6.6 Router Improvements

Saturday, November 30, 2019 8:21 PM

If you know me, you know that the router is very important to every application I build. And like every iteration of Ext JS' router, Sencha Fiddle is the application that has the most contribution and the most experimentation (Sencha Fiddle has a custom router with overrides to add new features to the Ext JS version it's using) on the router. Once again, with Ext JS 6.6, Sencha Fiddle exposed some features that would be great but it's also talking to the community that has provided many improvements as well. I want to talk about four new features that have landed in 6.6.

Exit Handler

This feature came from a community member (LesJ from the Sencha Forums).

There are many times when you create things in response to a route being executed that you may need to cleanup. For example, you may have opened an Ext.window.Window but when the route is executed, you need to close that window. So in your `action` handler, you create the window and likely cache it somewhere and when another route is executed you check to see if any windows are shown and clean that up. Ok, not the end of the world but this means code starts to get ugly and brittle and you start engineering things where Ext JS could make this easy.

With Ext JS 6.6, we are going to make this easy with a new `exit` handler and the use of it is just like the `action` and `before` handler:

```
routes : {
    'foo/bar' : {
        action : 'onFoo',
        exit   : 'onExit',
        name   : 'foo'
    }
},

onFoo : function () {},

onExit : function () {}
```

So now your route can handle when the route no longer recognizes anything in the hash. Internally, each route keeps track of the last token it was executed on which allows it to know if it was being exited which is needed if you use multiple tokens as is the case in my apps like Sencha Fiddle.

Single Routes

This improvement comes from a need for Sencha Fiddle.

At the start of Sencha Fiddle (and the support portal) I have to check user session and I also wait for certain stores to load before I want to continue with showing views and such. Currently, I have a global route with a before, here is a small example:

```
routes : {
    '*' : {
        before : 'onBeforeGlobal'
    }
},

onBeforeGlobal : function () {
```

```
    return new Ext.Promise(function (resolve, reject) {
        // check if things have finished loading
    });
}
```

I have an array of stores I go through and check if it has loaded. However, I felt like this was wasteful because it's going to check if these stores are loaded for each route as the * wildcard route fires before all other routes; I only needed it to check upfront. Taking a page out of the Observable listeners where you can pass `single: true`, routes will be able to define their singleness:

```
routes : {
    '*' : {
        before : 'onBeforeGlobal',
        single : true
    }
}
```

Tough syntax huh? This will remove the route from the router after the `before` and `action` handlers have been executed. Once removed, the route will not longer be connected and will not execute anymore. Perfect!

Or is it? Why wait for a full, successful execution to remove the route? Why not always remove it? Knowing that all applications (or developers) do not have the same requirements, we decided to give a bit more control and so the `single` option can also accept `'before'` and `'after'`. These string values are about when to remove the route.

`'before'` will remove the route before the configured `before` handler is executed. This means the route will always get removed regardless if the `before` action is resolved or rejected.

`'after'` will remove the route right after the `before` handler has been resolved but prior to the `action` handler to be removed. This can be seen as very similar to the `true` value since the `action` handler cannot be stopped (assuming to errors occur), however, the difference is when you have other routes that have a `before`. Those other `before`s may reject which would then not execute our single route's `action` handler from executing which means the route would not get removed.

Kind of hard to grok fully from just text so I'll try another way of explaining it. When you defined the `routes` config, it will actually only create one route instance matching the route name (or pattern) and the `before`/`action` handlers are then added to that route's `handlers` array. So when a hash changes, it finds the matching route instance and then executes it. It creates two arrays: the `before`s and the `action`s. Execution will go through each before in an async stack (each before has to resolve before moving onto the next before) and then it will go through the action stack. So there are three insert points on where the route can be removed: beginning of the before stack, within the before stack after the matching `before` action, and on the end of the action stack. So it's very important to know when you'd want the route to be removed in that execution but the main point is whether to wait for a `before` or all `before`s to resolve.

We picked `true` to remove on the end of the action stack but are still discussing whether it should go on the beginning of the before stack so these values may change. This will be documented in the API Docs when 6.6 is released so I would recommend checking the final docs. Also, comment below if you feel strongly about this. I'll even see if I can create an example and/or image to show this a bit better, I'm still trying to find a great way to explain it with so many duplicate terms.

Lazy Routes

This feature came out from the coworkee app that was developed internally among some posts on the forum.

One issue with using routes in ViewControllers is that the ViewController may not be instantiated when

the route was already executed. I personally don't spread routes around too much but people have been asking for this and it was easy to implement so ok. People asked for the routes defined on their ViewController to execute if it matches a current hash. We are calling this `lazy` and it can be defined like so:

```
routes : {
    'user/:id' : {
        action  : 'onUser',
        lazy    : true,
        name    : 'user'
    }
}
```

This means, if the hash is already `#user/1234` when the ViewController is instantiated, this route will execute.

I'm personally not a fan of defining routes all around my application but I do see some value it in since I've built some heinous enterprise UIs and routes may only update a portion of the screen not simply swap out what view is in the center/content region.

Named Types

This feature came from a mix of things from Fiddle to support tickets to a conversation between Don Griffin and I. This feature also aims to fix two different problems.

Ok, I teased this on Twitter and here is what this was about.

The first "problem" is people currently have to define custom matching RegExp strings to control the format of a url parameter. What I mean is say you have a user route that accepts an ID but that ID will only be numeric so you can define it as such:

```
routes : {
    'user/:id' : {
        action     : 'onUser',
        name       : 'user',
        conditions : {
            ':id' : '([0-9]+)'
        }
    }
},

onUser : function (id) {}
```

Now, this route will only execute for hashes where `:id` is numeric such as `user/1234` but will not execute if there are other characters such as `user/1234abc`. One important note to make here is the defaultMatcher of a route is `([%a-zA-Z0-9\\-\\_\\s,]+)` so it will accept a few different characters that you may or may not want.

My initial proof of concept had a syntax such as :

```
routes : {
    'user/:num' : {
        action : 'onUser',
        name   : 'user'
    }
}
```

And that `:num` would automatically use the `([0-9]+)` RegExp. This is pretty simple right? The issue

with this syntax is what if you had a `:num` parameter in a route but didn't want that exact RegExp? There was also `:alpha` and `:alphanum`. The chance of collision here was very tiny but if you were one of the unlucky people that would have a name conflict, it likely would be a silent failure and you'd find out about it in production which is not good at all. So we discussed how we can prevent this and a couple different ideas came up like `:$num` or some character to signify this was to be treated as a named type. But we decided on this format:

```
routes : {
    'user/:{id:num}' : {
        action : 'onUser',
        name   : 'user'
    }
},

onUser : function (params) {
    var id = params.id;
}
```

So we surround the parameter name and type in curly braces. Quick note, the type (`:num` in this case) is optional so you can have `:{id}` and the `defaultMatcher` will then be used. Other default types are `:alpha` and `:alphanum`. Also, for `:num` and `:alphanum`, numbers will be cast using `parseFloat` so you can have `#foo/10.4` and 10.4 will be a float instead of just a string.

Also notice in the `onUser` we have `params` which will be an object and the parameter names will not be the key of that object. This was another of the problems where having separate arguments in the functions could get a bit wildly and it can be a good rule of thumb if you have more than 3 arguments, an object will be more friendly. So if you use these new named typed, an object will always be passed to the handlers.

We have some detection if you reuse a param name, an error will be thrown in the console with what name was duplicated and what route this was defined in. So if you have `user/:{id}/:{id}`, you'd get an error in the console such as:

```
"id" already defined in route "user/:{id}/:{id}"
```

Also, if you have a type that is not known, you'll also get an error. So say you have `user/:{id:foo}`, `:foo` is not a default type so you'd get this error:

```
Unknown parameter type "foo" in route "user/:{id:foo}"
```

Since there is a difference between how the parameters are passed to the handlers, we don't want the two syntaxes mixed in a route. The older syntax is called "positional" and the newer being called "named". If you mix these syntaxes like so: `foo/:bar/:{baz}` you will get the error:

```
URL parameter mismatch. Positional url parameter found while in named mode in
the route "foo/:bar/:{baz}".
```

A couple thoughts here on the duplicate parameters. Right now, you cannot have duplicate parameter names in the same route. However, we've discussed turning the single value into an array to support duplicate names, however, we don't really think this will be something that many will use and an error may be more helpful. Once again, let us know your thoughts as we can have a config to allow this if people would want it, just provide us with a real world scenario where you'd want it.

You can define your own custom named types via an override. For example, in Sencha Fiddle a fiddle id will only be `[a-z0-9]` and I can have the route defined in a couple places so having a central place to have this and be able to be used anywhere would be a big pro. So you can add one:

```
Ext.define('Override.route.Route', {
    override : 'Ext.route.Route',

    config : {
        types : {
            'fiddleid' : {
                re : '([a-z0-9]+)'
            }
        }
    }
});
```

Now I can define routes such as:

```
routes : {
    'fiddle/:{id:fiddleid}' : {
        action : 'onFiddle',
        name   : 'fiddle'
    }
},

onFiddle : function (params) {
    Ext.Ajax
        .request({
            url : '/fiddle/get/' + params.id
        })
        .then(function (fiddle) {
            // ...
        });
}
```

So since we are now parsing parameters and their types, there is one thing that I've seen people ask for and that would be supporting an arbitrary number of parameters in a hash. For example, say you want to support hashes like `#view/foo` and `#view/foo/bar/baz` in a single route. Before, you'd have to do something complex with conditions and if you didn't want to get too complex, you'd have a static number of parameters defined in your pattern like `view/:foo/:bar/:baz` (this isn't a very good, full example of what would be needed). So we are attempting to handle this with this syntax:

```
routes : {
    'view/:{args...}' : {
        action : 'onView',
        name   : 'view'
    }
},

onView : function (params) {
    // Ext.isArray(params.args) === true
}
```

By default, this syntax will match everything (uses `(.+)?`) and split on `/` so that you get an array of values. Like the `:num` and `:alphanum`, this will also attempt to cast numbers into floats. So if you have this hash `#view/foo/12.45/bar`, this is what the params object would be:

```
{
    args : [ 'foo', 12.45, 'bar' ]
```

```
}
```

As you would expect, you don't need to have just the one parameter, you can have other parameters with their own types:

```
routes : {
    'view/:{which}/:{amount:num}:{args...}' : {
        action : 'onView',
        name   : 'view'
    }
}
```

The params object would then be:

```
{
    amount : 12.45,
    args   : [ 'bar' ],
    which  : 'foo'
}
```

Most people would be fine with splitting by / but if you wanted another character you can override this via:

```
Ext.define('Override.route.Route', {
    override : 'Ext.route.Route',

    config : {
        types : {
            '...' : {
                split : '-'
            }
        }
    }
});
```

This also shows there are 3 things you can use when defining your own type. The `...` is just a type defined as such:

```
'...': {
    re: '(.+)?',
    split: '/',
    parse: function (values) {
        var length, i, value;

        if (values) {
            length = values.length;

            for (i = 0; i < length; i++) {
                value = parseFloat(values[i]);

                if (!isNaN(value)) {
                    values[i] = value;
                }
            }
        } else if (Ext.isString(values)) {
            // IE8 may have values as an empty string
```

```
            // if there was no args that were matched
            values = undefined;
        }

        return values;
    }
}
```

Simple to understand, the `re` is the string that will control what pattern will be matched, `split` will turn the value into an array (or undefined if no matches) and `parse` allows you to parse the value into something else.

One type that could be useful and has come up a couple times in the last couple weeks is supporting query parameters in a hash. So you could use something like `#foo?bar&baz=2` and the query string like part would be turned into a nested object. You coul duse this override to support this:

```
Ext.define('Override.route.Route', {
    override : 'Ext.route.Route',

    config : {
        types : {
            queryString : {
                re    : '(?:\\?(.+))?',
                parse : function (params) {
                    var name, value;

                    if (params) {
                        params = Ext.Object.fromQueryString(params);

                        for (name in params) {
                            value = params[name];

                            if (value) {
                                if (/^[0-9]+$/.test(value)) {
                                    // let's turn this into a float
                                    params[name] = parseFloat(value);
                                }
                            }
                            // an empty string would be ?foo
                            // which can be seen as truthy
                            else if (Ext.isString(value)) {
                                params[name] = true;
                            }
                        }
                    }

                    return params;
                }
            }
        }
    }
});
```

You'd then defined your route as:

```
routes : {
```

```
    'foo:{query:queryString}'   : {
        action : 'onFoo',
        name   : 'foo'
    }
}
```

So if you had the hash as `#foo?bar&baz=2.1` then the params send to the `onFoo` method would be:

```
{
    query : {
        bar : true,
        baz : 2.1
    }
}
```

Summary

Once again, Sencha has listened to it's community and along with what we learn as we use the router, the router is only getting better. More and more people are using in different applications and knowing how people use the router has been immensely helpful in crafting new features. Keep letting us know what you'd like. Open a forum thread and get my attention (Twitter is a great way, @LikelyMitch) and I'd be more than happy to discuss it with you and give my honest personal opinion. :)

[source: 6.6 Router Improvements]

Abstract vs Override

Saturday, November 30, 2019 8:24 PM

Developing large single-page applications can be a daunting task. I've seen web applications approach a million lines of code. That's quite a lot of code and performance is a real issue there (luckily it was a B2B application not a consumer facing application). A golden rule in software development, well most things in life really, is that there is always a better way to do something. You may think you have the perfect way to make a pizza but there's likely a better way to make it. The same goes for software development.

Abstracting code

One way to minimize code is to create an abstract class. Ext JS has a great class system that allows simple inheritance. Taking advantage of this class system by creating an abstract class can minimize the code required because you're not having to have the same code spread among several classes.

A simple example I could provide is in my grids I tend to want a couple different things:

- A top docked toolbar with:
    - A search field
    - Button to add a record
- A paging toolbar
- A refresh tool in the title bar

Normally I would extend `Ext.grid.Panel` for all my grids but for those common UI features I'd have to implement them in each grid which is not optimal. There's an better way to do it, create an abstract grid class that extends `Ext.grid.Panel` and has configs to turn on/off the UI features. Here is an example abstract grid class (please don't get overwhelmed by the amount of code):

```
Ext.define('MyApp.view.abstracts.Grid', {
    extend : 'Ext.grid.Panel',
    xtype  : 'myapp-abstracts-grid',

    requires : [
        'Ext.button.Button',
        'Ext.form.field.Text',
        'Ext.grid.feature.Grouping',
        'Ext.toolbar.Paging'
    ],

    config : {
        /**
         * @cfg {Boolean/Object} [createNew=true] `true` to add a {@link
Ext.button.Button} to create a new record.
         *
         * Can also be a config object for the button.
         */
        createNew   : true,
        /**
         * @cfg {Boolean/Object} [grouped=false] `true` to add the {@link
Ext.grid.feature.Grouping} feature to the grid.
         *
         * Can be a config object that will be passed or a boolean.
         */
        grouped     : false,
```

```
        /**
         * @cfg {Boolean/Object} [pageable=true] `true` to add a {@link
Ext.toolbar.Paging}. Will set the grid's
         * store on the toolbar automatically.
         *
         * Can be a config object for the paging toolbar.
         */
        pageable    : true,
        /**
         * @cfg {Boolean/Object} [refreshable=true] `true` to add a refresh
{@link Ext.panel.Tool} to the title bar.
         *
         * Can be a config object for the tool.
         */
        refreshable : true,
        /**
         * @cfg {Boolean/Object} [searchable=true] `true` to add a search
{@link Ext.form.field.Text}.
         *
         * Can also be a config object for the text field.
         */
        searchable  : true
    },

    initComponent : function() {
        var me          = this,
            store       = me.store = Ext.data.StoreManager.lookup(me.store),
            dockedItems = me.dockedItems || [],
            features    = me.features    || [],
            tbar        = me.tbar        || [],
            tools       = me.tools       || [],
            createNew   = me.getCreateNew(),
            grouped     = me.getGrouped(),
            pageable    = me.getPageable(),
            refreshable = me.getRefreshable(),
            searchable  = me.getSearchable();

        if (createNew) {
            tbar.unshift(Ext.apply({
                text    : 'Create',
                iconCls : 'fa fa-plus',
                handler : 'onCreateClick'
            }, createNew));
        }

        if (searchable) {
            //add it first
            tbar.unshift(Ext.apply({
                xtype           : 'textfield',
                enableKeyEvents : true,
                emptyText       : 'search...',
                width           : 300,
                triggers        : {
                    search : {
                        cls     : 'fa fa-search',
```

```
                        handler : 'doSearch'
                    }
                },
                listeners       : {
                    keydown : 'onSearchKeyDown'
                }
            }, searchable));
        }

        if (grouped) {
            features.push(Ext.apply({
                ftype : 'grouping'
            }, grouped));
        }

        if (pageable) {
            dockedItems.push(Ext.apply({
                xtype       : 'pagingtoolbar',
                dock        : 'bottom',
                displayInfo : true,
                store       : store
            }, pageable));
        }

        if (refreshable) {
            tools.push(Ext.apply({
                type    : 'refresh',
                tooltip : 'Refresh Grid',
                scope   : me,
                handler : 'refreshStore'
            }, refreshable));
        }

        if (tbar.length) {
            docks.push({
                xtype : 'toolbar',
                dock  : 'top',
                items : tbar
            });
        }

        me.dockedItems = docks;
        me.features    = features;
        me.tools       = tools;
        me.tbar        = null;

        me.callParent();
    }
});
```

We now have an abstract grid class that has 5 configs to add certain UI features that are common among grids. Within the `initComponent` method, we go through these configs and add the classes to the grid. The configs can be a simple `Boolean` or can be config `Objects` for the different components associated with that config.

Abstract Benefits

The issue with this is it will cause a little bit of performance as any extra code will. Not only do you have an extra `initComponent` call for each class instance, you also have the configs that get the getter/setter methods created, the conditionals in the `initComponent`, and the `Ext.apply` calls. That's narrow vision thinking though. If you look at it from an entire application perspective, you will save on performance as it should lead to smaller code; likely not smaller code to be executed as the code has to be executed somewhere but smaller code for the browser to download and evaluate.

I haven't sold you on the idea yet? Let's look at it from a maintenance perspective. No doubt, somewhere down the line of development someone will make the decision to add a UI feature. Would you rather go through each grid and add that in there or make a new config in an abstract class that all grid's will inherit? Change one class or many? Still not sold? You're a hard sale. Let's look at it from a QA perspective. Would you rather test one class for these features or many classes duplicating the tests? Still not sold? Guess you aren't in charge of QA then. Maybe you're a project lead or in management. How about time of development? Would you rather pay someone to make one change or many? Abstracts allow you to work faster, implement application features quicker when you have the abstracts setup upfront.

I do this for many different classes that are used frequently: form panel, window, tree but also view controllers (before action in a route maybe?) and stores.

Using overrides

Abstracts are incredibly useful. However, abstracts cannot solve everything. I'm currently working on a new support portal for Sencha, employees (admins) should see different items than customers (users). I could hook into `initComponent` and build the items array there but I wanted to use simple configs and stay away from overriding methods so I have `adminItems` and `userItems` configs I implement like this:

```
Ext.define('MyApp.view.Foo', {
    extend : 'Ext.container.Container',
    xtype  : 'myapp-foo',

    adminItems : [
        //...
    ],

    userItems : [
        //...
    ]
});
```

If I were to handle this in an abstract, I'd have to spread the code around to `Ext.container.Container`, `Ext.panel.Panel`, `Ext.grid.Panel`, etc abstract classes so that all components would handle the different admin/user configs. With the abstract approach, I'd have to spread the logic which is not what I want to do. I could create a singleton utility class and in the abstracts execute a common method on that utility class but there is a better way to do it. Let's keep taking advantage of the class system and override certain Ext JS classes. An example to pick certain items based on if the user is an admin or user we need to determine where the class that handles the items is, which is `Ext.container.Container`, and override it like so:

```
Ext.define('Override.container.Container', {
    override : 'Ext.container.Container',

    adminItems : null,
    userItems  : null,
```

```
    buildItems      : null,
    buildAdminItems : null,
    buildUserItems  : null,

    initComponent : function() {
        var me      = this,
            user    = Portal.user,
            isAdmin = user && user.isAdmin();

        if (me.buildItems) {
            me.items = me.buildItems();
        } else if (me.buildAdminItems && isAdmin) {
            me.items = me.buildAdminItems();
        } else if (me.buildUserItems && user) {
            me.items = me.buildUserItems();
        } else if (me.adminItems && isAdmin) {
            me.items = me.adminItems;
        } else if (me.userItems && user) {
            me.items = me.userItems;
        }

        me.callParent();
    }
});
```

I also go a bit further and allow for `buildItems`, `buildAdminItems` and `buildUserItems` to be methods that I can execute and return items making this override a bit more flexible.

The same process can be put into place for docked items but we need to override `Ext.panel.Panel`:

```
Ext.define('Override.panel.Panel', {
    override : 'Ext.panel.Panel',

    adminDockedItems : null,
    userDockedItems  : null,

    buildDockedItems      : null,
    buildAdminDockedItems : null,
    buildUserDockedItems  : null,

    initComponent : function() {
        var me      = this,
            user    = Portal.user,
            isAdmin = user && user.isAdmin();

        if (me.buildDockedItems) {
            me.dockedItems = me.buildDockedItems();
        } else if (me.buildAdminDockedItems && isAdmin) {
            me.dockedItems = me.buildAdminDockedItems();
        } else if (me.buildUserDockedItems && user) {
            me.dockedItems = me.buildUserDockedItems();
        } else if (me.adminDockedItems && isAdmin) {
            me.dockedItems = me.adminDockedItems;
        } else if (me.userDockedItems && user) {
            me.dockedItems = me.userDockedItems;
        }
```

```
        me.callParent();
    }
});
```

I now have reusable logic for any class. Before, if I wanted to use adminDockedItems in a grid, I'd have to implement the code in `Ext.panel.Panel` and `Ext.grid.Panel` abstracts but with override I only have to impelemnt the code in the `Ext.panel.Panel` override as `Ext.grid.Panel` eventually extends `Ext.panel.Panel`.

The same benefits for abstracts is the same for overrides. Testability, reusability and time are more optimized.

Conclusion

The title of the blog is a bit misleading, it's not one versus the other; I use both methods in my applications. I find doing things like this allows me to be more flexible and quicker and have great success. That million line app I worked on a few years could have been optimized using abstracts and overrides. It was but only to an extent, there were many teams working on the one application and I was only to work on my section.

[source: Abstract vs Override]

Abstract vs Override

Saturday, November 30, 2019 8:24 PM

Developing large single-page applications can be a daunting task. I've seen web applications approach a million lines of code. That's quite a lot of code and performance is a real issue there (luckily it was a B2B application not a consumer facing application). A golden rule in software development, well most things in life really, is that there is always a better way to do something. You may think you have the perfect way to make a pizza but there's likely a better way to make it. The same goes for software development.

Abstracting code

One way to minimize code is to create an abstract class. Ext JS has a great class system that allows simple inheritance. Taking advantage of this class system by creating an abstract class can minimize the code required because you're not having to have the same code spread among several classes.

A simple example I could provide is in my grids I tend to want a couple different things:

- A top docked toolbar with:
    - A search field
    - Button to add a record
- A paging toolbar
- A refresh tool in the title bar

Normally I would extend `Ext.grid.Panel` for all my grids but for those common UI features I'd have to implement them in each grid which is not optimal. There's an better way to do it, create an abstract grid class that extends `Ext.grid.Panel` and has configs to turn on/off the UI features. Here is an example abstract grid class (please don't get overwhelmed by the amount of code):

```
Ext.define('MyApp.view.abstracts.Grid', {
    extend : 'Ext.grid.Panel',
    xtype  : 'myapp-abstracts-grid',

    requires : [
        'Ext.button.Button',
        'Ext.form.field.Text',
        'Ext.grid.feature.Grouping',
        'Ext.toolbar.Paging'
    ],

    config : {
        /**
         * @cfg {Boolean/Object} [createNew=true] `true` to add a {@link
Ext.button.Button} to create a new record.
         *
         * Can also be a config object for the button.
         */
        createNew   : true,
        /**
         * @cfg {Boolean/Object} [grouped=false] `true` to add the {@link
Ext.grid.feature.Grouping} feature to the grid.
         *
         * Can be a config object that will be passed or a boolean.
         */
        grouped     : false,
```

```
        /**
         * @cfg {Boolean/Object} [pageable=true] `true` to add a {@link
Ext.toolbar.Paging}. Will set the grid's
         * store on the toolbar automatically.
         *
         * Can be a config object for the paging toolbar.
         */
        pageable    : true,
        /**
         * @cfg {Boolean/Object} [refreshable=true] `true` to add a refresh
{@link Ext.panel.Tool} to the title bar.
         *
         * Can be a config object for the tool.
         */
        refreshable : true,
        /**
         * @cfg {Boolean/Object} [searchable=true] `true` to add a search
{@link Ext.form.field.Text}.
         *
         * Can also be a config object for the text field.
         */
        searchable  : true
    },

    initComponent : function() {
        var me          = this,
            store       = me.store = Ext.data.StoreManager.lookup(me.store),
            dockedItems = me.dockedItems || [],
            features    = me.features    || [],
            tbar        = me.tbar        || [],
            tools       = me.tools       || [],
            createNew   = me.getCreateNew(),
            grouped     = me.getGrouped(),
            pageable    = me.getPageable(),
            refreshable = me.getRefreshable(),
            searchable  = me.getSearchable();

        if (createNew) {
            tbar.unshift(Ext.apply({
                text    : 'Create',
                iconCls : 'fa fa-plus',
                handler : 'onCreateClick'
            }, createNew));
        }

        if (searchable) {
            //add it first
            tbar.unshift(Ext.apply({
                xtype           : 'textfield',
                enableKeyEvents : true,
                emptyText       : 'search...',
                width           : 300,
                triggers        : {
                    search : {
                        cls     : 'fa fa-search',
```

```
                        handler : 'doSearch'
                    }
                },
                listeners       : {
                    keydown : 'onSearchKeyDown'
                }
            }, searchable));
        }

        if (grouped) {
            features.push(Ext.apply({
                ftype : 'grouping'
            }, grouped));
        }

        if (pageable) {
            dockedItems.push(Ext.apply({
                xtype       : 'pagingtoolbar',
                dock        : 'bottom',
                displayInfo : true,
                store       : store
            }, pageable));
        }

        if (refreshable) {
            tools.push(Ext.apply({
                type    : 'refresh',
                tooltip : 'Refresh Grid',
                scope   : me,
                handler : 'refreshStore'
            }, refreshable));
        }

        if (tbar.length) {
            docks.push({
                xtype : 'toolbar',
                dock  : 'top',
                items : tbar
            });
        }

        me.dockedItems = docks;
        me.features    = features;
        me.tools       = tools;
        me.tbar        = null;

        me.callParent();
    }
});
```

We now have an abstract grid class that has 5 configs to add certain UI features that are common among grids. Within the `initComponent` method, we go through these configs and add the classes to the grid. The configs can be a simple `Boolean` or can be config `Objects` for the different components associated with that config.

Abstract Benefits

The issue with this is it will cause a little bit of performance as any extra code will. Not only do you have an extra `initComponent` call for each class instance, you also have the configs that get the getter/setter methods created, the conditionals in the `initComponent`, and the `Ext.apply` calls. That's narrow vision thinking though. If you look at it from an entire application perspective, you will save on performance as it should lead to smaller code; likely not smaller code to be executed as the code has to be executed somewhere but smaller code for the browser to download and evaluate.

I haven't sold you on the idea yet? Let's look at it from a maintenance perspective. No doubt, somewhere down the line of development someone will make the decision to add a UI feature. Would you rather go through each grid and add that in there or make a new config in an abstract class that all grid's will inherit? Change one class or many? Still not sold? You're a hard sale. Let's look at it from a QA perspective. Would you rather test one class for these features or many classes duplicating the tests? Still not sold? Guess you aren't in charge of QA then. Maybe you're a project lead or in management. How about time of development? Would you rather pay someone to make one change or many? Abstracts allow you to work faster, implement application features quicker when you have the abstracts setup upfront.

I do this for many different classes that are used frequently: form panel, window, tree but also view controllers (before action in a route maybe?) and stores.

Using overrides

Abstracts are incredibly useful. However, abstracts cannot solve everything. I'm currently working on a new support portal for Sencha, employees (admins) should see different items than customers (users). I could hook into `initComponent` and build the items array there but I wanted to use simple configs and stay away from overriding methods so I have `adminItems` and `userItems` configs I implement like this:

```
Ext.define('MyApp.view.Foo', {
    extend : 'Ext.container.Container',
    xtype  : 'myapp-foo',

    adminItems : [
        //...
    ],

    userItems : [
        //...
    ]
});
```

If I were to handle this in an abstract, I'd have to spread the code around to `Ext.container.Container`, `Ext.panel.Panel`, `Ext.grid.Panel`, etc abstract classes so that all components would handle the different admin/user configs. With the abstract approach, I'd have to spread the logic which is not what I want to do. I could create a singleton utility class and in the abstracts execute a common method on that utility class but there is a better way to do it. Let's keep taking advantage of the class system and override certain Ext JS classes. An example to pick certain items based on if the user is an admin or user we need to determine where the class that handles the items is, which is `Ext.container.Container`, and override it like so:

```
Ext.define('Override.container.Container', {
    override : 'Ext.container.Container',

    adminItems : null,
    userItems  : null,
```

```
    buildItems      : null,
    buildAdminItems : null,
    buildUserItems  : null,

    initComponent : function() {
        var me      = this,
            user    = Portal.user,
            isAdmin = user && user.isAdmin();

        if (me.buildItems) {
            me.items = me.buildItems();
        } else if (me.buildAdminItems && isAdmin) {
            me.items = me.buildAdminItems();
        } else if (me.buildUserItems && user) {
            me.items = me.buildUserItems();
        } else if (me.adminItems && isAdmin) {
            me.items = me.adminItems;
        } else if (me.userItems && user) {
            me.items = me.userItems;
        }

        me.callParent();
    }
});
```

I also go a bit further and allow for `buildItems`, `buildAdminItems` and `buildUserItems` to be methods that I can execute and return items making this override a bit more flexible.

The same process can be put into place for docked items but we need to override `Ext.panel.Panel`:

```
Ext.define('Override.panel.Panel', {
    override : 'Ext.panel.Panel',

    adminDockedItems : null,
    userDockedItems  : null,

    buildDockedItems      : null,
    buildAdminDockedItems : null,
    buildUserDockedItems  : null,

    initComponent : function() {
        var me      = this,
            user    = Portal.user,
            isAdmin = user && user.isAdmin();

        if (me.buildDockedItems) {
            me.dockedItems = me.buildDockedItems();
        } else if (me.buildAdminDockedItems && isAdmin) {
            me.dockedItems = me.buildAdminDockedItems();
        } else if (me.buildUserDockedItems && user) {
            me.dockedItems = me.buildUserDockedItems();
        } else if (me.adminDockedItems && isAdmin) {
            me.dockedItems = me.adminDockedItems;
        } else if (me.userDockedItems && user) {
            me.dockedItems = me.userDockedItems;
        }
```

```
        me.callParent();
    }
});
```

I now have reusable logic for any class. Before, if I wanted to use adminDockedItems in a grid, I'd have to implement the code in `Ext.panel.Panel` and `Ext.grid.Panel` abstracts but with override I only have to impelemnt the code in the `Ext.panel.Panel` override as `Ext.grid.Panel` eventually extends `Ext.panel.Panel`.

The same benefits for abstracts is the same for overrides. Testability, reusability and time are more optimized.

Conclusion

The title of the blog is a bit misleading, it's not one versus the other; I use both methods in my applications. I find doing things like this allows me to be more flexible and quicker and have great success. That million line app I worked on a few years could have been optimized using abstracts and overrides. It was but only to an extent, there were many teams working on the one application and I was only to work on my section.

[source: Abstract vs Override]

A cause in poor performance

Saturday, November 30, 2019 8:24 PM

Over the years of working with Ext JS and Sencha Touch, I've gotten to see lots of code both from projects I've worked on and for the many code reviews I've done. One common thing I always see, and I'm not exaggerating when I say always, is something that can degrade performance quickly both on the DOM and JavaScript execution level. It's called "overnesting" and it's an epidemic.

What is "overnesting"?

Ext JS and Sencha Touch work with Component and Layout systems where you layout your UI by instantiating Components and position/size them using a layout. Overnesting is when you nest a component within a container when that component does not need to be nested. When I say "component", I mean any component not just `Ext.Component`; likewise, "container" doesn't just mean `Ext.container.Container` it also means for any subclass like `Ext.form.Panel` or `Ext.grid.Panel`.

Affect of "overnesting"

The side affects of overnesting can include:

- Extra DOM nodes
- Extra JavaScript execution
- Extra layout runs by the browser and/or the framework
- Layout issues, components not laid out properly on all platforms
    - IE may display differently than Chrome

Any of these are bad and will happen if you have overnesting in your application.

Example

Let's look at a sample of overnesting:

```
Ext.create('Ext.tab.Panel', {
    items : [
        {
            xtype  : 'panel',
            title  : 'Users',
            layout : 'fit',
            tbar   : [
                { text : 'Create User' }
            ],
            items  : [
                {
                    xtype   : 'gridpanel',
                    store   : 'Users',
                    columns : [
                        {
                            text      : 'Name',
                            dataIndex : 'name',
                            flex      : 1
                        }
                    ]
                }
            ]
```

```
        }
    ]
});
```

In this example, we create a tab panel with a single tab that is a panel. The panel has a top docked toolbar with a Create User button and also a grid that is laid out using fit layout. While this will work and look as you would expect, it's overnesting. The panel is a layer that is not needed just to have a top docked toolbar, the grid can have a top docked toolbar and will look exactly the same:

```
Ext.create('Ext.tab.Panel', {
    items : [
        {
            xtype   : 'gridpanel',
            store   : 'Users',
            tbar    : [
                { text : 'Create User' }
            ],
            columns : [
                {
                    text      : 'Name',
                    dataIndex : 'name',
                    flex      : 1
                }
            ]
        }
    ]
});
```

Spotting "overnesting"

That was a very simple example, applications are going to be much more complex. The first tip I could give is you need to be aware of what your code is doing. A browser does a very good job at showing the DOM structure, it's a tree of nodes. Your application's UI is also a tree. The overnesting example could look like this in a tree format:

```
--Ext.tab.Panel
----Ext.panel.Panel
------Ext.grid.Panel
```

If you start to visualize your layout as a tree, you'll start to realize how things are laid out and can then spot possible overnesting.

Another tip would be to use tools or scripts. For example, Sencha AppInspector can display your components in an actual `Ext.tree.Panel` and give you a bit more information on each component that will help you debug your application. I personally prefer having intimate knowledge over the projects I work on but AppInspector can aid your understanding.

Conclusion

Overnesting can cause poor performance and all sorts of issues in your application. Even if overnesting isn't a cause of a bug in an application I'm looking at, it's still something that has to be addressed. To me, it's not optional; overnesting must be solved.

[source: A cause in poor performance]

Store Listeners

Saturday, November 30, 2019 8:24 PM

No doubt, you likely have a store or two in your Ext JS/Touch application. Have you had to add a listener to the store to do something? Maybe update a component when the store loads? Or take any action based on it? Like anything, there's a good way to do it but there may be a better way. Let's discuss some of these ways.

Disclaimer: Some of these techniques use private methods and therefore need to use at your own risk. I have used them and feel confident they will not change but it always could.

Listening to a store load in a component

One of the most popular components in Ext JS is the grid (`Ext.grid.Panel`) and it so happens to use a store which likely loads data remotely. A configuration I have in my abstract grid class for my projects (for more on abstracts, please see my blog on Abstract vs Override) is to automatically setup a paging toolbar. With it, I like to hide the paging toolbar when there is only one page of data. So with that requirement, I need to take action on when the store loads to then show or hide the paging toolbar.

First, let's look at an example abstract grid that will setup the paging toolbar:

```
Ext.define('Abstracts.grid.Panel', {
    extend : 'Ext.grid.Panel',
    xtype  : 'abstracts-gridpanel',

    config : {
        autoHidePaging : true,
        pageable       : true
    },

    initComponent : function() {
        var me = this,
            dockedItems = me.dockedItems || [],
            pageable = me.getPageable();

        me.initStore();

        if (pageable) {
            dockedItems.push(pageable);
        }

        me.dockedItems = dockedItems;

        me.callParent();
    },

    applyPageable : function(pageable, oldPageable) {
        if (pageable) {
            if (!Ext.isObject(pageable)) {
                pageable = {};
            }

            Ext.applyIf(pageable, {
                dock  : 'bottom',
                store : this.initStore()
```

```
            });
        }

        return Ext.factory(pageable, Ext.toolbar.Paging, oldPageable);
    },

    initStore : function() {
        var me = this,
            store = me.store;

        if (store && !store.isStore) {
            store = me.store = Ext.data.StoreManager.lookup(store);

            //overwrite this method to simply return the store
            me.initStore = me.getStore;
        }

        return store;
    }
});
```

This is just a snippet but will add a paging toolbar to the grid with the benefit of automatically setting the grid's store on the toolbar. If we use it now, the grid will work and display the paging toolbar and allow users to navigate within the pages.

Now we need to work with our requirement to show/hide the toolbar dependant on number of pages available. I'm sure most would add a load listener to the store like this:

```
store.on('load', me.handleStoreLoad);
```

and then in the handleStoreLoad method take action which would work but there is a better way. We know the grid already takes action on store loading right? Otherwise how could it display the data? So isn't there already a load listener somewhere? Yes there is! `Ext.grid.Panel` extends `Ext.panel.Table` which sets a load listener to execute a `onStoreLoad` method. Instead of adding our own listener, why not just take advantage of this function? First, we need to understand the method. On `Ext.panel.Table`, it's defined as such:

```
// template method meant to be overriden
onStoreLoad: Ext.emptyFn,
```

So it's just a template method not doing anything. Checking `Ext.grid.Panel` it doesn't override it so there is no ancestor that does anything with the method and therefore we now know that we don't need to execute `this.callParent` when we override it. So we can add our `onStoreLoad` method to show and hide the paging toolbar as such:

```
onStoreLoad : function(store) {
    if (this.getAutoHidePaging()) {
        var toolbar = this.getPageable();

        if (toolbar) {
            if (store.getTotalCount() / store.pageSize <= 1) {
                //only one page of data available
                toolbar.hide();
            } else {
                //more than one page is available
                toolbar.show();
```

```
            }
        }
    }
}
```

We just acted on a store load without setting our own store load listener and taking advantage of the class system.

Listening within a store

What if we wanted to do something within a subclass of `Ext.data.Store` when the store loads? We could do this:

```
Ext.define('MyApp.store.Users', {
    extend : 'Ext.data.Store',
    alias  : 'myapp-users',

    proxy : {
        type : 'ajax',
        url  : 'foo.json'
    },

    constructor : function(config) {
        this.callParent([config]);

        this.on('load', this.doSomething);
    },

    doSomething : function() {}
});
```

Where we set a listener but like what we did with the grid, we can do some research to see if we can do something without setting an event listener. Without going into details on the process a store and proxy go through to load, the proxy is what does the actual request and the store is just a holder of records. But the store has to take action when the proxy finishes it's loading so it can add the records to it's internal collection. The method on the store is cleverly named `onProxyLoad` which is also where the load event is actually fired. So we can then hook into this method like such:

```
Ext.define('MyApp.store.Users', {
    extend : 'Ext.data.Store',
    alias  : 'myapp-users',

    proxy : {
        type : 'ajax',
        url  : 'foo.json'
    },

    onProxyLoad : function(operation) {
        this.callParent([operation]);

        this.doSomething();
    },

    doSomething : function() {}
});
```

And now `doSomething` will execute whenever the proxy returns it's loading without setting any listener.

Conclusion

I hope this method of reacting to store loads can be thought of for other functions not just store loading. Overriding methods can perform better than listening to events but do have some drawbacks. When overriding methods, you always need to make sure to research that method so you will not break anything.

[source: Store Listeners]

mon vs on

Saturday, November 30, 2019 8:24 PM

A common method when developing an application is to be event driven. This allows your code to be very flexible. Not being event driven can lead to code that is very intimate to the current state of your application. Change or add something later on and you must go through your application and change the different parts where as being event driven allows an event to be fired and forgotten about. Anything listening to that event can then take action on that event but not be dependant on what fired the event.

A simple example is an email application. Say a user sends an email, when the email has been sent and the server returns success, the callback may fire an event to tell anything in the application that the user sent an email. The Sent folder list may listen to this event in order to reload the list. This email may have been sent from different forms, a create email form may be different than a quick reply form but that doesn't matter, either form may fire the same event and the Sent folder list doesn't care where it came from. This is flexibility and in my eyes, creates much safer code.

Adding listeners in Ext JS

Ext JS allows many different ways to add listeners but today I want to speak about `mon` and `on` and the differences. First, let's start with `on`.

on

The simplest of the two is the `on` method which can simply add an event listener:

```
component.on('foo', someFunc, component);
```

You can also pass an object for a convenient way to add multiple listeners:

```
component.on({
    scope : component,
    foo   : someFunc,
    bar   : someOtherFunc
});
```

This will add listeners for the `foo` and `bar` events and scope them to the `component` variable. In Ext JS 4 and newer, listeners defined this way will automatically get removed when the component is destroyed thanks to the `clearListeners` method being executed in the `destroy` method of `Ext.Component`.

mon

`mon` works just like `on` when defining a listener only the first argument must be a class to add the listeners too. That's a bit confusing of a statement right? Let's look at an example:

```
component.mon(subClass, 'foo', someFunc, component);
```

What this is doing, is adding a listener for the `foo` event that will be fired on the `subClass` variable. Once fired, it will execute the `someFunc` function scoped to the `component` variable but the listener is added to the `component` but the event will be fired on the `subClass`.

The benefit here is the component can listen to an event on something else but when the component is destroyed, this listener is removed. The component may be destroyed but the `subClass` variable may still live on; the component is managing the `foo` event listener.

Like the on method, you can also pass an object:

```
component.mon(subClass, {
    scope : component,
    foo   : someFunc,
    bar   : someOtherFunc
});
```

Difference

Let's think of an `Ext.Component` instance that wants to listen to a `store` load. You can use on like this:

```
store.on('load', this.onStoreLoad, this);
```

However, when the `Ext.Component` is destroyed, the listener will not be removed because the event listener is on the `store` not the `Ext.Component`. This causes a memory leak and the next `store` load after the component is destroyed will likely throw an error, both bad things. This is where `mon` would be a better use:

```
this.mon(store, 'load', this.onStoreLoad, this);
```

This adds a `load` listener on the store but is being managed by the `Ext.Component`. When the `Ext.Component` is destroyed, the listener is removed. The `store` is then free to live on and fire it's `load` event without errors being thrown and no memory leaks are present.

[source: mon vs on]

A Naming Strategy

Saturday, November 30, 2019 8:25 PM

You just got contracted or assigned a new app! Wow, that's exciting! All you want to do is just jump in and start coding right? Well, you need to slow down and think about the application first. Don't skip the pre-planning stage of the programming. This is separate from pre-planning the application where you come up with all the features the app will have, as developers (the implementors) we need to think of how to develop the app before we actually develop the app.

Classes

In an Ext JS or Sencha Touch application, you will no doubt be creating a lot of classes. Before we start programming, we need to think of how we are going to organize the code. You will likely be refactoring as you develop but thinking a bit up front will help quite a bit.

First, you have a list of features. You need to take that list and come up with a logical class structure. For example, maybe you have a user management part of the application. You'll likely need a grid to display a list of users, maybe a detail view of the user and maybe a form to edit and create a user. So right there we know we need 3 classes that belong to a user "feature":

- user
    - Detail
    - Form
    - Grid

To add to that, say your application is an email program so you'll need a form to send an email and of couse a view to display the email so now you have an email and user "features":

- email
    - Detail
    - Form
- user
    - Detail
    - Form
    - Grid

As you can see, we are already starting to think of what classes our application is likely to have and even a structure.

Naming our classes

Since we have a structure of classes, the naming of the classes should follow that structure. Say our application name is `Email`, therefore our classes should be named:

- `Email.view.email.Detail`
- `Email.view.email.Form`
- `Email.view.user.Detail`
- `Email.view.user.Form`
- `Email.view.user.Grid`

You can logically see the class names describe the same structure that we planned out. With Ext JS 5, you're likely to have some `ViewControllers` for your forms and grids:

- `Email.view.email.Detail`

- `Email.view.email.Form`
- `Email.view.email.FormController`
- `Email.view.user.Detail`
- `Email.view.user.Form`
- `Email.view.user.FormController`
- `Email.view.user.Grid`
- `Email.view.user.GridController`

Aliases

Another part of developing an Ext JS and Sencha Touch application is providing aliases (or xtypes for components). These are string representations of our classes that can be used for lazy instantiation. I've seen many naming schemes for naming their aliases to save on space. I see xtypes being like `usergrid` and `emailform` which are pretty descriptive in a simple application but the issue with this is down the road in development the more complex an application gets and not having a naming standard may not be the easiest to remember what class is for an alias when you or someone new to the project comes across the alias somewhere in the application. I keep my alias names closely associated to my class names so when I come across an alias, I know exactly what class that alias is associated with. Here is a mapping of what I'd give my views:

- `email-email-detail`
    - class: `Email.view.email.Detail`
- `email-email-form`
    - class: `Email.view.email.Form`
- `email-user-detail`
    - class: `Email.view.user.Detail`
- `email-user-form`
    - class: `Email.view.user.Form`
- `email-user-grid`
    - class: `Email.view.user.Grid`

You can see the alias for my views is very close to the class name. In fact, all I did was removed the `.view` portion, lowercased it and replaced periods with dashes. But that's exactly what I want. Now when I come across an xtype of `email-user-form` I know that it correlates to the `Email.view.user.Form` and I know exactly where to go to look at the source of that view. I'm even fine with the duplication of `email-email-detail` as it needs to conform to my naming standard. The standard needs to be adhered to regardless of duplication.

For the `ViewControllers`, I alias them like so:

- `controller.email-email-form`
    - class: `Email.view.email.FormController`
- `controller.email-user-form`
    - class: `Email.view.user.FormController`
- `controller.email-user-grid`
    - class: `Email.view.user.GridController`

Here I did pretty much the same thing as the views' xtype. I removed the `.view` portion, lowercased the name, replaced periods with dashes only with a `ViewController` I moved the controller part to the

beginning of the alias as you have to use the `alias` property and therefor need to give it the `controller` prefix. This gets confusing without seeing the actual code so here is the few lines for the user grid view and controller:

```
Ext.define('Email.view.user.Grid', {
    extend : 'Ext.grid.Panel',
    xtype  : 'email-user-grid',

    requires : [
        'Email.view.user.GridController
    ],

    controller : 'email-user-grid',

    //...
});

Ext.define('Email.view.user.GridController', {
    extend : 'Ext.app.ViewController',
    alias  : 'controller.email-user-grid',

    //...
});
```

Hopefully the code makes a bit more sense about what I mean. If you look at the `Email.view.user.Grid` source and you see the `controller` property, you should be able to immediately know where to look at for the `ViewController`. The reason why I remove the controller from the end of the `controller.email-user-grid` alias is because it's redundant, the prefix is telling you what the alias is so `controller.email-user-gridcontroller` is redundant but if you want it on your alias there is nothing wrong with it.

You may be wondering why I have my aliases prefixed with the `email` part that is the application name lowercased. I use Sencha Cmd and often times have packages and each package has it's own name and the classes are named with that package name. Having the aliases prefixed with that application or package name tells me what class that is. I may have a `MyPackage.view.user.Grid` class so I need to have an xtype of `mypackage-user-grid`. If I just had it as `user-grid` then does that tell me its for `MyPackage.view.user.Grid` or is it for `Email.view.user.Grid`? Do I need to now name my package classes dependent on what I have in my applications that the package may added to? No. Prefix the aliases to allow for present and future-proofing.

Reasons

The main reason I structure my class names and aliases is for maintenance. Six months down the road, I may not remember what I did but since I have a naming standard I don't rely on my memory. Six months down the road, I may hire someone to work on my project and with a naming standard they could more easily hit the ground running instead of searching through the source trying to find what class some alias belongs too. Yes, `Ext.ClassManager` holds a mapping of aliases to classes but IMO you shouldn't have to run the application in order to understand the code especially when it's so easy to structure your names.

Conclusions

I'm a big believer in self-deprecation programming. I know many like to program in their own way which can easily give them job security. I want to develop the project so that if something happens to me, like I change jobs and let's face it, this happens often in our programming field, that someone else can pick up where I left off and the project remains successful. One way to do this, is proper class name structuring; you can read the source and understand the structure very easily.

[source: A Naming Strategy]

What is callParent and callSuper

Saturday, November 30, 2019 8:25 PM

One of the best things about Ext JS is it's class system. Even though JavaScript doesn't have a class system with inheritance, Ext JS utilizes different JavaScript methods to create inheritance. This enables classes to extend one another to give power, flexibility and code reuse in a simple api otherwise not present in native JavaScript.

Introduction

First, let's setup a couple classes `Bar` which extends `Foo`:

```
Ext.define('Foo', {
    someProp : 3,

    someMethod : function() {
        console.log('Foo', 'someMethod');
    }
});

Ext.define('Bar', {
    extend : 'Foo',

    anotherProp : 'test',
    someProp    : 4
});
```

If we look at this code, `Foo` is defined with `someProp` and `someMethod` members on that class. `Bar` is defined and extends `Foo`, adds an `anotherProp` member and overwrites `someProp` to have a different value than was defined on `Foo`. In this case, `Bar` will inherit the `someMethod` and therefore instances of `Bar` can execute the `someMethod` method. Let's look at instantiation and execution:

```
var foo = new Foo();

foo.someProp === 3;
foo.someMethod(); //logs `Foo someMethod` to the console

var bar = new Bar();

bar.anotherProp === 'test';
bar.someProp === 4;
bar.someMethod(); //logs 'Foo someMethod' to the console
```

Using `callParent`

We have a need to add logic to `Bar`'s `someMethod` method but we still need to execute the `someMethod` that was inherited from `Foo`. If we simply add `someMethod` to `Bar` like this:

```
Ext.define('Bar', {
    extend : 'Foo',

    anotherProp : 'test',
    someProp    : 4,
```

```
    someMethod : function() {
        console.log('Bar', 'someMethod');
    }
});
```

When we execute `bar.someMethod();` it will now only log out `Bar someMethod` and will not execute the `someMethod` method on the `Foo` class because we overrode the method. We can still call `Foo`'s `someMethod` by using the `callParent` method:

```
Ext.define('Bar', {
    extend : 'Foo',

    anotherProp : 'test',
    someProp    : 4,

    someMethod : function() {
        console.log('Bar', 'someMethod');

        this.callParent();
    }
});
```

Now when `bar.someMethod();` is executed, we will get `Bar someMethod` **and** `Foo someMethod` both logged to the console because `this.callParent();` calls the superclass' (`Foo`) `someMethod` method.

Function arguments

Thus far, the `someMethod` method did not accept any arguments and therefore we didn't need to deal with them and how to pass the arguments to the superclass' method. Let's modify `Foo` to handle a couple arguments:

```
Ext.define('Foo', {
    someProp : 3,

    someMethod : function(callback, scope) {
        console.log('Foo', 'someMethod');

        if (callback) {
            callback.call(scope || this);
        }
    }
});
```

We changed the signature of the `someMethod` method to accept `callback` and `scope` arguments which are optional but we need to now handle passing the arguments in our `someMethod` method in the `Bar` class in order to keep the behavior of the `Foo someMethod` method.

We can do this two ways. The preferred way is to know what the arguments are and pass an array of the arguments to the `callParent` method call. This may take some educating yourself on what arguments are valid for that method by reading the source for the superclass and it's ancestors. The other way is to use the special `arguments` keyword which holds the arguments for the current function block. Here are the two methods in action:

```
Ext.define('Bar', {
    extend : 'Foo',
```

```
    anotherProp : 'test',
    someProp    : 4,

    someMethod : function(callback, scope) {
        console.log('Bar', 'someMethod');

        this.callParent([callback, scope]);
    }
});
```

or with the `arguments` keyword:

```
Ext.define('Bar', {
    extend : 'Foo',

    anotherProp : 'test',
    someProp    : 4,

    someMethod : function(callback, scope) {
        console.log('Bar', 'someMethod');

        this.callParent(arguments);
    }
});
```

I personally always try to know what arguments are possible and use the first method. Documentation can really help here or simply reading the source from that class' ancestors. If you know a method will never have any arguments, `initComponent` in `Ext.Component` for example, then you can simply execute `callParent` without passing any parameters: `this.callParent();`

Using `callSuper`

Like `callParent`, `callSuper` allows you to call an ancestor method when overwriting one. There are two differences:

- `callSuper` is only usable within an override
- While `callParent` calls the inherited method, `callSuper` will skip the directly inherited method and call the next level's method.

Let's create an override to `Bar` to explore how to use the `callSuper` method:

```
Ext.define('Override.Bar', {
    override : 'Bar',

    someMethod : function(callback, scope) {
        console.log('Override.Bar', 'someMethod');
    }
});
```

In this example, we are globally overwriting the `someMethod` method on `Bar`. When we execute `bar.someMethod();` we will only see the `Override.Bar someMethod`. Our need to is not call the `someMethod` method on the `Bar` class but skip it and call the method on the `Foo` class. We can do this with `callSuper`:

```
Ext.define('Override.Bar', {
    override : 'Bar',
```

```
    someMethod : function(callback, scope) {
        console.log('Override.Bar', 'someMethod');

        this.callSuper([callback, scope]);
    }
});
```

Now when `bar.someMethod();` is executed, we will get both `Override.Bar someMethod` and `Foo someMethod` logged to the console. Also note that passing arguments works exactly like it did with `callParent`.

Handling returning something

What if the `someMethod` in the `Foo` class returned something? How do we handle this with `callParent` and `callSuper`? Well, it's actually quite simple. First, let's look at the changes to `Foo`:

```
Ext.define('Foo', {
    someProp : 3,

    someMethod : function(callback, scope) {
        console.log('Foo', 'someMethod');

        if (callback) {
            callback.call(scope || this);
        }

        return this;
    }
});
```

Now `someMethod` is chainable as it returns `this` which is likely the `Foo` class. Now we need to handle this in the `Bar` class where we use `callParent`:

```
Ext.define('Bar', {
    extend : 'Foo',

    anotherProp : 'test',
    someProp    : 4,

    someMethod : function(callback, scope) {
        console.log('Bar', 'someMethod');

        return this.callParent([callback, scope]);
    }
});
```

All I did here was add `return`, `callParent` is just a placeholder for calling the superclass' method and therefore will return whatever that method returns. The same goes for when `callSuper` is used:

```
Ext.define('Override.Bar', {
    override : 'Bar',

    someMethod : function(callback, scope) {
        console.log('Override.Bar', 'someMethod');

        return this.callSuper([callback, scope]);
    }
```

```
});
```

What if you need to do something with what is returned from `callParent` or `callSuper`? Just cache the execution to a local variable and return that variable. Once again this is exactly the same between `callParent` and `callSuper` so I will only show one:

```
Ext.define('Override.Bar', {
    override : 'Bar',

    someMethod : function(callback, scope) {
        console.log('Override.Bar', 'someMethod');

        var ret = this.callSuper([callback, scope]);

        console.log('Override.Bar', ret.someProp);

        return ret;
    }
});
```

With all the classes and the override in place, when `bar.someMethod();` is executed it will return `this` (which will be the instance of Bar) but will now log these in this order: `Override.Bar someMethod`, `Foo someMethod` and `Override.Bar 4`. Couple things to note here, since we used `callSuper` in the override, the `someMethod` defined on the `Bar` class was not executed as it was skipped due to using `callSuper` and the scope is the instance that was created which is that of the `Bar` class (since we used `var bar = new Bar();` in the beginning) which is why `ret.someProp` returns 4 **not** 3.

[source: What is callParent and callSuper]

Overriding methods safely

Saturday, November 30, 2019 8:25 PM

In the article Abstract vs Override, I talked about how to use overrides in an application to globally change Ext JS to work for your application. I was recently asked for the best way to override a method by a colleague and that made me think that talking about some techniques may help. It's not enough to know how to do it, but to know some real-world techniques is helpful.

Introduction

There are a couple use cases for overriding, most popular is to globally change the behavior or to fix a bug. If you're not familiar with other programming languages that have a class structure with inheritance, the best way to think of a class structure is a tree struture or like a file system. A class (called a subclass) can extend another class (called a superclass) where the subclass will inherit all members (configs, properties and methods) from the superclass. Overriding allows you to hook into any of these classes to override that member.

`Ext.define` vs `Ext.override`

First, let's look at the anatomy of how to override. In previous Ext JS versions you could use `Ext.override`:

```
Ext.override(Ext.Component, {
    foo : 'bar'
});
```

Or the class has a static `override` method also:

```
Ext.Component.override({
    foo : 'bar'
});
```

And while this still works, it's better to use `Ext.define`:

```
Ext.define('Override.Component', {
    override : 'Ext.Component',

    foo : 'bar'
});
```

While this is more characters, it comes with two benefits:

- Similar API to using `Ext.define` to extend a class.
- Override can be dynamically loaded and required using Cmd or plain `Ext.Loader` and then can also be organized in a file system.

It's recommended to use `Ext.define` and defining one override per file. Cmd has the ability to auto-include overrides for you and include them when you build an application.

Overriding Properties

One of the basic and simplest usages of an override is overriding properties. Looking at the previous code snippets, we actually see how to override properties. Of course in that snippet the override didn't actually override anything, it added a property onto `Ext.Component`. Looking at the API documentation for `Ext.Component` we can see many properties/configs such as the `hideMode` config which will control how the component is hidden. By default it's value is `display` which will hide the

component but using the CSS `display:none` style which will cause the space taken up by the component not to be reserved. If we change `hideMode` to `visibility` then when the component is hidden the space is still reserved for the component it's just not visible due to now using the CSS `visibility:hidden` style. We can override this to affect any `Ext.Component` or subclass like so:

```
Ext.define('Override.Component', {
    override : 'Ext.Component',

    hideMode : 'visibility'
});
```

Now when we hide a component, the space where the component was is still reserved for the component, you just cannot visually see the component anymore. Of course beware, this will override every single `Ext.Component` and subclass.

Overriding Methods

Let's say we have a `Foo` class that has a `someMethod` method defined on it:

```
Ext.define('Foo', {
    someMethod : function() {
        console.log('Foo', 'someMethod');
    }
});
```

If we instantiate `Foo` and execute the `someMethod`, we will get `Foo someMethod` logged out to the console.

We find that there is a bug in it but we don't have control over the `Foo` class, say it's in some third party code where we cannot edit the source so we need to override the `someMethod` method. We can do this just like how we overrode a property:

```
Ext.define('Override.Foo', {
    override : 'Foo',

    someMethod : function() {
        console.log('Override.Foo', 'someMethod');
    }
});
```

Now if we instantiate `Foo` and execute `someMethod`, we will only get `Override.Foo someMethod` logged out to the console. We will **not** see the `Foo someMethod` logged because we overrode the method; it's like the `someMethod` defined in the `Foo` class never existed.

Using `callParent`

In the last override, we overrode the `someMethod` method on the `Foo` class to fix a bug. It's possible that the fix to the bug still needs to call the method that is being overridden, maybe we need to still have the `Foo someMethod` logged out. This is where the `callParent` method comes in, it tracks what method was overridden to make it still executable:

```
Ext.define('Override.Foo', {
    override : 'Foo',

    someMethod : function() {
        console.log('Override.Foo', 'someMethod');

        this.callParent();
```

```
    }
});
```

So now if we instantiate Foo and execute someMethod we will get Override.Foo someMethod logged to the console but we will also get Foo someMethod logged out thanks to the this.callParent(); call.

Using callSuper

Ext JS has a large class system where there are multiple levels of inheritance. There may be a time when one class extends another class and the subclass calls it's superclass' method. Say we need to override the subclass' method to fix a bug but instead of calling the overridden method using callParent we need to skip that method and call the superclass' method. This is where callSuper comes in handy. First, let's look at a Bar class that extends Foo and uses callParent:

```
Ext.define('Bar', {
    extend : 'Foo',

    someMethod : function() {
        console.log('Bar', 'someMethod');

        this.callParent();
    }
});
```

If we instantiate Bar and execute someMethod we will get all Bar someMethod, Override.Foo someMethod and Foo someMethod logged out to the console. Remember, we still have Override.Foo override in place.

Let's see what callSuper does when we override the someMethod on Bar with this override:

```
Ext.define('Override.Bar', {
    override : 'Bar',

    someMethod : function() {
        console.log('Override.Bar', 'someMethod');

        this.callSuper();
    }
});
```

Now if we instantiate Bar and execute someMethod we should **not** see the Bar someMethod logged to the console because this.callSuper(); should skip that method. And sure enough, we see Override.Bar someMethod, Override.Foo someMethod and Foo someMethod logged to the console.

Difference of callParent vs callSuper

The difference between callParent and callSuper is what method ends up being executed. callParent will always call the method being overridden where callSuper will skip the method being overridden and call a superclass' method.

Overriding Singletons

I'm a big fan of having a singleton class to manage certain things like application configs. Let's say we have something like this for a singleton (simplified than what I have):

```
Ext.define('MyApp.Config', {
    singleton : true,
```

```
    config : {
        configs : null
    },

    constructor : function() {
        this.initConfig();
        this.callParent();
    },

    get : function(key) {
        return this.getConfigs()[key];
    }
});
```

Without our code we can do things like `MyApp.Config.get('foo')`. But there in an issue with the `get` method, what if `this.getConfigs()` returns a non-Object such as the `get` method was executed before an Xhr call finishes and therefore `this.getConfigs()` will return `null`. We need to handle this case.

For demonstration purposes, say we cannot edit the source for `MyApp.Config` and therefore we need to use an override. Of course, if you have control over the file, edit the source instead of creating an override.

```
Ext.define('Override.Config', {
    override : 'MyApp.Config',

    get : function(key) {
        var configs = this.getConfigs();

        return configs ? configs[key] : null;
    }
});
```

Now when we use `MyApp.Config.get('foo')` we are protected from whether or not there are configs. `callParent` and `callSuper` work just like overrides on class definitions:

```
Ext.define('Override.Config', {
    override : 'MyApp.Config',

    get : function(key) {
        return this.getConfigs() ?
            this.callParent([key]) :
            null;
    }
});
```

This override will execute `this.callParent([key])` if `this.getConfigs()` returns truthy else will return `null`.

More Reading

I also split off some more reading about `callParent` and `callSuper` in another blog: [What is callParent and callSuper](#).

[source: Overriding methods safely]

rootProperty as a function!

Saturday, November 30, 2019 8:25 PM

When I design an endpoint to return some JSON data for consumption in a grid, I always nest the data in the `data` property so that I can provide a total count in order to support pagination. So most of my server side scripts are setup to nest data. Something like this:

```
{
    "success": true,
    "total": 4375,
    "data": [...]
}
```

I setup my store to read this via something like this:

```
Ext.define('MyApp.view.Foo', {
    extend : 'Ext.data.Store',

    proxy : {
        type   : 'ajax',
        url    : '/foo',
        reader : {
            rootProperty : 'data'
        }
    }
});
```

However, when using `Ext.data.TreeStore`, it wants to use the same property to find the children of nodes. By default, it uses the `children` property and you can change this by using the `rootProperty` like we have above (there are a couple different ways really). So if you have your nodes nested within the `data` property but still use `children` to have your child nodes in, it won't find any child nodes because you have two different roots, `data` and `children`.

So how can we handle this use case? Well, first I'd say fix your server to not nest the data so that things will work great within Ext JS. But let's be real for a second, the people in charge of your server-side code are likely a bit reluctant to change. You can buy them a steak dinner and they may still not change, you just have to deal with it.

Don't worry, Ext JS actually has a way to handle this. If you look at the documentation for the `rootProperty` config it says that it accepts a string for a value. Well, that's not entirely true and the docs will get fixed to reflect this but the `rootProperty` can actually accept a function and you return child nodes from the data provided. Let's look at some sample data:

```
{
    "data" : [
        {
            "text"     : "Foo",
            "expanded" : true,
            "children" : [
                {
                    "text" : "Bar",
                    "leaf" : true
                }
            ]
```

```
        }
    ]
}
```

So we have our child nodes nested within the `data` property but we also use `children` to denote our child nodes. We can define out `rootProperty` to work with this like:

```
rootProperty : function(node) {
    return node.data || node.children;
}
```

With this it returns `node.data` if it's truthy or it will return `node.children`. So if `data` exists, return it, else return `children`. It's as easy as that! Here's a live fiddle:

[source: rootProperty as a function!]

Use of a base url in a utility class

Saturday, November 30, 2019 8:25 PM

I had an idea the other day and I'm not sure why I hadn't thought of it sooner. For the longest time I've always advocated to have a singleton utility class that holds onto configs like a base url. Then in your store, you can get the base url and prepend it. Something like this:

```
Ext.define('MyApp.Util', {
    singleton : true,

    config : {
        baseUrl : 'http://foo.com/'
    },

    constructor : function(config) {
        this.initConfig(config);
    }
});

Ext.define('MyApp.store.Foo', {
    extend : 'Ext.data.Store',
    alias  : 'store.myapp-foo',

    requires : [
        'MyApp.Util',
        'MyApp.model.Foo'
    ],

    model : 'MyApp.model.Foo',

    proxy : {
        type : 'ajax',
        url  : MyApp.Util.getBaseUrl() + 'foo.json'
    }
});
```

When the `MyApp.store.Foo` class is loaded by the browser, that `MyApp.Util.getBaseUrl()` is automatically evaluated and if you had `MyApp.Util` loaded prior to the store being loaded, it will concat the http://foo.com/ and `foo.json` and everything is good. The biggest drawback to this was that you had to require the `MyApp.Uti` class in both the `Ext.application` and the store's `requires` array. (side note, Sencha Cmd may be able to detect `MyApp.Util` but I've seen where this falls down so I'm usually more explicit with my `requires` array) Also, in your code you have to spread the `MyApp.Util.getBaseUrl()` calls around everywhere.

Instead, I had a thought to make this workflow much easier! What if you were able to define your store like this?

```
Ext.define('MyApp.store.Foo', {
    extend : 'Ext.data.Store',
    alias  : 'store.myapp-foo',

    requires : [
        'MyApp.model.Foo'
```

```
    ],

    model : 'MyApp.model.Foo',

    proxy : {
        type : 'ajax',
        url  : 'foo.json'
    }
});
```

No requiring `MyApp.Util`, no `MyApp.Util.getBaseUrl()` spread everywhere. Looks like I can solve both my caveats right? But how can I get the base url in there? Just by this code, there isn't anything that will prepend it. Where's the magic?

The magic is all in the `MyApp.Util` class with a small change. `Ext.data.Connection` (`Ext.Ajax` is an instance off) will fire a `beforerequest` event allowing you to cancel the request or you can also modify the request by changing the params but you can also change the url of the request. So why not have `MyApp.Util` automatically prepend the baseUrl for us? Like so:

```
Ext.define('MyApp.Util', {
      singleton : true,

      config : {
          baseUrl : 'http://foo.com/'
      },

      constructor : function(config) {
          this.initConfig(config);

          Ext.Ajax.on('beforerequest', this.onBeforeRequest, this);
      },

    onBeforeRequest : function(connection, options) {
          options.url = this.getBaseUrl() + options.url;
      }
  });
```

All I did was add the `beforerequest` event listener, and then changed the `options.url` and it will automatically prepend the base url. Here it is in action:

Notice this will of course work with `Ext.Ajax.request` calls, this is what `Ext.data.proxy.Ajax` does under the hood anyway.

There is always a better way to do things and always many different ways to do things. There are more performant ways to do this but I like this approach as it's very logical to me in how it works. I can come back to this code in 2 years and easily pick up what I was doing.

[source: Use of a base url in a utility class]

Delaying launch for stores to load

Saturday, November 30, 2019 8:25 PM

A common request I hear from many people is how to delay the application's `launch` method from being executed. First, let's investigate the problem. Say you have a normal `app/Application.js` that looks like:

```
Ext.define('Fiddle.Application', {
    extend : 'Ext.app.Application',

    name : 'Fiddle',

    stores : [
        'Store1',
        'Store2',
        'Store3'
    ],

    launch : function() {
        var store1 = Ext.getStore('Store1'),
            store2 = Ext.getStore('Store2'),
            store3 = Ext.getStore('Store3');

        console.log('store1 isLoaded', store1.isLoaded());
        console.log('store2 isLoaded', store2.isLoaded());
        console.log('store3 isLoaded', store3.isLoaded());

        Ext.Msg.alert('Fiddle', 'All stores are loaded!');
    }
});
```

In the `launch` method you have some code that depends on the stores being loaded. Ok, this example just has some `console.log`s so use your imagination some. I could add logic to the launch method and execute some other method on this class sure but there may be a more elegant way. If you were to run this right now, your `Ext.Msg.alert` would happen but all the `console.log`s would show `false` for the `isLoaded` method calls because the load hasn't happened yet. So what we need to do is add `load` listeners to the stores and only execute the `launch` method when all stores have loaded.

What I like to do is keep `app/Application.js` looking like it is and create an override on the `Ext.app.Application` class. What this override will do is change the `launch` method so that we can check if the stores have loaded. When they have all loaded, then execute the original `launch` method.

Let's take a look at a Sencha Fiddle as an example:

Looking at the different files, we have the 3 stores defined. `Store1` and `Store3` have `autoLoad` set to `true` but `Store2` does not so we only need to wait for `Store1` and `Store3` to load.

If you want, you can take a peak at `overrides/app/Application.js` to see how I take control of the `launch` method. Here's a rundown. The `launch` method gets transformed in the cleverly named `transformLaunch` method into a function that checks if all stores are loaded. The other method that is important is the `isStoreLoaded` method, this is important because this tells the override if the store has loaded in order to add the `load` listener and to tell if all stores have loaded. You may need to add additional logic dependent on your application's situation. If you need to modify that logic, I like to keep the override like it is and add that logic into my `app/Application.js` class so the logic is upfront in

the app.

[source: Delaying launch for stores to load]

Pluralize text

Saturday, November 30, 2019 8:25 PM

One of my gripes is when I see text that should be singular but shows plural when the value before it is one. It's 2015, surely we can detect when we should show singular there. Now admittedly, I've missed these opportunities but I do try to show the appropriate form of a word. So if you see `1 items` wrong! Should be `1 item` and `3 items`. I mean all that has to be done is a simple ternary expression:

```
value + ' item' + (value === 1 ? '' : 's'); //returns "1 item" or "3 items"
```

Done! So what does this have to do with Ext JS? Ext JS makes this a bit easier and has a function that you can call. That code there can be replaced with:

```
Ext.util.Format.plural(value, 'item'); //returns "1 item" or "3 items"
```

All the `plural` method does is adds the 's' onto the term you pass but also prepends the value you pass in. The `plural` method doesn't do too much magic in determining whether or not to add the 's' so for words that need to be changed more than just adding the 's' then you can pass the plural version as the third argument:

```
Ext.util.Format.plural(value, 'company', 'companies'); //returns "1 company"
or "3 companies"
```

Ok, so this isn't really getting me much except a friendly API. Where I really love using this is in an `XTemplate`. Code like this:

```
new Ext.XTemplate('{rows.length} item{[values.rows.length === 1 ? "" :
"s"]}');
```

can be replaced with this:

```
new Ext.XTemplate('{rows.length:plural("item")}');
```

Super clean! In a grouped grid, I use `plural` in the `groupHeaderTpl` and love how clean it is. Just for completeness, you can still pass the plural version too:

```
new Ext.XTemplate('{rows.length:plural("company", "companies")}');
```

[source: Pluralize text]

Format your numbers!

Saturday, November 30, 2019 8:25 PM

Yesterday, I posted about a gripe on not pluralizing text based on the value that text was associated to. Today, I visited a site and noticed some of their numbers were not formatted (some were). Once again, I was surprised how much this annoyed me...

> **147 out of 1000000 claimed**

You can see the '1000000' is not formatted, in fact I had to count a couple times to make sure I had enough zeros because my eyes were playing tricks on me when trying to count the number on the site. That's why you format the numbers! So your users don't have to struggle!

Once again, Ext JS to the rescue! `Ext.util.Format` has a `number` method that will take a value and a format string. The format string has quite a lot of options and luckily is pretty well documented. Let's look at a simple example that will format that text correctly (least for my locale):

As you can see, it outputs '1,000,000' and I can easily see what the actual value. Before formatting, it could have easily been 10,000,000 or 100,000 but now I know for sure it's 1,000,000. Let's look at usage in an `Ext.XTemplate`:

Ext JS making it simple! Let's look at doing this in a grid, there's three ways to do it. Here's a grid with three columns that's doing it the three different ways:

The first column has a renderer that uses `Ext.util.Format.number` directly, nothing really wrong with this but we can simplify this version by using `Ext.util.Format.numberRenderer` as in the second column. What the `numberRenderer` method does is basically returns a function and within this function uses `Ext.util.Format.number` just like in the first column. In fact, it's exactly the same thing, just a shorthand version. Those two are more for manual usage and backwards compatibility. The third column, for a grid, is the preferred way using `Ext.grid.column.Number` with a simple format config. Just like before, `Ext.grid.column.Number` just defines the `renderer` config for you and uses `Ext.util.Format.number` method just like in the first column.

When should you use one over the other. `Ext.grid.column.Number` was introduced in Ext JS 4.0 so if you are using Ext JS 3 or older, of course you cannot use it then, maybe you should update! :) The benefit of using `Ext.grid.column.Number` is it sets up the `updater` method on the column which means potential faster grid cell updates. I still use the other two when I need to use a different column but still need to format the number. For example, `Ext.tree.Panel` is actually just a grid but with a special column to show the expand and node icons. That means you have to use `Ext.tree.Column` and cannot use the `Ext.grid.column.Number` so you have to use one of the manual ways, like this:

Here, we have to use the `treecolumn` so we can expand/collapse the Items node so we have to define the renderer. Since some `text` values may not be a number and therefore do not want to format them, I use a quick ternary to check if it's a number then format it, otherwise return it.

[source: Format your numbers!]

Custom Two Way Binding!

Saturday, November 30, 2019 8:25 PM

When Ext JS 5 was released it brought with it data binding including two-way data binding. Although some argue that two way data binding is an anti-pattern, it does have it's uses. Many Ext JS components have two-way binding ready by default (for example, this Kitchen Sink example) but there may be cases where you want to setup a custom bindings. At a high level, what I want is when one component changes, another component changes in response automatically and if that second component changes, the first component also changes.

First Attempt

As a use case, I was recently asked how to collapse/expand a `Ext.form.FieldSet` using data binding. Without any custom things, it's possible like so:

In this fiddle, the`Ext.form.field.Checkbox` under the `FieldSet` will toggle whether the `FieldSet` is expanded or not. However, that's kind of weird because we are binding to `expanded` but that's not a config of `FieldSet`. The reason I used it is because binding requires a setter function so in this case the `FieldSet` needs (and does albeit being private) have a `setExpanded` method in order for binding to work. `FieldSet` does have a `collapsed` config but binding needs a setter and there is no `setCollapsed` method. We could create a simple override to add one:

That looks better! However, if we manually collapse/expand the `FieldSet`, the `Checkbox` will not reflect this change so it's only one-way currently. This is where we need to do some custom stuff in order to have two-way binding. There are two ways to have two-way binding and which to use depends on if the config you want to change is within the config object (so setter and getter methods are automatically created for you).

Manual Two-Way Binding

To get the new value up to the `Ext.app.ViewModel` on the form, the `publishState` method must be executed passing in the name and value. The `FieldSet` will fire `collapse` and `expand` events when it has been collapsed/expanded so we can add listeners to it in order to execute that `publishState` method:

In the `initComponent` method, I add listeners to the `collapse` and `expand` events which will execute the `publishCollapsed` method. If rendered, this will execute `publishState` meaning two-way binding is setup.

Automatic Two-Way Binding

Like I said before, the way to know if you need to hookup the `publishState` method execution or if Ext JS can handle it automatically for you is whether the config you are binding to is within the config object. The `collapsed` config is not part of the config object but what if we had a custom config, how can we get that custom config to be two-way bindable? This is where `twoWayBindable` comes in handy to hook things up for us:

When `setFoo` is executed, we check to see if the `FieldSet` is already collapsed/expanded and then execute the `setExpanded` if not. We had to add the `setFoo` call to `setExpanded` also to hook into when the `FieldSet` is manually collapsed/expanded by the user. That has nothing to do with the binding tho, the bit to pay attention to is the config object and the `twoWayBindable` property. The `twoWayBindable` property is an array of configs and it will then hook into the `updater` for that config to then automatically execute that `publishState` method. So the binding is automatic, the other code is to handle collapsing/expanding when the `foo` config changes.

Summary

Data binding can be a bit mysterious but having some form of understanding the sequence of things helps go a long way. Knowing that `publishState` and having an associated setter method are important pieces of the binding puzzle. For more, check out the guide that goes over binding.

[source: Custom Two Way Binding!]