woof
```


- Note that mixins are just javascript hashes,
- Create and extend both takes any number of hashes as their bodies, which can be seen a ‘mixin literals’.
- For example, mixins in both Barkable and the ‘mixin literal’ that overides name and sayName

KEY-VALUE OBSERVING

- Observer pattern central to Sproutcore
- App connected through observers and bindings to object properties
- Syntax for creating bindings via Property Paths & Observes on methods
- Computed properties: Annotate methods so they behave as properties
- Use .set and .get to access properties. (almost) never dot syntax.

Thursday, March 10, 2011

(trim the text on slides, move some down here)

The KVO is fabulous when it works, it eliminates a huge amount of ‘glue code’ and can lead to very lean apps. The consistency is also very refreshing and simplifies the programming model: you connect your views, data, and logic through bindings and observers. period.

The downside is it can lead to difficult debugging (stack traces aren’t particularly useful when methods are called/triggered indirectly at the end of a runloop) and can create performance problems if used too willy nilly (two way bindings, multiple triggers, etc...)

```
MyApp = SC.Application.create();

MyApp.Frank = SC.Object.create({
    petsName: "Rover"
});

MyApp.FranksDog = SC.Object.create({
    // create a (two way) binding between properties
    nameBinding: "MyApp.Frank.petsName",

    // create a computed property
    nameLength: function(){
        return this.get("name").length;
    }.property(),

    // create a method that observes FranksDog's
    // name property (is triggered whenever name changes)
    echoName: function(){
        console.log("My name is " + this.get('name') +
                " " + this.get("nameLength"));
    }.observes("name")
});

MyApp.FranksDog.echoName();
// > My name is Rover 5

MyApp.Frank.set("petsName", "Rufussius");
// > My name is Rufussius 9
```

MVC + DATASTORE

- A Model-View-Controller framework, similar to Rails.
- A 'whole sink' framework - significant infrastructure from datastore to UI components
- Views bind to properties on controllers
- Models (SC.Record) map relationships and mirror the server side schema
- Datastore manages interaction with server and sits 'below' model layer.

Controllers either 'pass through' bound data directly to models or perform transformations on the data

Datastore responsible for updating synchronizing model data and keeping models in valid sates. (could show model state diagram from wiki http://wiki.sproutcore.com/w/page/12412876/DataStore-About%20Records)

Will show example of this in Stickies

COMPILATION

- Each app is compiled to a single compressed js file, a single css, and a single html file. Entirely server agnostic, a simple app could be entirely statically served.

- Modules are included via sc_require a compiler macro indicating dependencies between source files.

- Development server compiles reloads on the fly

- The libraries are included separately actual page load involves 5-10 files.

- sc_require can be a bit of a burden: more on that later.

- Development server sends multiple uncompressed source files for simpler debugging.

- Smallish apps that could be entirely statically served can still persist data through the HTML5 localstore API (imagine a shopping list app, for example).

STICKIES

Let's see some code! Quick 'Full App' example

Pulling out the desktop sticky logic from Matygo and will quickly tour through a 'full app' that allows users to create sticky notes, reposition on the dashboard, and be persisted. Modelled on OS X dashboard stickies.

PROBLEM: OBSERVER CHAINS

- Can have very long property chains that are onerous and brittle.
- Example".parentView.parentView.parentView.Foo" in a view

SOLUTION: USE/ABUSE INIT()

- inside init can use this.foo = bar; before calling sc_super for equivalent results of statically laying out foo: bar,
- Benefit: nested views are in the root view scope, can refer to root bindings directly in their closure.

Thursday, March 10, 2011

- Given example, if you changing the nesting level of the view the app breaks.
- If important data is bound to 'root' view, calling multiple parentViews are the only way to obtain a reference to the root view and thus the data.

- Simple technique to moving properties into init and really cleaned up our code. pass in property path to root data as a string variable in scope and views, at any level of nesting, can bind directly to the data.

Result: more robust & flexible construction and cleaner, less reptitive code.

Important: Much of this may be reduced in the sproutcore 1.5 with SC.TemplateView

```
// Problem: brittle binding chains
Matygo.EditAnnouncementView = SC.ScrollView.extend({
    controllerBinding: this.controllerPath,
    contentView:  SC.View.design({
        childViews: ['form'],
        form: SC.View.design({
            childViews: "subjectValue".w(),
            subjectValue: DarkHorse.MatygoFormTextField.design({
                labelValue: 'Subject',
                // WTF??
                valueBinding: '*parentView.parentView.parentView.controller.editSubject',
                errorValueBinding: '*parentView.parentView.parentView.controller.editSubjectError
            })
        })
    })
});
```

```
// solution: get controller path into a var
// May not be less code, but more readable & robust to change
Matygo.EditAnnouncementView = SC.ScrollView.extend({
    init: function() {
        var controllerPath = this.controllerPath;  // BINGO

        this.contentView = SC.View.design({
            childViews: ['form'],
            form: SC.View.design({
                childViews: "subjectValue".w(),
                subjectValue: DarkHorse.MatygoFormTextField.design({
                    layout: { ... },
                    labelValue: 'Subject',
                    valueBinding: controllerPath + '.editSubject',
                    errorValueBinding: controllerPath + '.editSubjectError'
                })
            })
        });
        sc_super();
    }
});
```

SINGLETON CONTROLLERS

- Sproutcore assumes that views are bound to singleton controllers than have a statically known property path.

- We have windowed multitasking, thus # of views and controllers unknown ahead of time

SOLUTION

- create a controller registry that creates and registers controllers on the fly.

- Have unique property paths to controllers based on numbering scheme, that can be passed to views at runtime.

Thursday, March 10, 2011

- Your controller definitions go from being .creates to .extends

```
Matygo.NestedController = SC.Object.extend({

    // Registers a controller with this NestedController
    registerController: function(name, controller){
        this.set(name, controller);
        return controller;
    },

    // protect against multiple controller instantiation for the same name
    registerControllerIfUnregistered: function(name, controller){
        var currently = this.get(name);
        if (currently) controller = currently;
        else this.registerController(name, controller);
        return controller;
    },

    //  Removes the controller with this name
    unregisterController: function(name){
        this.set(name, null);
    }
});
```

```
// Class for all offering controllers
Matygo.OfferingsController = Matygo.ListViewerBaseController.extend({
    controllerId: 0,
    controllerBasePath: function(){
        return "Matygo.offeringsController." + this.get('controllerId');
    }.property('controllerId')
});

// singleton registry for offering controllers
// bath of path to all offering controller instances
Matygo.offeringsController = Matygo.NestedController.create({
    offeringCounter: 0,
    spawnNewView: function(){
        var controllerId = "controller" + this.get('offeringCounter');
        this.incrementProperty('offeringCounter');

        var controller = this.registerController(controllerId,
              Matygo.OfferingsController.create({controllerId: controllerId}));

         var offeringsAdapter = (..);
         controller.set('content', offeringsAdapter);

        var path = controller.get('controllerBasePath');
        var view = Matygo.CoursesView.create({
              controllerPath: path
        });
        return view;
    }
});
```

MODULE DEPENDENCIES

- Managing the sc_requires can be onerous and mistakes can lead to strange bugs

SOLUTION

- wrote a custom script to parse our code and plunk in the right sc_required.
- available: https://github.com/joegaudet/SCRequireProcessor

Thursday, March 10, 2011

We periodically run this script which works quite well and has made this entirely a non-issue

Open source on github (joegaudet)

OTHER DIFFICULTIES

- No URLS? hidden from search engines, breaks back button
- Datastore doesn't support nested REST routes
- Performance and Browser support issues

Thursday, March 10, 2011

URL solution partially addressed using SC.Route

REST routes: for example, if a discussion has a topic, to update it would be a natural PUT to /discussion/:d_id/topics/:t_id. But the datastore updateRecord() call has no idea 'who' triggered the record's updating, and cannot construct this URL naturally. You must either interrogate the model which is an abuse of layer separation (if it has the information) or if the model doesn't have the information, create a hack around or live with uglier URLS.

Browsers: We have performance problems in even the latest firefox, especially with the animation framework. IE before 9? Good luck.

RECOMMENDATIONS

- Great for thick client web apps. Provides essential structure and libraries.

- IF:

- Your product is naturally more an ‘app’ than a ‘site’

- Your users all have modern browsers (preferably WebKit)

- Having all your data easily accessible to search engines isn’t critical

Things we like: simple programming model, great UI component toolkit, active and helpful community

- Obviously these are just my experiences. As always, your mileage may vary.

RESOURCES

- Project home: http://www.sproutcore.com/
- Sproutcore Wiki: http://wiki.sproutcore.com
- Sproutcore Guides: http://guides.sproutcore.com/
- Google Group: http://groups.google.com/group/sproutcore
- #sproutcore on freenode
- Source: https://github.com/sproutcore/sproutcore

QUESTIONS?

Thank you!

presented by

Paul Lambert

http://www.paulitex.com