Sencha

Building an Email Client from Scratch

PART 7

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 sees us add deep linking to our Email Client application using Ext JS router classes. We will add 3 routes to the application and refactor our app to use them and make our application shareable and preserve state across refreshes.

Key Concepts / Learning Points

- Working with Ext JS router classes
- Refactoring our application to work with the new workflow using routes
- Added `before` guards to our routes to ensure our app has data in it before we proceed.
- Using named routes so we can have multiple active routes at any given time

Code along with Stuart!

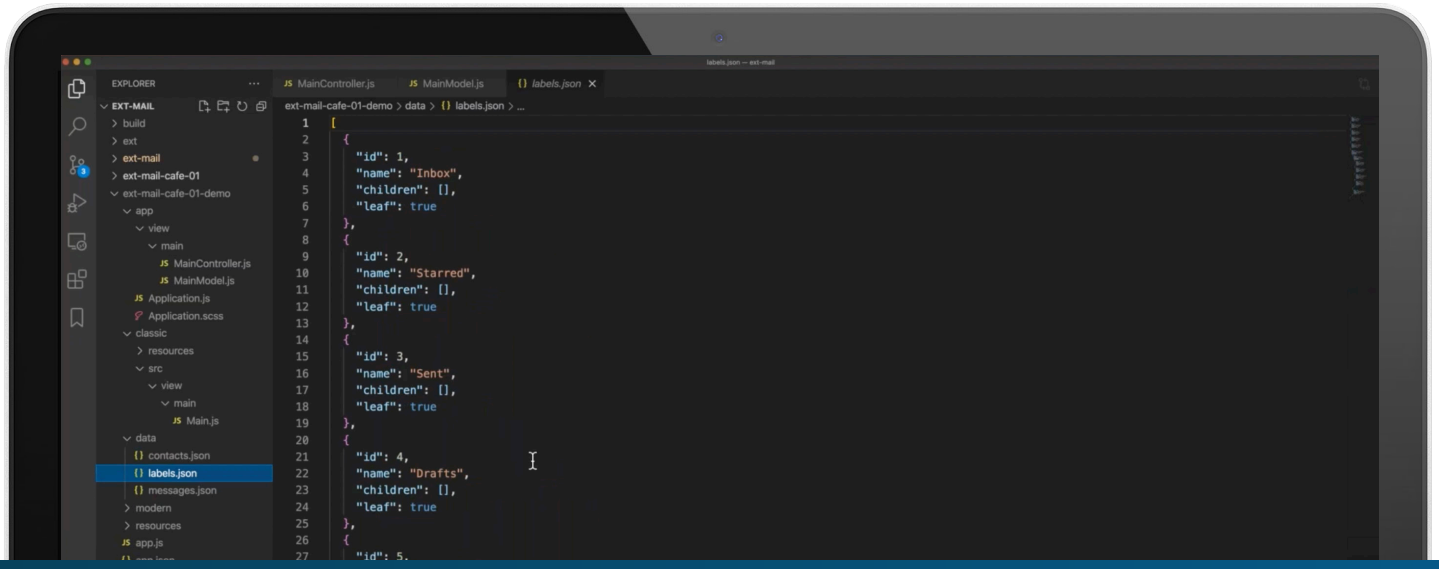
Start buddy coding with Stuart on-demand!

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

Introduction

Up until this point, we've built our Email Client application under a single URL that doesn't change throughout the entire lifecycle of the application. This means that refreshing the browser resets the app, losing the user's place in the application.

In this article we will be adding Deep Linking support to the application so state is preserved, URLs are shareable and the back button works as expected.



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.

What is Deep Linking

Deep Linking refers to users being able to navigate directly to a section of your application via the URL.

With traditional server-side rendered applications each section of the application would have its own distinct URL so navigating directly to a particular section was trivial, for example, to go directly to the Users section you might go to www.myapp.com/users.

However, modern web applications are often designed as Single-Page Applications (SPAs) so the user never leaves the root page when moving through the application and so each section doesn't have a unique URL to navigate to.

With this pattern, we break the traditional concept of linking which is an integral part of the web. It means that URLs can't be shared and refreshing the browser will reset the application to its base state, rather than taking the user back to where they were.

It also breaks the browser's Back button functionality which is something most users rely on and have a mental model of what to expect when clicking it. For example, if we navigate to www.myapp.com from a Google search result screen, navigate to a section of the app and then click the browser's back button, we would get taken back to Google rather than back to the starting page of the app. This is counterintuitive and can result in lots of frustration for users.

We will talk about bringing the concept of deep linking back to our single-page Ext JS applications to make it easy and convenient for users to go directly to sections of our apps, and navigate our web app as they do in other parts of the web.

Benefits of Deep Linking

The benefits of enabling deep linking in our applications are clear:

- It maintains the web's standard navigation concept, which users know and comes to expect.
- It keeps the back button working as expected.
- It allows specific sections of the app to be shared and bookmarked.

Explore What is Deep Linking

Watch this Chapter!

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

Deep Linking & Ext JS

Ext JS has a set of classes built into the framework that allows us to implement our own routing within our application so deep linking is possible.

These classes fall under the `Ext.route` namespace and together they let us handle and make URL changes as we move through our application.

The routing system doesn't change the full URL but instead manipulates the URL's hash value, whose changes are reacted to by our application, triggering changes to the apps state.

For example, the URL for viewing a User with an ID of 123 might look like this: www.myapp.com#user/view/123

Explore Deep Linking & Ext JS

Watch this Step!

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

Features of Ext JS Router

Ext JS' Router has a few key features that we will make use of in our demonstration application, namely:

- We can define multiple parameters in each Route, for example, `user/:action/:id` would parameterize the action and the id within the URL.
- It allows for optional parameters.
- We can validate the parameter values so we match the routes precisely. For example, we might want an ID parameter to only be numeric and reject it if it doesn't follow this pattern.
- With the nature of SPAs we might have multiple sections open in windows at a single time. The Router supports multiple active routes at the same time to handle this.
- We can supply routes with 'before' guards to check the state of the application before applying the route. This can be used for things like checking session state or if certain data stores have been loaded yet.

Demonstration of Deep Linking in Email Client Application

We will be demonstrating adding deep linking to our example Email Client application which we've built over the course of this series. We will be adding 3 routes to the application to handle navigation between labels, the viewing of a message, and the composing of a new message.

These routes could be active at the same time and will allow the application to be returned to the current state on refresh and navigated using the back button.

Discover Deep Linking in Email Client Application

Watch This Part!

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

Adding Label Slug Field

Before we can add the first route we need to give each label a URL-friendly name since the current names could contain mixed case, spaces, and special characters.

We do this by adding a new calculated field to the Label model which will take the label's name and normalise it into a slug that can be used in a URL.

```
...
{
    name: 'slug',
    calculate: function(data) {
        return (data.name || '').toLowerCase()
                                .replace(/ /g, '-')
                                .replace(/[^\w-]+/g, '');
    }
}
...
```

It will convert the name to lowercase, replace space with hyphens and removes any special characters (except hyphens).

We can now use this slug when referencing a Label in a URL.

Discover How to Add a Label Slug Field

Watch This Phase!

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

Adding Label Viewing Route

Generally, we add our route definitions to View Controllers, which is what we will do here. We will add a `routes` config to the `MainControllerBase` class and create a definition that will be triggered to show a particular label.

```
...
routes: {
    'label/:label': {
        name: 'label',
        before: 'onBeforeViewLabel',
        action: 'onViewLabel'
    }
},
...
```

The key for the route is the pattern that is used to match the URL and trigger the route to run. We can add as many parameters as we like using the ":" prefix to the parameter name - these values will then be passed to the `before` and `action` handlers.

In this case, we want to match the values such as "label/all-mail" and "label/starred" where we use the second part to find and select the correct item in the tree, and subsequently filter the messages list.

As we mentioned before we can have multiple active routes at the same time and this is why we provide a `name` config which allows us to add, update or remove this particular route while leaving the others untouched.

The `before` config allows us to provide a function that is called before the `action` and lets us do any checks or setups before the route is fulfilled. This function has the ability to block the route from running if certain criteria aren't met - such as the user being logged out or not having permissions.

In our case we use it to make sure we have the correct data loaded before proceeding.

```
...
onBeforeViewLabel: function(label, action) {
    var labelsStore = this.getViewModel().getStore('labels');

    if (labelsStore.loadCount > 0) {
        action.resume();
    } else {
        labelsStore.on('load', function() {
            action.resume();
        }, this, { single: true });
    }
}
...
```

The `before` method has two parameters - the first is the parameter we defined in the route, and the second is an `Ext.route.Action` instance. If multiple parameters have been defined then the first parameter would be an object containing each of the parameters with their name as the key.

Our `onBeforeViewLabel` method checks to see if the `labels` store has been loaded already. If it has, then we continue the route's process by calling the action's `resume` function. If it hasn't then we attach a handler to the store's `load` event and only when it does load do we continue with the route.

The route's `action` config is the method that performs the work to update the interface according to the route's parameters. In this case, we want the UI to select the defined label.

```
...
onViewLabel: function(label) {
    var labelsStore = this.getViewModel().getStore('labels');
    var labelRecord = labelsStore.findRecord('slug', label);

    if (!labelRecord) {
        labelRecord = labelsStore.first();
    }

    this.getViewModel().set('selectedMessage', null);
    this.getViewModel().set('selectedLabel', labelRecord);
}
...
```

We first have to find the corresponding Label record from the `labels` store (which we have verified is loaded) based on the slug passed in. If we don't find a match then we just select the first one.

Next, we set the `selectedMessage` property to null so that we navigate back to the Messages list if we happened to be viewing a message. Then we assign the Label to the `selectedLabel` property. This will trigger the UI to change accordingly.

This setup works for when you load the application with a label in the URL but it won't update the URL as the user navigates to different labels by clicking the tree. To do this we have to listen for the Label Trees `selectionchange` event and update the URL at that point.

We add an `onLabelSelectionChange` method to the MainControllerBase which will get the `slug` from the selected record and redirect the browser to the right URL.

```
...
onLabelSelectionChange: function(labelTree, selectedLabelRecords) {
    var selectedLabelRecord = selectedLabelRecords[0]; // always use the first
one
    var slug = '';

    // grab the 'slug' for the label
    if (selectedLabelRecord) {
        slug = selectedLabelRecord.get('slug');
    }

    this.redirectTo({
        label: Ext.String.format('label/{0}', slug), // redirect to the found
label
        message: null // navigate away from a Message View route if we have
one, so we go back to the list view
    });
}
...
```

We get the selected label's `slug` value and use the View Controller's `redirectTo` method to route to a new URL. This method can take an object whose keys correspond to the `name` we gave to the route in the initial setup.

Here we want to change the `label` route to "label/" plus the selected Label's slug.

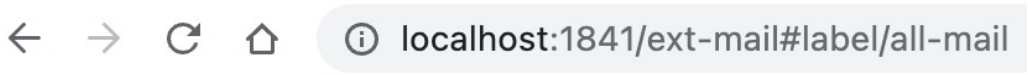
We also want to remove any `message` route value (since we want to move back to the Message list). We do this by setting its route to null.

> We can call the `redirectTo` method and pass a simple string with the new route we want to move to (for example, `this.redirectTo('label/all-mail');`) but this would clear any other routes that are active at the time.

Finally we hook this handler up to the event LabelsTree's configuration code.

```
...
{
    xtype: 'labels-LabelsTree',
    region: 'west',
    width: 300,
    bind: {
        store: '{labels}',
        selection: '{selectedLabel}'
    },
    listeners: {
        compose: 'onCompose',
        selectionchange: 'onLabelSelectionChange'
    }
}
...
```

With all of this code in place we can see the URL change as we navigate through the labels in the application.



Explore How to Add Label Viewing Route

Watch this Step!

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

Adding Message Viewing Route

Next we will follow a similar pattern to add a route that will handle the viewing of a particular message.

We first add the new route definition to the `routes` config in the MainControllerBase.

```
...
routes: {
    ...
    'view/:messageId': {
        name: 'message',
        before: 'onBeforeViewMessage',
        action: 'onViewMessage'
    },
    ...
}
...
```

In this route we look for the string “view/123” where the “123” would be the ID of a Message record in our `messages` store.

Again, we give it a name so we can modify it while maintaining any other existing routes, a `before` hook and an `action` method.

Our `before` hook is identical to the Labels route where we check that the Messages store has been loaded before continuing with the route.

```
...
onBeforeViewMessage: function(messageId, action) {
    var store = this.getViewModel().getStore('messages');

    if (store.loadCount > 0) {
        action.resume();
    } else {
        store.on('load', function() {
            action.resume();
        }, this, { single: true });
    }
}
...
```

The `onViewMessage` function then looks up the message ID in the `messages` store and sets it as the `selectedMessage` property in the ViewModel.

If the Message record was not found then we clear the `message` route by calling the `redirectTo` method and setting it to `null`.

```
...
onViewMessage: function(messageId) {
    var store = this.getViewModel().getStore('messages');
    var messageRecord = store.getById(messageId);

    // if messageRecord is null then we reset it anyway
    this.getViewModel().set('selectedMessage', messageRecord);

    // if we didn't find a message record we reset the route
    if (!messageRecord) {
        this.redirectTo({
            message: null
        });
    }
}
...
```

The last piece of the puzzle is to ensure that when a user interacts with the UI in order to view a Message (i.e. clicks a row in the Messages grid) instead of setting the `selectedMessage` property with the clicked item (like we do just now) but instead trigger the URL to change and let the routing configuration pick it up and perform the UI transition.

This shift in the process is likely where you will need to alter your mindset and possibly refactor areas of your application. We have to ensure that the routing handles all the changes in UI and that the UI interactions simply trigger a route change (i.e. a URL redirect).

For the Message viewing, we must update the `handleMessageClick` method so it calls the `redirectTo` method instead of setting the `selectedMessage` in the ViewModel. This logic is now handled in the `onViewMessage` route handler function as we saw above.

```
...
handleMessageClick: function(messageRecord) {
    if (messageRecord.get('draft')) {
        this.showComposeWindow(messageRecord);
    } else {
        this.redirectTo({
            message: Ext.String.format('view/{0}', messageRecord.getId())
        });
    }
}
...
```

Explore How to Add Message Viewing Route

Watch this Stage!

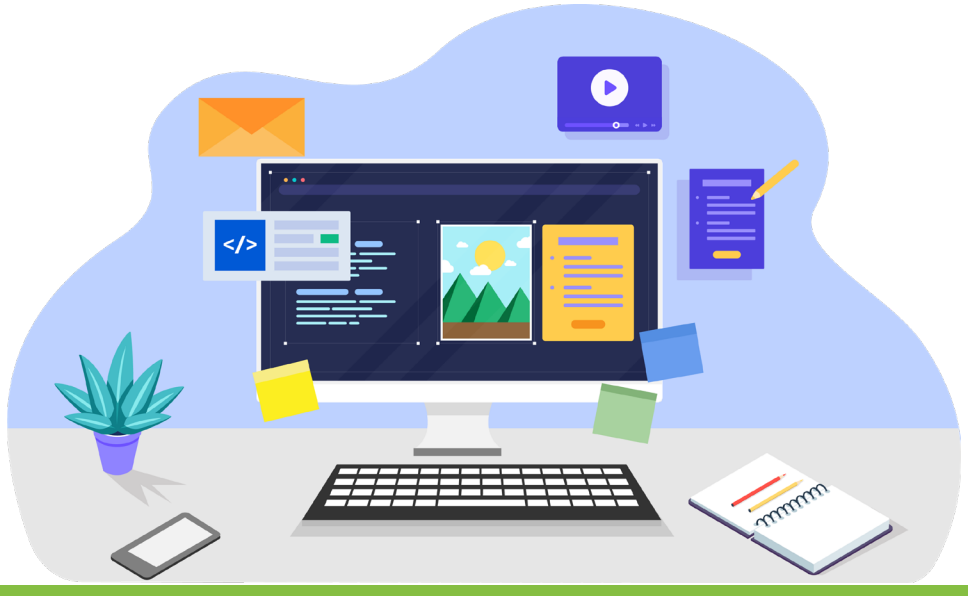
Use the bite-sized video links in this e-book to instantly watch the section you are reading.

Adding Message Compose Route

The third route we will add is one to handle the opening of the message compose form. For this route we use the pattern “draft/<messageId>” - we don’t need a `before` guard so we just define an `action` property and a name.

```
...
routes: {
    'draft/:messageId': {
        name: 'draft',
        action: 'onDraftMessage'
    }
}
...
```

The `onDraftMessage` handler simply grabs the Message record from the store and calls the `showDraftWindow` function that we already have. It will also clear the `draft` route if the Message record isn’t found.



```
...
onDraftMessage: function(messageId) {
    var messageRecord = this.getViewModel().getStore('messages').
getById(messageId);

    if (!messageRecord) {
        this.redirectTo({
            draft: null
        });
    } else {
        this.showDraftWindow(messageRecord);
    }
}
...
```

Once again we need to update the code which currently triggers the Compose Window so that it triggers the URL change instead of changing the UI directly.

In this case, we update the `onComposeMessage` method in the MainControllerBase class and replace the call to `showDraftWindow` with a call to `redirectTo`.

```
...
onComposeMessage: function() {
    var messageRecord = Ext.create('ExtMail.model.Message', {
        labels: [ ExtMail.enums.Labels.DRAFTS ],
        outgoing: true,
        draft: true
    });

    messageRecord.addLabel(ExtMail.enums.Labels.DRAFTS);

    this.getViewModel().getStore('messages').add(messageRecord);
    this.getViewModel().getStore('messages').commitChanges(); // commit changes
immediately since we aren't persisting to backend
```

```
        this.redirectTo({
            draft: Ext.String.format('draft/{0}', messageRecord.getId())
        });
    },
    ...
```

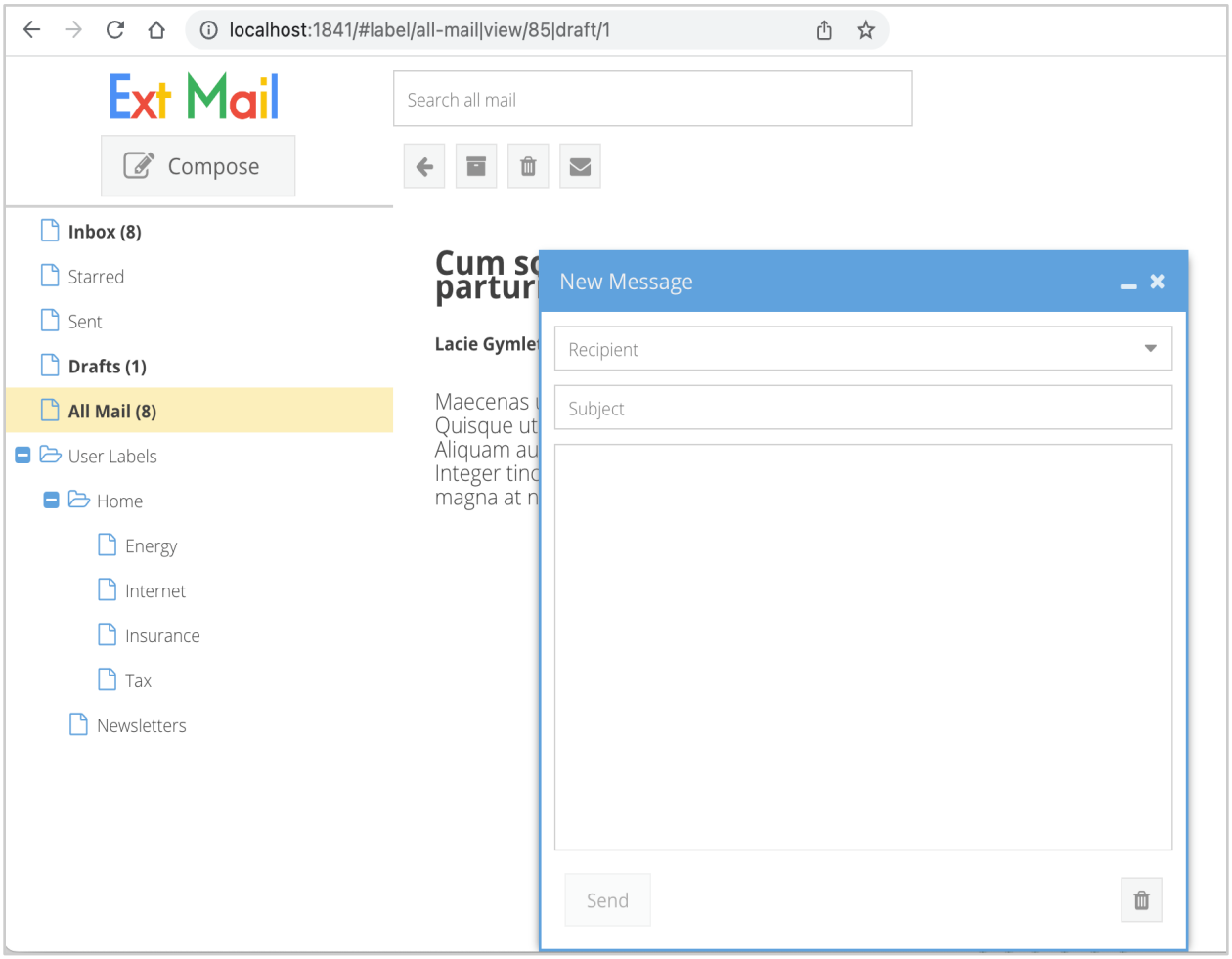
We must also do the same in the code that handles a user clicking on a draft in the Messages list. So the `handleMessageClick` function can be refactored to look like this:

```
    ...
    handleMessageClick: function(messageRecord) {
        var destination = {};

        if (messageRecord.get('draft')) {
            destination = {
                draft: Ext.String.format('draft/{0}', messageRecord.getId())
            };
        } else {
            destination = {
                message: Ext.String.format('view/{0}', messageRecord.getId())
            };
        }

        this.redirectTo(destination);
    }
    ...
```

With the 3 routes set up now, we can see them in action within our application and even how all 3 can be active at the same time as in the screenshot below.

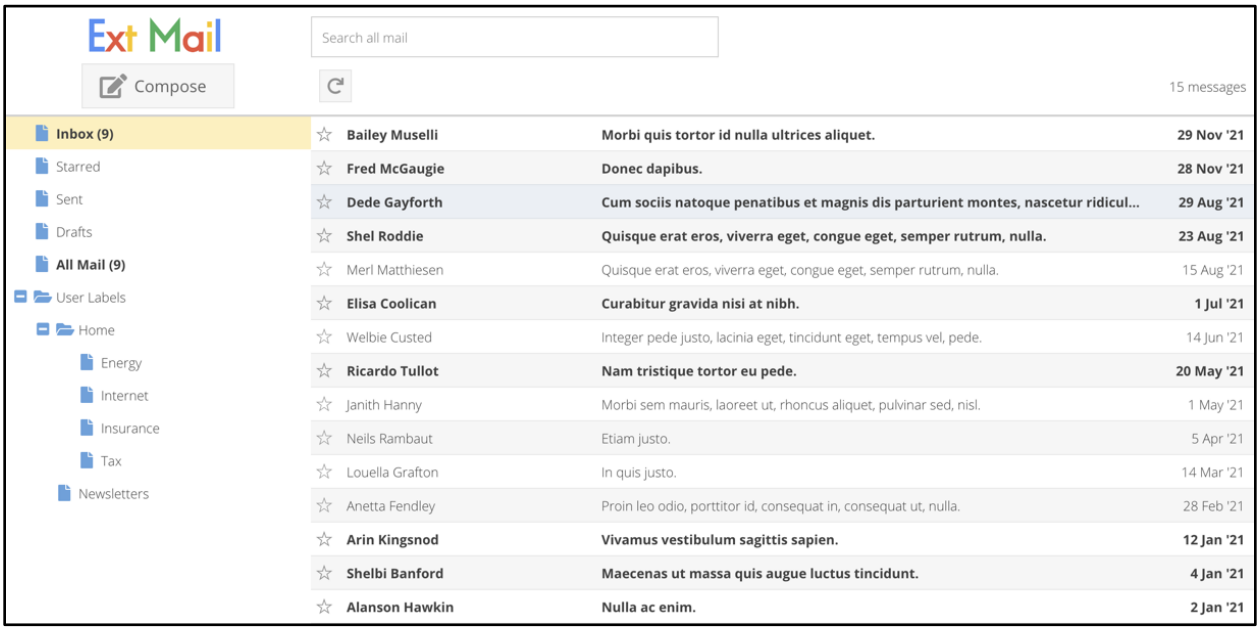


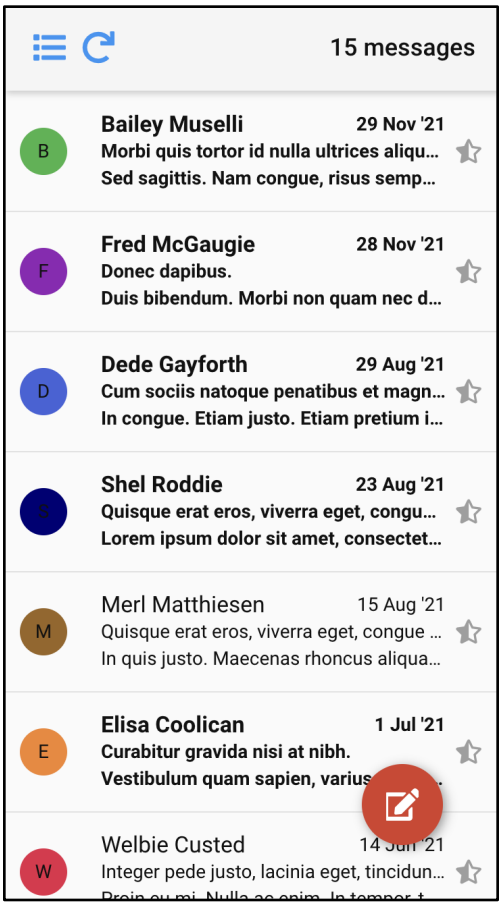
Learn How to Add Message Compose Route

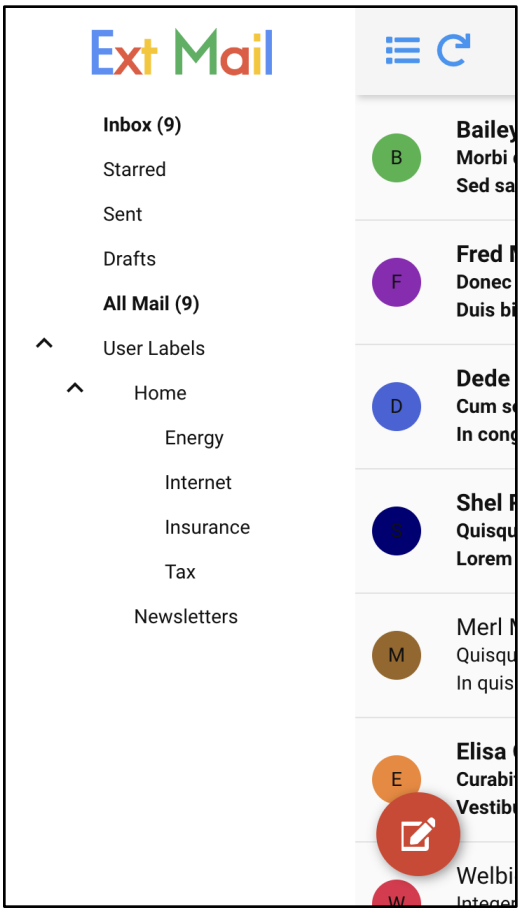
Watch this Part!

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

Ext Mail Application Home Screen







SUMMARY

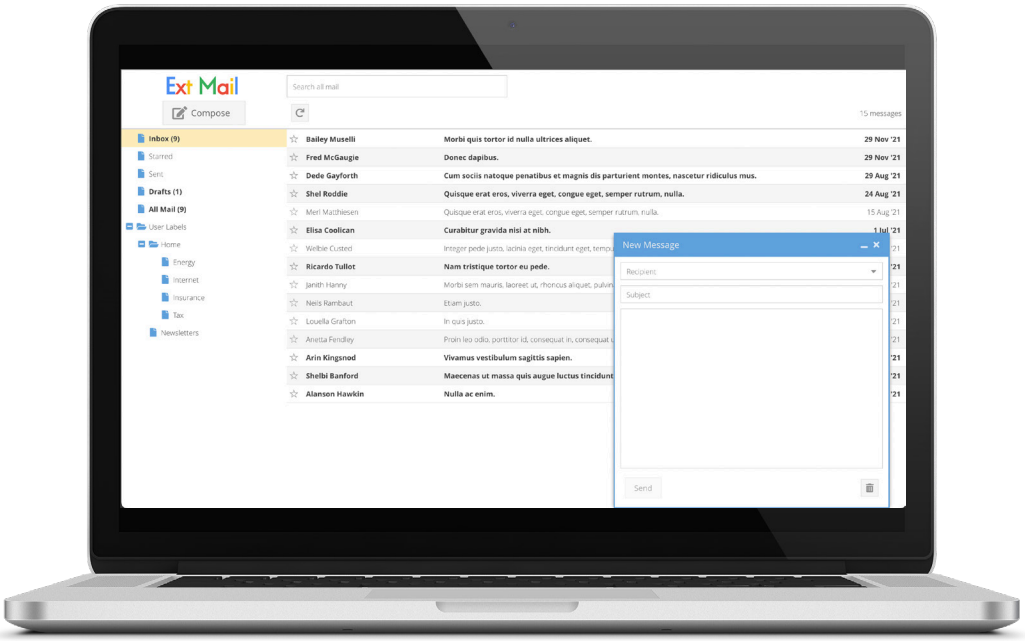
We have now added support for deep linking to our application with 3 routes supporting viewing labels, viewing messages, and viewing the draft window.

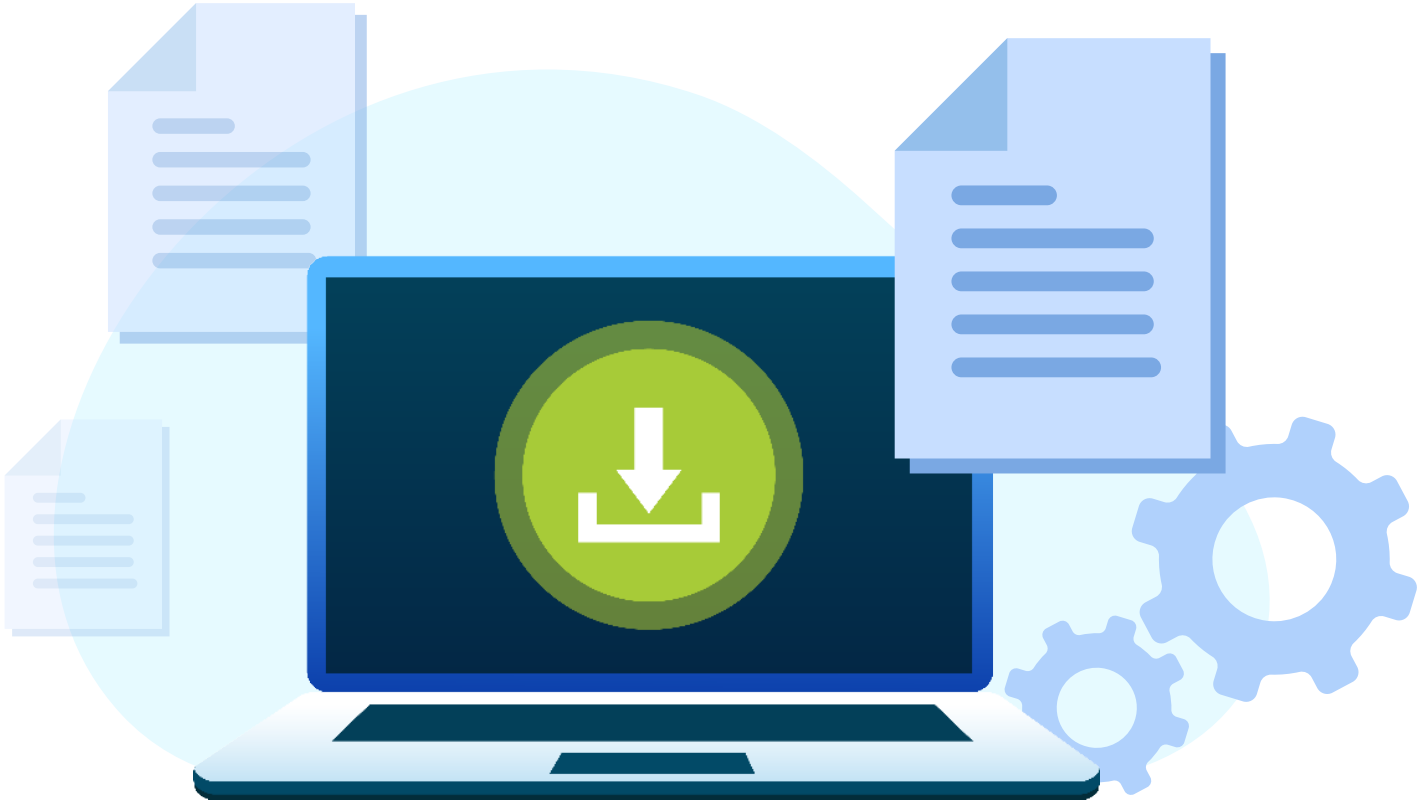
We coded them in a way that multiple routes are supported at once giving us fully shareable URLs and making the app's state persistent across page refreshes.

We also refactored our code so all UI navigation is handled by the routes and any user interactions result in a route change, a tricky mindset to get into and sometimes hard to implement after the fact.

Thank you for reading **Part 7** of **Building an Email Client from Scratch.** We hope that you found this series informative and helpful in your own email client development project.

If you have any questions or feedback, please don't hesitate to reach out to us. We're always happy to help and support you in your development journey. And if you haven't already, why not give it a try and start building your own email client today? With the knowledge and insights gained from this e-Book series, you'll be well on your way to creating a successful and robust email client that meets the needs of your users.





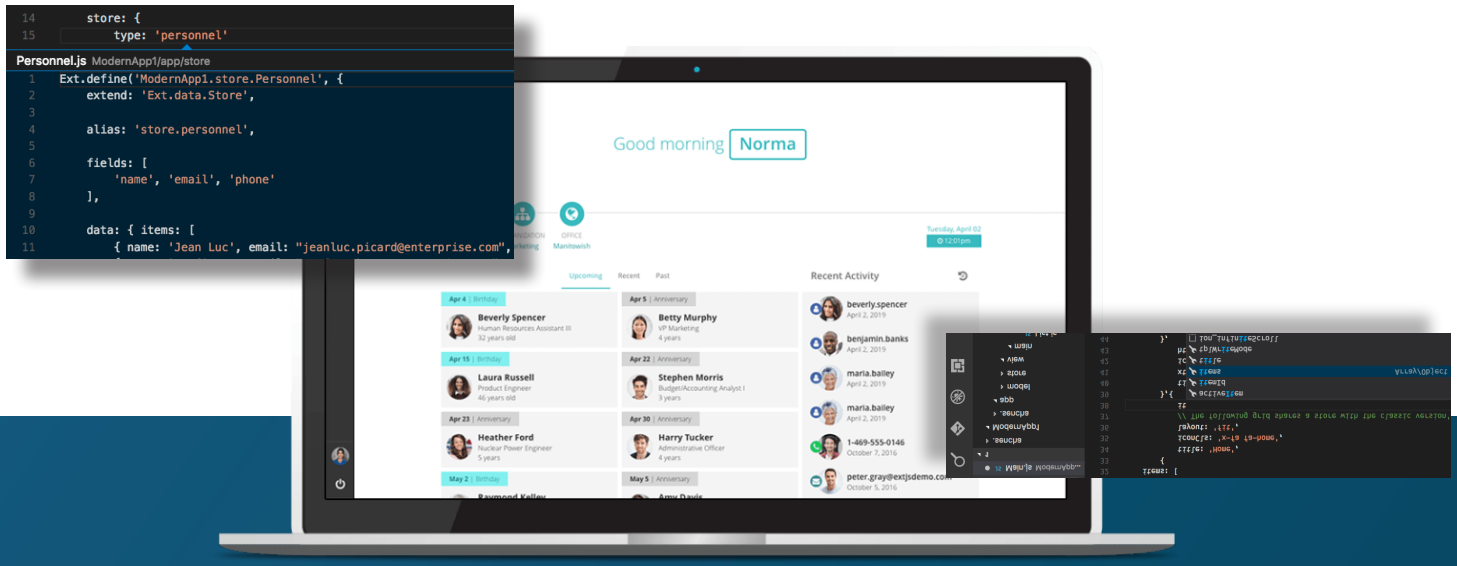
Thank you for reading!

Part-7 of Building an Email Client from Scratch

We hope you found this series informative and helpful in your development projects.

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