Sencha ExtJS MVC

SevOne

Crash Course

Presented by Brandon Ryall

Why would we use an MVC architecture?

As our development team grows, we have an even greater need for more efficient, maintainable, and readable code. Overall it will really begin to improve the way we think about the creation of our projects and the methods we go about architecting them.

- multi-developer teams can edit files without many conflicts or modifying similar files
- code is divided into logical parts that should make following work-flow easier
- time to finding and resolving syntactical errors is reduced

Why would we avoid an MVC architecture like the plague?

When you bring up the discussion of heading towards an MVC architecture, people often want to know both sides, when to use it and when not to use it. Unfortunately in my opinion, there aren't many reasons you wouldn't want to use the architecture.

So I pulled these out of the sky:

- Your application doesn't really deal with data. No data means no model. And of course with no model, you have no need for a controller. Have you ever heard somebody say "we should start using the V architecture?" No, no you have not.
- Another reason would be the scale of your application. If your application is so un-advanced technologically and can only be broken up into 1 view - then don't you dare use something as awesome as the MVC architecture.

Understanding The Structure

When dealing with an MVC architecture, the structure of your file system will almost always begin with the most common structure:

- **Model** - *How we represent our data.*
- **View** - *How we visualize the data we have within our model.*
- **Controller** - *The actions we want to take within the application, typically against our model.*

Understanding The Structure Part Deux

However, in ExtJS we add 1 more directory to the structure to complete our build.

- **Model** - *How we represent our data. In ExtJS, we create data stores and utilize a CRUD proxy within the model.*
- **Store** - *An ExtJS data store that can hold our data locally.*
- **View** - *How we visualize the data we have within our model.*
- **Controller** - *The actions we want to take within the application, typically against our model.*

Where To Begin

When a front end developer is handed a project, typically the first thing they will do is begin mocking up (AKA the best phase ever because you get to scribble) their future baby. Said front end-er would usually go on to begin programming immediately after.

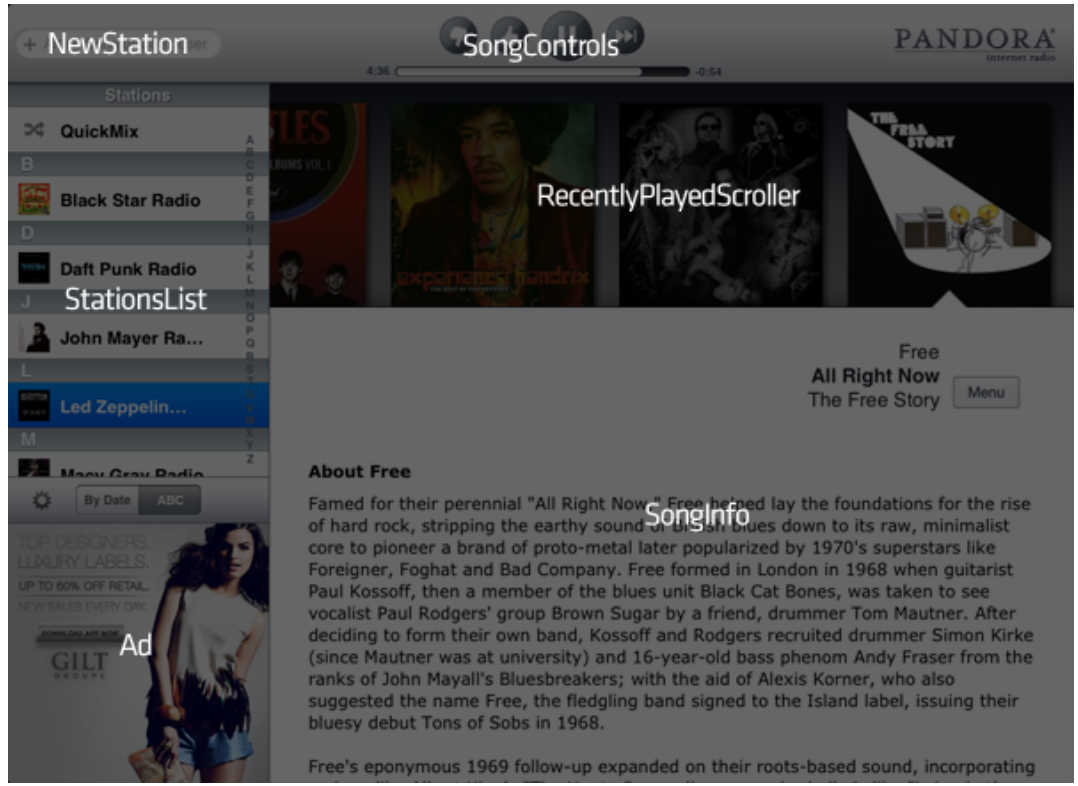
HOLD THE PHONE BILLY! Now that we know about this marvelous MVC architecture, we can begin breaking down our ***views***.

When we're breaking down our mockup:

- Breakdown by functionality
- Remember K.I.S.S and D.R.Y
- Refrain from being overly complicated or overly generic. You want to meet somewhere in the middle

View Breakdown

So now that we know how our view should breakdown, let's look at an example. For this - I've stolen Sencha's Pandora mockup.

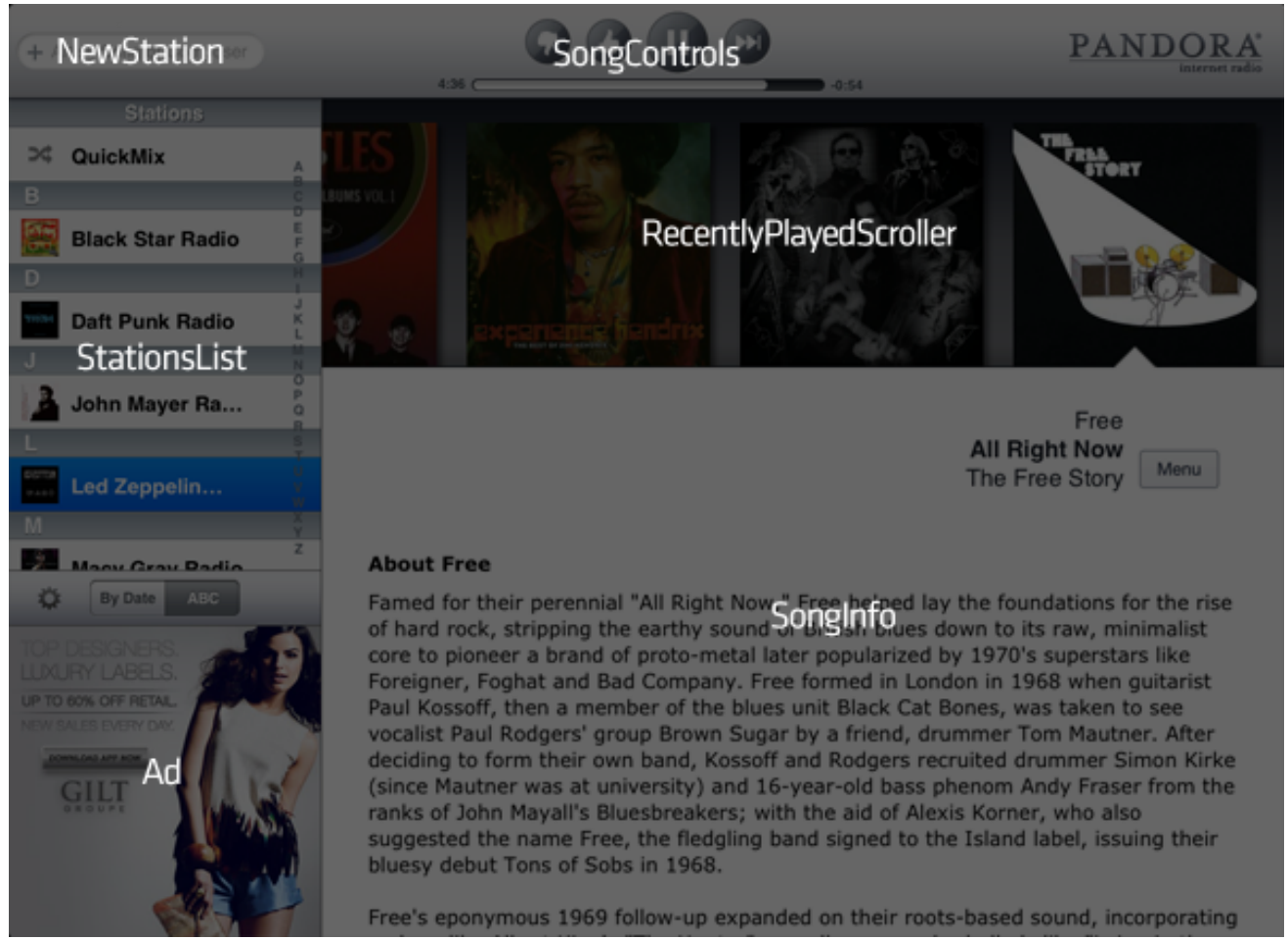


View Breakdown Part Deux

Looking at this, we see how we would breakdown our views:

- NewStation
- SongControls
- StationsList
- RecentlyPlayedScroll
- Ad
- SongInfo

For each of these we would have a file with a relevant name located in `/app/view/`



Building Our Models

So now that we have our views in place, we can start building out our models. By looking at the type of **dynamic** data in our UI, we can get an idea of what needs to go into each model.

It should be noted that not every view will necessarily have its own model. Remember, when building out your applications K.I.S.S and D.R.Y. We want to try to reuse the same data that is in our local store as much as possible to avoid additional data calls to the database.

Building Our Models Part Deux

Another thing to remember - AJAX calls can become expensive. The more you can bulk into 1 call the better.

One of the great things about modern browsers is how well they manage their memory, so we could have 200k records in a store and our browser won't crap out on us. One of the pitfalls with having this many records, however, is rendering. Having your browser attempt to render 200k records at one time will almost certainly cause it to crash. So we would want to break those records up during render and loop through smaller sets we can send to our rendering function.

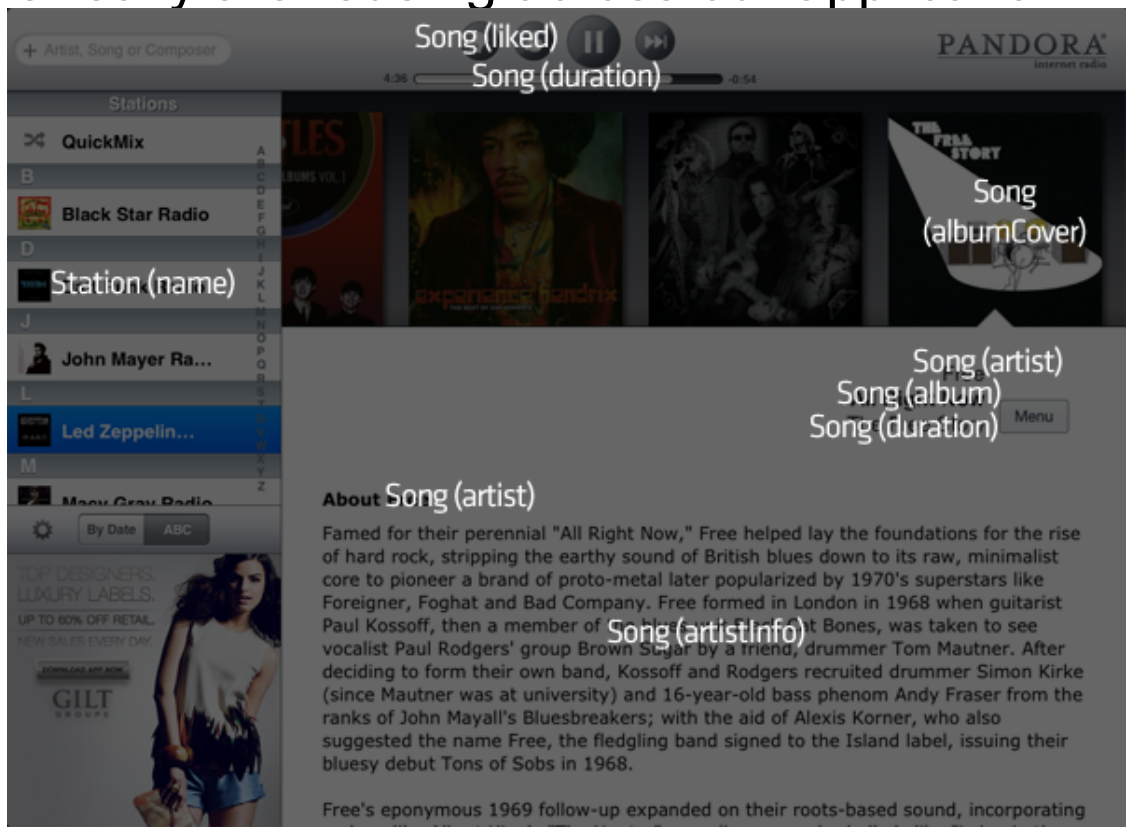
We should be mindful of our IE users when we're traversing. D':

Building Our Models Part Trois

Looking at our mockup below, we can see we have mapped out 2 models - Song and Station.

You can now see how much data we really are reusing across our application.

Since most of our data is based around the artist/song/album, we can use that information across the board - and since it's stored locally now, we've just greatly enhanced our speed.



Model + Proxy = Love

When we're using a model and proxy, we're going to need somewhere to offload that data. So what do we need?

Model + Proxy = Love, or just a really fine tuned data store

When we're using a model and proxy, we're going to need somewhere to offload that data. So what do we need?

A store of course!

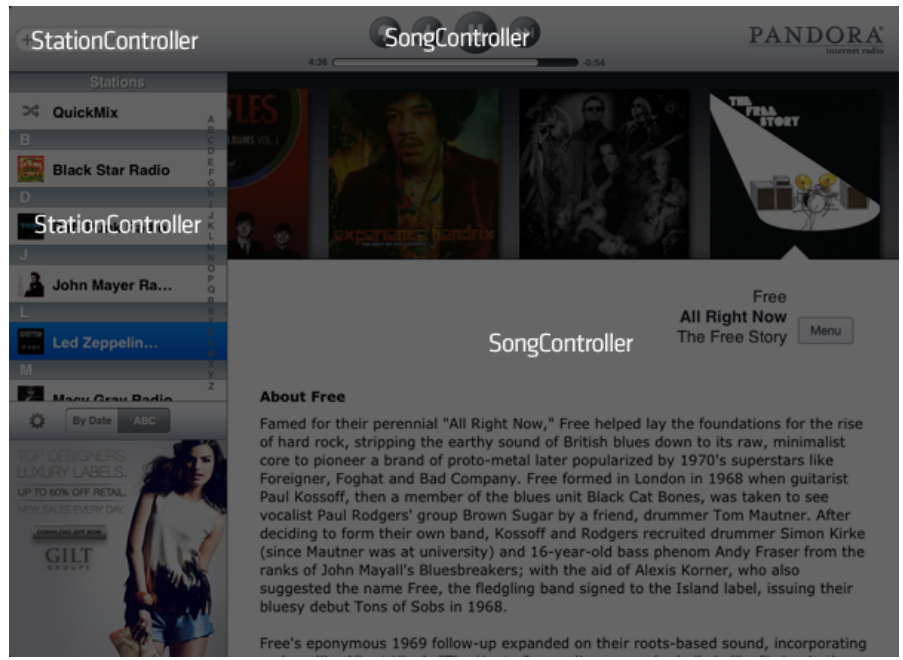
There isn't much to stores, the model fields should match up appropriately to our store fields. Mismatches can happen, but you will almost always see them during execution. Things like that are why we have an awesome filled QA team that can catch all of our terrible JS errors!

Let The Controller Do The Work

When we're diving into an MVC architecture we should know that the purpose is to separate our functionality from our data and visualizations.

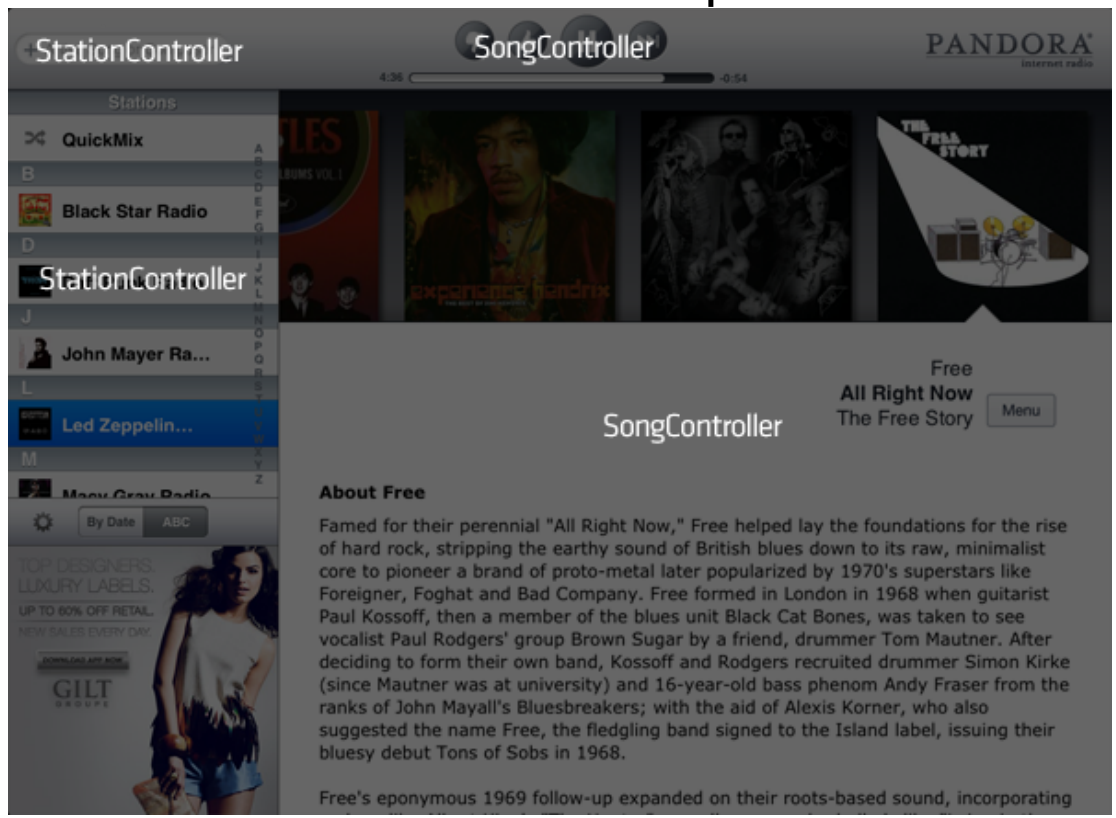
The role of responsibility is held by our controller. This guy can often be small but packs some heat. More bang for your buck, if you will.

So looking back at our application, we can create 2 straight-forward controllers.



Let The Controller Do The Work Part Deux

We can see our 2 controllers created will be SongController and StationController. Our StationController would hold the logic of creating new stations as well as loading our favorite stations. Our SongController would hold the logic of managing the song info (within our model) as well as knowing what to do when a user clicks on the "thumbs up" or "thumbs down" icons respectively.



Afterthoughts

Now that we're all on the same page, we can see how fast, easy, and maintainable it is to build our newly architected applications.

A few things to consider:

- This is a very loose MVC
- Multiple MVC architected applications are possible on any given page
- A model != store, but a model should always relate to a store

Now get out there slugger!

And Remember...

SevOne