Managing Data

Data management the Ext way.

presented by Brandon Ryall

What is a data store?

To get started, lets talk about what a data store is.

A data store is exactly what is in the name - it’s a repository of information. A data store can be anything from a piece of paper on your desk to a more robust system such as MySQL.

What truly defines a data store is how it is represented - with a schema model.

Example 1

Take the following into example:

I have a book filled with contact information but it’s completely unsorted. I know for a fact that I have most of their contact information (First Name, Last Name, Phone #), but in general it takes me a long time to find my friends information due to the disorganization. How can I possibly make this easier for myself with little effort?

Example 1

Take the following into example:

I have a book filled with contact information but it's completely unsorted. I know for a fact that I have most of their contact information (First Name, Last Name, Phone #), but in general it takes me a long time to find my friends information due to the disorganization. How can I possibly make this easier for myself with little effort?

Since I know the name is always there I can create a data store grouped and sorted by First Name containing the following:

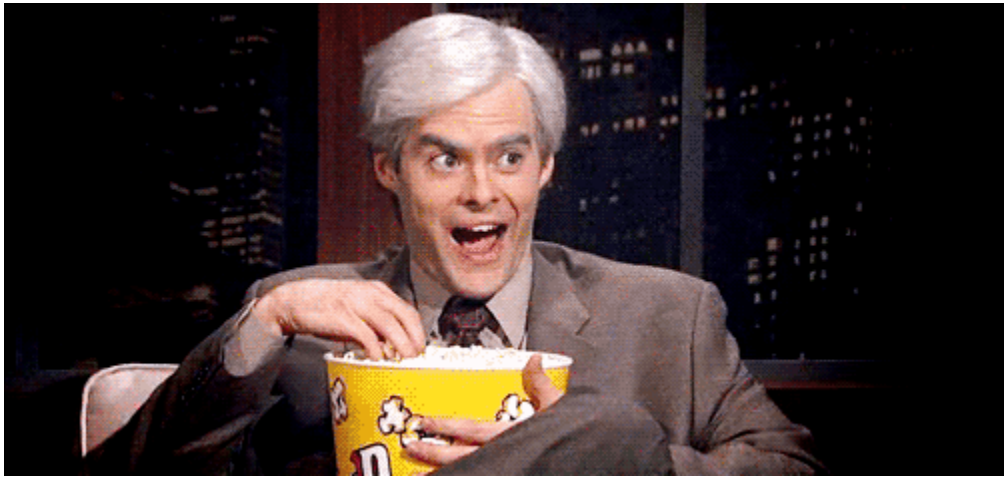
- First Name, Last Name
- Page the full contact information is on

Ok I get it, move on...

The reason we just went over that is so we can start talking more in depth about data management with extjs.

We’re going to talk about quite a few things from the different types of storage engines available, how to pick the best one, to infinite scrolling, and other goodies.

Sit back and enjoy...



What is a Store?

Like we discussed previously a store in extjs is a data store.

There are 3 major types of stores:

1. JSON
    a. This is the most common type of store where your model corresponds to the keys in your data
2. Array
    a. Array stores are fairly uncommon but can be used for small data set, their model indexes correspond to the data indexes
3. Tree
    a. Tree stores are essentially JSON stores but contain children mappings

What does a store need?

There are 2 major parts that go into a store:

1. Model
    a. Models represent how our data is being returned and have functionality for validating this data
2. Proxy
    a. A proxy defines how we read, write, and remove data

There are many more items, but we will only focus on the important stuff...well maybe some “the more you know”.

Intermission Break Dance

We’re about to get down and dirty with some modeling, let’s clear the brain cache for a moment.

Model - Basics

A model is a representation of how our data is returned and how we want it organized and validated.

There are 2 major parts to a model:

1. Fields
    a. An array of either strings or objects defining what keys and data types a data set will have.
2. idProperty
    a. The default is “id”, but you can set it to any unique identifier from your fields list.

Model - Fields

A field defines a key that a model contains. A field can be defined in 2 ways, simply putting the key name in a string or defining an object.

Below is an example of a model's field setup containing both simple and complex fields:

```
Ext.define( 'Alert', {
    extends: 'Ext.data.Model',
    fields: [
        'id',
        'device',
        { name: 'first', type: 'date', dateFormat: 'Y-m-d' }
    ]
} );
```

Model - Fields - Continued

In our example we defined “id” and “device” as simple fields.

The one thing to remember when using a simple field is that the data type will be automatic. It is the same as stating:

```
{ name: 'id', type: 'auto' }
```

What this means is that there will be no conversion, so above we could assume that “id” should be an integer, but if the model is given a string then “id” will continue to be a string.

Model - Fields - Continued

In our example we also defined “first” which would be a more complex field. It’s ideal to use complex fields for their type checking and validation abilities. Let’s talk more about what goes into a complex field.

In our example we had the following:

```
{ name: 'first', type: 'date', dateFormat: 'Y-m-d' }
```

Let’s evaluate what each piece of this means:

- name
  - This is the corresponding key in your data set. This is stating that “first” will be a key in the data set.
- type
  - The type of data for conversion. Ext will ensure any value passed in gets converted to this type. Possible values: auto (default), string, int, float, boolean, date
- dateFormat
  - Only available for date data types, the format in which to display the date. Uses Ext.Date formatting.

Model - Fields - Continued

We've already got the basics covered for a complex field, but what else can go into it?

- defaultValue
  - What the default value should be in the event nothing is returned
- persist
  - Tells our writer (we'll get to those) whether or not to include the fields when dealing with data changes
- useNull
  - Whether or not null values are allowed to be used. This could be anything from conversion issues to setting the value to null. A good example is forcing updates for boolean fields.
- convert
  - A custom conversion function that can be used during data reading

There are a few more, but in general you probably don't need to use them since *most* of that functionality that they handle can also be done outside of the fields.

Model - Data Validation

Model's can also handle data validation to make life a bit simpler. Supported validations can be found in the Ext.data.validations singleton.

Let's continue with our previous model "Alert" and add some validation to it:

```
Ext.define( 'Alert', {
    extends: 'Ext.data.Model',
    fields: [
        'id', 'device', { name: 'first', type: 'date', dateFormat: 'Y-m-d' }
    ],
    validations: [
        { field: 'device', type: 'presence' },
        { field: 'device', type: 'length', min: '1' }
    ]
} );
```

Model - Associations

One of the undervalued abilities of a model is associating. Models can have associations with other models via Ext.data.associations.*.

Possible relationships:

- BelongsTo
  - Many to one relationship, where the owner model is expected to have a foreign key which is the primary key for the associated models
- HasMany
  - One to many relationship
- HasOne
  - One to one relationship, where the owner model is expected to have a foreign key which is the primary key for the associated model

Model - Associations - Continued

Let's look at an example using our "Alert" model and newly created "Device" model. (we're going to leave out the field information, etc. for space)

```
Ext.define( 'Alert', {
     extends: 'Ext.data.Model',
     belongsTo: 'Device'
} );
Ext.define( 'Device', {
     extends: 'Ext.data.Model',
     hasMany: [
          { model: 'Alert', name: 'alerts' }
     ]
} );
```

Model - Associations - Continued

In our example we stated that a “Device” can have many “Alert” models. So how does this get used?

Lets assume that we have a device with an ID of 23 and a reference to device store already.

```
var myDevice = this.getDeviceStore().getById( 23 );

var myDeviceAlerts = myDevice.alerts();
```

The function “alerts” was automatically generated by the name we provided for the *many* reference on “Device”. This function will automatically create a store scoped to the set of alerts for device 23.

This would assume your response data would be something like:

```
{ data: [ { id: 23, name: “Jay”, alerts: [ { id: 1, device: “Jay”, first: “2014-08-08 01:03:04 } ] } ] }
```

Model - Associations - Continued

What are the benefits of using associations?

- Less calls to the server
- Dependency errors go down
- Easier to manage on the client side

Disadvantages?

- Slower requests due in part to larger data sets
- Slightly more complex client side

Model - Did You Know?

A model can also have its own proxy setup

This can be beneficial when we're dealing with associations as we previously discussed. Lets say we want to update the alert records, we can do just that by specifying a proxy with writability.

Model - Summary

Let’s summarize what we just learned about models.

- Powerful! Models have a lot of functionality by themselves
- We can specify the types of data we want to view and actively convert them
- Validation can be done at the model level
- Associations are possible and extremely useful for larger applications
- Have the ability to run proxy configurations

Model - Summary

Let’s summarize what we just learned about models.

- Powerful! Models have a lot of functionality by themselves
- We can specify the types of data we want to view and actively convert them
- Validation can be done at the model level
- Associations are possible and extremely useful for larger applications
- Have the ability to run proxy configurations

Proxy - Basics

A proxy is used by the store (or model) to process data actions. For the most part, your code should never need to interact with the proxy directly but it is important to understand how to set it up and how they work.

There are 2 major types of proxies:

1. Client
    a. Local storage
    b. Session storage
    c. Memory storage
2. Server
    a. AJAX
    b. JSONP
    c. REST
    d. Direct

There are also 2 items that should be in proxy setups:

1. reader
    a. Readers interpret the data to be loaded into the model instance.
2. writer
    a. Writers are used for taking operation objects and passing it through to the request. Writers are only needed for server proxies.

Proxy - Basics

A proxy is used by the store (or model) to process data actions. For the most part, your code should never need to interact with the proxy directly but it is important to understand how to set it up and how they work.

There are 2 major types of proxies:

1. Client
   a. Local storage
   b. Session storage
   c. Memory storage
2. Server
   a. AJAX
   b. JSONP
   c. REST
   d. Direct

There are also 2 items that should be in proxy setups:

1. reader
   a. Readers interpret the data to be loaded into the model instance.
2. writer
   a. Writers are used for taking operation objects and passing it through to the request. Writers are only needed for server proxies.

Proxy - Client

Client side proxies are used to store data in memory or web storage. Below is a list of possible client side proxies:

1. Local Storage
    a. Uses HTML5 localStorage API to store model data. This API is a key=>value type storage and cannot handle overly complex JSON so it relies on serializing and deserializing data. The downside to this type of proxy is browser compatibility. Older browser users would be met with a error.
2. Session Storage
    a. Uses HTML5 sessionStorage API to store model data. This is similar to local storage with the exception that session storage will die when the browser session ends. This should only be used for temporary data.
3. Memory Storage
    a. Uses a local variable for storage, any data changes will be lost on page refresh.

Proxy - Server

Server proxies are used for data that interacts with server side requests. Below is a list of possible server proxies:

1. AJAX
   - Easily the most used proxy for applications. It works by making AJAX requests to handle operations.
2. JSONP
   - Works by injecting script tags into the page to load data. JsonP proxies are useful for loading data from a remote domain.
3. REST
   - Based on the AJAX proxy, it works by making AJAX requests but supports CRUD functionality (POST, GET, PUT, DELETE)
4. Direct
   - Uses Ext.Direct to streamline communication between the client and server. There's a lot more to this than we will probably cover.

Proxy - Hammer Time

Before we get any further with proxies we need to learn about 2 more things. We touched on them briefly already, but now we need to know about them in depth.

1. Readers
2. Writers

Reader

As mentioned earlier, readers are used to interpret the data loaded into our model.

Readers work by looking at the response data, extracting the data, creating accessors, and then passing them off to the model.

There are 3 types of readers:

1. Array
    a. Creates an array based on the model's field setup
2. JSON
    a. Used to interpret data presented in JSON format
3. XML
    a. Used to interpret data presented in XML format

For the sake of this presentation, we're only going to focus on the JSON reader.

Reader - JSON

Below is an example of a JSON reader setup, we'll break it down next:

```
Ext.create( 'Alerts', {
      model: 'Alert',
      proxy: {
            type: 'ajax',
            url: 'alerts.json',
            reader: {
                  type: 'json',
                  root: 'data',
                  successProperty: 'success',
                  totalProperty: 'total',
                  messageProperty: 'msg'
            }
      }
} );
```

Reader - JSON - Continued

Example response:

{ success: true, msg: ‘Here is your data!’, data: [{ ... }] }

Break Down:

- **type**
  - This is the type of reader that corresponds with the type of data being returned. Remember, there are 3: Array, JSON, and XML
- **root**
  - Key for our response data. In the example up top “data” would be our root property, it is a property that distinguishes the key of our data.
- **successProperty**
  - Key for our success property. In the example above “success” would be our successProperty. This is used to tell whether an operation passed or failed.
- **totalProperty**
  - Key for the total number of records. This is generally used for paging.
- **messageProperty**
  - Key for the response message. Ext has the ability to display this message to the user.

Writer

Writers are responsible for taking a set of data operation objects and a request object and modifying the request based on the operation. In short, they find the modified fields in a model and then pass them off to the request. Writers are only used for server proxies.

There are 2 types of writers:

1. JSON
    a. Used for passing data back to the server in JSON format
2. XML
    a. Used for passing data back to the server in XML format

Writer - JSON

For the sake of this presentation, we will only cover JSON writers.

Below is an example of a JSON writer setup within a proxy:

```
...
    writer: {
        type: 'json',
        allowSingle: true,
        encode: false,
        root: 'data',
        writeAllFields: false
    }
...
```

Writer - JSON - Continued

Break Down:

- **allowSingle**
  - Whether or not you want the ability to process multiple actions at once. The default of this is true, if set to false it will wrap your modified record sets in an array
- **encode**
  - Whether or not you want to attempt to encode the data before the request. By default, this is set to false and will send as a raw post - otherwise true will send as parameters.
- **root**
  - Key that the records will be placed under, good practice is to use the same root key as your reader, but this could be anything.
- **writeAllFields**
  - Whether or not to send all fields regardless if they are the modified field. By default, all fields will be sent - so setting this to false will only send the modified fields.

Proxy - Back in Time

Great, so now we know how our data is going to be processed with our proxies so let’s start checking out how proxies work.

For the sake of this presentation we’re going to evaluate only one type of proxy configuration - AJAX.

Proxy - AJAX

The AJAX proxy works by handling data actions from the store (create, read, update, destroy). For each action a data operation is formed followed by a request being built. This request object is going to contain a few things - params, action, records, operation, url, and a reference to the proxy. If you didn’t figure it out already, this object is created with the writer! Once the request object is finalized, an AJAX request is built containing - headers, timeout, scope, callback, and method (POST or GET).

After the request has been made the proxy will now process the response data by checking the success property and passing it off to the reader.

Let’s see how we build one now.

Proxy - AJAX - Build It

```
Ext.create( 'Ext.data.Store', {
    model: 'Alert',
    proxy: {
        type: 'ajax',
        url: 'alerts.json',
        reader: { type: 'json', root: 'data' },
        writer: { type: 'json', root: 'data', writeAllFields: false }
    }
} );
```

Intermission Break Dance

We just covered a lot, lets take a break dance.

Infinite Scrolling

So we've talked about what is behind a basic store, but what about when we have large datasets that we want to display to the user quickly? That's when we bring infinite scrolling into the mix, also known as a buffered store.

Buffered stores work by prefetching and caching data in a page cache, which is a map of page => records.

Buffered stores are generally used in part with the grid component, which in turn utilizes a PagingScroller which acts as the "brain" of the buffered store.

Infinite Scrolling - Paging Scroller

The PagingScroller implements infinite scrolling on a grid which allows the user to scroll through N records without performance impacts since only a subset of the data is rendered at one time.

The PagingScroller is responsible for maintaining the the scroll height based on the total record count and handling loading data based on scroll position.

Infinite Scrolling - Paging Scroller

What goes into a paging scroller?

- **leadingBufferZone**
  - This is the number of extra rows to render on the leading side of scrolling outside the numFromEdge buffer as scrolling proceeds. As an example if set to 100, an additional 100 records will also cache *after* what is currently displayed.
- **trailingBufferZone**
  - This is the number of extra rows to render on the trailing side of scrolling outside the numFromEdge buffer as scrolling proceeds. As an example if set to 100, an additional 100 records will also cache *before* what is currently displayed.
- **numFromEdge**
  - Number of records from the edge of the viewport which causes a refresh of the view (view is moved).
- **scrollToLoadBuffer**
  - This is the number of milliseconds to buffer load requests when scrolling.

Infinite Scrolling - Loading Data

While loading for infinite scrolling still depends on the proxy to handle the requests, there is still a lot done pre-proxy.

In order for a request to be made, the store first needs to check if the range of data exists within our page map. For each page requested (trailing, page, leading) the store will check if a guaranteed range exists from the map. This is done by finding an index based on the scroll position, obtaining a page from that index, and checking if it has a value in the page map.

If the page cannot be guaranteed from the page map a prefetch is fired off. Prefetches are basically the same as a typical load but rather than forcing data into the store immediately, they are passed into a page map for caching.

Infinite Scrolling - Page Map

The page map is a cache map of page => records.

You can ask the page map to maintain as many pages as you want, but keep in mind the browser can get slow if you exceed a large amount. The default page map size is 5 pages. As more pages are added previously added pages are purged off.

As a note, we have found a bug within this functionality wherein once the purge count is hit and again accessing the first page the user is met with a blank grid. We have fixed this internally and have requested Sencha address it within their own codebase.

Infinite Scrolling - Implementing

So now that we understand how it works, lets check out how to implement it.

```
var alertStore = Ext.create( 'Ext.
data.Store', {
    …
        buffered: true,
        pageSize: 100,
    ...
} );
```

```
var alertGrid = Ext.create( 'Ext.
grid.Panel', {
    …
        verticalScroller: {
            trailingBufferZone:
100,
            leadingBufferZone: 100
        }
    ...
} );
```

Infinite Scrolling - Modifications

Over the past few years we've had to make modifications to how stores and infinite scrolling work to handle the large data sets. Below are a few of what we've done:

- Fixed an issue with reloading a store and the page map not updating
- Force all requests as a POST to prevent JSON hijacking
- Allow grouping grids to allow infinite scrolling
- PageMap now prunes pre-page add to account for newly added data

Stores - What Else?

Now that we know what goes into a store, what else is there to know?

- Once a store is initiated on a component, in order to replace it you need to use the bind mixins, otherwise your store will not operate properly and will likely show incorrect data on your component
- If your store has the possibility of exceeding 500 records, use infinite scrolling to improve the user experience
- Don’t localize model values!
- Never ever use a model’s internalId. If your code is relying on this, you’re doing it wrong.
- Ext.data.Operation.commitRecords() has a nasty bug that can cause IE to go haywire if you batch operations of 50+ items together! We have a fix for it :)
- You should never trust that Sencha’s code is great. It may be “good enough” but not great. Their code is often not optimized to handle large data sets. An example is we had a loop that was taking 3 seconds because of a poorly written loop. After fixing it, it was improved down to ~400 milliseconds.

Questions?

A lot of ground was just covered and we only scratched the surface.

What other questions are there?