Sencha

Building an Email Client from Scratch

PART 2

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

This article continues our series where we are building an email client with Ext JS. Picking up from the last part where we set up our data models and created Message grid and Message Reader components, tying them together with events and shared data.

We will now go a step further and introduce a Labels tree to our application so users can view subsets of Messages based on the labels that they have.

We will also add additional actions both in our dynamic toolbar and in the Message grid itself.

Finally, we will be calculating unread counts for each Label and, using a custom tree column renderer, displaying them to the user. All while automatically staying in sync with the Messages store.

Key Concepts / Learning Points

- Working with Action Columns
- Working with custom events
- Binding to store shorthand values
- Working with Store filters and calculating stats
- Working with Trees and customizing display values

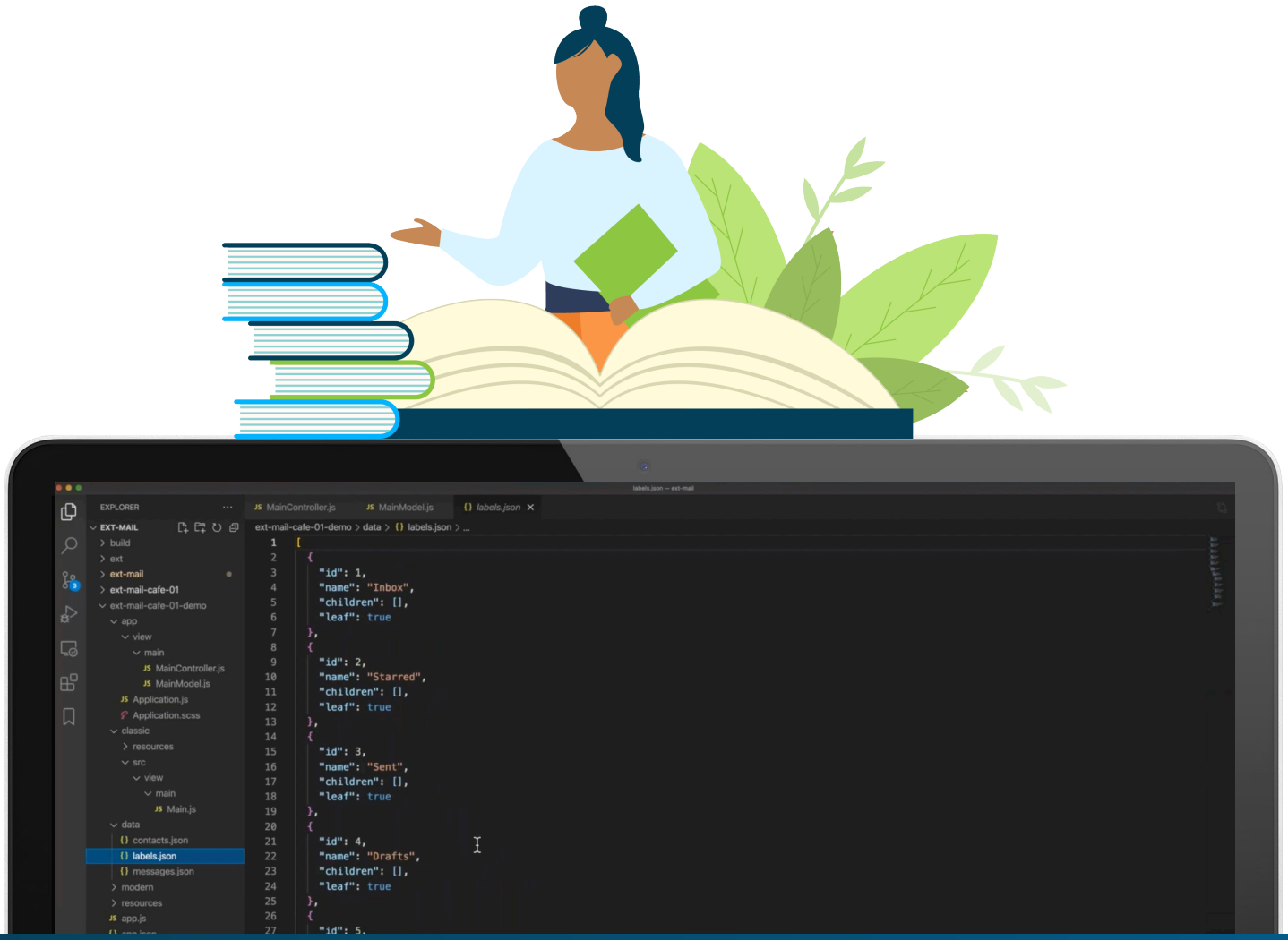
Code along with Stuart!

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

We assume you have followed the first part of this series where we set up the foundations of our application, if you haven't, we suggest going back to that article to get a grasp of the application's code before jumping in to add these new features.



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.

Starring and Unstarring Messages

We start by adding the ability for the user to star and unstar a message by adding an action button to each row of the message grid. This will render a solid star when it has been starred and an empty one when it hasn't.

We do this by adding an `actioncolumn` as the grid's first column. An action column accepts an `items` array of actions which we will give a `glyph` and a `tooltip`. Rather than having a single action that toggles its action we will define two - one for starring, one for unstarring - and show and hide them as required.

```
columns: [
    {
        xtype: 'actioncolumn',
        width: 25,
        items: [
            {
```

```
columns: [
    {
        xtype: 'actioncolumn',
        width: 25,
        items: [
            {
                glyph: '★',
                tooltip: 'Star',
            },
            {
                glyph: '★',
                tooltip: 'Un-star',
            }
        ]
    }
    ...
```

To ensure only one of the actions is visible at a time we add a `getClass` config which allows us to define a function that will return a CSS that should be added to the action. In this function for the 'Star' button we will add the `x-hidden-display` CSS class (a utility class that adds `display: none` to the element) if the record's `starred` field is true. We will do the opposite for the 'Unstar' button so it is hidden when the record is not starred.

```
columns: [
    {
        xtype: 'actioncolumn',
        width: 25,
        items: [
            {
                glyph: '★',
                tooltip: 'Star',
                getClass: function(v, meta, rec) {
                    if(rec.get('starred')) {
                        return 'x-hidden-display';
                    }
                }
            },
```

```
            {
                glyph: '★',
                tooltip: 'Un-star',
                getClass: function(v, meta, rec) {
                    if(!rec.get('starred')) {
                        return 'x-hidden-display';
                    }
                }
            }
        ]
    }
    ...
```

Finally we hook up the functionality to toggle the `starred` field when the buttons are clicked. To do that we define a `handler` config which we will use to raise a custom event for each of the actions on the parent grid. The event will have the row's record attached to it.

```
...
{
    glyph: '★',
    tooltip: 'Star',
    handler: function(view, rowIndex, colIndex, item, e, rec) {
        view.grid.fireEvent('starmessage', rec);
    },
    ...
},
{
    glyph: '★',
    tooltip: 'Un-star',
    handler: function(view, rowIndex, colIndex, item, e, rec) {
        view.grid.fireEvent('unstarmessage', rec);
    },
    ...
}
...
```

With the Message Grid raising these two events we can add handlers for them in the Main component. These handlers will reference function names that will be part of the MainControllerBase as these actions will be common across both toolkit applications.

```
...
{
    xtype: 'messages-MessageGrid',
    bind: {
        store: '{messages}'
    },
    listeners: {
        itemclick: 'onMessageClick',
        starmessage: 'onStarMessage',
        unstarmessage: 'onUnStarMessage'
    }
}
...
```

These handler functions will add or remove the Starred label to the Message record, set the `starred` flag to true or false and then call the record's `commit` method to set the changes.

```
...
onSatarMessage: function(messageRecord) {
    messageRecord.addLabel(ExtMail.enums.Labels.STARRED);
    messageRecord.set('starred', true);
    messageRecord.commit();
},
onUnStarMessage: function(messageRecord) {
    messageRecord.removeLabel(ExtMail.enums.Labels.STARRED);
    messageRecord.set('starred', false);
    messageRecord.commit();
}
...
```

By making these updates the UI will automatically update to reflect the new values. This means the grid will re-render making the Star/Unstar actions swap around showing the correct action.

When we created the `addLabel` and `removeLabel` methods on the Message model you will remember they accepted an ID, but in this code, we pass in a property of the `Labels` enum. We have created this enum so we can deal with English-readable values for labels instead of their IDs. This enum class is a simple singleton with properties for each of the labels, with their values being the ID that is found in the messages.json data.

```
Ext.define('ExtMail.enums.Labels', {
    singleton: true,
    INBOX: 1,
    STARRED: 2,
    SENT: 3,
    DRAFTS: 4,
    ALL_MAIL: 5
});
```

The advantage of this approach is that we aren't using 'magic IDs' in our code which could be changed at a later date. We also don't have to remember which ID refers to which label which makes our code more readable.

Explore Starring and Unstarring Messages

Watch this Section!

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

Adding Extra Toolbar Actions

We now want to allow users to delete, archive, and mark as unread messages when viewing their details. To achieve this we can add 3 new buttons to the MessagesToolbar component. We only want these buttons to be available when on the MessageReader view so we use the same binding to the `hidden` config as we used previously.

```
...
{
    tooltip: 'Archive',
    iconCls: 'x-fa fa-archive',
    handler: 'onArchiveClick',
    hidden: true, // hide from start
    bind: {
        hidden: '{!selectedMessage}'
    }
},
{
    tooltip: 'Delete',
    iconCls: 'x-fa fa-trash',
    handler: 'onDeleteClick',
    hidden: true, // hide from start
    bind: {
        hidden: '{!selectedMessage}'
    }
},
{
    tooltip: 'Mark as Unread',
    iconCls: 'x-fa fa-envelope',
    handler: 'onMarkAsUnreadClick',
    hidden: true, // hide from start
    bind: {
        hidden: '{!selectedMessage}'
    }
}
...
```

In each of the handler functions that we have referenced we raise a custom event that can be handled in the Main component, where we will attach methods from the MainControllerBase class.

```
...
{
    xtype: 'messages-MessagesToolbar',
    dock: 'top',
    height: 56,
    listeners: {
        refresh: 'onRefreshMessages',
        back: 'onBackToMessagesGrid',
        delete: 'onDeleteMessage',
        markunread: 'onMarkMessageUnread',
        archive: 'onArchiveMessage'
    }
}
...
```

First we define the `onDeleteMessage` method which will remove the `selectedMessage` record from the `messages` store and then take the user back to the Messages grid.

```
onDeleteMessage: function() {
    var vm = this.getViewModel();

    vm.getStore('messages').remove(vm.get('selectedMessage'));

    this.onBackToMessagesGrid();
}
```

Next we define the `onMarkMessageUnread` method which simply flips the `unread` flag on the `selectedMessage` record to `true` and moves the user back to the Messages grid.

```
onMarkMessageUnread: function() {
    this.getViewModel().get('selectedMessage').set('unread', true);

    this.onBackToMessagesGrid();
}
```

Finally, the archive action will remove the Archive label from the `selectedMessage` record's `labels` array using the `removeLabel` helper function. We then move back to the Messages grid.

```
onArchiveMessage: function() {
        this.getViewModel().get('selectedMessage').removeLabel(ExtMail.enums.
Labels.INBOX);

        this.onBackToMessagesGrid();
    }
```

> Note that we use the `INBOX` enum property to remove the label.

With these new actions hooked up, we can move on to adding another item to the toolbar to display the current number of messages on display in the grid.

Learn How to Add Extra Toolbar Actions

Watch this Step!

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

Adding Message Count Summary

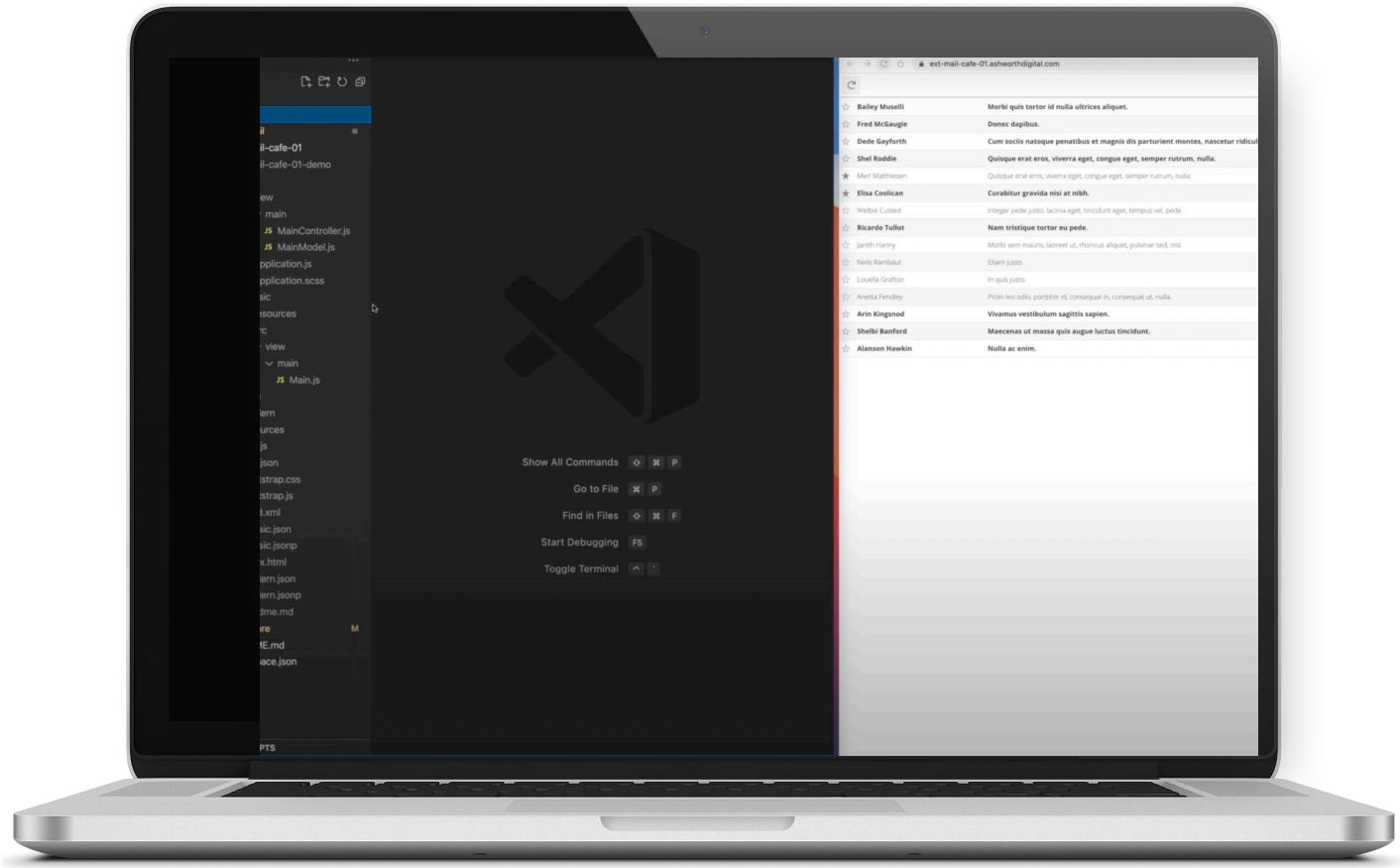
To do this we add a simple component to the MessagesToolbar component and push it all the way to the right-hand side by using the spacer component's shorthand. We want this component to be visible only on the Message grid view so we bind its `hidden` config accordingly.

```
items: [
    ...
    '->',
    {
        xtype: 'component',
        tpl: '{count} messages',
        data: {},
        bind: {
            hidden: '{selectedMessage}',
            data: {
                count: '{messages.count}'
            }
        }
    }
]
```

The template for this component is very simple, only displaying a count of the messages. We want this count value to be bound to the number of items in the Messages store so we define the `data` config (which will be used by the template to generate the final HTML) and use the special shorthand syntax of `{messages.count}` to bind it to the record count within the store.

This binding will cause the template to be re-rendered when a Message is added or removed from the store. This shorthand is a hugely valuable feature that means we don't need to set up event handlers on the store's add and remove events and manually update a data property on each change.

> There are 5 shorthand properties of a store that we can bind to: `count`, `first`, `last`, `loaded`, and `totalCount`. The `first` and `last` properties will return the first and last record in the store respectively, and re-evaluate should that record change, i.e. via reloading, sorting, or filtering.



Discover How to Add a Message Count Summary

Watch this Part!

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

Creating the Labels Tree

We now want to allow users to navigate the different labels that messages are part of. For this, we will create a tree on the left-hand side of the application that will display all of the labels that we set up in the `ExtMail.store.Labels` store.

Add Labels Store to View Model

We have the Labels store defined and ready but we haven't yet told the application to instantiate it. We do this by adding it to the `MainModel.js` by adding it to the `requires` array so the store is available.

```
...
requires: [
    'ExtMail.store.Messages',
    'ExtMail.store.Labels'
],
...
```

Then we add a `labels` key to the `stores` configuration, and configure it using the `Labels` alias we defined.

```
...
stores: {
    ...
    labels: {
        type: 'Labels'
    }
}
...
```

Define the Tree

The Labels tree component is defined in the `classic` folder and named `ExtMail.view.labels.LabelTree` and extends the `Ext.tree.Panel` component.

```
Ext.define('ExtMail.view.labels.LabelsTree', {
    extend: 'Ext.tree.Panel',
    alias: 'widget.labels-LabelsTree',

    initComponent: function() {
        Ext.apply(this, {
            rootVisible: false
        });

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

As usual we give it a widget alias mirroring the class' file system location. In this component, we demonstrate the use of the `initComponent` lifecycle hook where we can merge our component's config onto the class at instantiation time.

We give set our tree to have `rootVisible` false because we don't want our labels to show as children of a single item - we want the top level to be all of our top-level labels

The tree component is similar to a grid in that it requires a set of columns to be defined, the difference being that the rows are expandable to reveal child items in a nested structure. They also use a special column type of `treecolumn` which supports this functionality.

For this tree, we want a single column showing the label's `name` field. We want to hide the header and we want it to take up the entire width of the panel.

```
...
columns: [
    {
        xtype: 'treecolumn',
        header: false,
        text: 'Name',
        dataIndex: name,
        flex: 1
    }
]
...
```

Short of a store bound to it, this is all we need for our tree to render as we want. We can now require it into our Main component and add it as the `west` region of our border layout.

```
...
items: [
    {
        xtype: 'labels-LabelsTree',
        region: 'west',
        width: 300,
        bind: {
            store: '{labels}'
        }
    },
    ...
]
```

We bind the `labels` store from our View Model to the tree and this will populate it with the loaded labels data.

Binding the Selected Label

With our labels tree in place we need to react to the user selecting one of the labels and filter the Messages grid appropriately.

To do this we bind the `selection` property of the tree to a View Model property. In this case, we will create a data property called `selectedLabel`.

```
// main/MainModel.js
data: {
    // holds the ExtMail.model.Label instance of the selected label in the
tree
    selectedLabel: null,
    ...
}
```

The binding can then be added to the component's config in Main.js.

```
 items: [
     {
         xtype: 'labels-LabelsTree',
         region: 'west',
         width: 300,
         bind: {
             store: '{labels}',
             selection: '{selectedLabel}'
         }
     },
     ...
 ]
```

With this added, whenever a user clicks a label in the tree the `selectedLabel` data property will be updated with the Label model instance of that selected label.

Filter the Messages By Label

Next we can react to this change and filter the Messages store to only show the ones relevant to that label.

To do this we add a binding to the `selectedLabel` data property in the MainModel.js and attach a handling function called `onSelectedLabelChange`.

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

    // filter the Messages store when the selectedLabel changes
    this.bind('{selectedLabel}', this.onSelectedLabelChange, this);
    ...
}
```

First, this method will clear any existing filters on the Message store so we start with the entire data set. We then use the store's `filterBy` method which accepts a function as it's the first parameter. This function will be passed each of the store's records in turn and if the function returns true it will be kept, otherwise, it will be excluded from the store.

> It's important to note that when filtering a store the full data set is still there, the omitted items are just excluded from the 'working' data set which any bound views will use.

In this case, we want to use the Message record's `hasLabel` method to determine if the Message has the selected Label ID in its labels array - if it does then we keep it, otherwise, we omit it.

We also check that the `labelRecord` is truthy because the selection could be empty.

```
...
onSelectedLabelChange: function(labelRecord) {
    this.getStore('messages').clearFilter();

    this.getStore('messages').filterBy(function(messageRecord) {
        return labelRecord ? messageRecord.hasLabel(labelRecord.getId()) :
false;
    });
}
...
```

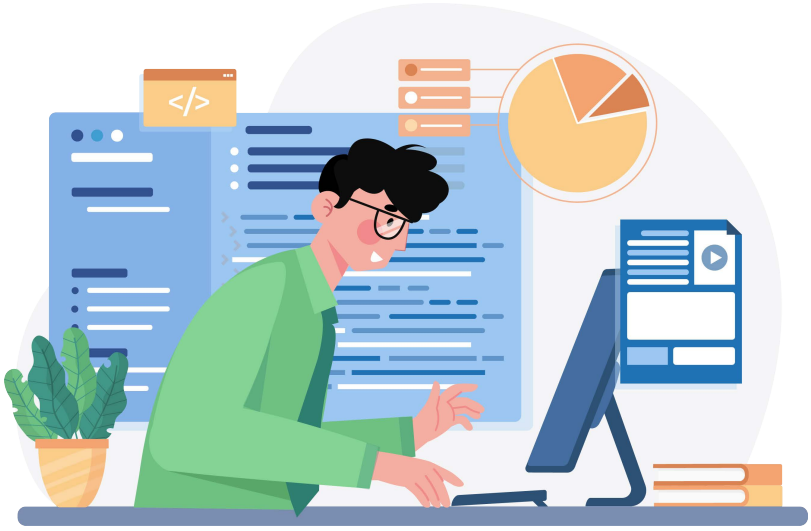
Testing this feature as it stands shows that it works well, but on the initial load of the app the messages grid is empty - this is because by default no label is selected on app load.

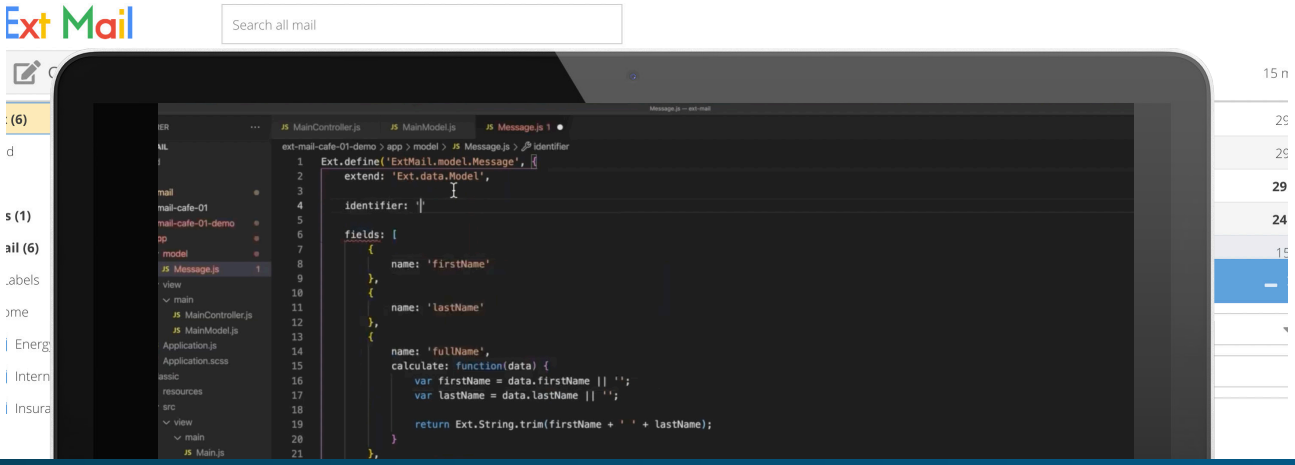
To fix this we add another binding to the Label store's first record, using the `.first` syntax. When this binding changes we know the store is loaded. When this happens we will set the `selectedLabel` property to the first record. This will then trigger the filtering and start the app off with a label selected.

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

    // select the first label (Inbox) on load
    this.bind('{labels.first}', this.onFirstLabelRecordChange, this);
    ...
},
```

```
onFirstLabelRecordChange: function(firstLabelRecord) {
    this.set('selectedLabel', firstLabelRecord);
},
...
```





Explore How to Create the Labels Tree

Watch this Step!

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

Adding Label Unread Count

The final feature we will add is to show the number of unread messages within each Label.

To do this we want to calculate each label's `unreadCount` whenever the Messages store's data changes. Essentially when the `unread` flag of a record has been changed we need to recalculate all of the unread counts.

In addition to unread counts, we want to display the number of drafts in the draft label so we will refer to drafts as unread items in this case.

We start by binding to the Message store's `datachanged` event in the MainModel.js.

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

    // recalculate the unread count on each label after the Messages store
changes
    this.getStore('messages').on('datachanged', this.calculateUnreadCounts,
this);

    ...
},
...
```

> The `calculateUnreadCounts` function goes through the following logic:
> - loop through all of the Labels
> - For each label find all of the Messages that:
>   - Has the current Label Id in its collection
>   - Has the `unread` field set to `true`

> OR
> - it has the DRAFT label
> - Has the `draft` field set to `true`

```
calculateUnreadCounts: function() {
    this.getStore('labels').each(function(labelRecord) {
        // count the unread messages in each Label, OR the number of messages
in DRAFT status
        var count = this.getStore('messages').queryBy(function(messageRecord) {
            return (
                messageRecord.hasLabel(labelRecord.getId()) &&
                messageRecord.get('unread')
            ) || (
                messageRecord.hasLabel(labelRecord.getId()) &&
                messageRecord.get('draft')
            );
        }).getCount();

        labelRecord.set('unreadCount', count);
    }, this);
}
...
```

With this event handler in place the `unreadCount` on the labels will always be in sync with the contents of the Message store. Now we can render that value into the LabelTree's column.

Like we said earlier the tree's columns are just like a grid's so we can give the name column a renderer to customise the markup that is output for each of the tree nodes.

```
{
    xtype: 'treecolumn',
    header: false,
    text: 'Name',
    dataIndex: 'name',
    flex: 1,
    renderer: function(value, meta, record) {
        var hasUnread = record.get('unreadCount') > 0;

        meta.tdStyle = 'font-weight: ' + (hasUnread ? 'bold' : 'normal');

        var unreadTpl = Ext.String.format('<span> ({0})</span>', record.
get('unreadCount'));

        return Ext.String.format('<span>{0}</span>{1}', value, hasUnread ?
unreadTpl : '');
    }}
```

First we determine whether this label has any unread messages, simply by checking the `unreadCount` field. We then use this to add bold styling to the wrapping `<td>` element if we have at least 1 unread message.

We will then set up a basic HTML string that will have the unread count in the parenthesis, wrapped in a `<span>`.

The final line will then output the label's name and, if there are unread messages, append the unread HTML on the end.

Learn How to Add Label Unread Count

Watch this Stage!

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

SUMMARY

That concludes the second part of the series where we have:

- Added row actions to the Messages grid using an `actioncolumn`

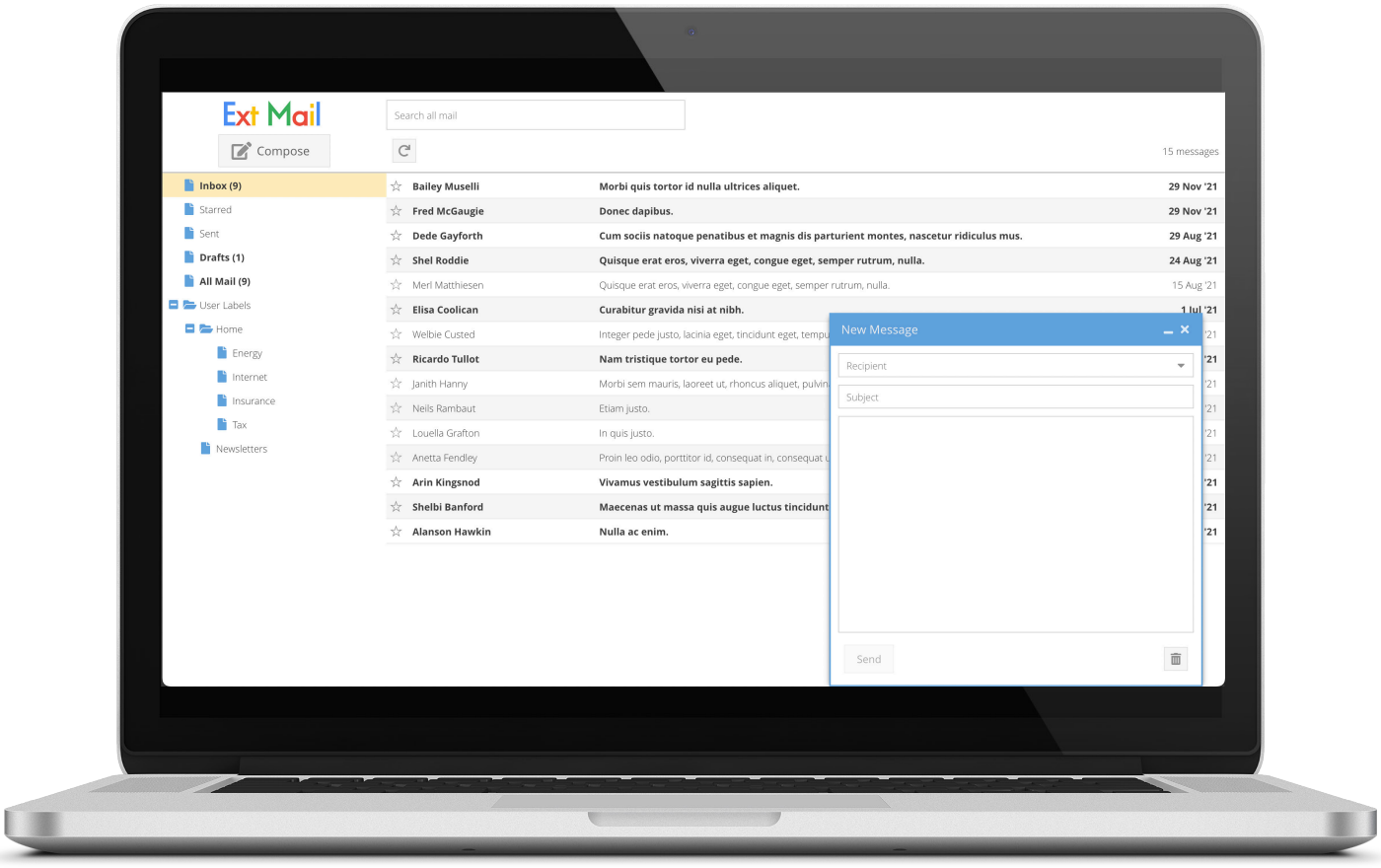
- Added additional Message actions

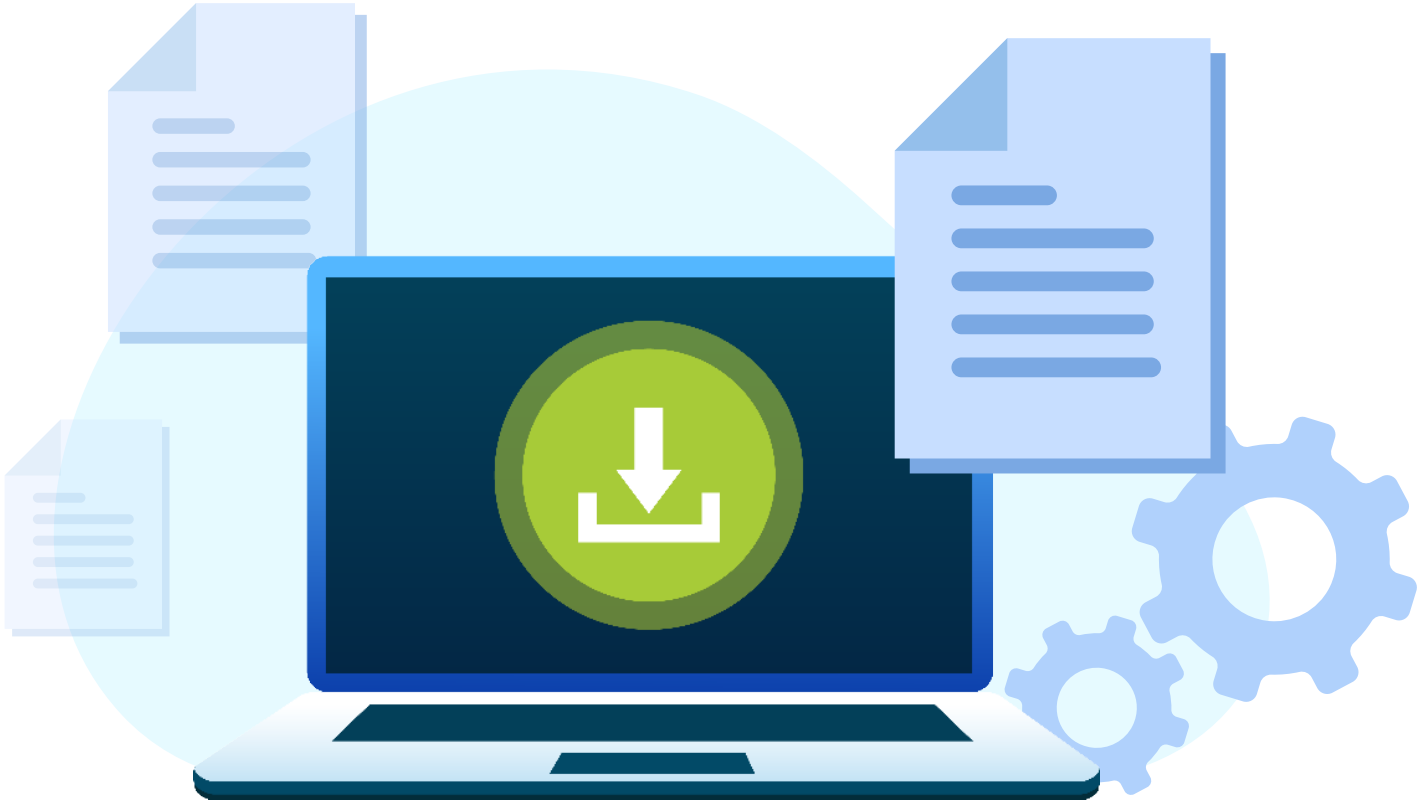
- Created template components bound directly to store properties

- Added a Label Tree component and used it to filter the Messages grid

- Finally, we calculated each Label's unread Message count and customized the Label Tree to display it through a custom renderer.

We've touched on a number of different components and solved some somewhat challenging problems in this article. We will continue the implementation of our email client in the next part, where we will add a Compose feature for sending new emails.





Thank you for reading!

Part-2 of Building an Email Client from Scratch

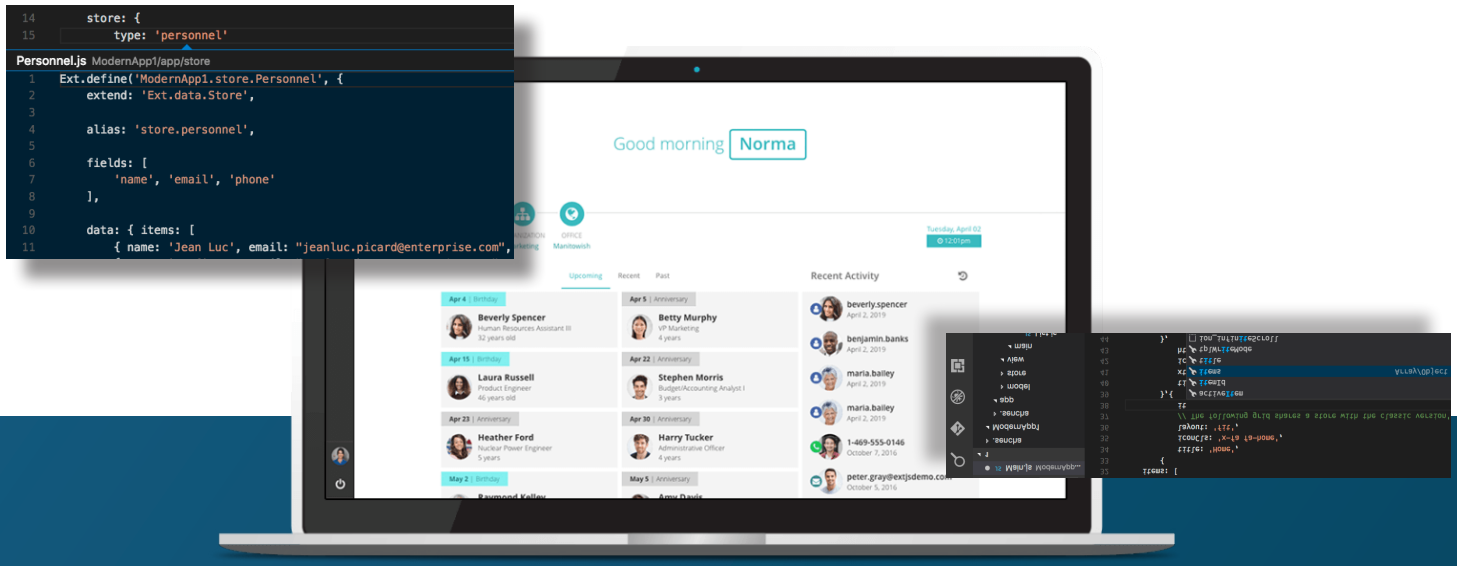
We hope you found it informative and helpful in your development projects. We have 5 more parts lined up to take you through the entire process of building an email client from scratch.

Download the Part-3 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