« ExtJS 4.2 Walkthrough — Post-Upgrade Bug ExtJS 4.2 Diversion: Custom Panel UIs »

ExtJS 4.2 Walkthrough — Part 4: Steppin' in Some CRUD

We've finally arrived! After some setup, a bit of layout work, the creation of our first controller, a full version upgrade and even a bug fix, we're ready to get into the heart of our application…specifically, interacting with data from our server. In order to do this, we'll need to explore the concepts of **Models**, **Stores** and **Proxies**, as well as creating a new controller and even a fancy data grid with inline editing capability.

We're in for a long haul in this installment, but it's worth it. When we come out the other side, we'll not only have real functionality working in our app, but we'll also have covered some very important concepts that will let us ramp up our development of the rest of the application. Ready? Let's do it.

> ***NOTE:*** *The code for this session can be found on* ***GitHub****.*

Data Models

As a developer, you've no doubt utilized data models on a regular basis. They are handy things, for not only do they let you describe objects via their properties and relationships, but they also semantically describe your application. In ExtJS, the **Ext.data.Model** fills this role, providing a really nice way to manage objects which can be grouped together in caches (a Store) and, more interestingly or even bound to model-aware components like the **Ext.grid.Panel**

Especially in the domain of managing data coming from the server, the **Model** is a crucial means of translating the raw stream of content returned from the server into an object that your ExtJS app can leverage to do some pretty awesome stuff.

Simple Example

So let's start with a very simple example. In our application, we're going to have several tables in our database which are "option" or "lookup" tables. Consider, for example, the **Color** lookup table which stores options for car colors:

```
------------------------------
| ColorID     | INT           |
| LongName    | VARCHAR(255)  |
| ShortName   | VARCHAR(45)   |
| CreatedDate | TIMESTAMP     |
| Active      | BIT           |
------------------------------
```

To model this data in our app, we simply need to define an **Ext.data.Model** that will translate the data we send from the server into an object that ExtJS can use. Here's an example of our potential model:

```
/**
 * Model representing a Color object
 */
Ext.define('CarTracker.model.option.Color', {
```

```
    extend: 'Ext.data.Model',
    idProperty: 'ColorID',
    fields: [
        // id field
        {
            name: 'ColorID',
            type: 'int',
            useNull : true
        },
        // simple values
        {
            name: 'LongName',
            type: 'string'
        },
        {
            name: 'ShortName',
            type: 'string'
        },
        {
            name: 'CreatedDate',
            type: 'date',
            persist: false
        },
        {
            name: 'Active',
            type: 'boolean',
            defaultValue: true
        }
    ]
});
```

A few things.

First, our data model extends **Ext.data.Model**. By doing this, we gain access to a TON of really helpful methods and properties that will help us manage our model instances. Be sure to ***memorize*** the docs on this one.

Next, we have a **fields[]** config wherein we define

- The properties, or "fields" that we want to manage
- The data type of the property (string, boolean, date, int, float, auto)
- Whether or not the property is persistent (if false, it won't be sent back to the server, ever)
- Default values (if any)

By default, field names map 1-to-1 with the keys of the data structure retrieved from the server. So, in order to populate this model with data from the server, we would expect to see the following in the response:

```
// full response
...
{ ColorID: 1, LongName: 'Metallic Blue', ShortName: 'Blue', CreatedDate:
'2013-05-27...', Active: 1 }
```

(We'll get around to actually populating a data model later).

Extensible Models

While we could plunk this **Model** in our app without any issues, we'll take a bit of a longer view from the start. Namely, if we look at this data model, we might see some potential for abstraction:

- **CreatedDate** and **Active** are going to be used on *EVERY TABLE* in our database.

- **LongName** and **ShortName** are going to be used on *every OPTION table* in our database

Since specifying these properties on every single data model in our application would be redundant, we'll expand on our policy of abstraction, and create some base models that we can reuse as we expand our app.

First, in the model folder, let's create a new file named **Base.js:**

```
/**
 * Base {@link Ext.data.Model} from which all other models will extend
 */
Ext.define('CarTracker.model.Base', {
   extend: 'Ext.data.Model',
   fields: [
       // non-relational properties
       {
           name: 'CreatedDate',
           type: 'date',
           persist: false
       },
       {
           name: 'Active',
           type: 'boolean',
           defaultValue: true
       }
   ]
});
```

Pretty simple. Just a base, abstract model that specifies the common properties that every one of our models will inherit.

Next, since all of our option tables share common properties, let's make another, higher-level abstract model that all of our option models can use. In the model folder, first create a new folder called **option**. Next, create a new file named **Base.js**:

```
/**
 * Base {@link CarTracker.model.Base} from which all other "Option"
models will extend
 */
Ext.define('CarTracker.model.option.Base', {
   extend: 'CarTracker.model.Base',
   fields: [
       // non-relational properties
       {
           name: 'LongName',
           type: 'string'
       },
       {
           name: 'ShortName',
           type: 'string'
       }
   ]
});
```

This has the same principle as our first base model, but instead of extending **Ext.data.Model**, this more specific abstract model extends our other abstract model, **CarTracker.model.Base**.

Excellent. Now that we have these in place, let's finish by creating our **Color** model. In the **model/option** folder, create a new file named **Color.js**:

```
/**
```

```
 * Model representing a Color object
 */
Ext.define('CarTracker.model.option.Color', {
   extend: 'CarTracker.model.option.Base',
   idProperty: 'ColorID',
   fields: [
       // id field
       {
           name: 'ColorID',
           type: 'int',
           useNull : true
       }
   ]
});
```

Very straightforward. Now that we've abstracted, we have a new **Color** model that, while still retaining precisely the properties as were defined in the original example, is better constructed. Not only will this let us write less repetitive code when we create other option models, but it will also save us a lot of headaches if we need to add another shared property…instead of updating each and every file, we can just update the one base model. And finally, if we ever need to make one of the options divergent from the others, that's still possible since we've subclassed each option model. Beautiful.

Testing Our Model

Let's take our new model for a test drive. To do this, open your browser, load this application, and go into Web Developer or Firebug. In the console, enter the following:

```
color = Ext.create( 'CarTracker.model.option.Color', {
    LongName: 'Silvery Red',
    ShortName: 'Red',
    Active: false }
)
```

After running the command, you should see something like the following:

```
▼constructor {raw: Object, modified: Object, data: Object, hasListeners: HasListeners,
 ▼data: Object
    Active: false
    ColorID: null
    CreatedDate: null
    LongName: "Silvery Red"
    ShortName: "Red"
```

As you can see, we have created a new instance of our **CarTracker.model.option.Color** model, and pre-populated it with a bit of data. Now that our raw data (the object we passed to the creation of the model instance) has been converted into a legit model, we can call any of the built-in methods that **Ext.data.Model** affords.

Moving On

As great as creating our first model has been, it doesn't do us a lot of good yet. After all, we're not really interested in creating model instances like this: we want them to be populated with data coming from the server. While we could certainly configure our model to do just that, a more common way of managing models is via a store. So let's turn our attention there now.

Our First Store

At its most basic level, a store is simply a cache of model instances within your app. But it's so much more! Not only is it one of the most common gateways for interacting with the server within ExtJS apps, but it is also

a powerful object that can be wired to a large number of data-aware components. For example, let's say we have a data grid with a column for each property in our model. If we take a store which has been configured with the particular model and wire it up to the grid, the grid will automatically render a row for each model in our store, matching the properties in the store's model to the columns in the grid. What's more, if the data in our store changes, these will be automatically communicated to the grid, and the displayed data will update to represent the change. This is game changing, since you can really turn your attention to more interesting aspects of your application and just let this awesomeness do its thing.

Enough talk; let's make one.

As we saw when setting up our application, **Sencha Cmd** automatically created a **store** folder in our application's app folder. This is where we will create all of our stores. As with our views and models, we'll want to be smart about packaging our stores so that our file organization doesn't suck.

Since we're creating a store for our **Color** model, let's go ahead and create a new folder called **option** in the **store** folder, and then create a new file called **Colors.js**:

> *NOTE: While not required by any means, a common convention for naming stores is to use the plural form of the model name. So if our model is Store.js, the store would be Stores.js. The idea is that since a model represents a single instance, and the store represents a collection of model instances, the plural fits.*

```
/**
 * Store for managing car colors
 */
Ext.define('CarTracker.store.option.Colors', {
    extend: 'Ext.data.Store',
    alias: 'store.option.color',
    requires: [
        'CarTracker.model.option.Color'
    ],
    storeId: 'Colors',
    model: 'CarTracker.model.option.Color'
    remoteFilter: true,
    remoteSort: true,
    proxy: {
        type: 'rest',
        url: '/api/option/colors'
        reader: {
                type: 'json',
                root: 'data',
                totalProperty: 'count'
        }
    }
})
```

Let's take this line by line:

- **extend**: We're extending **Ext.data.Store**…no surprise here 😀
- **alias**: As with components, we can alias our stores. Instead of "widget", however, we use "store" as the prefix
- **requires**: Here, we're requiring the model that we want to use for this store in order to make sure it's loaded before our store tries to use it
- **storeId**: Not required, but a nice way to uniquely identify the store. Plus, if we specify a storeId, this store will automatically be registered with **Ext.StoreManager**, making it easier to reference later

- **model**: The full class path to the model we'd like to use with the store (notice the correspondence with requires[]….)
- **remoteFilter**: If true, we'll do all our filtering via SQL/or other server-side mechanism…a must if our result set is "paged"
- **remoteSort**: If true, we'll do all our sorting via SQL/or other server-side mechanism…a must if our result set is "paged"
- **proxy**: The proxy our store will use for communicating with the server…more below

Getting Data to Our Store

Before we can test our Store, we need to make sure that our application server is prepared for the request. There are two main things to consider.

Request URL

In the proxy config, you can see that we've specified both a type of "rest" and a url of "/api/option/colors". The URL can be whatever you want it to be, but the type of proxy is important. With the "rest" proxy (Ext.data.proxy.Rest), ExtJS will dynamically build the request URL based on the type of request being made. So assuming that our url is configured as above, the normal CRUD urls that ExtJS creates will be:

- **read**: /api/option/colors.json (GET)
- **create**: /api/option/colors.json (POST)
- **update**: /api/option/colors/{id}.json (PUT)
- **delete**: /api/option/colors/{id}.json (DELETE)

As you can see, with the REST-style proxy, the same URL is being used for each request. The only difference is the HTTP verb sent, as well as the ID property of the model instance in the case of updates and deletes.

With this is in mind, we'll need to make sure that our server will accept these types of requests and that our application server (ColdFusion, PHP, .NET, whatever) is configured to properly route the requests to the same URL based on the HTTP verb and existence of the ID value.

Handling the Response

Assuming that our server has properly handled the request, we also need to ensure that the data is returned to our **Store** in a way that it can handle. While there are a number of formats that the Ext.data.reader.Reader can accomodate, the most used is the Ext.data.reader.Json. For our purposes, the configuration of this reader revolves around two configurations: **root** and **totalProperty**. The **root** config instructs the **Reader** where it should begin looking for data within the JSON-formatted response. In our example, we've specified that it should look in a key called "data". Next, we specify that the **totalProperty** for our result set can be found at the "count" key. This will let our store know the total number of records that exist in our data, which is especially helpful when using techniques for paging (e.g., the total data set size is known, but only a partial set is returned at a time).

Putting this all together, our response from the server should look something like:

```
{
    "count":10,
    "data":[
        {"LongName":"Antique Gold
Metallic","ColorID":1,"ShortName":"Gold","Active":true},
        {"LongName":"Aqua Pearl
Metallic","ColorID":2,"ShortName":"Aqua","Active":true},
        {"LongName":"Arctic
White","ColorID":3,"ShortName":"White","Active":true},
        {"LongName":"Black
Metallic","ColorID":4,"ShortName":"Black","Active":true},
        ...
```

```
    ]
}
```

When the response is returned, the **Reader** will evaluate the response and convert it into an array of model instances within our **Store**.

Cool? So let's try it.

The Test

Back in the browser, load up our app and go to the Console in Web Developer or Firebug. Run the following:

```
colorStore = Ext.create('CarTracker.store.option.Colors')

colorStore.load()
```

First, we create a new instance of our **Store**. Next, we call the **load()** method which will kick off a remote request to the server (assuming everything is configured correctly). Once the response is received, run the following:

```
colorStore.getCount()
```

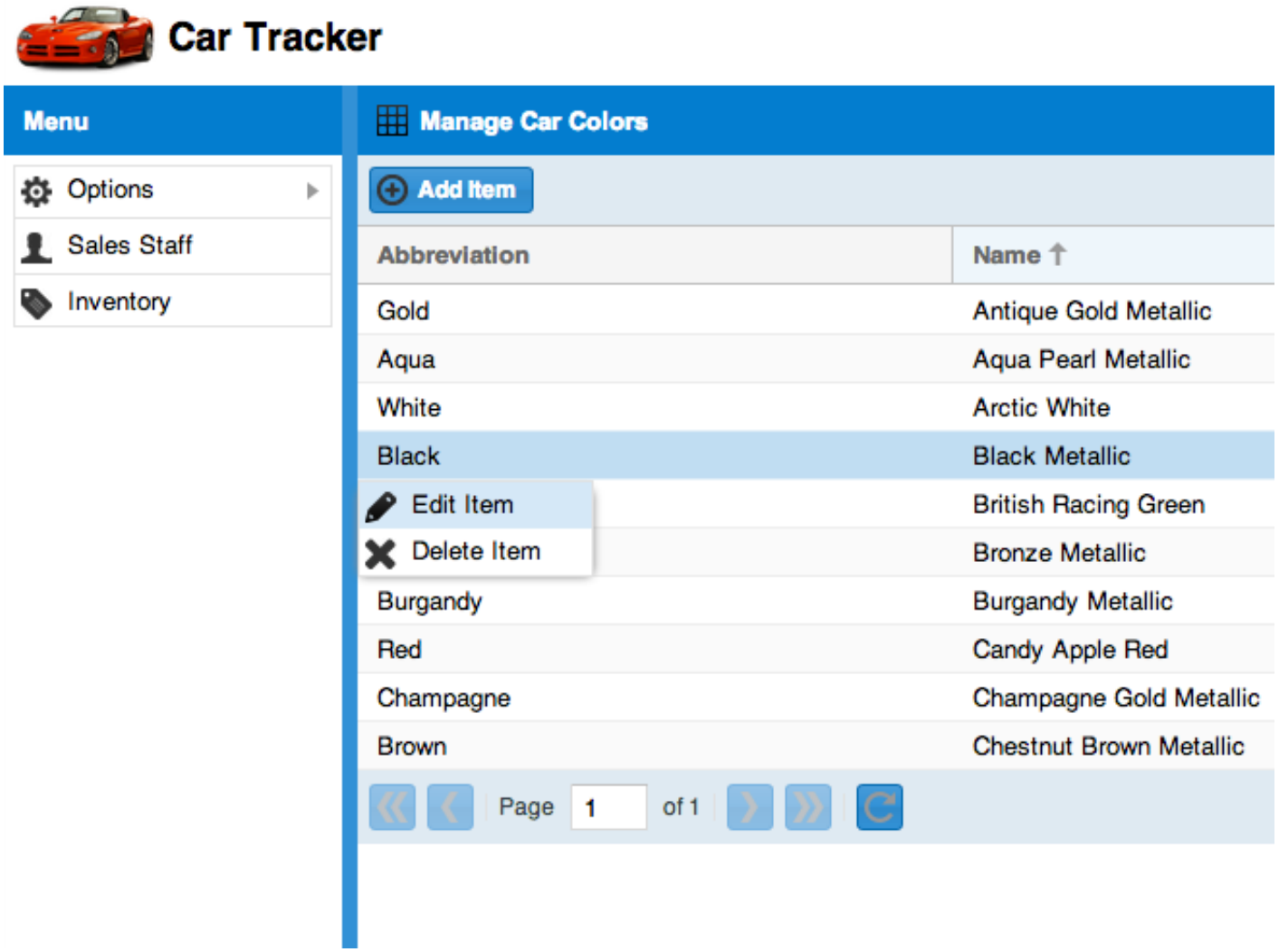
Assuming that you had data to return, this should print the total number of model instances in the store.

We now have a store that can manage our model instances, and this store is successfully communicating with the server. Awesome.

Of course, there is some refactoring and abstracting that we'll want to do. But since we've come so far, let's reward ourselves by pressing on and attaching our store to something that we can actually use.

The Option Grid

Now that we have our first **Model** and **Store**, we can start using them within our application to make some cool stuff. Our first objective is to make a management console for our **Color** options. While there are a lot of ways we can approach this, a pretty shiny one is to create an editable grid that will not only allow us to display all of the data, but will also let us create, edit and delete records…all in the same view. When we're finished, it should look kind of like this:



A Fancy Editable Grid

Let's create the grid. In view, create a folder named option, and then a new file called List.js:

```
Ext.define('CarTracker.view.option.List', {
    extend: 'Ext.grid.Panel',
    alias: 'widget.option.list',
    requires: [
        'Ext.grid.plugin.RowEditing',
    'Ext.toolbar.Paging'
    ],
    initComponent: function() {
        var me = this;
        Ext.applyIf(me,{
            selType: 'rowmodel',
            plugins: [
                {
                    ptype: 'rowediting',
                    clicksToEdit: 2
                }
            ],
            columns: {
                defaults: {},
                items: [
                    {
                        text: 'Abbreviation',
                        dataIndex: 'ShortName',
                        editor: {
                            xtype: 'textfield'
                        },
```

```
                        flex: .2
                    },
                    {
                        text: 'Name',
                        dataIndex: 'LongName',
                        editor: {
                            xtype: 'textfield',
                            allowBlank: false
                        },
                        flex: .5
                    }
                ]
            },
            dockedItems: [
                {
                    xtype: 'toolbar',
                    dock: 'top',
                    ui: 'footer',
                    items: [
                        {
                            xtype: 'button',
                            itemId: 'add',
                            iconCls: 'icon_add',
                            text: 'Add Item'
                        }
                    ]
                },
                {
                    xtype: 'pagingtoolbar',
                    ui: 'footer',
                    defaultButtonUI: 'default',
                    dock: 'bottom',
                    displayInfo: true,
                    store: me.getStore()
                }
            ]
        });
        me.callParent( arguments );
    }
});
```

By now, we've created a number of components, so this should look familiar. I'll only point out a few items of interest:

- **requires[]:** This grid will utilize a special plugin (**Ext.grid.plugin.RowEditing**) and a special toolbar (**Ext.toolbar.Paging**). As with our other components, we want to make sure that these classes are loaded before we use them
- **plugins[]:** We use the **plugins[]** config to define the plugins to be used by our component. In this example, our grid is using the **Ext.grid.plugin.RowEditing** plugin, which can be initialized via its ptype "rowediting"
- **dockedItems[]:** This allows us to configure items that will be "docked" to our component, which essentially binds them in place to their configured position (top, left, right, bottom). In this example, we have a top toolbar with some buttons, and on the bottom we have a toolbar to manage paging of our recordset.

Displaying the Grid

Now that are grid is defined, we need a way to display it. In our **view/layout/Menu.js** file, let's change the "Options" button to have a sub-menu. This will let us put all of our "option" links in one place and not take up space in our navigation menu.

```
{
    text: 'Options',
    itemId: 'options',
    iconCls: 'icon_gear',
    menu: [
        {
            text: 'Car Colors',
            itemId: 'option/color',
            iconCls: 'icon_color'
        }
    ]
}
```

Now let's open the **App** controller and wire up the **Color** option to our history/navigation management:

```
dispatch: function( token ) {
    ...
    switch( token ) {
        ...
        case 'option/color':
            config = {
                xtype: 'option.list',
                title: 'Manage Car Colors',
                iconCls: 'icon_color',
                store: Ext.create( 'CarTracker.store.option.Colors', {
         pageSize: 30
       })
            };
            break;
        ...
    }
    ..
}
```

A few things to notice:

- We specified an **xtype** of "option.list". If you remember, we gave our grid the same xtype. Therefore, when this component is rendered, a new instance of our grid will be created and placed into the center region of our app.
- We configured this instance of our grid to use a store via **Ext.create(…).** One benefit of this approach is that it allows us to override configuration of the store for the particular instance. In this case, we are setting a pageSize of 30.

A New Controller

As we discussed earlier, it's ideal for us to create controllers for each main section of our application. While we could create a separate **Colors** controller, we've already discussed that our **Color** model really only represents a specific implementation of a more generic **Option** model. Since all options will have the same data structure, it also makes sense that we manage them in the same way as well. Therefore, instead of creating a specific **Colors** controller, we'll settle for a more general **Options** controller.

> ***NOTE:*** *The beauty of ExtJS MVC-style is that we can always subclass Colors if we want/need to. That is, if the needs of the Color model diverge from what Option can reasonably accomodate, we can always create a controller that extends the Options controller, or create a new one altogether.*

```
/**
 * Generic controller for managing simple options
 */
Ext.define('CarTracker.controller.Options', {
    extend: 'CarTracker.controller.Base',
    stores: [
        'option.Colors'
    ],
    views: [
        'option.List'
    ],
    refs: [
        {
            ref: 'OptionList',
            selector: '[xtype=option.list]'
        }
    ],
    init: function() {
        this.listen({
            controller: {},
            component: {
                'grid[xtype=option.list]': {
                        edit: this.save,
                        canceledit: this.cancel,
                        beforerender: this.loadRecords,
                        itemcontextmenu: this.showContextMenu
                },
                'grid[xtype=option.list] button#add': {
                        click: this.add
                },
                'grid[xtype=option.list] gridview': {
                        itemadd: this.edit
                }
            },
            global: {},
            store: {},
            proxy: {}
        });
    },
    /**
     * Displays context menu
     * @param {Ext.view.View} view
     * @param {Ext.data.Model} record
     * @param {HTMLElement} item
     * @param {Number} index
     * @param {Ext.EventObject} e
     * @param {Object} eOpts
     */
    showContextMenu: function( view, record, item, index, e, eOpts ) {
        var me = this;
        // stop event so browser's normal right-click action doesn't
continue
        e.stopEvent();
        // if a menu doesn't already exist, create one
        if( !item.contextMenu ) {
                // add menu
                item.contextMenu = new Ext.menu.Menu({
                        items: [
                                {
                                        text: 'Edit Item',
                                        iconCls: 'icon_edit',
                                        handler: function( item, e ) {
                                                var grid =
me.getOptionList(),
                                                        plugin =
```

```
grid.editingPlugin;
                                                    // start row edit
                                                    plugin.startEdit( record,
0 );
                                            }
                                        },
                    {
                        text: 'Delete Item',
                        iconCls: 'icon_delete',
                        handler: function( item, e ) {
                            me.remove( record );
                        }
                    }
                        ]
                })
        }
        // show menu relative to item which was right-clicked
        item.contextMenu.showBy( item );
    },
    /**
     * Loads the grid's store
     * @param {Ext.grid.Panel}
     * @param {Object}
     */
    loadRecords: function( grid, eOpts ) {
        var me = this,
                store = grid.getStore();
        // clear any fliters that have been applied
        store.clearFilter( true );
        // load the store
        store.load();
    },
    /**
     * Cancels the edit of a record
     * @param {Ext.grid.plugin.Editing} editor
     * @param {Object} context
     * @param {Object} eOpts
     * @param {}
     * @param {}
     * @param {}
     */
    cancel: function( editor, context, eOpts ) {
        // if the record is a phantom, remove from store and grid
        if( context.record.phantom ) {
                context.store.remove( context.record );
        }
    },
    /**
     * Begins edit of selected record
     * @param {Ext.data.Model[]} records
     * @param {Number} index
     * @param {Object} node
     * @param {Object} eOpts
     */
    edit: function( records, index, node, eOpts ) {
        var me = this,
                grid = me.getOptionList(),
                plugin = grid.editingPlugin;
        // start edit of row
        plugin.startEdit( records[ 0 ], 0 );
    },
    /**
     * Creates a new record and prepares it for editing
     * @param {Ext.button.Button} button
     * @param {Ext.EventObject} e
```

```
     * @param {Object} eOpts
     */
    add: function( button, e, eOpts ) {
        var me = this,
                grid = me.getOptionList(),
                plugin = grid.editingPlugin,
                store = grid.getStore();
        // if we're already editing, don't allow new record insert
        if( plugin.editing ) {
                // show error message
                Ext.Msg.alert( 'Attention', 'Please finish editing before
inserting a new record' );
                return false;
        }
        store.insert( 0, {} );
    },
    /**
     * Displays context menu
     * @param {Ext.grid.plugin.Editing} editor
     * @param {Object} context
     * @param {Object} eOpts
     */
    save: function( editor, context, eOpts ) {
        var me = this,
                store = context.record.store;
        // save
        store.save();
    },
    /**
     * Displays context menu
     * @param {Ext.data.Model[]} record
     */
    remove: function( record ) {
        var me = this,
                store = record.store;
        // show confirmation before continuing
        Ext.Msg.confirm( 'Attention', 'Are you sure you want to delete
this item? This action cannot be undone.', function( buttonId, text, opt
) {
                if( buttonId=='yes' ) {
                        store.remove( record );
                        store.sync({
                                /**
                                 * On failure, add record back to store
at correct index
                                 * @param {Ext.data.Model[]} records
                                 * @param {Ext.data.Operation} operation
                                 */
                                failure: function( records, operation ) {
                                        store.rejectChanges();
                                }
                        })
                }
        })
    }
});
```

Since there's so much going on here, I'm not going to go into a huge amount of detail. If you follow the logic, however, we've implemented logic for viewing the records (loadRecords), creating new data (add), editing data (save) and deleting data (remove). We've also let our controller know about the Colors store (stores[] config), our grid (views[] config), and have set up a number of listeners within init() to handle the various events that will occur while managing the data for this particular store and grid.

Wiring Up Our Controller

The very last thing we need to do is to let our app know about our new controller. To do this, let's open **app.js** and add our controller to the **controllers[]** config:

```
Ext.application({
    name: 'CarTracker',
    ...
    controllers: [
        'App',
        'Options'
    ],
    ...
    }
});
```

That's all there is to it. If everything is implemented correctly, we should be able to reload our app in the browser and see the awesome new functionality. Hooray!

Some Refactoring

There is no rest for the wicked! Our new functionality looks awesome, but we know there's some refactoring we should do. Let's get to it.

An Abstract Proxy

First, let's refactor our proxy. If you remember above, we more or less hard-coded our proxy configuration onto our **Colors** store. While this obviously works, it's definitely not great since we'd have to duplicate that proxy config over and over for each and every store. What would be better would be to create an abstract proxy that we can simply use anywhere it's needed. To do this, let's start by creating a folder named **proxy** at the root of our app. In this **proxy** folder, let's also create a file called **Rest.js**:

```
/**
 * Abstract REST proxy
 */
Ext.define('CarTracker.proxy.Rest', {
    extend: 'Ext.data.proxy.Rest',
    alias: 'proxy.baserest',
    format: 'json',
    limitParam: 'max',
    startParam: 'offset',
    sortParam: 'sortorder',
    writer: {
        type: 'json',
        writeAllFields: true
    },
    reader: {
        type: 'json',
        root: 'data',
        totalProperty: 'count'
    }
});
```

Very straightforward. We've more or less moved the **proxy** config off of the **Colors** store, and wrapped it into a new class. Notice that we also gave it an alias of "**baserest**"–this is what we'll use when applying it to our stores.

A Base Store

You knew it was coming. Since we don't want repeat ourselves over and over when defining stores, let's create an abstract **Store** in the store folder called **Base.js**:

```
/**
 * Base {@link Ext.data.Store} from which all other application stores
will extend
 */
Ext.define('CarTracker.store.Base', {
    extend: 'Ext.data.Store',
    requires: [
        'CarTracker.proxy.Rest'
    ],
    /**
     * @cfg {String} restPath End point for store requests
     */
    restPath: null,
    constructor: function( cfg ){
        var me = this;
        cfg = cfg || {};
        me.callParent([Ext.apply({
            storeId: 'Base',
            remoteSort: true,
            remoteFilter: true,
            remoteGroup: true,
            proxy: {
                type: 'baserest',
                url: me.restPath
            }
        }, cfg)]);
    }
})
```

Again, we have more or less applied much of our **Colors** store config and created a base class out of it. Notice, though, that we are requiring the custom **Proxy** that we created, and then using it within the **proxy** config of our store.

Another thing to notice is the custom **restPath** config. This will allow us to specify a custom URL on our individual store instances, which will be then be applied to our proxy's url config. This is especially useful if your ExtJS application is not at the webroot, and needs to specify some kind of root URL to which the **restPath** is appended.

A(nother) Base Store

This one's definitely not necessary, but since our **Colors** store is really an implementation of an idea of an "option", let's go ahead an make an abstract store within the **option** package called **Base.js**. At the moment, it won't have anything specific in it besides a custom sorter, but may prove even more useful later on as the specificity and requirements of our applications grows:

```
/**
 * Store from which all other option stores will extend
 */
Ext.define('CarTracker.store.option.Base', {
    extend: 'CarTracker.store.Base',
    constructor: function( cfg ){
        var me = this;
        cfg = cfg || {};
        me.callParent([Ext.apply({
            storeId: 'option_Base'
        }, cfg)]);
    },
```

```
    sorters: [
    {
      property: 'LongName',
      direction: 'ASC'
    }
  ]
})
```

Colors Store

Now that we have our abstract stores completed, we can finally refactor our **Colors** store to be much more compact:

```
/**
 * Store for managing car colors
 */
Ext.define('CarTracker.store.option.Colors', {
    extend: 'CarTracker.store.option.Base',
    alias: 'store.option.color',
    requires: [
        'CarTracker.model.option.Color'
    ],
    restPath: '/api/option/colors',
    storeId: 'Colors',
    model: 'CarTracker.model.option.Color'
});
```

Beautiful. Now our **Colors** store is only configuring options that are specific to itself, and allowing the inheritance of its parent classes to take care of the rest.

Wrapping Up…and Some Homework

Phew! That was epic. In that whirlwind of development, we made data models, implemented stores and proxies, and even created a super-fancy editable grid. But best of all, in the process of working through this, we've covered a lot of topics that will be incredibly useful to us as we move forward and start developing more functionality.

But before we continue, you have some homework

Before the next installment, go ahead and finish out the rest of the options. Once finished, you should have the following options available:

- Categories
- Colors
- Features
- Positions
- Statuses

Each of these should appear in the "Options" menu on the interface, and selecting any of the options should load the Option Management grid with the appropriate data from the server (you'll have to make that). Have fun!

COMMENTS (13) **TRACKBACKS (1)**

#1 written by Jardalu
about 6 months ago

Awesome ! I think we will be revisiting this again and again to get inspired. Pretty impressive, especially with the UI.

#2 written by budhines
about 3 months ago

Thank you for providing this great tutorial.

I have been able to follow along and use your examples until I hit the ColdBox code. You have now raised my interest in ColdBox.

I loaded the latest release of ColdBox, and was was able to get the super simple template page running.

I modified car tracker application cfc to point to ColdBox (as defined on my local environment) instead of ColdBox_3.5.3 but I keep getting 404 failures on my api calls.

I usually go into Firebug and look at the xhr request but these calls through ColdBox just show failures on my api calls.

I am searching through the ColdBox documentation to determine how to debug but I would I appreciate it if you could pass along any hints on how to interrogate api failures.

thanks,
Bud

I defined my database, my table names, my table columns according to the specs.

#3 written by existdissolve
about 3 months ago

Hi Bud–

If you're getting 404's, I'm guessing it's probably something to do with the URL rewriting. Are you on IIS or Apache?

#4 written by budhines
about 3 months ago

Hi existdissolve,

Thank you for responding so quickly.

I am using Apache. I have IIS turned off.

My call to the api looks like this:
http://127.0.0.1:8500/api/option/colors.json?_dc=1375539291061&page=1&offset=0&max=30&sortorder=%5B%7B%22property%22%3A%22L

I find it interesting that the api is showing at my root. Is this what ColdBox does?

I know your focus is presenting The Sencha ExtJS MVC framework but I would appreciate any insight you could provide.

thanks,
Bud

#5 written by existdissolve
about 3 months ago

Hi Bud-

I just realized that I failed to upload the .htaccess file that my site is using for URL rewriting. I just pushed it to the repo if you want to see if that helps: https://github.com/existdissolve/CarTracker/blob/master/cf/.htaccess

RE: the root issue, that's a combination of how your site is setup and how the URLs in Ext JS are defined. In the CarTracker source, all store URLs (or restPath) are relative to the site root (e.g., '/api/cars'). This setup will always produce urls that are "rooted" to the base of the site. To use this, I setup a virtual host for my site called "cartracker.dev". This lets me access the site like so: http://cartracker.dev, and produces API urls like http://cartracker.dev/api/…

If you need to access your site as a subdirectory of the site root, you can specify a global variable in your index.html file and set the full proxy root there. For example, before the inclusion of app.js, add something like so:

```
var APPLICATION_PROXY_ROOT = "http://127.0.0.1:8500/cartracker" // or whatever the path you're using looks like
```

And then, in your Base store, set the proxy URL like so:

```
proxy: {
type: 'baserest',
url: APPLICATION_PROXY_ROOT + me.restPath
}
```

I would definitely suggest putting this in the index.html file, since you don't want the path to be in code that will ultimately get compiled.

Hope this helps!

#6 written by budhines
about 3 months ago

Hi existdisolve,

I apologize for the delay in providing feedback but had some issues arise with our production systems this weekend.

Thank you for your guidance. I am now able to redirect my api's appropriately but I am still getting 404s.

Without the redirects, where should I be seeing the api calls – from ColdBox at the root of my application?

I am still running into 404s with my api calls. Json is being appended onto the call which is what I believe ColdBox expects.

I think I this will be a long learning process for me as I am just learning coldbox, orm and restful services. I do not want to burden you with with a series of questions.

I am going to go back to PartThree and add on the option features without rest and coldbox and try to go through the tutorial that way. I am sure it will not be as efficient as this code.

Thank you for your help,
Bud

#7 written by existdissolve
about 3 months ago

No worries at all. I'm quite involved in the ColdBox community, so more than happy to help out on that side of things as much as I can.

Without the rewrite rules, your path would be something like this: http://127.0.0.1:8500/cartracker/car/list (where "car" is the handler name and "list" is the method)

(Assuming your site is in a folder called "cartracker")

Re: the 404's, can you try adding "index.cfm" into your URL, like: http://127.0.0.1:8500/cartracker/index.cfm/api/option/colors.json. If this works, that would suggest that the rewriting isn't quite working as expected.

#8 written by Bud Hines
about 4 weeks ago

HI existdisolve,

I am the person that you are discussing the ajax calls with on stack overflow. I appreciate your help. I am looking how to get the data passed as request.jsonData passed as request.params.

I tried to work with the json data inside my cfc by dumping the contents of the cgi variables:

But the data is still showing as empty when I dump it.

So I really have two questions, how to pass the jsonData as parameters in the post and how do you read the jsonData passed to a cfc?

thanks,
Bud

#9 written by existdissolve
about 4 weeks ago

Hi Bud–

What do you get when you dump the form scope?

#10 written by Bud Hines
about 4 weeks ago

HI existdissolve,

The form shows as empty (empty struct) and my url shows url – struct
_dc 1382872989469
method addcolors.

thanks,
Bud

#11 written by Bud Hines
about 4 weeks ago

Hi Existdissolve,
In my cfc I was checking to see if the cgi.content_type EQ "application/json;

It did not because it actually contained "application/json; charset=UTF-8 "

So my if statement did not work. I was able to use deserializeJSON(ToString(getHTTPRequestData().content and get the data that was passed! Yeah!!!

Thank you for your patience and all your help,
Bud

#12 written by Christian Segota
about 1 week ago

Hi Existdissolve,

I was working through this walk through at a great pace until I hit this section. I work on a dev team and primarily do front end work web work. In fact I have only been doing professional web development for less than a year. I'm kind of at a loss on how to set up the server side to this project. It seems to me that it might not be an extremely simple task? I'm really not even sure where to start.

Did I miss a section in this project that had to do with data/server set up?

Should I just be able to assume the set up of the server/tables/database based on what I've done in the front end?

Obviously there is a lot of code that I can see on GitHub that isn't included in this part of the walk through, should that be my main point of reference to finish this section?

Please don't take this next question the wrong way but: Isn't the "data" portion of this walk through a reasonably extensive and important aspect of this project as a whole? Is this(server/data) something that I (as a Jr web developer) should be able to just throw together in an afternoon while doing this walk through? Is this not like a whole project within a project?

I'm just trying to put a scope on this thing. For example if you respond and say its not too extensive of a set up, I'll know it could take me any where from maybe a day to a week to figure out. But if you say its reasonably extensive, given my almost complete lack of prior server side development, then I'll know it may take me much more than a week to work out the kinks.

I would greatly appreciate any advise on where to start looking(guides?) and perhaps a list of requirements that this portion of the project concerns.

Maybe I'm making a mountain out of a mole hill?

Thanks again Existdissolve!
Christian

#13 written by existdissolve
about 1 week ago

Hi Christian–

You didn't miss the section on the server setup. As I mention in my introduction to the series, my purpose in doing this walkthrough was to specifically address the Ext JS side of things…I tried to steer clear of the server-side stuff as much as possible (after all, Ext JS doesn't particularly care what it's connecting to…).

That being said, you are right: there is a significant amount of work to do on the backend in order to make this a workable project. Definitely not something that could be thrown together in an afternoon, unless of course you're using the same setup and just drop the repo in your project.

On the other hand, this app does not presume any life-altering server-side code voodoo. It is a pretty straight-forward CRUD type app. So if you are looking for tutorials, you could easily pick your language of choice (PHP, .NET, ColdFusion, whatever) and search for tutorials on creating basic CRUD-style applications, specifically tutorials that focus on heavy AJAX integration.

However, there's really not much to what the server side needs to be able to do. The short list is basically:

- Process AJAX request
- Turn request into logic that interacts with the database (read, create, update, delete, etc.)
- Product JSON response to send back to Ext JS

Literally, that would get you to about 99% of the functionality in the app.

Let me know if you have any additional questions–thanks!

Name (required)

E-mail (required, will not be published)

Website

Notify me of follow-up comments by email.

Notify me of new posts by email.

Submit Comment

Search

Adobe *(1)*

Audio *(10)*

Books *(8)*

ColdFusion *(61)*

ColdBox *(12)*

ContentBox *(7)*

Cool Stuff *(39)*

Flex *(5)*

General *(26)*

JavaScript *(88)*

AJAX *(2)*

CKEditor *(3)*

ExtJS *(54)*

ExtJS 4.2. App Walkthrough *(16)*

- *jQuery* (1)
- *Sencha Fiddle* (2)
- *Sencha Touch* (9)
- *Spry Framework* (18)

Microsoft (8)

Mobile (4)

- *Sencha Touch* (4)

Music (26)

Philosophy (16)

PHP (2)

Ruby (6)

SharePoint (8)

Sitecore (5)

Social Media (5)

Theology (81)

Travel (1)

Uncategorized (8)

Video Games (4)

Web Design (45)

- *CSS3* (3)
- *HTML5* (9)

Web Development (12)

WordPress (1)

MY LATEST TWEETS

Warning: Invalid argument supplied for foreach() in **/home/existdis/public_html/wp-content/themes/mystique/lib/widgets.php** on line **26**

Error while retrieving tweets (Twitter down?)