Sencha

Building an Email Client from Scratch

PART 3

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 the previous sections, we set up our data models for our Messages grid and Labels tree and also created the UI for users to view Messages in each label and to view a Message in full. We also created a number of action buttons in a dynamic toolbar for refreshing the grid and deleting and archiving a single message.

In this article we will cover the Compose workflow, allowing a user to create a new message, select a recipient from a data-backed combo box and 'send' or discard the message.

We will also let users re-open a draft message to continue editing their email.

Key Concepts / Learning Points

- Creating data-backed combo boxes
- Working with forms
- Working with windows
- Creating shared and toolkit-specific controller logic
- Bubbling and handling custom events

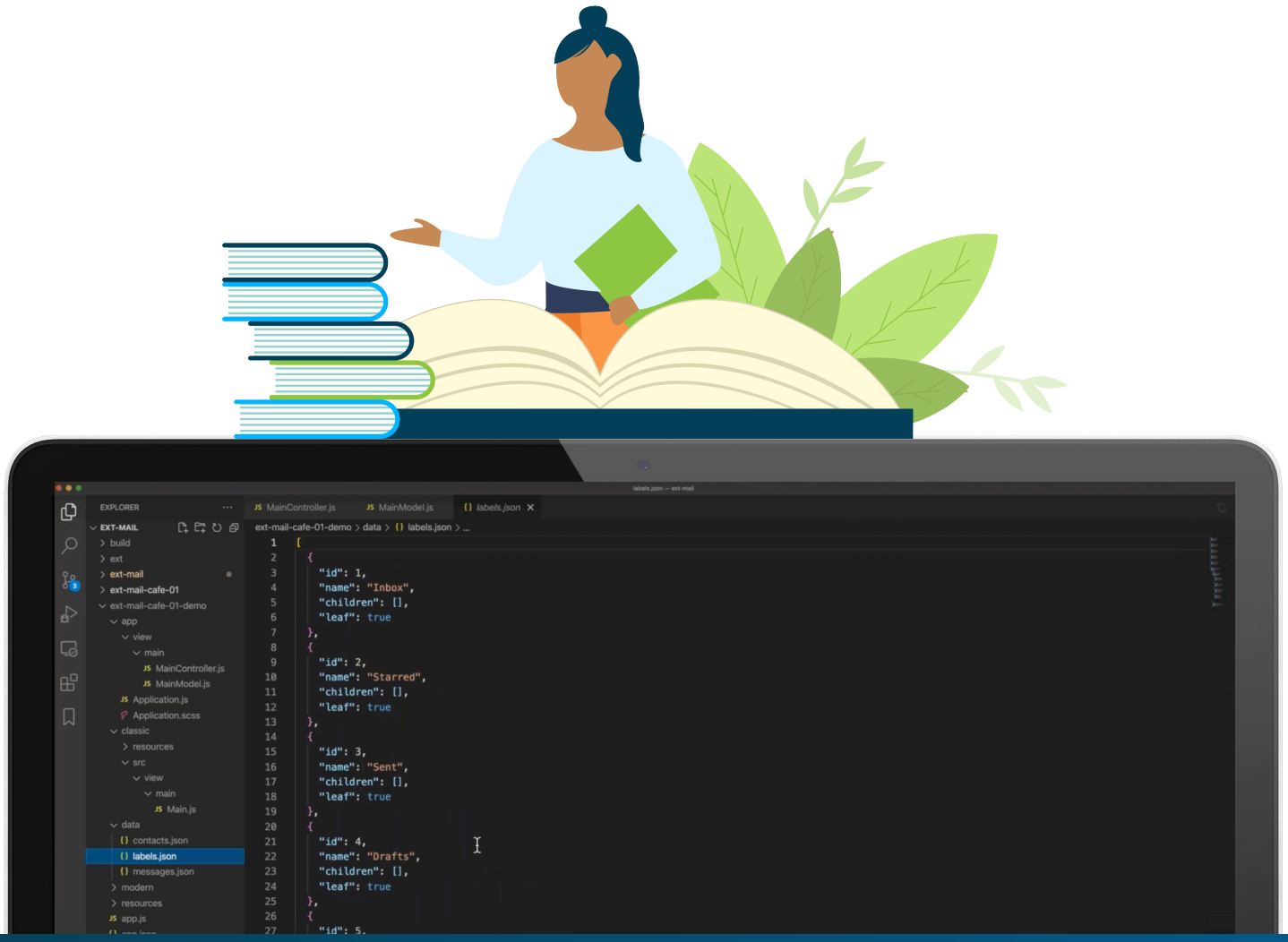
Code along with Stuart!

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 & second parts of this series where we set up the foundations of our application and built a dynamic toolbar and unread message count display for label-based message filtering, 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.



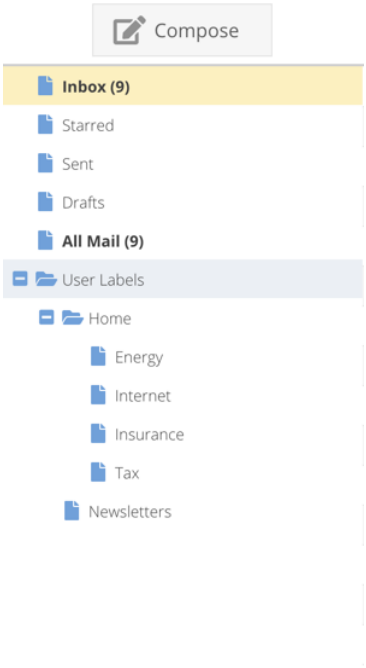
Code along with Stuart!

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.

Compose Button

At the moment we can't create a new message so we will create a compose button that will sit above the Labels tree as a docked item.



To add the button we add the `dockedItems` config to the `LabelsTree` component's `initComponent` method.

> Docked items can be added to any sub-component of an Ext.Panel and they can be docked to the `top`, `bottom`, `left` or `right` of the panel with the main `items` being shifted across to accommodate.

In this case, we want to add a `toolbar` with 3 children, 2 special spacer components to center the button, and the button itself.

```
initComponent: function() {
    Ext.apply(this, {
        rootVisible: false,
        dockedItems: [
            {
                xtype: 'toolbar',
                dock: 'top',
                weight: -1,
                items: [
                    '->',
                    {
                        xtype: 'button',
                        scale: 'large',
                        text: 'Compose',
                        iconCls: 'x-fa fa-edit',
                        width: 150,
                        handler: function () {
                            this.fireEvent('compose');
                        },
                        scope: this
                    },
                    '->'
                ]
            }
        ],
        ...
    });
    ...
}
```

We use the `dock` config to locate the toolbar at the top of the LabelsTree.

Our Compose button is a simple `button` configuration. The `scale` is set to “large” so it is as big as possible and we use the `x-fa` CSS class to make use of the built-in Font Awesome icon set.

The `handler` function will be executed when the button is clicked and will run the scope of the LabelsTree itself (thanks to the `scope: this` config).

We don’t actually create the compose form here, instead, we fire a custom event named `compose` and let the parent component handle it.

Learn How to Create a Compose Button

Watch this Part!

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

Compose Form

To compose an email we want a form with 3 fields - a recipient, a subject and a message field.

We add this as a new component named `ExtMail.view.compose.ComposeForm` and extend the `Ext.form.Panel` class.

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

    defaultListenerScope: true,
    padding: 10,
    layout: {
        type: 'vbox',
        align: 'stretch'
    },

    items: []
});
```

We set the `defaultListenerScope` to `true` so any event listener function names are resolved within this component. We add some padding and make use of a VBox layout so that the message field will always be stretched to fill the form's remaining space.

Next, we add the Subject and Message fields (we will talk about the Recipient field shortly). The former will just be a simple `Ext.form.field.Text` component, using the `textfield` xtype. We bind it to the `subject` field of the `messageRecord` object (which we will add later) so the value in the field will be stored in this field.

The Message field is a TextArea component and is given a `flex: 1` config so it fills the form's height. We bind its value to the `message` property of the `messageRecord`.

```
...
items: [
    {
        xtype: 'textfield',
        emptyText: 'Subject',
        bind: {
            value: '{messageRecord.subject}',
        },
    },
    {
        xtype: 'textareafield',
        flex: 1,
        bind: {
            value: '{messageRecord.message}',
        },
    }
]
...
```

Finally, we add the Send and Discard buttons to a toolbar docked to the bottom of the form.

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

The Send button has `formBind: true` set which will automatically disable the button if any of the form's fields are invalid. This is really useful for preventing invalid form submissions.

The `handler` for each button is set to a string which will be resolved to a method of that name on the component itself.

We add them now and simply fire a custom event for each action, passing the `messageRecord` currently active in the form as its single parameter.

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

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

Learn How to Create a Compose Form

Watch this Step!

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

Recipient Field

Our Compose form is now ready for the Recipient field to be added. For this field, we want to use a combo box that will autocomplete the recipient from a list of Contacts stored in the application.

To achieve this we need to add some more data infrastructure.

Contact Data Models

We start by adding a new data model to our shared `model` folder. The `Contact` model will have 5 fields: firstName, lastName, email, phone, and a calculated name field that combines the first and last names together.

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

    fields: [
        {
            name: 'name',
            calculate: function (data) {
                var firstName = data.firstName || '';
                var lastName = data.lastName || '';

                return Ext.String.trim(firstName + ' ' + lastName);
            },
        },
        'firstName',
        'lastName',
        'email',
        'phone',
    ],
});
```

Now we can add a Contacts store to load our contact data into and have it available for our Recipients field.

The Contacts store uses the new Contact model and a simple AJAX proxy to load our static `contacts.json` data. It has `autoLoad: true` set so when it is instantiated the contacts data is loaded.

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

    alias: 'store.Contacts',

    model: 'ExtMail.model.Contact',

    autoLoad: true,

    proxy: {
        type: 'ajax',
        url: 'data/contacts.json',
        reader: {
            type: 'json',
            rootProperty: 'rows'
        }
    }
});
```

Finally we add this store to our MainViewModel class so it is created and loaded when the application starts.

```
...
stores: {
    ...
    contacts: {
        type: 'Contacts'
    }
}
...
```

Recipients Field

Now we have our data store ready we can add the Recipient combo box and bind it to the store.

```
...
{
    xtype: 'combobox',
    emptyText: 'Recipient',
    width: '100%',
    displayField: 'email',
    valueField: 'email',
    queryMode: 'local',
    allowBlank: false,
    bind: {
        store: '{contacts}',
        selection: '{selectedRecipient}',
        value: '{messageRecord.email}'
    }
}
...
```

Explore Contact Data Models

Watch this Part!

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

We set it up so the `displayField` (the Contact model field that is shown to the user) is the same as the `valueField` (the field that is submitted with the form) to the `email` field.

We also use `queryMode: local` so when a user types in the field it will just query the local data we already have, rather than sending a query to the server requesting new data that matches the value entered. If you were dealing with large datasets using `queryMode: remote` might be favourable to avoid loading all the data upfront.

We bind the `store` config to the `contacts` store we added to the MainViewModel class and the `value` to the `email` field of the `messageRecord`.

We have also bound the `selection` property, which stores the record that is currently selected in the combobox, to the `selectedRecipient` View Model property, which doesn't yet exist. So we will add it now.

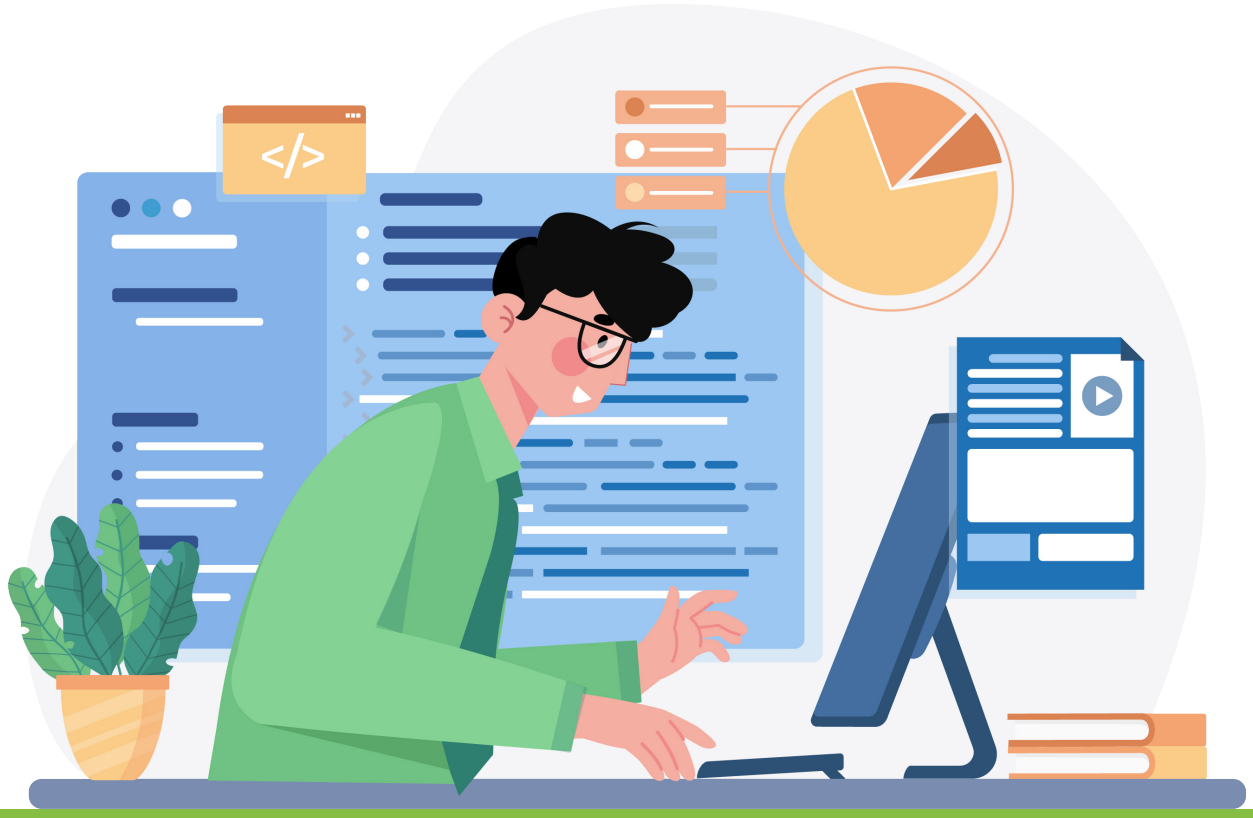
Instead of creating a standalone View Model for this form, we will just create an inline one, as we only need one data field.

```
...
viewModel: {
    data: {
        selectedRecipient: null
    }
}
...
```

The reason we want to keep track of the `selection` value, as well as the `value` (the email), is that we want to add the first and last name fields to the underlying Message record so we can display them in other parts of the app. We have to extract the parts of the Contact model that we want manually, so we add a binding to the `selectedRecipient` data field and do this ourselves.

We start by setting up the binding in the `constructor` function:

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

    this.getViewModel().bind('{selectedRecipient}',
this.onSelectedRecipientChange, this);
}
...
```



Then we can define the handler function to add the first name, last name, and email to the `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('firstName');
        lastName = selectedRecipientRecord.get('lastName');
        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
        });
    }
}
...
```

Discover How to Add a Recipients Field

Watch this Stage!

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

Compose Window

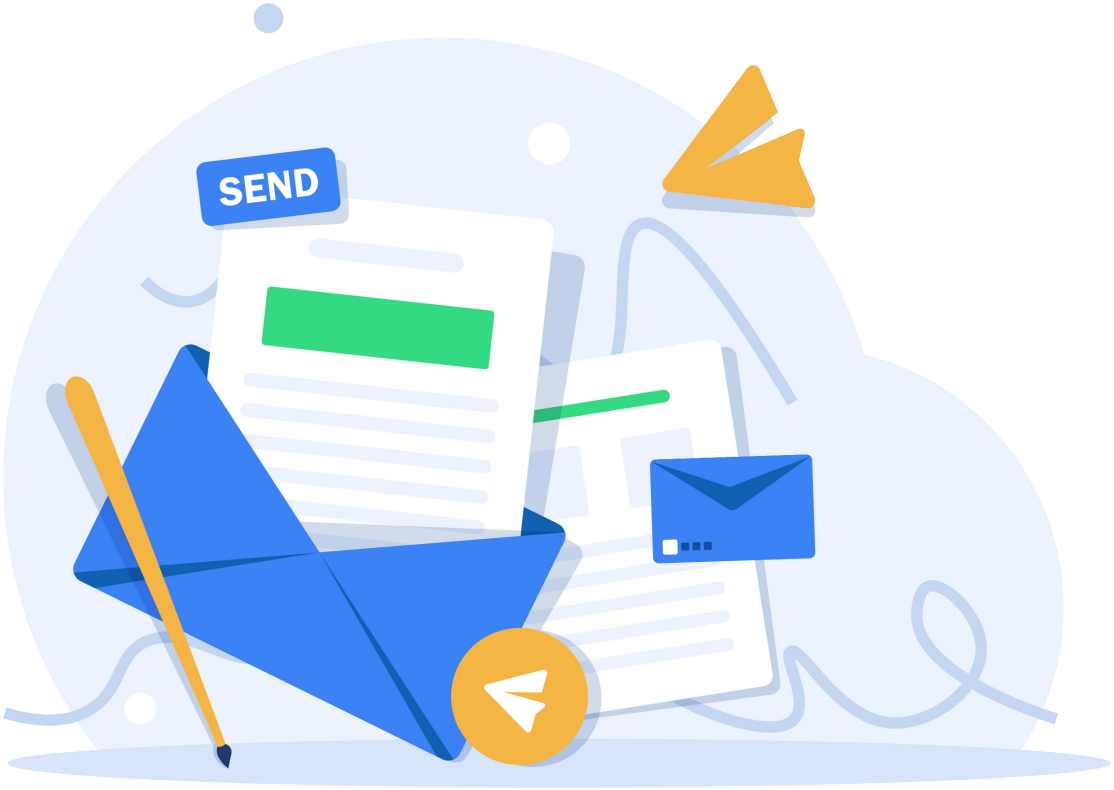
With the Compose Form complete we need to be able to show it to the user. We will display this in a Window and will create our own sub-class so we can add some additional functionality.

This will be a simple window with a single item - the ComposeForm - and it will have a basic inline View Model that holds the `messageRecord` (which we referenced multiple times in the ComposeForm). We will also add a `messageRecord` config that we will use to pipe a bound value into the view model.

```
Ext.define('ExtMail.view.compose.ComposeWindow', {
    extend: 'Ext.window.Window',
    alias: 'widget.compose-ComposeWindow',

    viewModel: {
        data: {
            messageRecord: null,
        },
    },

    config: {
        messageRecord: null,
    },

    minimizable: true,
    resizable: false,
    draggable: false,
    title: 'New Message',
    layout: 'fit',
    constrain: true,
    constrainHeader: true,
    items: [
        {
            xtype: 'compose-ComposeForm',
            bubbleEvents: ['send', 'discarddraft'],
        },
    ],

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

When configuring the ComposeForm we use the `bubbleEvents` config to tell it that we want the `send` and `discarddraft` events to be bubble up and emitted from the ComposeWindow too, this means we can listen for them on the ComposeWindow rather than having to delve inside and attach handlers to the form itself.

We also add an `updater` for the `messageRecord` config which simply sets the View Model's property with the passed value, keeping these in sync.



Learn How to Create a Compose Window

Watch this Section!

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

Start Compose Flow

Our Compose interface is in place so now we can tie it all together. We start by adding an `onComposeMessage` function to the `MainControllerBase` class which will be shared across toolkit apps.

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

    this.getViewModel().getStore('messages').add(messageRecord);
    this.getViewModel().getStore('messages').commitChanges();

    this.showComposeWindow(messageRecord);
}
```

First, we create a new Message instance, marking it as `outgoing` and `draft`, along with giving it the `DRAFTS` label.

We then add it to the Messages store and then pass it to the `showComposeWindow` function.

> We immediately call the `commitChanges` method of the Messages store because we don't have a proper backend so we want the record to appear as 'saved' straight away.

The `showComposeWindow` method will need to be toolkit-specific as the window will be shown differently depending on the toolkit, so we add this method to the Classic toolkit's sub-class.

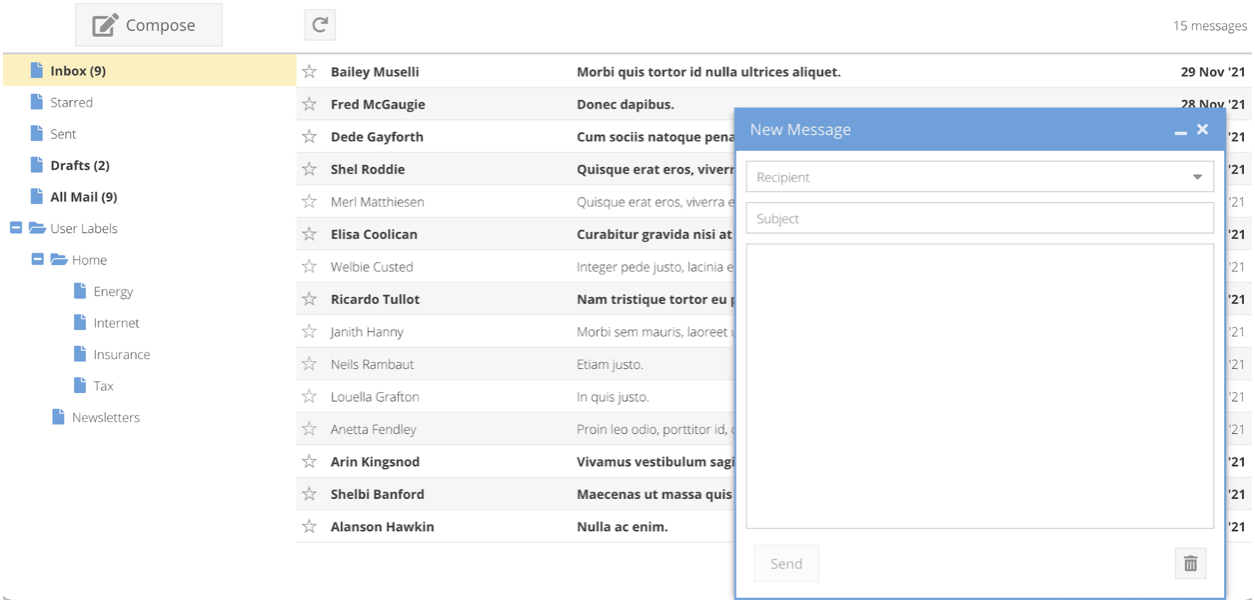
```
showComposeWindow: function (messageRecord) {
    var win = Ext.create('ExtMail.view.compose.ComposeWindow', {
        messageRecord: messageRecord,
        height: 500,
        width: 500
    });

    win.show();
}
```

This method simply creates a new ComposeWindow, passing in the newly created Message record, and shows it to the user.

Finally we need to hook the `onComposeMessage` method to the `compose` event that the new Compose button we added at the start fires. We do this in the `LabelsTree` configuration in the `Main` component.

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

Clicking the Compose button will now open our new form and let us compose a message.



Explore Compose Flow

Watch this Phase!

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

Compose Window Events

We can compose our email now but we can't yet send or discard it since we haven't hooked up the ComposeWIndow's buttons yet, we'll do that now!

In the `showComposeWindow` method we hook up the `send` and `discarddrafts` events using the window's `on` method.

```
showComposeWindow: function (messageRecord) {
    var win = Ext.create('ExtMail.view.compose.ComposeWindow', {
        messageRecord: messageRecord,
        height: 500,
        width: 500
    });

    win.on({
        send: Ext.bind(this.onSendMessage, this, [win], true),
        discarddraft: Ext.bind(this.onDiscardDraftMessage, this, [win],
 true),
        scope: this
    });

    win.show();
}
```

We want to be able to close the Compose window after these operations are complete so we use the `Ext.bind` function to wrap our handler functions and tack on the `win` variable as an extra parameter.

The `onSendMessage` method will remove the `DRAFTS` label from the Message and add the `SENT` label. It will then mark `draft` as false, `sent` as true, and stamp it with a `date`. We will then commit the changes, and close the window.

```
onSendMessage: function (messageRecord, e, composeWindow) {
    messageRecord.removeLabel(ExtMail.enums.Labels.DRAFTS);
    messageRecord.addLabel(ExtMail.enums.Labels.SENT);

    messageRecord.set({
        draft: false,
        sent: true,
        date: new Date()
    });

    messageRecord.commit();

    composeWindow.close();
}
```

> Note the third parameter is the `composeWindow` reference that we added using the `Ext.bind` method when we hooked up the event handler.

The `onDiscardDraftMessage` handler is very simple and removes the record from the `Messages` store and then closes the window.

```
onDiscardDraftMessage: function(messageRecord, e, composeWindow) {
    this.getViewModel().getStore('messages').remove(messageRecord);

    composeWindow.close();
}
```

Learn How to Compose Window Events

Watch this Stage!

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

Editing a Draft

After creating a new message and closing the window we will see it appear in the `Drafts` folder. Clicking on this message will move us to the default MessageReader screen. Ideally, we want this to reopen the Compose Window so the user can continue to write their email. We will do this now.

To do this we head to the `onMessageClick` method in the `MainControllerBase` class and add some logic for when the message has the `draft` flag set to true. When this is the case we call the `showComposeWindow` method, passing in the Message record that was clicked. Our existing code will take care of populating the form with the existing record and it can be edited and sent or discarded from there.

```
onMessageClick: function(grid, messageRecord, rowEl, index, e) {

    if (e.getTarget('.x-action-col-icon')) {
        return;
    }

    if (messageRecord.get('draft')) {
        this.showComposeWindow(messageRecord);
    } else {
        this.getViewModel().set('selectedMessage', messageRecord);
    }
}
```

Learn How to Edit a Draft

Watch this Step!

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

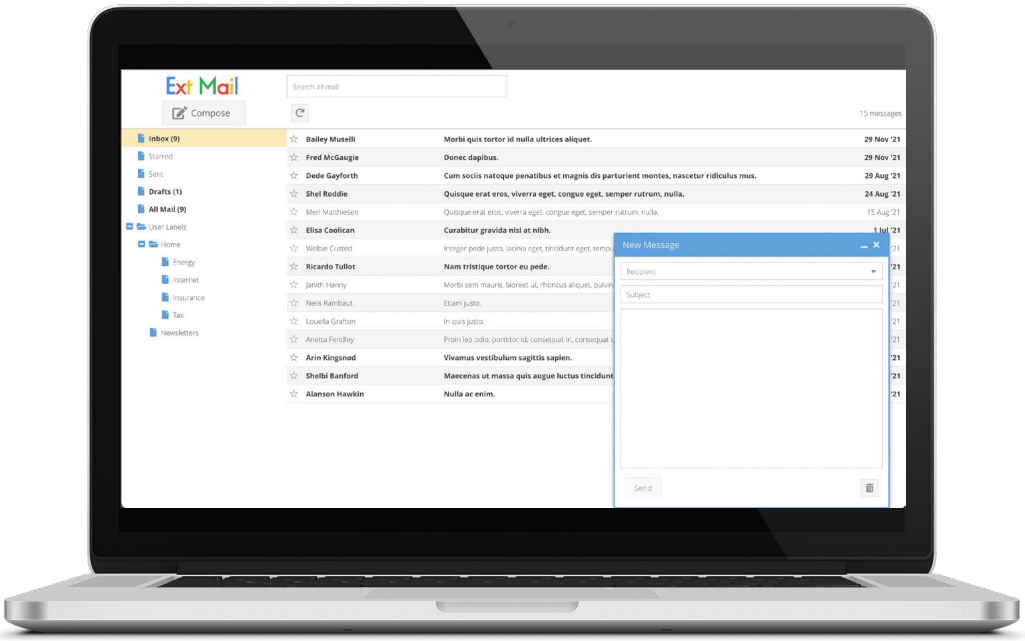
SUMMARY

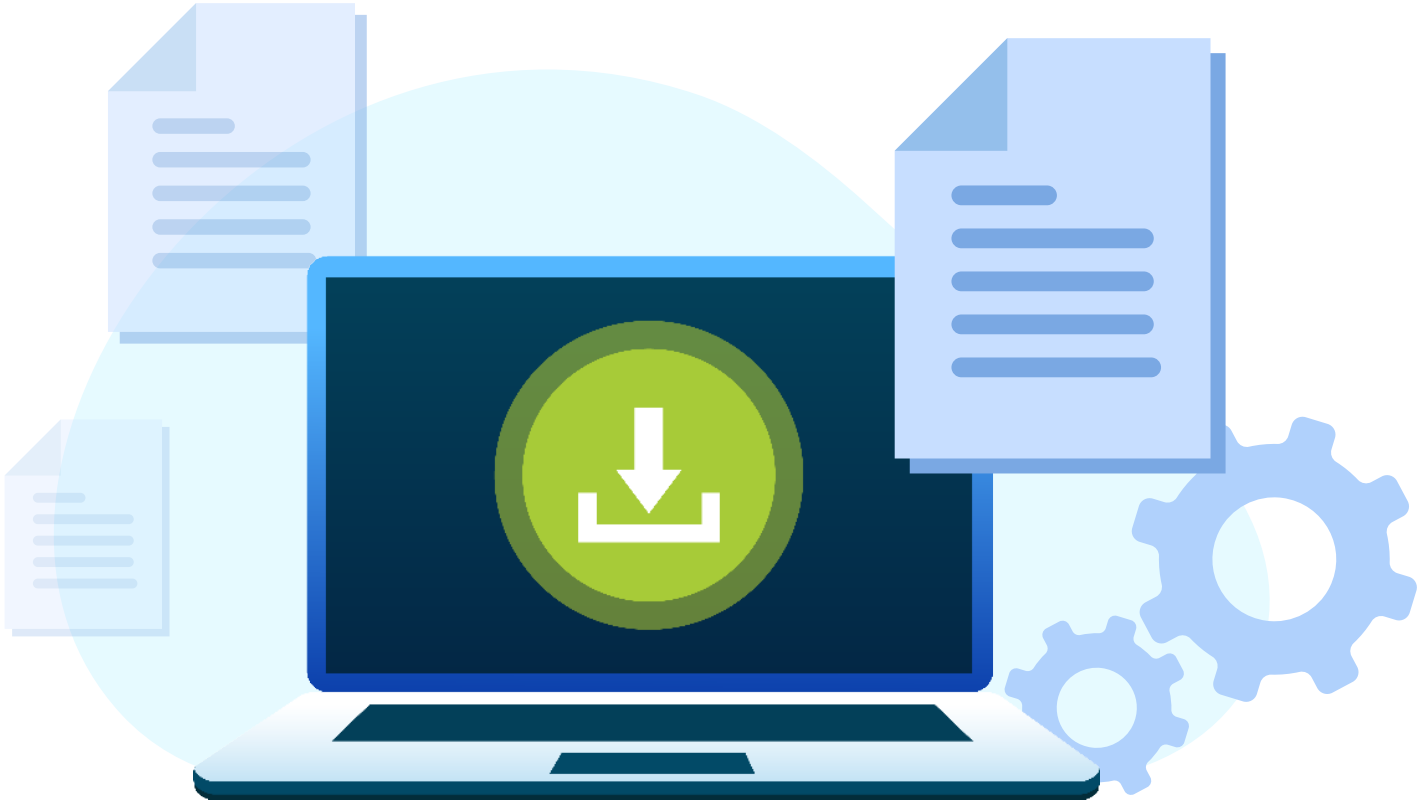
In this article we've extended our Email Client to allow users to compose their own new messages, having them appear in the Drafts folder in our main navigation.

We made use of the common base classes to enable our composing business logic to be shared across toolkits - something we will be exploring in the next article.

We also created a simple form and added 3 different field types - text field, text area, and a more complex type-ahead combo box. Each of these was bound to a backing View Model using the `bind` configuration.

As well as the Form Panel component, we used the Window component to allow the compose form to float about the main interface and be positioned correctly. In the next part, we will be integrating the Modern toolkit into our application to bring support to mobile and touch devices.





Thank you for reading!

Part-3 of Building an Email Client from Scratch

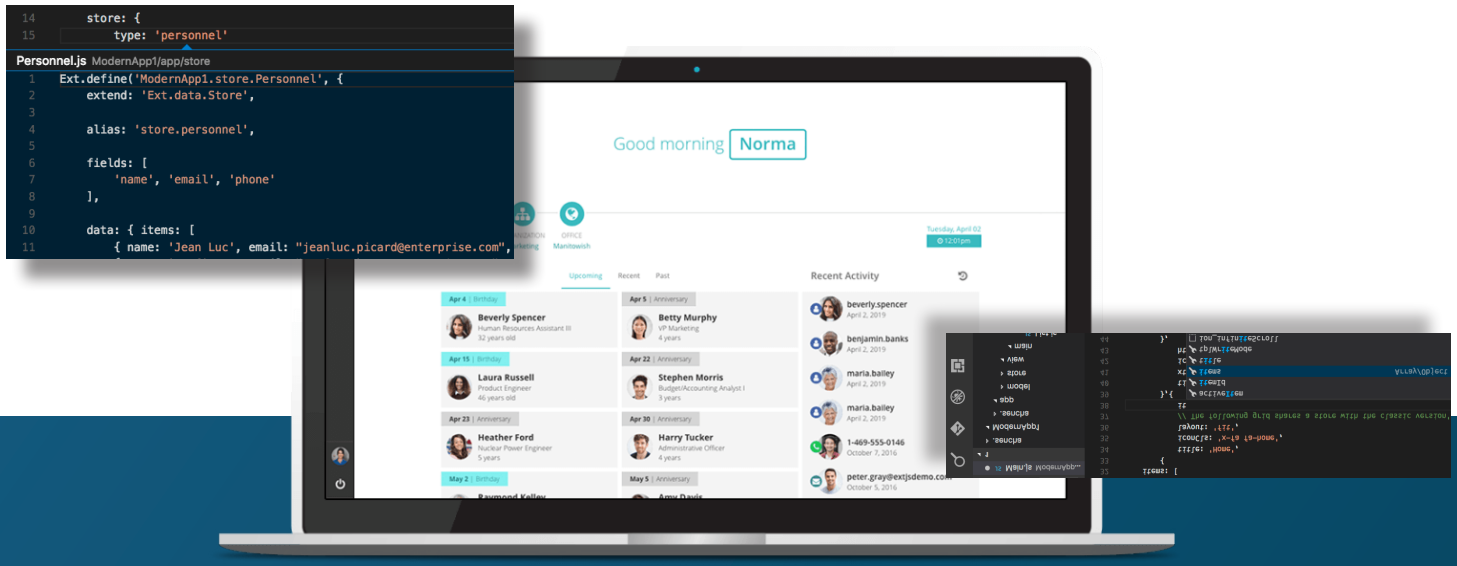
We hope you found it informative and helpful in your development projects. We have 4 more parts lined up to take you through the entire process of building an email client from scratch.

Download the Part-4 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