Sencha

Building an Email Client from Scratch

PART 5

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

We continue implementing our Modern toolkit interface for our email client, bringing feature parity between it and the Classic toolkit's application.

We will create a tree component and add it to a native-like sliding menu, allowing users to switch message categories. We will then create the Compose functionality with a floating compose button and a Modern form.

All of these features will utilize the shared business logic and data models that we already have in place.

Key Concepts / Learning Points

- Creating native-like sliding menusCreating Modern grid components
- Creating Tree components
- Creating Modern forms
- Overriding and expanding base class functionality

Code along with Stuart!

Start buddy coding with Stuart on-demand!

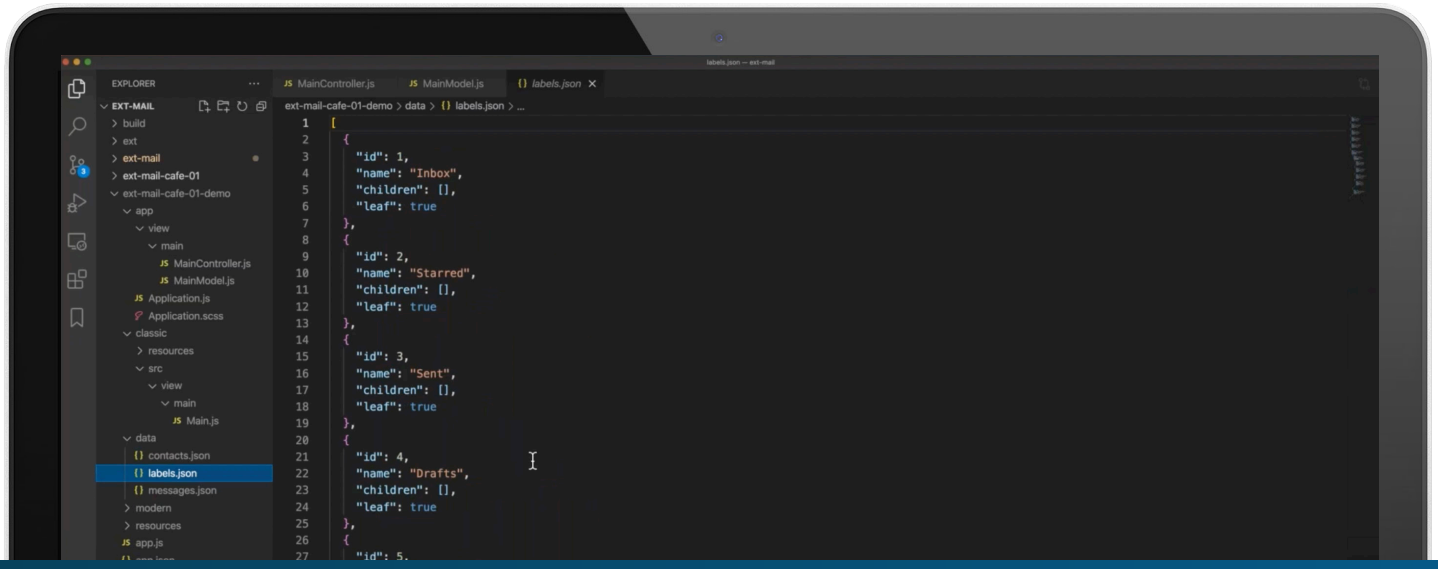
Use the bite-sized video links in this e-book to instantly watch the section you are reading.

Introduction

In this article, we will be completing our implementation of the Modern toolkit interface for our email client by adding a slide-out menu allowing users to switch between message categories.

We will then add Message composition to the application using a floating button and a new compose form.

These two features will make use of the shared business logic we already have in place and demonstrate just how powerful universal applications can be.



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 the Labels Tree

The first thing we want to create is a tree component that will display all of the Message labels/ categories that we have and allow users to select one to show all of the messages that fall in that category.

We will mirror the structure that the Classic toolkit has for the Labels Tree component and create a new file called LabelsTree.js in the `view/labels` folder. The class will be called `ExtMail.view.labels.LabelsTree` and will extend the `Ext.list.Tree` component.

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

    defaults: {
        xtype: 'treelistitem',
        textProperty: 'combined'
    }
});
```

We tell the component what `xtype` to use when creating an item in the tree - in this case the based `treelistitem` component and configure which field of the data model to display for each item. We make use of the `combined` field we created previously which displays the label name and the number of unread messages within it. This combined field also wraps it in an HTML element with an `unread` CSS class so we can apply bold styling to it.

With that in mind, we must create a corresponding SASS file for the component where we will put this custom styling. We create a file named `LabelsTree.scss` as a sibling to the JS file and add this style rule.

```
.unread {
    font-weight: bold;
}
```

That is all the code we need to create a fully functioning tree component that will mirror that of the Classic Toolkit app.

Learn How to Create the Labels Tree

Watch this Chapter!

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

Creating the Slide-out Menu Component

We want the LabelsTree to appear in a slide-out menu component that is commonplace in mobile applications. To do this we create a new component called `ExtMail.view.menu.Menu` which will be attached to the main application viewport using the built-in `menu` config.

The menu will be a simple `Ext.Panel` component with the LabelsTree component as its only child item, using a `fit` layout.

```
Ext.define('ExtMail.view.menu.Menu', {
    extend: 'Ext.panel.Panel',
    alias: 'widget.menu-Menu',

    requires: [
        'ExtMail.view.labels.LabelsTree'
    ],

    defaultListenerScope: true,
    scrollable: 'vertical',
    layout: 'fit',
    items: [
        {
            xtype: 'labels-LabelsTree',
            style: {
                background: 'white'
            },
            bind: {
                store: '{labels}',
                selection: '{selectedLabel}'
            },
            listeners: {
                selectionchange: 'onLabelSelectionChange'
            }
        }
    ],

    // when the selected label changes then we fire the `closemenu`
event
```

```
        onLabelSelectionChange: function() {
            this.fireEvent('closemenu');
        }
    });
```

We bind the LabelsTree to the `labels` store found in our shared `MainViewModel` and the selected label to the `selectedLabel` data property.

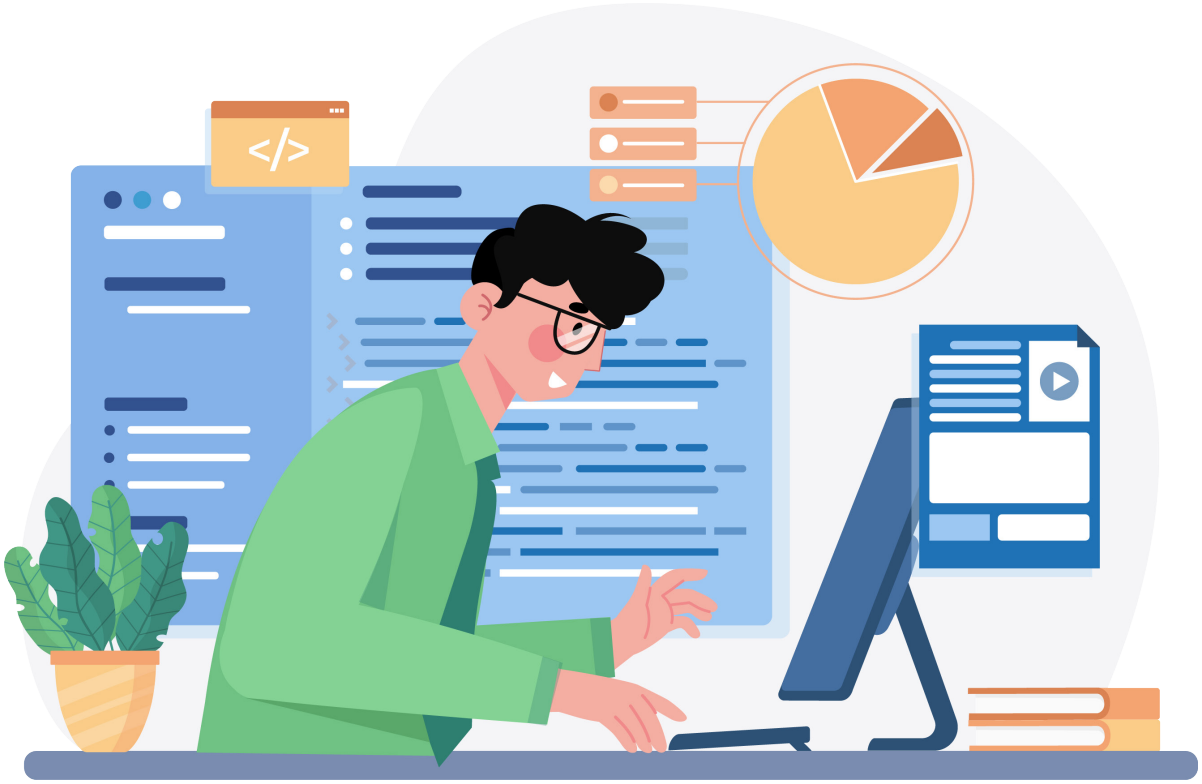
We also listen for the `selectionchange` event and bind it to the `onLabelSelectionChange` method which will fire the custom `closemenu` event that will allow us to ensure the menu is closed once a new label has been selected.

> Note that we set the `defaultListenerScope` to `true` so the handler is resolved to the Menu component, rather than to any attached View Controller.

The last thing we need to do to get this basic panel to behave as a menu is to add three getter functions that will define how the menu behaves - the `getReveal` method decides whether the menu is shown by the main component being moved out of the way, with the menu underneath. The `getCover` method defines whether the menu comes out and over the top of the main component. FInally, the `getSide` method decides from which side of the screen the menu sits.

In our case we want the menu to sit underneath the main component, on the left side, and be revealed as the main component moves to the right.

```
...
getReveal: function() {
    return true;
},
getCover: function() {
    return false;
},
getSide: function() {
    return 'left';
}
...
```



Learn How to Create the Slide-out Menu Component

Watch this Section!

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

Adding the Menu to the App

The menu will be configured and created as part of the App's launch process - in the `Application.js` file.

We start by creating a method called `setupMenu` which will create a new instance of our `ExtMail.view.menu.Menu` class and attach it to the Viewport instance using the `setMenu` method.

```
...
setupMenu: function() {
    Ext.Viewport.setMenu(Ext.create('ExtMail.view.menu.Menu', {
        width: 250,
        viewModel: this.getMainView().getViewModel(),
        listeners: {
            // close the menu when instructed
            closemenu: function() {
                Ext.Viewport.hideMenu('left');
            }
        }
    }));
}
...
```

We give the menu a fixed width of 250 pixels and pass in a reference to the Main View's ViewModel so it shares the data modelling code we already have in place. By attaching the Menu to the Viewport (as opposed to the Main view) it doesn't become a child of the Main view and so doesn't inherit access to its ViewModel. For this reason we must share its ViewModel instance manually.

Lastly, we listen to the custom `closemenu` event that we fire when a label is selected and tell the menu to hide. We pass in "left" so it knows which menu to hide since we could have other menus attached to the other sides of the viewport.

With the setup code in place we can execute the function during the app's launch process.

```
...
launch: function() {
    this.setupMenu();
},
...
```

Adding a Menu Button

The menu now exists on the app but we don't have a way to open it. We want to add a new button to our `MessagesToolbar` which will show and hide the menu. We do this by overriding the `buildItems` method that lives in the base class, so we can add the menu button to that array of items.

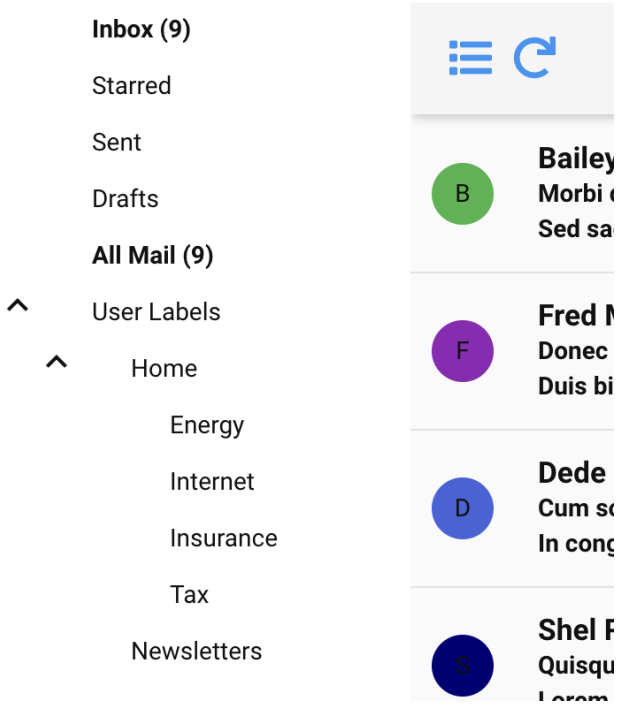
```
...
buildItems: function() {
    var items = this.callParent(arguments);

    // add menu button to start of toolbar
    items.unshift({
        xtype: 'button',
        iconCls: 'x-fa fa-list',
        bind: {
            hidden: '{selectedMessage}'
        },
        handler: this.onMenuButtonTap,
        scope: this
    });

    return items;
},

onMenuButtonTap: function() {
    if(Ext.Viewport.getMenus().left.isHidden()){
        Ext.Viewport.showMenu('left');
    } else {
        Ext.Viewport.hideMenu('left');
    }
}
```

First we call the base class' `buildItems` method using the `callParent` function. This gets us the output from the original function that we will modify with the Modern specific items.

From there we can add the menu button to the start of the array. We bind the `hidden` config to the `selectedMessage` data property so that the menu is only visible when we are on the list view.

Tapping on the button triggers the `onMenuButtonTap` function which hides the menu if it is already visible, or shows it if it isn't.

Now we can open and shut our menu from the toolbar.



Discover How to Add the Menu to the App

Watch this Part!

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

Adding the App Logo

To finish off the menu we will add our Ext Mail logo to the top of the menu as a docked component.

```
Ext.define('ExtMail.view.menu.Menu', {
    extend: 'Ext.panel.Panel',
    alias: 'widget.menu-Menu',

    ...

    items: [
        {
            xtype: 'component',
            docked: 'top',
            style: {
                textAlign: 'center',
            },
            html: '<img src="resources/images/ext-mail-logo.png" />',
        },
        {
            xtype: 'labels-LabelsTree',
            ...
        },
    ],
    ...

});
```

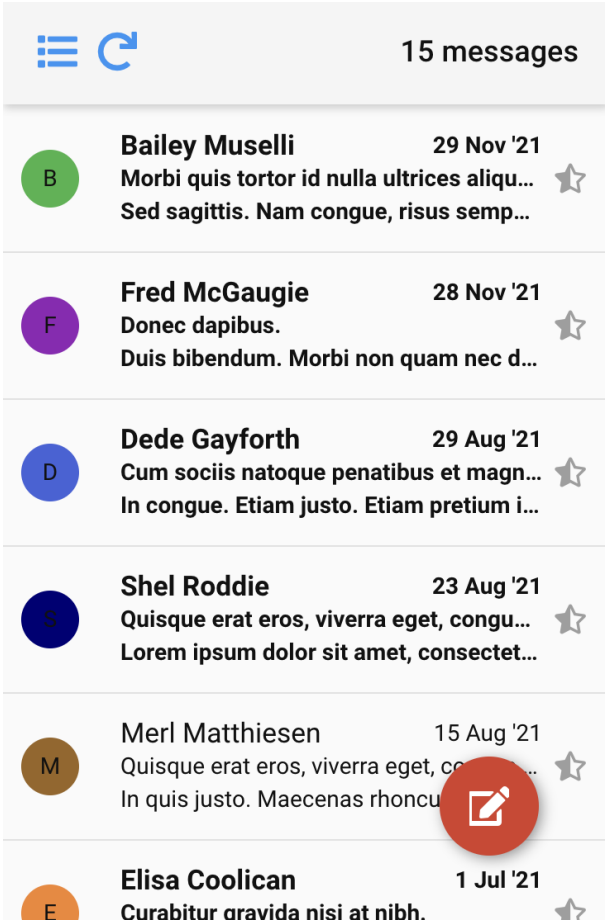
Explore How to Add the App Logo

Watch this Step!

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

Add a Floating Compose Button

To trigger our compose workflow we're going to use a floating button in the lower right of our screen which, when tapped, will open our compose form.



We start by creating a sub-class of the `Ext.Button` class in a new namespace folder called `compose`, called `ComposeButton`. We configure this with a width, height, icon, and a custom CSS class, along with the `floated: true` config which will tell the component to be absolutely positioned on top of our interface.

```
Ext.define('ExtMail.view.compose.ComposeButton', {
  extend: 'Ext.Button',

  alias: 'widget.compose-ComposeButton',

  floated: true,
  width: 60,
  height: 60,
  cls: 'compose-button',
  iconCls: 'x-fa fa-edit'
});
```

We also create a SASS file alongside it to give the button its appearance and positioning.

```
.compose-button.x-button {
  background: #d8372d;
  border-radius: 50%;
  position: absolute;
  right: 40px;
  bottom: 40px;

  .x-icon-el {
    color: #FFF;
  }
}
```

Just like with the Menu component we will initialize this button in the `Application.js` file with a `setupComposeButton` method.

```
setupComposeButton: function() {
    this.composeButton = Ext.create('ExtMail.view.compose.ComposeButton', {
        hidden: false,
        handler: function() {
            this.getMainView().getController().onComposeMessage();
        },
        scope: this
    });

    // hide the composeButton when viewing/composing a message, show when
viewing
    // message list
    this.getMainView().getViewModel().bind('{selectedMessage}',
function(selectedMessage) {
        // if we're animating to a message view screen then do composeButton
hide immediately,
        // otherwise we wait for the animation to complete and then toggle it.
        var setDelay = selectedMessage ? 0 :
this.getMainView().getLayout().getAnimation().getDuration();

        setTimeout(Ext.bind(function() {
            this.composeButton.setHidden(selectedMessage);
        }, this), setDelay);
    }, this);
}
```

In this method we create a new instance of the ComposeButton class and trigger the `onComposeMessage` method of the Main View's controller when it is clicked.

We also bind to the `selectedMessage` property of the MainViewModel and hide the button when moving from the list to the message details screen, and show it when moving back.

We perform this show/hide logic using a `setTimeout` so it happens after the animation when showing it, but immediately when hiding it.

We then call the `setupComposeButton` method during the `launch` process:

```
...
launch: function() {
    this.setupMenu();
    this.setupComposeButton();
},
...
```

Discover How to Add a Floating Compose Button

Watch this Stage!

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

Creating the Compose Form

The Compose form that we created for the Classic toolkit has some logic that can be shared with the Modern, namely the inline ViewModel, the button action handling and the recipient details population. Unfortunately, the form fields themselves have slightly different APIs and so can't be shared.

To enable this to be shared we need to create a base class which each toolkit's form will extend from. We will create this in the `app` folder and call it `ComposeFormBase.js`

```
Ext.define('ExtMail.view.compose.ComposeFormBase', {
    extend: 'Ext.form.Panel',

    viewModel: {
        data: {
            selectedRecipient: null,
        },
    },

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

        // when we select a recipient we need to extract the firstname,
lastname and email from the record and put it into the message
        this.getViewModel().bind(
            '{selectedRecipient}',
            this.onSelectedRecipientChange,
            this
        );
    },

    onSendClick: function () {
        this.fireEvent('send', this.getViewModel().get('messageRecord'));
    },

    onDiscardClick: function () {
        this.fireEvent(
            'discarddraft',
            this.getViewModel().get('messageRecord')
```

```
        );
    },

    onSelectedRecipientChange: function (selectedRecipientRecord) {
        var firstName, lastName, email;

        // if we have a recipient record then pull the properties from it
        if (selectedRecipientRecord) {
            firstName = selectedRecipientRecord.get('first_name');
            lastName = selectedRecipientRecord.get('last_name');
            email = selectedRecipientRecord.get('email');
        }

        // assign them to the messageRecord if we have one
        if (this.getViewModel().get('messageRecord')) {
            this.getViewModel().get('messageRecord').set({
                firstName: firstName,
                lastName: lastName,
                email: email,
            });
        }
    },
});
```

With this in place we can clean out the extracted code from the Classic toolkit's ComposeForm component and change its base class to the new one we created.

We can then create our Modern ComposeForm with the same Recipient combobox, subject field, and message text area. The action buttons are docked to the bottom of the form using the modern pattern.

```
Ext.define('ExtMail.view.compose.ComposeForm', {
    extend: 'ExtMail.view.compose.ComposeFormBase',
    alias: 'widget.compose-ComposeForm',

    defaultListenerScope: true, // makes string method names resolve to methods
on this component
    padding: 10,
    layout: {
        type: 'vbox',
        align: 'stretch'
    },
    items: [
        {
            xtype: 'combobox',
            placeholder: 'Recipient',
            displayField: 'email',
            valueField: 'email',
            queryMode: 'local',
            required: true,
            bind: {
                store: '{contacts}',
                selection: '{selectedRecipient}',
                value: '{messageRecord.email}'
            }
        },
        {
            xtype: 'textfield',
            placeholder: 'Subject',
            bind: {
                value: '{messageRecord.subject}'
            }
        },
        {
            xtype: 'textareafield',
            placeholder: 'Compose email',
            flex: 1,
            bind: {
                value: '{messageRecord.message}'
            }
        },
        {
            xtype: 'toolbar',
            docked: 'bottom',
```

```
            margin: 0,
            items: [
                {
                    xtype: 'button',
                    scale: 'medium',
                    text: 'Send',
                    handler: 'onSendClick'
                },
                '->',
                {
                    xtype: 'button',
                    iconCls: 'x-fa fa-trash',
                    tooltip: 'Discard',
                    handler: 'onDiscardClick'
                }
            ]
        }
    ],

    onSendClick: function() {
      if (this.validate()) {
        this.callParent(arguments);
      }
    }
});
```

We override the `onSendClick` method to prevent the form from being submitted if it isn't valid. This is handled in the Classic toolkit by the `formBind` config of the Send button but that isn't available in the Modern toolkit.

Since the buttons use the same methods as the Classic toolkit's equivalent they will emit the same custom events that can be handled by our MainController as needed.

Learn How to Create the Compose Form

Watch this Part!

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

Integrating the Compose Form

With the Compose Form ready, we can integrate it into our app using the same pattern as we did in the Classic toolkit.

We start by overriding the `showComposeWindow` method in our MainController class. Here we want to set the ViewModel's `selectedMessage` property with the passed in `messageRecord`.

```
...
showComposeWindow: function(messageRecord) {
    this.getViewModel().set('selectedMessage', messageRecord);
}
...
```

Next we can update the `onSelectedMessageChange` handler function to move to the ComposeForm if the selected message is a `draft`.

```
...
onSelectedMessageChange: function(selectedMessage) {
    if (selectedMessage && !selectedMessage.get('draft')) {
        this.getView().push(this.getMessageDetailsConfig(selectedMessage));
    } else if(selectedMessage && selectedMessage.get('draft')) {
        this.getView().push(this.getComposeMessageConfig(selectedMessage));
    } else {
        this.getView().pop();
    }
},
...
```

The `getComposeMessageConfig` returns a basic config for the form where it hooks up the `send` and `discarddraft` events to their corresponding MainControllerBase methods and sets up the form's ViewModel with the Message record that has been passed in.

```
...
getComposeMessageConfig: function(messageRecord) {
    return {
    xtype: 'compose-ComposeForm',
    viewModel: {
        data: {
        messageRecord: messageRecord
        }
    },
    listeners: {
        send: this.onSendMessage,
        discarddraft: this.onDiscardDraftMessage,
        scope: this
    }
    };
}
...
```

Again, this makes use of the shared functionality we have already in place and so we are just hooking up the modern interface components with the business logic that should be triggered by their events. Using custom events like this is a great way of normalizing the actions away from the interface elements themselves.

Learn How to Integrate the Compose Form

Watch this Phase!

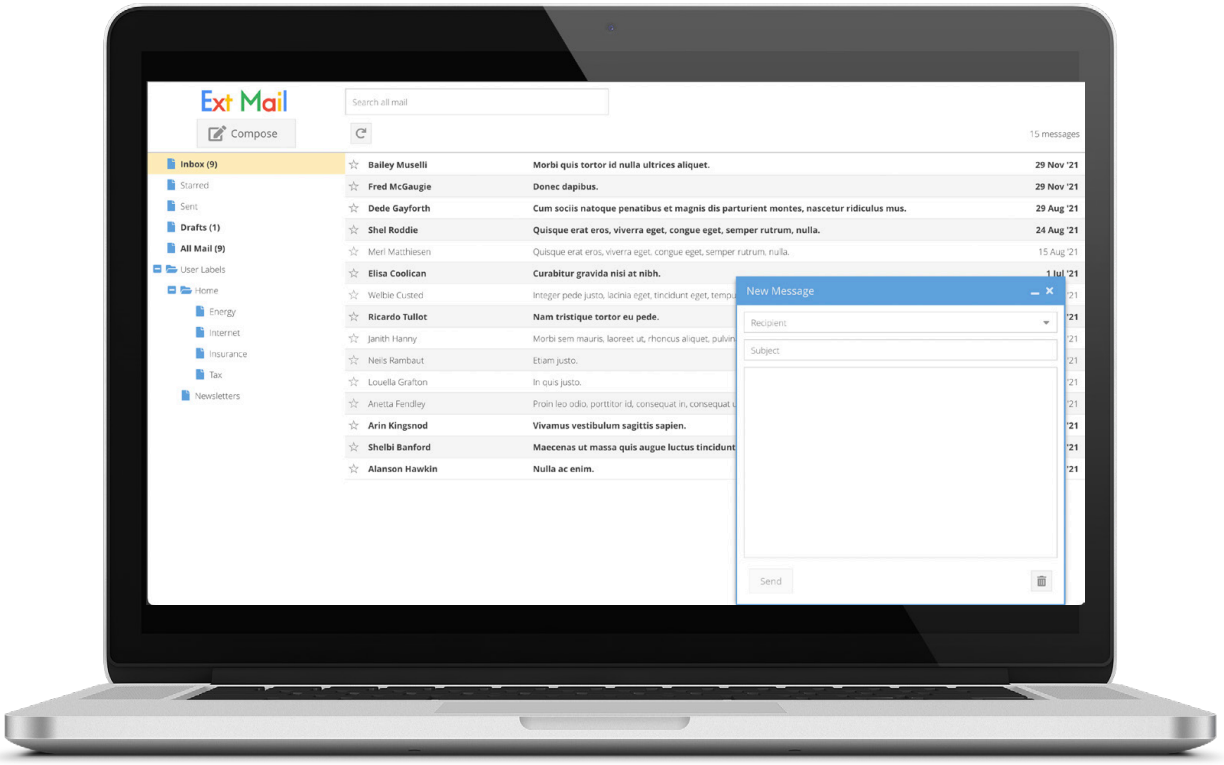
Use the bite-sized video links in this e-book to instantly watch the section you are reading.

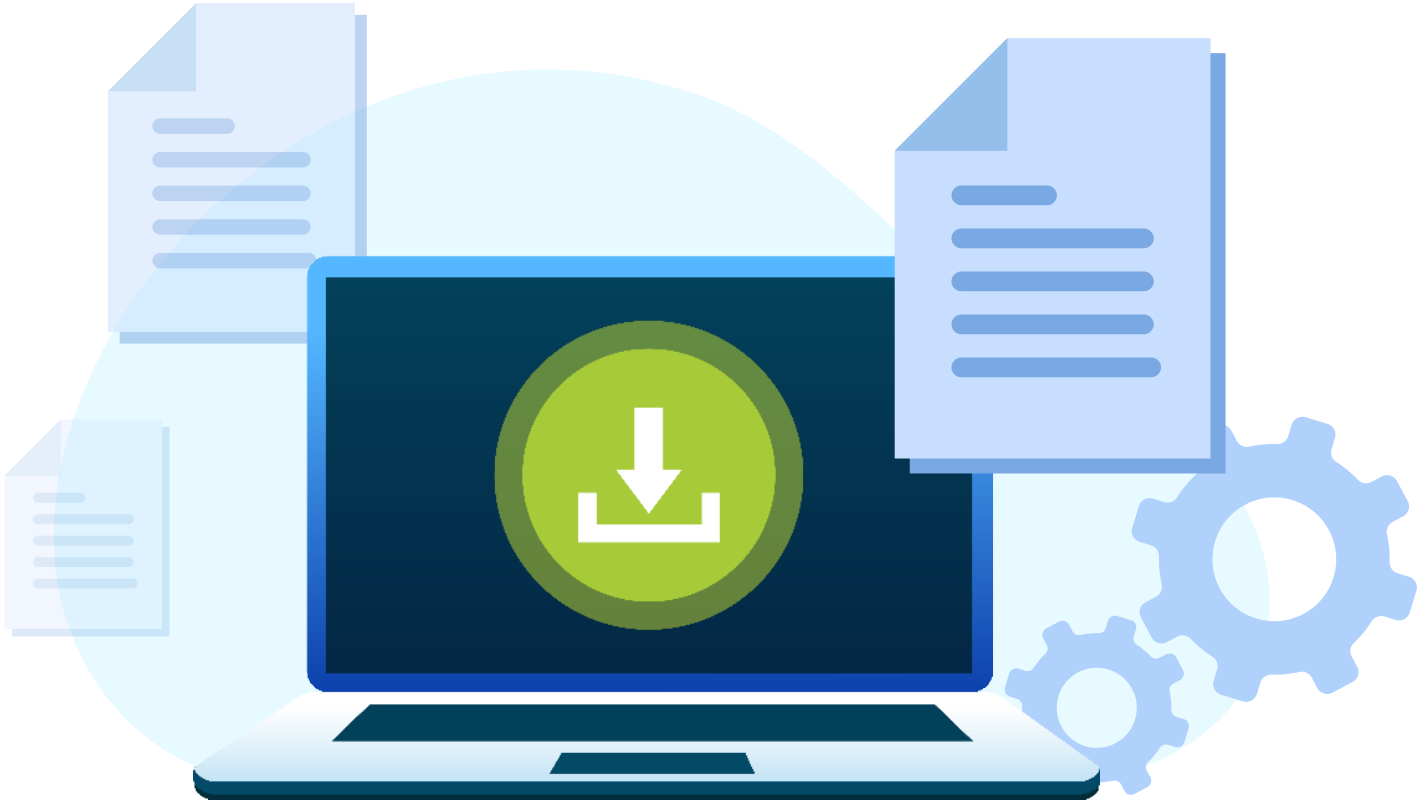
SUMMARY

With these features in place we have a fully functional Modern email client application with feature parity with its Classic counterpart.

We created a Labels tree within a sliding menu that gives the application a native-like feel while using out-of-the-box components almost everywhere.

The created Compose form builds upon a shared base class so all toolkit-agnostic code can be kept together with toolkit-specific APIs being used in the relevant sub-classes.





Thank you for reading!

Part-5 of Building an Email Client from Scratch

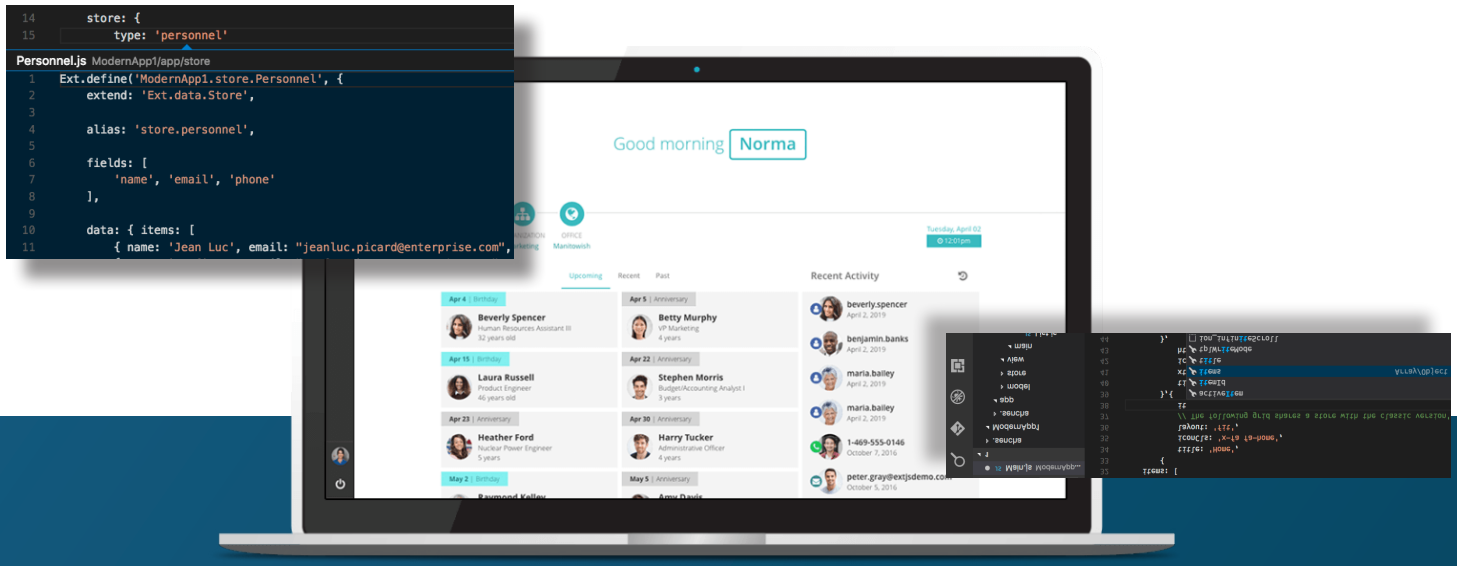
We hope you found it informative and helpful in your development projects. We have 2 more parts lined up to take you through the entire process of building an email client from scratch.

Download the Part-6 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