MVC Architecture

Large client side applications have always been hard to write, hard to organize and hard to maintain. They tend to quickly grow out of control as you add more functionality and developers to a project. Ext JS 4 comes with a new application architecture that not only organizes your code but reduces the amount you have to write.

Our application architecture follows an MVC-like pattern with Models and Controllers being introduced for the first time. There are many MVC architectures, most of which are slightly different from one another. Here's how we define ours:

- **Model** is a collection of fields and their data (e.g. a User model with username and password fields). Models know how to persist themselves through the data package, and can be linked to other models through associations. Models work a lot like the Ext JS 3 Record class, and are normally used with **Stores** to present data into grids and other components.
- **View** is any type of component - grids, trees and panels are all views.
- **Controllers** are special places to put all of the code that makes your app work - whether that's rendering views, instantiating Models, or any other app logic.

In this guide we'll be creating a very simple application that manages User data. By the end you will know how to put simple applications together using the new Ext JS 4 application architecture.

The application architecture is as much about providing structure and consistency as it is about actual classes and framework code. Following the conventions unlocks a number of important benefits:

- Every application works the same way so you only have to learn it once
- It's easy to share code between apps because they all work the same way
- You can use our build tools to create optimized versions of your applications for production use

File Structure

Ext JS 4 applications follow a unified directory structure that is the same for every app. Please check out the **Getting Started guide** for a detailed explanation on the basic file structure of an application. In MVC layout, all classes are placed into the **app** folder, which in turn contains sub-folders to namespace your models, views, controllers and stores. Here is how the folder structure for the simple example app will look when we’re done:
In this example, we are encapsulating the whole application inside one folder called 'account_manager'. Essential files from the Ext JS 4 SDK are wrapped inside ext-4.0 folder. Hence the content of our index.html looks like this:

```html
<html>
<head>
    <title>Account Manager</title>
    <link rel="stylesheet" type="text/css" href="ext-4.0/resources/css/ext-all.css">
    <script type="text/javascript" src="ext-4.0/ext-debug.js"></script>
    <script type="text/javascript" src="app.js"></script>
</head>
<body>
</body>
</html>
```

Creating the application in app.js
Every Ext JS 4 application starts with an instance of `Application` class. The Application contains global settings for your application (such as the app’s name), as well as maintains references to all of the models, views and controllers used by the app. An Application also contains a launch function, which is run automatically when everything is loaded.

Let's create a simple Account Manager app that will help us manage User accounts. First we need to pick a global namespace for this application. All Ext JS 4 applications should only use a single global variable, with all of the application’s classes nested inside it. Usually we want a short global variable so in this case we're going to use "AM":

```javascript
Ext.application({
    name: 'AM',
    appFolder: 'app',

    launch: function() {
        Ext.create('Ext.container.Viewport', {
            layout: 'fit',
            items: [
                {
                    xtype: 'panel',
                    title: 'Users',
                    html: 'List of users will go here'
                }
            ]
        });
    }
});
```

There are a few things going on here. First we invoked `Ext.application` to create a new instance of Application class, to which we passed the name "AM". This automatically sets up a global variable `AM` for us, and registers the namespace to `Ext.Loader`, with the corresponding path of 'app' set via the `appFolder` config option. We also provided a simple launch function that just creates a `Viewport` which contains a single `Panel` that will fill the screen.
Defining a Controller

Controllers are the glue that binds an application together. All they really do is listen for events (usually from views) and take some actions. Continuing our Account Manager application, let's create a controller. Create a file called `app/controller/Users.js` and add the following code:

```javascript
Ext.define('AM.controller.Users', {
    extend: 'Ext.app.Controller',

    init: function() {
        console.log('Initialized Users! This happens before the Application launch');
    }
});
```

Now let's add our newly created Users controller to the application config in app.js:

```javascript
Ext.application({
    ...

    controllers: [
        'Users',
    ],

    ...
});
```

When we load our application by visiting `index.html` inside a browser, the Users controller is
automatically loaded (because we specified it in the Application definition above), and its init function is called just before the Application's launch function.

The init function is a great place to set up how your controller interacts with the view, and is usually used in conjunction with another Controller function - control. The control function makes it easy to listen to events on your view classes and take some action with a handler function. Let's update our Users controller to tell us when the panel is rendered:

```javascript
Ext.define('AM.controller.Users', {  
    extend: 'Ext.app.Controller',
    
    init: function() {
        this.control({  
            'viewport > panel': {  
                render: this.onPanelRendered
            }
        });
    },

    onPanelRendered: function() {
        console.log('The panel was rendered');
    }
});
```

We've updated the init function to use this.control to set up listeners on views in our application. The control function uses the new ComponentQuery engine to quickly and easily get references to components on the page. If you are not familiar with ComponentQuery yet, be sure to check out the ComponentQuery documentation for a full explanation. In brief though, it allows us to pass a CSS-like selector that will find every matching component on the page.

In our init function above we supplied 'viewport > panel', which translates to 'find me every Panel that is a direct child of a Viewport'. We then supplied an object that maps event names (just render in this case) to handler functions. The overall effect is that whenever any component that matches our selector fires a render event, our onPanelRendered function is called.

When we run our application now we see the following:
Not exactly the most exciting application ever, but it shows how easy it is to get started with organized code. Let’s flesh the app out a little now by adding a grid.

**Defining a View**

Until now our application has only been a few lines long and only inhabits two files - app.js and app/controller/Users.js. Now that we want to add a grid showing all of the users in our system, it's time to organize our logic a little better and start using views.

A View is nothing more than a Component, usually defined as a subclass of an Ext JS component. We're going to create our Users grid now by creating a new file called app/view/user/List.js and putting the following into it:

```javascript
Ext.define('AM.view.user.List', {
    extend: 'Ext.grid.Panel',
    alias: 'widget.userlist',

    title: 'All Users',

    initComponent: function() {
        this.store = {
```
Our View class is nothing more than a normal class. In this case we happen to extend the Grid Component and set up an alias so that we can use it as an xtype (more on that in a moment). We also passed in the store configuration and the columns that the grid should render.

Next we need to add this view to our Users controller. Because we set an alias using the special 'widget.' format, we can use 'userlist' as an xtype now, just like we had used 'panel' previously.

```javascript
Ext.define('AM.controller.Users', {
    extend: 'Ext.app.Controller',

    views: [
        'user.List'
    ],

    init: ...

    onPanelRendered: ...
});
```

And then render it inside the main viewport by modifying the launch method in app.js to:

```javascript
Ext.application({
    ...

    launch: function() {
        Ext.create('Ext.container.Viewport', {
            layout: 'fit',
            items: {
                xtype: 'userlist'
            }
        });
    }
});
```
The only other thing to note here is that we specified `user.List` inside the views array. This tells the application to load that file automatically so that we can use it when we launch. The application uses Ext JS 4's new dynamic loading system to automatically pull this file from the server. Here's what we see when we refresh the page now:

![Screen capture of Ext JS grid](image)

**Controlling the grid**

Note that our `onPanelRendered` function is still being called. This is because our grid class still matches the `viewport > panel` selector. The reason for this is that our class extends Grid, which in turn extends Panel.

At the moment, the listeners we add to this selector will actually be called for every Panel or Panel subclass that is a direct child of the viewport, so let's tighten that up a bit using our new xtype. While we're at it, let's instead listen for double clicks on rows in the grid so that we can later edit that User:

```javascript
Ext.define('AM.controller.Users', {
    extend: 'Ext.app.Controller',

    views: [
        'user.List'
    ],
```

```javascript
```
init: function() {
    this.control({
        'userlist': {
            itemdblclick: this.editUser
        }
    });
},

editUser: function(grid, record) {
    console.log('Double clicked on ' + record.get('name'));
}
});

Note that we changed the ComponentQuery selector (to simply 'userlist'), the event name (to 'itemdblclick') and the handler function name (to 'editUser'). For now we're just logging out the name of the User we double clicked:

Logging to the console is all well and good but we really want to edit our Users. Let's do that now, starting with a new view in app/view/user/Edit.js:

```javascript
Ext.define('AM.view.user.Edit', {
   ...
```
extend: 'Ext.window.Window',
alias : 'widget.useredit',

title : 'Edit User',
layout: 'fit',
autoShow: true,

initComponent: function() {
    this.items = [
        {
            xtype: 'form',
            items: [
                {
                    xtype: 'textfield',
                    name : 'name',
                    fieldLabel: 'Name'
                },
                {
                    xtype: 'textfield',
                    name : 'email',
                    fieldLabel: 'Email'
                }
            ]
        }
    ];

    this.buttons = [
        {
            text: 'Save',
            action: 'save'
        },
        {
            text: 'Cancel',
            scope: this,
            handler: this.close
        }
    ];

    this.callParent(arguments);
}
});

Again we're just defining a subclass of an existing component - this time Ext.window.Window. Once more we used initComponent to specify the complex objects items and buttons. We used a 'fit' layout and a form as the single item, which contains fields to edit the name and the email address. Finally we created two buttons, one which just closes the window, and the other that will be used to save our changes.

All we have to do now is add the view to the controller, render it and load the User into it:
First we created the view using the convenient method `Ext.widget`, which is equivalent to `Ext.create('widget.useredit')`. Then we leveraged ComponentQuery once more to quickly get a reference to the edit window's form. Every component in Ext JS 4 has a `down` function, which accepts a ComponentQuery selector to quickly find any child component.

Double clicking a row in our grid now yields something like this:

**Creating a Model and a Store**
Now that we have our edit form it's almost time to start editing our users and saving those changes. Before we do that though, we should refactor our code a little.

At the moment the AM.view.user.List component creates a Store inline. This works well but we'd like to be able to reference that Store elsewhere in the application so that we can update the data in it. We'll start by breaking the Store out into its own file - app/store/Users.js:

```javascript
Ext.define('AM.store.Users', {
    extend: 'Ext.data.Store',
    fields: ['name', 'email'],
    data: [
        {name: 'Ed', email: 'ed@sencha.com'},
        {name: 'Tommy', email: 'tommy@sencha.com'}
    ]
});
```

Now we'll just make 2 small changes - first we'll ask our Users controller to include this Store when it loads:

```javascript
Ext.define('AM.controller.Users', {
    extend: 'Ext.app.Controller',
    stores: [
        'Users'
    ],
    ...}
});
```

Then we'll update app/view/user/List.js to simply reference the Store by id:

```javascript
Ext.define('AM.view.user.List', {
    extend: 'Ext.grid.Panel',
    alias: 'widget.userlist',

    // we no longer define the Users store in the `initComponent` method
    store: 'Users',

    ...
});
```

By including the stores that our Users controller cares about in its definition they are automatically loaded onto the page and given a storeId, which makes them really easy to reference in our views (by simply configuring store: 'Users' in this case).

At the moment we've just defined our fields ('name' and 'email') inline on the store. This works well enough but in Ext JS 4 we have a powerful Ext.data.Model class that we'd like to take advantage of when it comes to editing our Users. We'll finish this section by refactoring our Store to use a Model, which we'll put in app/model/User.js:

```javascript
Ext.define('AM.model.User', {
```
```javascript
extend: 'Ext.data.Model',
fields: ['name', 'email']
}
```

That's all we need to do to define our Model, now we'll just update our Store to reference the Model name instead of providing fields inline, and ask the Users controller to get a reference to the model too:

```javascript
// the Users controller will make sure that the User model is included on the page
Ext.define('AM.controller.Users', {
    extend: 'Ext.app.Controller',
    stores: ['Users'],
    models: ['User'],
    ...
});

// we now reference the Model instead of defining fields inline
Ext.define('AM.store.Users', {
    extend: 'Ext.data.Store',
    model: 'AM.model.User',
    data: [
        {name: 'Ed', email: 'ed@sencha.com'},
        {name: 'Tommy', email: 'tommy@sencha.com'}
    ]
});
```

Our refactoring will make the next section easier but should not have affected the application's current behavior. If we reload the page now and double click on a row we see that the edit User window still appears as expected. Now it's time to finish the editing functionality:
Saving data with the Model

Now that we have our users grid loading data and opening an edit window when we double click each row, we'd like to save the changes that the user makes. The Edit User window that the defined above contains a form (with fields for name and email), and a save button. First let's update our controller's init function to listen for clicks to that save button:

```javascript
Ext.define('AM.controller.Users', { 
    init: function() { 
        this.control({
            'viewport > userlist': { 
                itemdblclick: this.editUser,
            },
            'useredit button[action=save]': { 
                click: this.updateUser
            }
        });
    },

    updateUser: function(button) {
        console.log('clicked the Save button');
    }
});
```

We added a second ComponentQuery selector to our `this.control` call - this time 'useredit
button[action=save]'. This works the same way as the first selector - it uses the 'useredit' xtype that we defined above to focus in on our edit user window, and then looks for any buttons with the 'save' action inside that window. When we defined our edit user window we passed {action: 'save'} to the save button, which gives us an easy way to target that button.

We can satisfy ourselves that the `updateUser` function is called when we click the Save button:

Now that we've seen our handler is correctly attached to the Save button's click event, let's fill in the real logic for the `updateUser` function. In this function we need to get the data out of the form, update our User with it and then save that back to the Users store we created above. Let's see how we might do that:

```
updateUser: function(button) {
    var win   = button.up('window'),
        form  = win.down('form'),
        record = form.getRecord(),
        values = form.getValues();

    record.set(values);
    win.close();
}
```

Let's break down what's going on here. Our click event gave us a reference to the button that the user clicked on, but what we really want is access to the form that contains the data and the window itself. To
get things working quickly we'll just use ComponentQuery again here, first using
`button.up('window')` to get a reference to the Edit User window, then `win.down('form')` to get the
form.

After that we simply fetch the record that's currently loaded into the form and update it with whatever
the user has typed into the form. Finally we close the window to bring attention back to the grid. Here's
what we see when we run our app again, change the name field to 'Ed Spencer' and click save:

Saving to the server
Easy enough. Let's finish this up now by making it interact with our server side. At the moment we are
hard coding the two User records into the Users Store, so let's start by reading those over AJAX
instead:

```
Ext.define('AM.store.Users', {
    extend: 'Ext.data.Store',
    model: 'AM.model.User',
    autoLoad: true,

    proxy: {
        type: 'ajax',
        url: 'data/users.json',
        reader: {
            type: 'json',
            root: 'users',
            successProperty: 'success'
        }
    }
});
```
Here we removed the 'data' property and replaced it with a **Proxy**. Proxies are the way to load and save data from a Store or a Model in Ext JS 4. There are proxies for AJAX, JSON-P and HTML5 localStorage among others. Here we've used a simple AJAX proxy, which we've told to load data from the url 'data/users.json'.

We also attached a **reader** to the Proxy. The reader is responsible for decoding the server response into a format the Store can understand. This time we used a JSON reader, and specified the root and **successProperty** configurations (see the **Json Reader** docs for more on those configurations). Finally we'll create our **data/users.json** file and paste our previous data into it:

```json
{
    success: true,
    users: [
        {id: 1, name: 'Ed', email: 'ed@sencha.com'},
        {id: 2, name: 'Tommy', email: 'tommy@sencha.com'}
    ]
}
```

The only other change we made to the Store was to set **autoLoad** to **true**, which means the Store will ask its Proxy to load that data immediately. If we refresh the page now we'll see the same outcome as before, except that we're now no longer hard coding the data into our application.

The last thing we want to do here is send our changes back to the server. For this example we're just using static JSON files on the server side so we won't see any database changes but we can at least verify that everything is plugged together correctly. First we'll make a small change to our new proxy to tell it to send updates back to a different url:

```javascript
proxy: {
    type: 'ajax',
    api: {
        read: 'data/users.json',
        update: 'data/updateUsers.json'
    },
    reader: {
        type: 'json',
        root: 'users',
        successProperty: 'success'
    }
}
```

We're still reading the data from **users.json**, but any updates will be sent to **updateUsers.json**. This is just so that we can return a dummy response so we know things are working. The **updateUsers.json** file just contains `{ "success": true }`. The only other change we need to make is to tell our Store to synchronize itself after editing, which we do by adding one more line inside the **updateUser** function, which now looks like this:

```javascript
updateUser: function (button) {
    var win = button.up('window'),
        form = win.down('form'),
    record = form.getRecord(),
```
values = form.getValues();

record.set(values);
win.close();
this getUsersStore().sync();

Now we can run through our full example and make sure that everything works. We'll edit a row, hit the Save button and see that the request is correctly sent to updateUser.json.

Deployment

The newly introduced Sencha SDK Tools (download here) makes deployment of any Ext JS 4 application easier than ever. The tools allows you to generate a manifest of all dependencies in the form of a JSB3 (JSBuilder file format) file, and create a minimal custom build of just what your application needs within minutes.

Please refer to the Getting Started guide for detailed instructions.

Next Steps

We've created a very simple application that manages User data and sends any updates back to the server. We started out simple and gradually refactored our code to make it cleaner and more
organized. At this point it's easy to add more functionality to our application without creating spaghetti code. The full source code for this application can be found in the Ext JS 4 SDK download, inside the examples/app/simple folder.

In the next guide, we'll look at advanced Controller usage and patterns that can make your application code smaller and easier to maintain.