Sencha

Building an Email Client from Scratch

PART 6

Stuart Ashworth
Sencha MVP

This e-book series will take you through the process of building an email client from scratch, starting with the basics and gradually building up to more advanced features.

PART 1: Setting Up the Foundations

Creating the Application and Setting up Data Structures and Components for Seamless Email Management

PART 2: Adding Labels, Tree and Dynamic Actions to Enhance User Experience

Building a Dynamic Toolbar and Unread Message Count Display for Label-Based Message Filtering

PART 3: Adding Compose Workflow and Draft Messages

Streamlining Message Composition and Draft Editing for Seamless User Experience.

PART 4: Mobile-Optimized Email Client with Ext JS Modern Toolkit.

Creating a Modern Interface for Mobile Devices using Ext JS Toolkit

PART 5: Implementing a Modern Interface with Sliding Menu & Compose Functionality

Implementing Modern toolkit features for the Email Client: Sliding Menus, Compose Button, Forms, etc.

PART 6: Integrating with a REST API

Transitioning from static JSON files to a RESTful API with RAD Server for greater scalability and flexibility

PART 7: Adding Deep Linking and Router Classes to the Email Client Application

Integrating Deep Linking with Ext JS Router Classes for Improved Application Usability

By the end of all the 7 series, you will have a fully functional email client that is ready to be deployed in production and used in your daily life. So, get ready to embark on an exciting journey into the world of email client development, and buckle up for an immersive learning experience!

Tips for using this e-book

1 Start with Part-1 and work your way through each subsequent series in order. Each series builds upon the previous one and assumes that you have completed the previous part.

2 As you read each series, follow along with the code examples in your own development environment. This will help you to better understand the concepts and see how they work in practice.

3 Take breaks and practice what you have learned before moving on to the next series. This will help to reinforce your understanding of the concepts and ensure that you are ready to move on to the next step.

4 Don't be afraid to experiment and customize the code to meet your own needs. This will help you to better understand the concepts and make the email client your own.

5 If you encounter any issues or have any questions, don't hesitate to reach out to the community or the authors of the articles. They will be happy to help you and provide guidance along the way.

6 Once you have completed all the series, take some time to review the entire email client application and make any necessary adjustments to fit your specific needs.

7 Finally, enjoy the satisfaction of having built your own fully functional email client from scratch using Ext JS!

Table of Contents

Executive Summary

In this article, we will update our application to move from being backed by static JSON files to integrating with a fully persistent REST API created with RAD Server.

We will update the proxy configurations of our stores and models to point to our new API and ensure the data is compatible with the application through a series of transform functions.

We will also add a hasMany association between our Message and MessageLabel models so we can handle the normalized data structure that our new API serves.

A utility class will also be added to enable the base URL that the application uses to be easily switched based on the environment the application is running in.

Key Concepts / Learning Points

- Using the rest proxy class
- Utilizing the transform config to ensure compatibility with the API
- Adding hasMany associations between two models and syncing those associations with the API

Code along with Stuart!

Start buddy coding with Stuart on-demand!

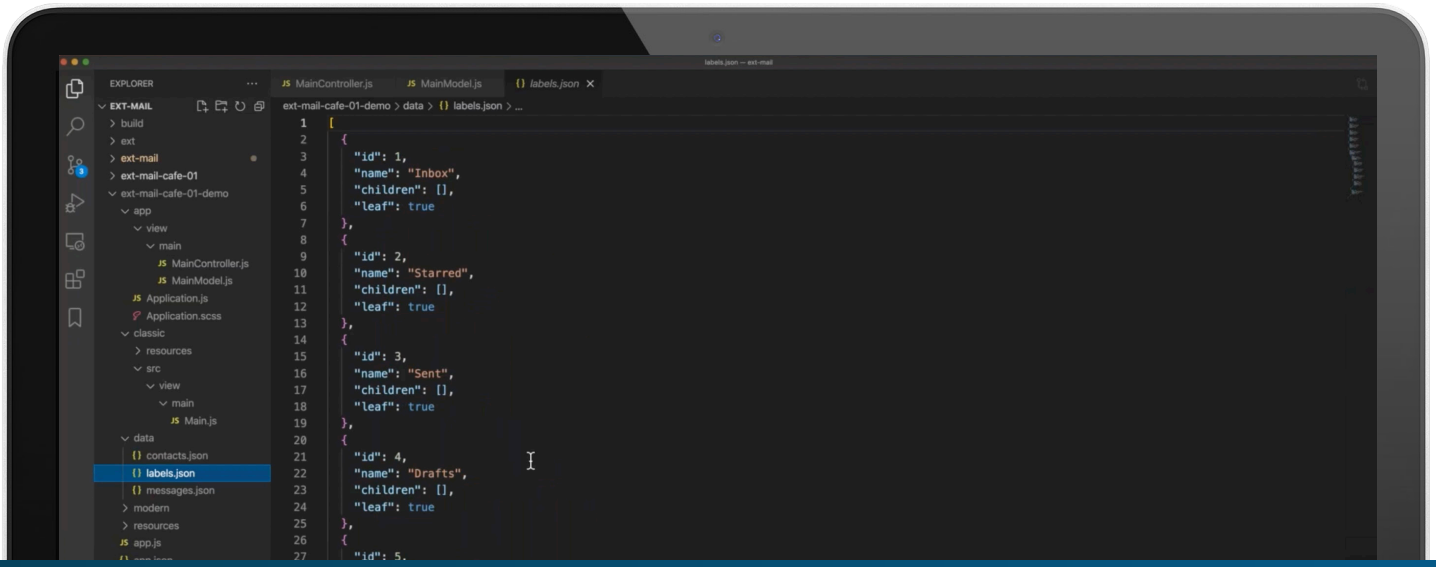
Use the bite-sized video links in this e-book to instantly watch the section you are reading.

Introduction

At the moment our application is just backed by static JSON files rather than a dynamic API. Although the data is loaded via AJAX it is never persisted and will be reset when the browser is refreshed.

We will be demonstrating how to update our data models and stores to handle loading and saving to a REST API.

To do this we will be replacing our uses of the ajax proxy and replacing them with the rest proxy class so our API calls follow the standard REST pattern.



Code along with Stuart!

Start buddy coding with Stuart on-demand!

Use the bite-sized video links in this e-book to instantly watch the section you are reading.

Creating a Base URL Utility Class

Before we update any of our stores we want to configure our application to make any AJAX calls point to our RAD Server API. We could add the API url to each store and model but that would be a lot of duplication and makes it hard to update when we deploy to different environments.

So to do this we create a utility singleton class that will add a hook to the Ext.Ajax class so the URL being called can be prefixed with our API's base URL. This class can then be required at the root of the application, attaching the hook before any stores or models are loaded.

```
Ext.define('ExtMail.util.BaseUrl',{
    singleton : true,
    requires:[
        'Ext.Ajax'
    ],

    config: {
        baseUrl: ''
    },

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

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

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

We add this class to our shared app code in a new folder called util. We set it up as a singleton and require the Ext.Ajax class because it is this class that we will add a global event handler.

We give it a single config of baseUrl which we will set with the API's URL when the app launches.

In the constructor, we set up the config and then add a handler to the Ext.Ajax class' beforerequest event is fired when a request is about to be made.

In our handler, we simply update the url property of the request's options and prefix it with the contents of the baseUrl config. This ensures all Ajax calls go to our API.

> In larger applications this might be a bit heavy-handed as some Ajax calls might go to different destinations so you might need to take that into consideration.

With the class in place it will be required automatically by our wildcard requires set up in the app.js file.

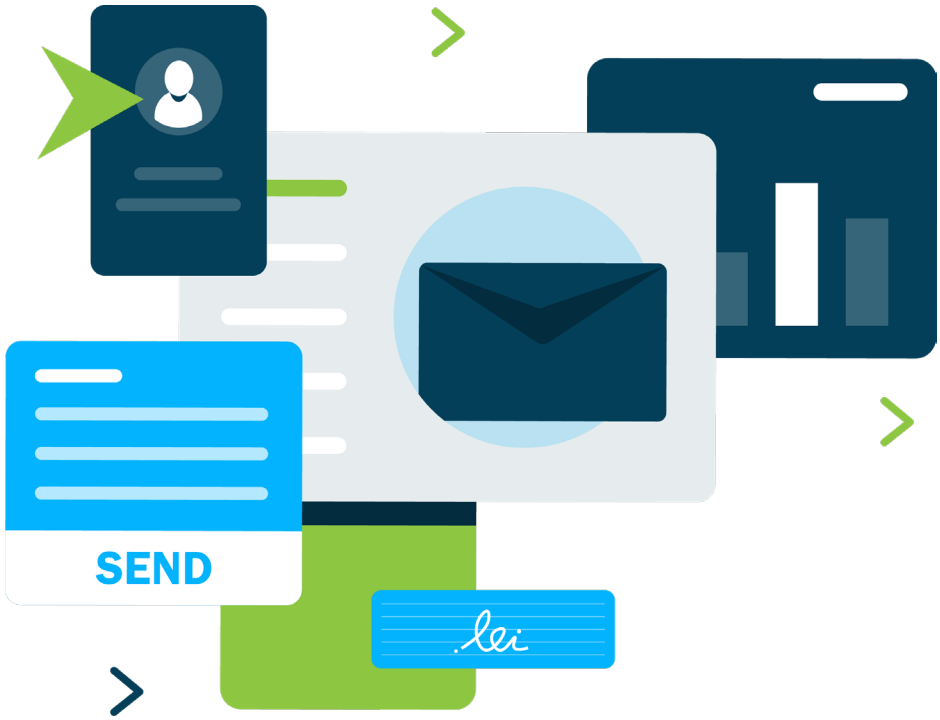
```
Ext.application({
    extend: 'ExtMail.Application',

    name: 'ExtMail',

    requires: [
        'ExtMail.*'
    ],
...
```

With the class loaded (and instantiated, because it is a singleton) we can set the baseUrl in the onBeforeLaunch function within the app.js so it is configured before any Ajax calls are made.

```
onBeforeLaunch: function() {
    ExtMail.util.BaseUrl.setBaseUrl('https://<your-base-url>/');

    this.callParent(arguments);
}
```

In a production application, you might pull this base URL from an environment config during the build process so it is specific to a build environment.

With this in place, we can update our data stores to hit the endpoints within our API.



Learn How to Create a Base URL Utility Class

Watch this Chapter!

Use the bite-sized video links in this e-book to instantly watch the section you are reading.

Hooking up the Contacts Store

The Contacts store is a simple one to start with as it is loaded when the application is launched and doesn't change throughout the app's lifecycle.

The store's proxy definition becomes this:

```
proxy: {
    type: 'rest',
    url: 'data/contacts/',
    reader: {
        type: 'json',
        transform: function(data) {
            return Ext.Array.map(data.result, ExtMail.util.Object.
snakeCaseToCamelCase);
        }
    }
}
```

We start by swapping the proxy type from ajax to rest because we want our calls to use the correct HTTP verb and to be constructed following the REST pattern.

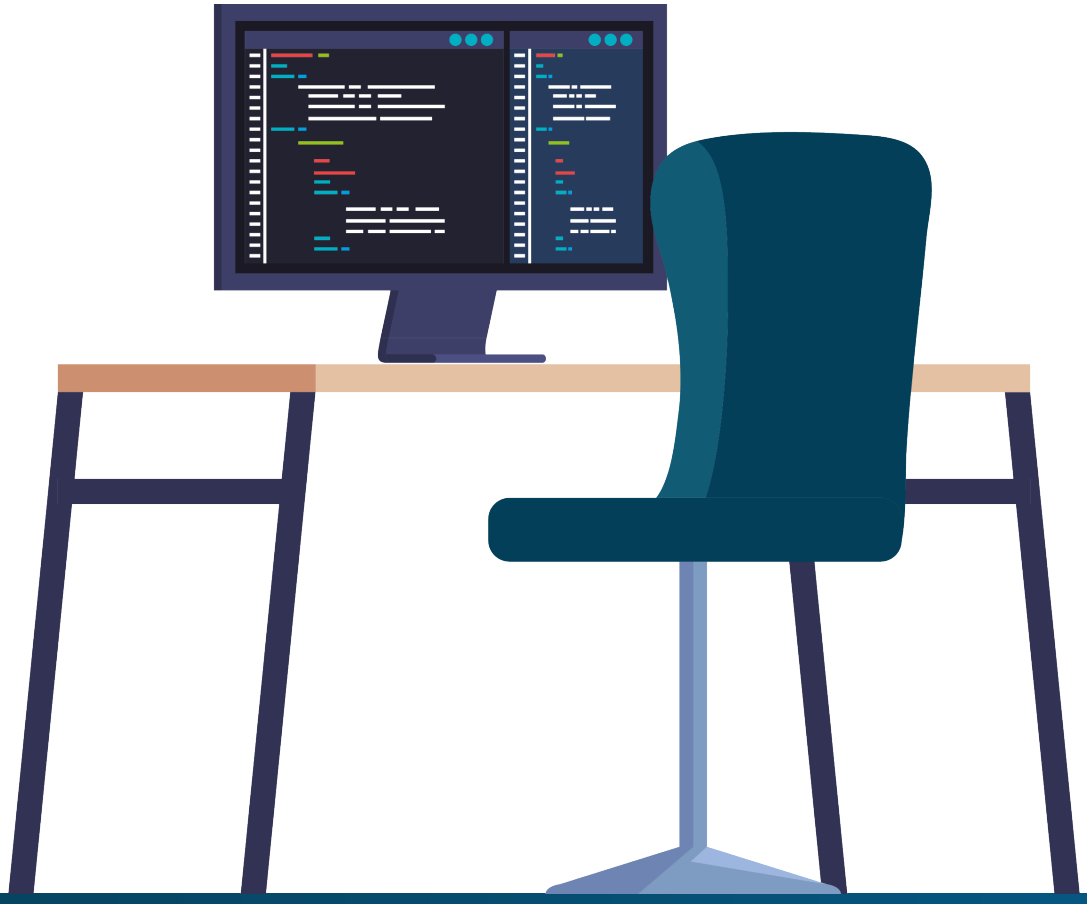
The url can then be changed from pointing to the static json file to the actual REST endpoint, in this case data/contacts. Remember, this will have the base URL prepended to it at the point the request is launched.

The final update we have made is to include a transform function which is called once the data is loaded and allows us to modify the data structure before it is pushed into the store. This isn't always necessary but in our case, we need to transform the data's property names from SNAKE_CASE to camelCase. This is purely to avoid us having to update the references to the properties elsewhere in the application but is a good way to demonstrate how we might handle this process.

> The ExtMail.util.Object class is a basic utility class that can transform an object's keys from snake case to camel case and back again.

In some cases, you might have to use this transform hook to modify the data structure entirely to suit your application's data model.

This is all that is needed to have the store load from the REST API and the application will run as it did before with the data now coming from a different source.



Learn How to Hook-up the Contacts Store

Watch this Step!

Use the bite-sized video links in this e-book to instantly watch the section you are reading.

Loading the Labels Store

Next we will hook the Labels store (which backs the tree view on the left side of the application) into the API.

We follow exactly the same process as we did for the Contacts store, including adding a transform step to update the object keys.

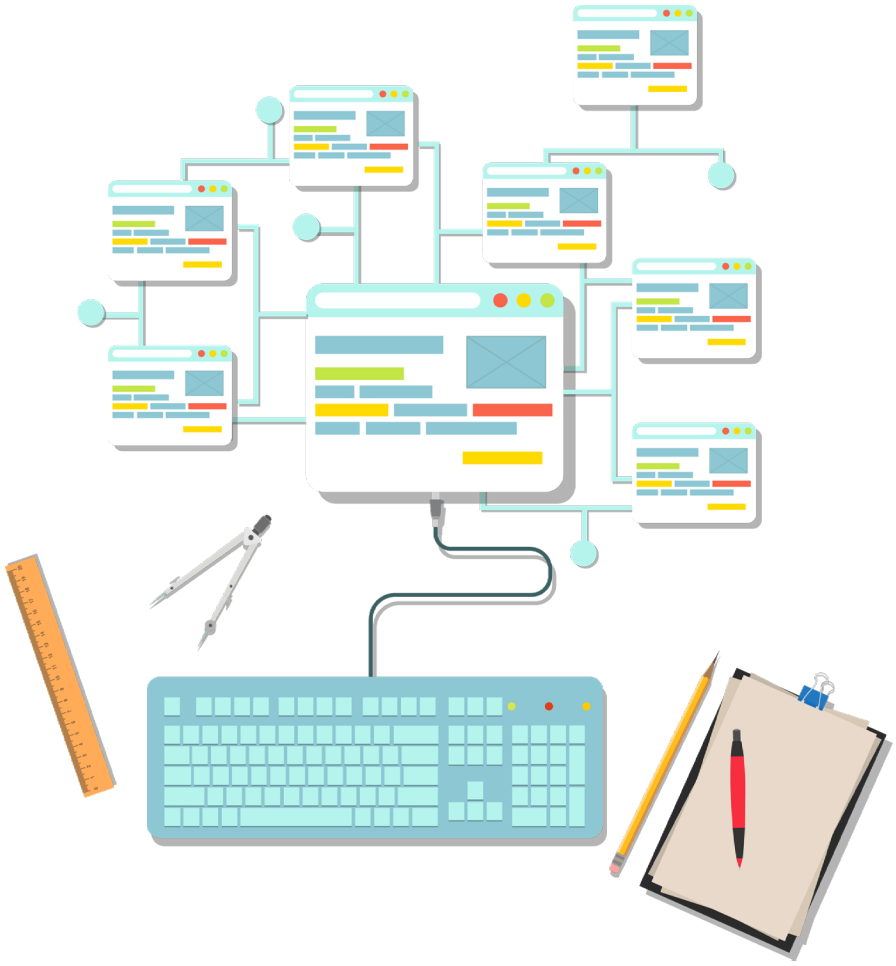
```
proxy: {
    type: 'rest',
    url: 'data/labelsnested',
    reader: {
        type: 'json',
        transform: function(data) {
            var transformRow = function(row) {
                row = ExtMail.util.Object.snakeCaseToCamelCase(row);

                if (row.children && Ext.isArray(row.children)) {
                    row.children = Ext.Array.map(row.children, transformRow);
                }

                return row;
            }

            return Ext.Array.map(data, transformRow);
        }
    }
}
```

First we switch the proxy type from ajax to rest, and then update the URL to use data/labelsnested which will give our labels back in a tree structure, with child nodes inside a property named children.

We then add an explicit JSON reader to the proxy so we can add a transform step.

In this case we need our transform to traverse down the tree processing every node. We define a transformRow function which converts the object and then recurses down into its children. This will then return us an array of labels and their children whose keys have all been converted to camelCase.

That is all that is required to hook up the API and as you can see the most complex bit is the transformation to make the data compatible with our existing application. Generally speaking, this step would not be necessary.



Discover How to Load the Labels Store

Watch this Part!

Use the bite-sized video links in this e-book to instantly watch the section you are reading.

Refactoring Message Labels

Before we can hook up the main Messages store we need to do some refactoring around how messages are attached to labels. The new API now has the association normalized with an intermediary message_labels database table, so we need to mirror this structure so we can load and save new links with our front-end.

Updating the Message Model

We start by making some updates to the Message model. First, we change its automatic ID generation scheme to use the Sequential class which gives us numeric IDs for any new Messages that are created. We do this because the API uses numeric IDs and we want to mirror its types.

```
Ext.define('ExtMail.model.Message', {
    extend: 'Ext.data.Model',

    requires: [
        'Ext.data.identifier.Sequential',
        'ExtMail.store.MessageLabels'
    ],

    identifier: 'sequential',
    ...
});
```

Next we want to stop the labels field in the model from being saved to the server when the Message details (content, recipient etc) are updated because the labels will be handled by a new model we'll add next.

```
{
    name: 'labels',
    type: 'auto',
    defaultValue: [],
    persist: false
}
```

Now we can create a hasMany relationship on a Message linking it to any number of labels. We use the hasMany config to define this relationship, which accepts an array of relationship configurations.

A hasMany relationship essentially creates a sub-store on the model which holds a set of associated model instances.

```
hasMany: [
    {
        model: 'ExtMail.model.MessageLabel',
        name: 'labels',
        storeConfig: {
            type: 'MessageLabels',
        },
    },
]
```

First we tell it what model class to use for each of the related entities. In this case, we reference a new MessageLabel model. We define the key where the associated data will be found - in our case the labels property. We can then provide it with some configuration details for how to create the sub-store, which we want to be of type MessageLabels which is a new store we will define now.

Adding MessageLabels Store & MessageLabel Model

```
Ext.define('ExtMail.store.MessageLabels', {
    extend: 'Ext.data.Store',

    alias: 'store.MessageLabels',

    model: 'ExtMail.model.MessageLabel',

    autoSync: true
});
```

This store is very simple and just describes the model it uses and is set up to automatically sync when its contents change. This means it will send a request to the API when a label is added or removed from a Message. In this case, we will define our proxy on the model definition.

The MessageLabel model has a simple structure with just two fields - the messageId and the labelId it is joined to:

```
Ext.define('ExtMail.model.MessageLabel', {
    extend: 'Ext.data.Model',

    fields: [
        'messageId',
        'labelId'
    ]
});
```

Next we add the proxy definition which defines how the label associations will be saved (added or deleted).

```
proxy: {
    type: 'ajax',
    noCache: false,
    url: 'labels/',
    actionMethods: {
        create: 'POST',
        destroy: 'DELETE',
    },
    writer: {
        type: 'json',
        writeAllFields: true,
        transform: function (data) {
            if (data) {
                data = ExtMail.util.Object.camelCaseToSnakeCase(data);
            }

            return data;
        },
    },
},
```

The proxy is similar to the other ones with the url set to the "labels" endpoint. However, we keep this one as an `ajax` type because we need to construct our URL manually, which we will do at the end of this section. We set the HTTP verb that each action uses because our setup differs slightly from the usual REST pattern.

The next thing we define is the writer config. We want it to write as JSON so we use the json writer type and we want it to send all the data's fields, regardless of what has changed, so we set the writeAllFields property to true.

We then set up a transform function - which the outgoing data will be put through before it is sent to the server - to turn the camelCase property names into SNAKE_CASE.

The final step is to configure the model's URL with the correct URL which would normally be done by the rest proxy but we want to combine multiple fields into the url which the rest proxy won't do out of the box.

For this reason, we call the proxy's setUrl method in the constructor passing in the messageId and labelId fields to give the URL the pattern of labels/<messageId>/<labelId>.

```
constructor: function () {
    this.callParent(arguments);

    this.getProxy().setUrl(
        Ext.String.format(
            'labels/{0}/{1}',
            this.get('messageId'),
            this.get('labelId')
        )
    );
}
```

Updating the Message Label Utility Methods

In our Message model we have 3 utility methods for manipulating the labels ID array on the model instance. Now that we have a hasMany association defined we need to query and manipulate the sub-store that forms that association.

The sub-store can be accessed by calling a method named the same as the name config of our association. We named our association labels so we can access the sub-store using the labels() method, which returns a store instance - in our case an instance of the MessageLabels store.

The first method to update is the hasLabel method which can now use the findExact method of the Store class to see if the label is already present. It will return the index of the record if it is found, so if the returned index is greater than or equal to zero then it is present.

```
hasLabel: function(labelId) {
    return this.labels().findExact('labelId', labelId) >= 0;
}
```

Next we update the addLabel method so it performs an add operation to the association's sub-store.

```
addLabel: function(labelId) {
    var labels = this.get('labels') || [];

    labels.push(labelId);

    this.set('labels', Ext.clone(labels)); // clone so it triggers an update
on the record

    this.labels().add({
        messageId: this.getId(),
        labelId: labelId
    });
}
```

The add operation populates the two fields of the MessageLabel model - the messageId which comes from the Message model instance we're acting on, and the labelId that is passed in.

Finally, we update the removeLabel method to find the index of a record that has the given labelId and then remove it from the sub-store.

```
removeLabel: function(labelId) {
    var labels = this.get('labels') || [];

    labels = Ext.Array.remove(labels, labelId);

    this.set('labels', Ext.clone(labels)); // clone so it triggers an update
on the record

    var index = this.labels().findExact('labelId', labelId);

    this.labels().removeAt(index);
}
```

When these add or remove operations happen it will automatically sync to the API because we set the autoSync: true config on the MessagesLabel store, meaning we don't have to manually trigger the sync to happen.

Explore How to Refactor Message Labels

Watch this Step!

Use the bite-sized video links in this e-book to instantly watch the section you are reading.

Hooking up the Messages Store

Next we can configure the Messages store to load from the API. We make use of the rest proxy and the api config to specify which endpoint we want to use for each of the CRUD operations.

```
proxy: {
    type: 'rest',
    noCache: false,
    api: {
        read: 'data/messagesfull',
        create: 'data/messages/',
        update: 'data/messages/',
        destroy: 'data/messages/',
    }
},
```

In this case we only want a different endpoint for the read operation but we still need to specify the same one for all of the others.

Next we define the reader which once again needs a transform function defined to transform the property names and also to convert the received array of label IDs into a structure compatible with the MessageLabel model. So for each ID in the labels property, we turn it into a simple object with the parent messageId and the labelId within it. These objects will then be turned into MessageLabel model instances in the association's sub-store.

```
reader: {
    type: 'json',
    transform: function (data) {
        if (data) {
            if (Ext.isArray(data)) {
                data = Ext.Array.map(
                    data,
                    ExtMail.util.Object.snakeCaseToCamelCase
                );

                // turn simple array of labelIds into MessageLabel object
                Ext.each(data, function (row) {
                    row.labels = Ext.Array.map(
                        row.labels || [],
                        function (labelId) {
                            return {
                                messageId: row.id,
                                labelId: labelId,
                            };
                        }
                    );
                });
            } else {
                data = ExtMail.util.Object.snakeCaseToCamelCase(data);
            }
        }

        return data;
    },
},
```

This will often be unnecessary as the API would return the data in this structure already.

Explore Hooking up the Messages Store

Watch this Phase!

Use the bite-sized video links in this e-book to instantly watch the section you are reading.

Saving the Message Updates

The last step is to set up the Message writer to save changes to the Message models.

```
writer: {
    type: 'json',
    writeAllFields: true,
    transform: function (data) {
        if (data) {
            data = ExtMail.util.Object.camelCaseToSnakeCase(data);

            delete data.ID;

            // format date as server expects
            data.DATE = Ext.Date.format(
                Ext.Date.parse(data.DATE, 'c'),
                'm/d/Y H:i:s'
            );
        }

        return data;
    },
},
```

We use a json writer and opt to include all fields in the request, even if they haven't changed. We also include a transform function that will, once again, convert the field names from camelCase to snake_case. In addition to this, we delete the ID property as this is included in the URL and we reformat the date field from a standard ISO format to a custom format that the API expects.

With this setup, the Message data will get sent to the server and saved correctly. The only problem that we will hit is that there will be a lot of AJAX calls sent when you type a new message as each keystroke will trigger a save.

To prevent this we will introduce a buffer to the store's sync method which will prevent any repeat calls from being dispatched within a 500ms timeframe.

We do this by overwriting the sync function in the constructor with a new version of the function returned by the Ext.Function.createBuffered method.

```
constructor: function () {
    this.callParent(arguments);

    // buffer sync operations so it doesn't happen too many times in quick
succession
    this.sync = Ext.Function.createBuffered(this.sync, 500, this);
}
```

Learn How to Save Message Updates

Watch this Stage!

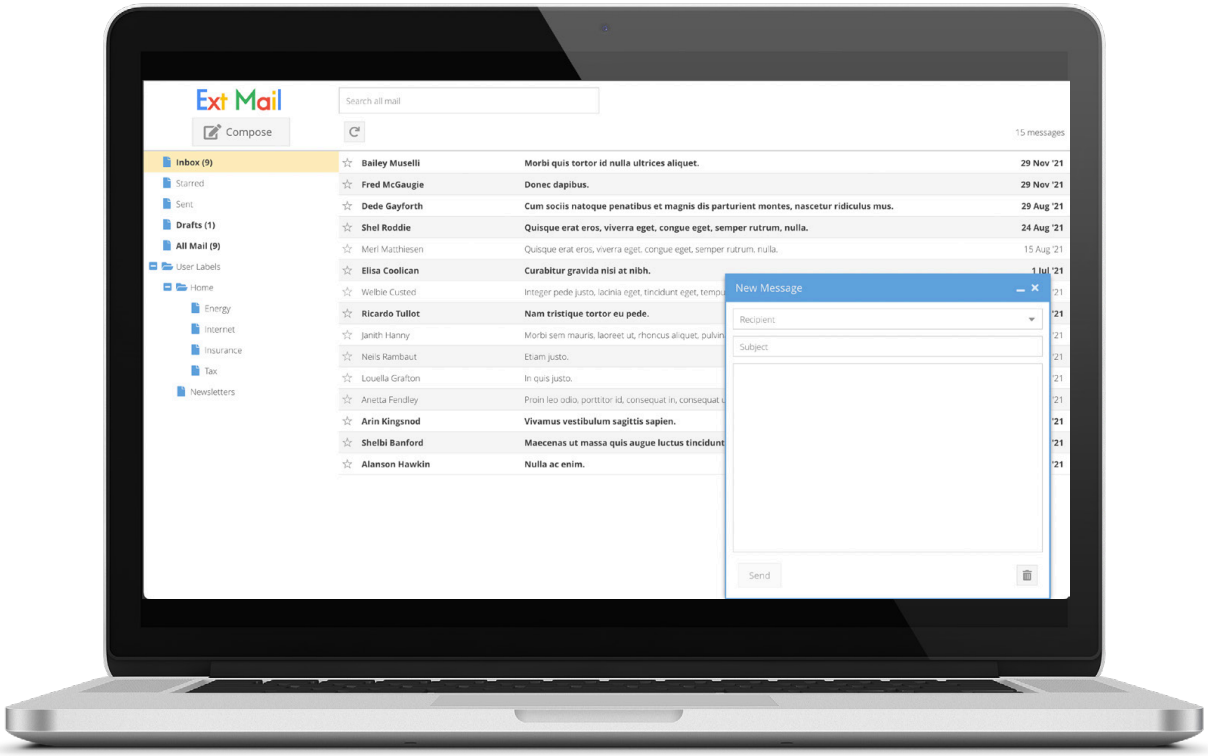
Use the bite-sized video links in this e-book to instantly watch the section you are reading.

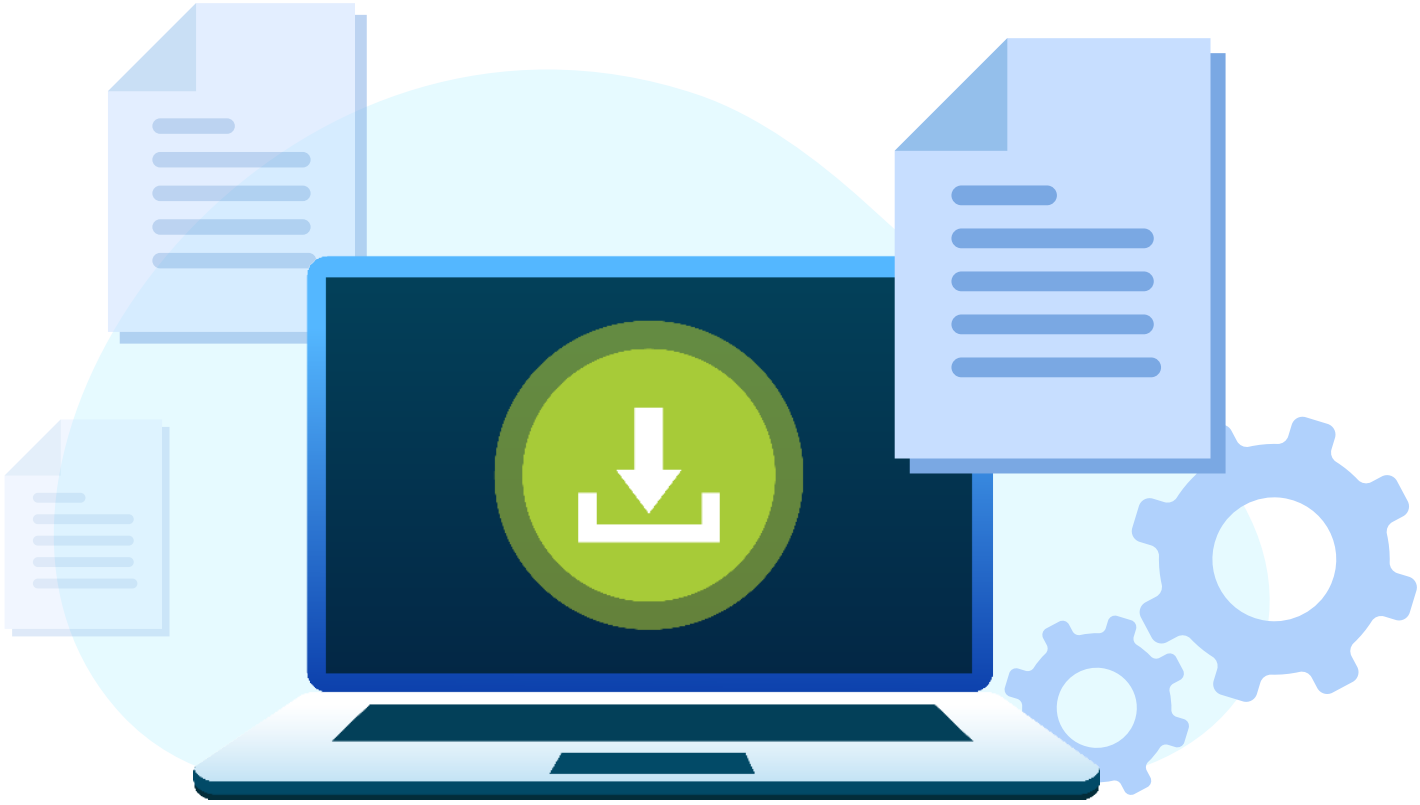
SUMMARY

With these relatively simple updates to our data models and stores, we have successfully hooked our application up to a persistent backend API allowing our application's state to be saved.

We made use of transform functions to ensure compatibility between our application and the API and refactored the way Messages and Labels are associated so it is mirroring the normalized data structure that the API has implemented.

We also created a utility class to ensure all AJAX calls are sent to the correct URL, making it easy to switch between environments during the different stages of the development cycle.





Thank you for reading!

Part-6 of Building an Email Client from Scratch

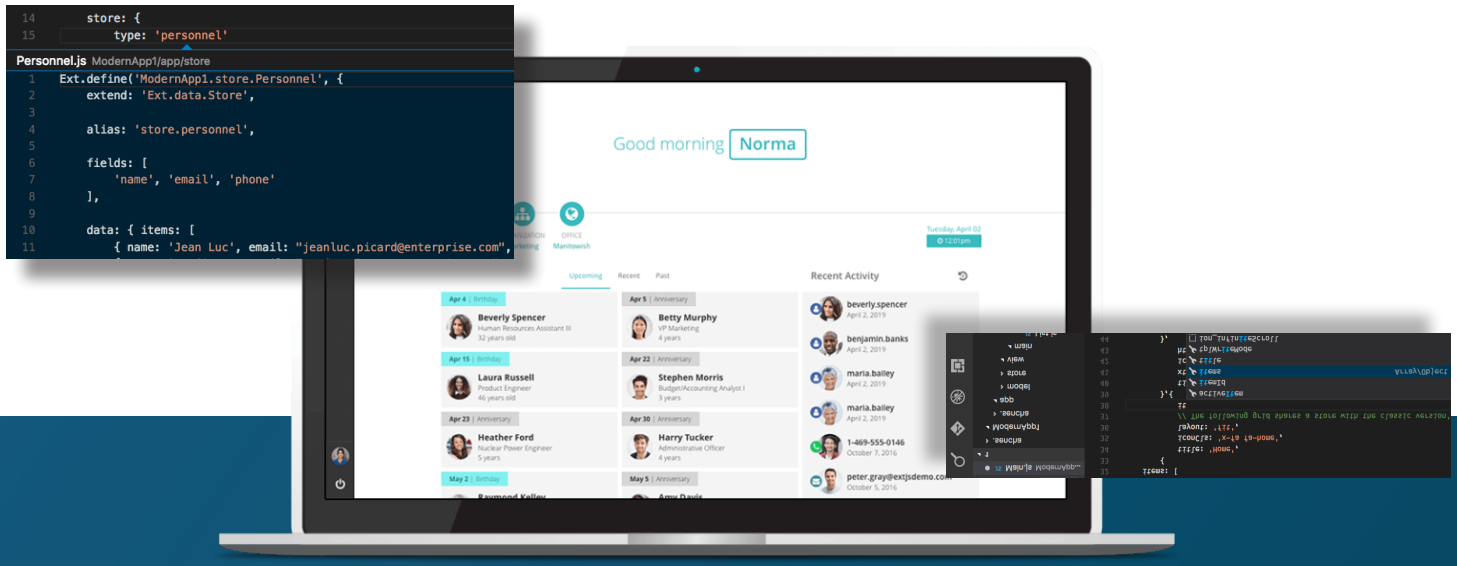
We hope you found it informative and helpful in your development projects. We have 1 more part lined up to take you through the entire process of building an email client from scratch.

Download the Part-7 of Building an Email Client from Scratch

Click Here to Download Now!

Try Sencha Ext JS

FREE for 30 DAYS



Save time and money.

Make the right decision for your business.

START YOUR FREE 30-DAY TRIAL

MORE HELPFUL LINKS:

See It in Action

Read the Getting Started Guides

View the tutorials

Sencha