Sencha

Building an Email Client from Scratch

PART 4

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

So far we have built our application focusing on the Classic toolkit which is ideal for targeting desktop/laptop computers, however, we want to optimize our application for use on touch and mobile devices as well.

To do this we build upon the applications shared business logic and data models to create a Modern toolkit interface using the toolkit's UI components.

We will create a Messages grid with custom rendering and row actions and refactor some of the existing code to enable maximum code reuse.

Key Concepts / Learning Points

- An Introduction to the Modern toolkit and its components and APIs
- Creating Modern grid components
- Hooking up grid actions
- Refactoring existing code to maximize reuse across toolkits
- Normalizing event handling code across toolkits

Code along with Stuart!

Start buddy coding with Stuart on-demand!

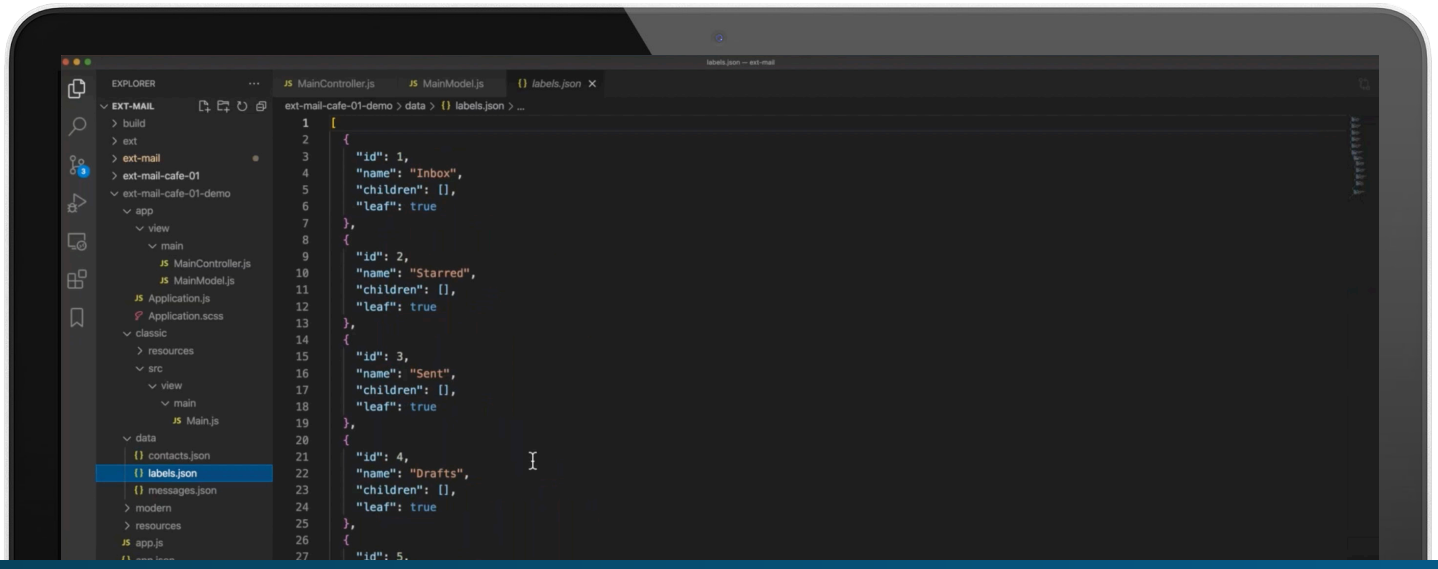
Use the bite-sized video links in this e-book to instantly watch the section you are reading.

Introduction

In this article, we will be expanding our application to include support for mobile and touch devices using Ext JS' Modern Toolkit.

So far we have built our app in a way that maximizes code reuse through the shared `app` folder and we will see the real benefits of this during this article. We will be building the mobile-friendly interface and harnessing all of the business logic and data modeling that we have already in place, making this a relatively simple process.

To launch our app to test the Modern toolkit we can run `sencha app watch modern` and then open it using Chrome's device emulation mode (or the equivalent in other browsers). By using this emulation it will tell Sencha Cmd to serve the Modern toolkit's app.



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.

Adding the Main View

The Modern toolkit's version of the application needs to have its own root component that will be the parent to the rest of the interface. In this case, we will create a Main component that will extend the `Ext.NavigationView` component. This component gives the application a 'native' feel without much effort at all.

The Navigation View component extends the basic Container class and gives us the ability to push and pop child components onto it, with a nice sliding transition when doing so. It also gives us a header that will display the current card's title and a back button allowing the user to move to a previous card easily.

```
Ext.define('ExtMail.view.main.Main', {
    extend: 'Ext.NavigationView',
    xtype: 'app-main',
    fullscreen: true,

    requires: [
        'ExtMail.view.main.MainModel'
    ],

    viewModel: 'main'

});
```

We link this view to the "main" View Model which is the one that lives in the shared `app` folder and is common to the Classic and Modern toolkit apps.

Next, we will create a Modern-specific Main View Controller which will extend the MainControllerBase that we created in the previous stages. This base class contains the majority of the business logic we will require and the sub-class we will now create will just add code to deal with modern-only components.

```
Ext.define('ExtMail.view.main.MainController', {
    extend: 'ExtMail.view.main.MainControllerBase',

    alias: 'controller.main'

});
```

With this created, we can require it and link it to the Main component.

```
Ext.define('ExtMail.view.main.Main', {
    ...
    requires: [
        'ExtMail.view.main.MainController',
        'ExtMail.view.main.MainModel'
    ],

    controller: 'main'
    ...
});
```

We don't have to tell the application to use this Main component as the framework will automatically look for a component with the `app-main` xtype when launching in modern-toolkit mode.

Learn How to Add the Main View

Watch this Section!

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

Adding the Messages Grid

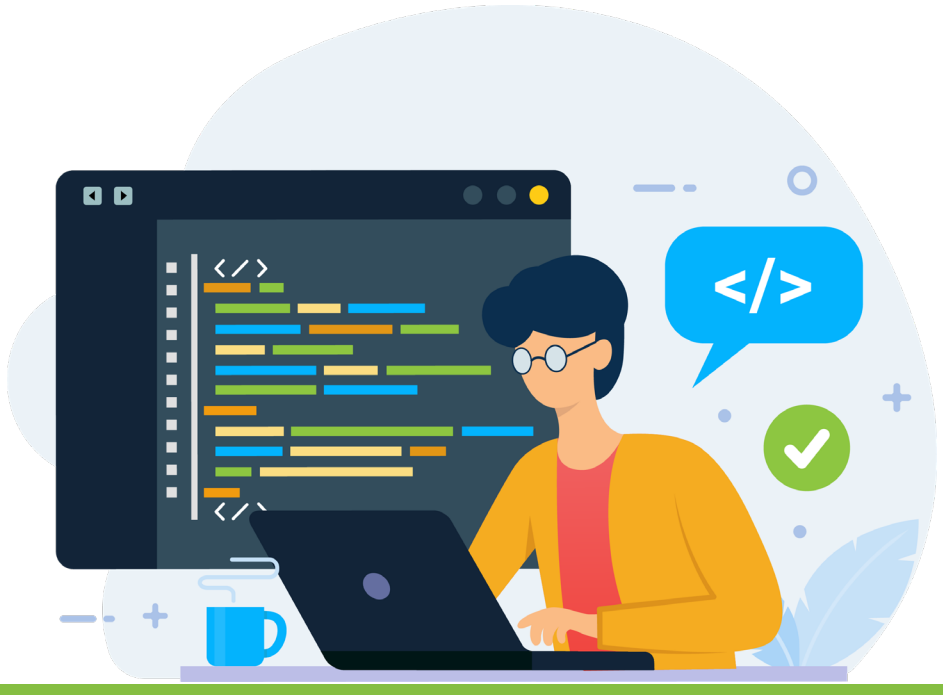
To display the messages we want to create a grid similar to the one in the Classic toolkit but using the Modern toolkit's grid component.

We create a new class called `ExtMail.view.messages.MessagesGrid` in the `view/messages` folder. This class extends the `Ext.grid.Grid` component (essentially the modern equivalent of the `Ext.grid.Panel` component).

We give it an alias of `messages-MessagesGrid` (again mirroring the Classic toolkit version so things are consistent), and add a CSS class called `messages-grid` which we will use to scope some styling to the component.

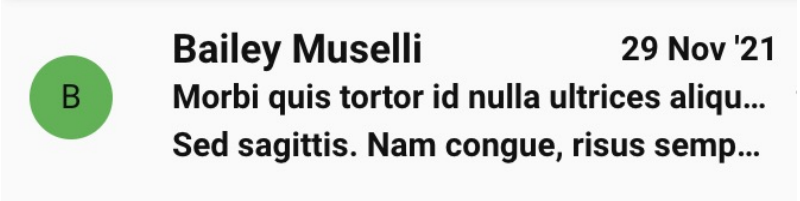
```
Ext.define('ExtMail.view.messages.MessagesGrid', {
    extend: 'Ext.grid.Grid',
    alias: 'widget.messages-MessagesGrid',

    cls: 'messages-grid'
});
```



Create the Columns

Now we can define the columns we want the grid to have. At this stage, we will create 2 columns - one for the 'avatar' (showing the sender's first name initial in a coloured circle) and another for the sender's full name, the subject, and a snippet of the message body.



The avatar column will be bound to the `firstName` property of the record using the `dataIndex` config and we define the HTML template using the `tpl` config.

```
...
columns: [
    {
        dataIndex: 'firstName',
        width: 60,
        cell: {
            encodeHtml: false
        },
        tpl: [
            [
                '<div class="avatar" style="background-color: {[this.
getAvatarColour(values.firstName)]};">',
                '   <span>{[values.firstName.substring(0, 1).toUpperCase()]}</
span>',
                '</div>',
            ].join(''),
            {
                getAvatarColour: function(name) {
                    var alphabet = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'.split('');
                    var colours = ['#e6194B', '#3cb44b', '#ffe119', '#4363d8',
'#f58231', '#911eb4', '#42d4f4', '#f032e6', '#bfef45', '#fabed4', '#469990',
'#dcbeff', '#9A6324', '#fffac8', '#800000', '#aaffc3', '#808000', '#ffd8b1',
```

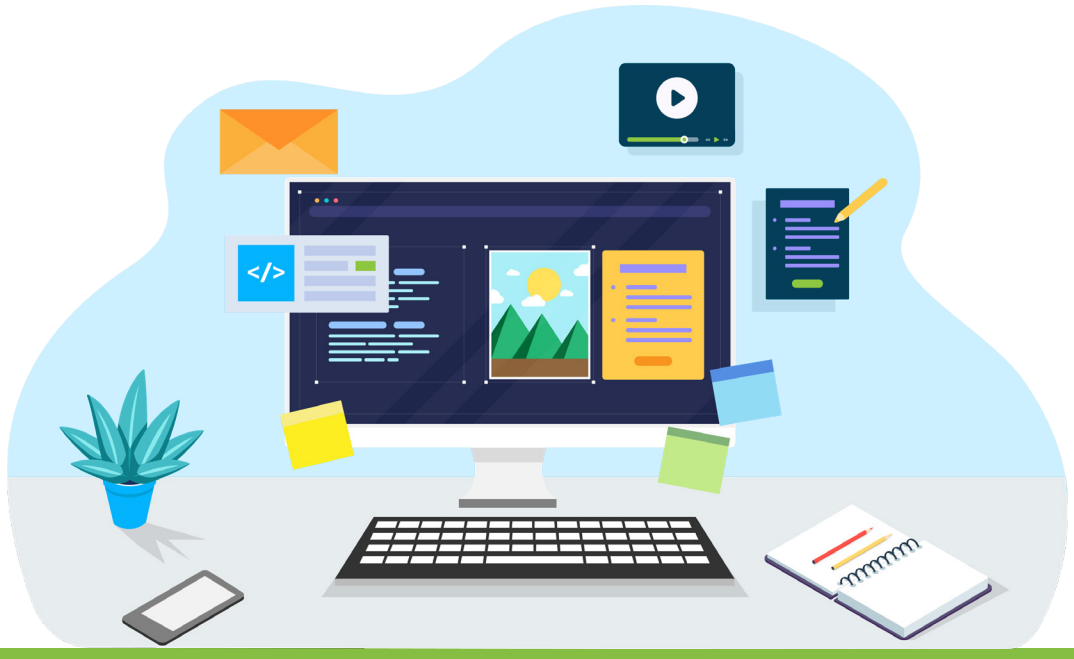
```
'#000075', '#a9a9a9', '#ffffff', '#000000'];
                    var initial = name.substring(0, 1).toUpperCase();

                    return colours[alphabet.indexOf(initial)] || '#e6194B';
                }
            }
        ]
    }
]
...
```

The template defines a wrapping `div` which is given a background colour based on the sender's first name initial, and then that initial is displayed inside it.

We give the column a width of 60 pixels and tell the cell to render the template output as HTML and not try to encode it.

Next we create the second column which will display the sender's full name, the subject and message snippet. This will look identical to the avatar's column but just with a different template.



```
...
{
    dataIndex: 'subject',
    flex: 1,
    cell: {
        encodeHtml: false
    },
    tpl: [
        [
            '<div class="{[values.unread ? \"unread\" : \"\"]}">',
            '   <div class="top-line">',
            '      <span class="name">{fullName}</span>',
            '      <span class="date">{date:date("j M \'y")}</span>',
            '   </div>',
            '   <div class="subject">{subject}</div>',
            '   <div class="message">{message}</div>',
            '</div>'
        ].join('')
    ]
}
...
```

The template adds a conditional “unread” CSS class for messages that are unread using the `{[ ... ]}` syntax which lets us execute arbitrary JavaScript code within the template.

The rest of the template simply outputs the message info in a stacked nature which will be styled with CSS.

Adding Component Styles

To add styling to this component we can create an `.scss` file as a sibling to the component with the same name, in this case `MessagesGrid.scss`. The contents of this file will be compiled by Sencha Cmd and added to our page automatically.

```
.messages-grid {
    .avatar {
        border-radius: 50%;
        display: flex;
        align-items: center;
        justify-content: center;
        width: 35px;
        height: 35px;
    }

    .unread {
        .name, .date, .subject {
            font-weight: bold;
        }
    }

    .top-line {
        display: flex;

    }

    .name {
        font-size: 1.05rem;
        flex: 1;
    }

    .subject, .message {
        text-overflow: ellipsis;
        overflow: hidden;
    }
}
```

Integrating the Messages Grid

With the Messages grid complete we can now add it to our Main component and bind it to the Messages data store from the shared View Model. We do this by simply requiring the class and adding it to the `items` array.

```
...
requires: [
    ...
    'ExtMail.view.messages.MessagesGrid'
],

items: [
    {
        xtype: 'messages-MessagesGrid',
        hideHeaders: true,
        titleBar: false,
        bind: {
            store: '{messages}'
        }
    }
]
...
```

We hide the grid's column headers using the `hideHeaders` and `titleBar` configs.

Finally, we bind the `messages` store (from our MainViewMModel) to the grid using the `bind` config.

Learn How to Add the Messages Grid

Watch this Chapter!

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

Handling a Message Tap

When a user clicks on a message we want to have the app move to a reader screen, showing the details of the message. We do this by listening for the `childtap` event of the Grid and adding a handler function to the Modern toolkit's MainViewController class.

```
// Main.js
...
items: [
    {
        xtype: 'messages-MessagesGrid',
        ...
        listeners: {
            childtap: 'onMessageTap'
        }
    }
]
...


// MainController.js
onMessageTap: function(grid, location) {

}
```

In the Classic toolkit when a Message is clicked we set the `selectedMessage` property in the View Model and the Controller then changes the UI when this property changes. We want the Modern toolkit to follow this same pattern as it means we keep the business logic the same, and the toolkit code can handle the stat changing as it needs to.

Refactoring the Message Selecting Code

To do this we need to refactor that code slightly. First, we extract the Message Click code from the Classic MainController and move it to the MainControllerBase class:

```
Ext.define('ExtMail.view.main.MainControllerBase', {
    ...
    handleMessageClick: function(messageRecord) {
        // if it's a draft then we show the compose window, otherwise we show
the message reader
        if (messageRecord.get('draft')) {
            this.showComposeWindow(messageRecord);
        } else {
            this.getViewModel().set('selectedMessage', messageRecord);
        }
    }
});
```

Having this function in the base class means we can call from the toolkit sub-class after extracting the Message Record from the event as needed. The `childtap` and `itemclick` events don't have the same signature so we couldn't share the handler function directly.

The Modern toolkit's `onMessageTap` method now looks like this, grabbing the Message record from the `location` parameter.

```
// modern/src/view/main/MainController.js
onMessageTap: function(grid, location) {
    this.handleMessageClick(location.record);
}
```

While the Classic toolkit's `onMessageClick` handler is updated to this:

```
// classic/src/view/main/MainController.js
onMessageClick: function(grid, messageRecord, rowEl, index, e) {
    // don't do the row action if we've clicked on the action column
    if (e.getTarget('.x-action-col-icon')) {
        return;
    }

    this.handleMessageClick(messageRecord);
}
```

Showing the Message Reader

At this point tapping on a Message will update the `selectedMessage` View Model property but the UI won't change. To do that we need to bind the `selectedMessage` property and tell the UI to change when it does.

First, we add an `init` method to the MainController class. This function is called when the controller is initialized and is a useful place to hook up bindings and other tasks to prepare the view. In it, we use the `bind` method of the View Model to run a function every time it changes.

```
init: function() {
    this.getViewModel().bind('{selectedMessage}',
this.onSelectedMessageChange, this);
},

onSelectedMessageChange: function(selectedMessage) {
    if (selectedMessage) {
        this.getView().push(this.getMessageDetailsConfig(selectedMessage));
    } else {
        this.getView().pop();
    }
}
```

In the handling function if the `selectedMessage` is truthy then we `push` a new component config onto the main view, otherwise we pop the current one-off (i.e. move back from a reader view to the list).

The `getMessageDetailsConfig` method simply returns a component configuration:

```
getMessageDetailsConfig: function(messageRecord) {
    return {
        xtype: 'reader-MessageReader',
        data: messageRecord.data,
        header: false
    };
}
```

This will create a MessageReader component and render it using the `messageRecord`'s data to populate the template.

> Don't forget to add the MessageReader class to the MainController's `requires` array.

Refactoring the Message Reader

Despite the MessageReader existing in the project, it is part of the `classic` toolkit folder and so will not be found when running the Modern toolkit app. If you run the app just now you will see an error about this missing component.

Fortunately, the Message Reader component is a basic component with a template and uses configs that are all common to both toolkits. This means we can move the component from the `classic` folder to the shared `app` folder.

Rerunning the app will now show the Message Reader for both toolkits.

Explore How to Handle a Message Tap

Watch this Step!

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

Adding an Actions Toolbar

In the Classic app we have a toolbar above the Message list and reader which has various actions for each screen - refresh, back, archive, delete etc - which we now want to show for the Modern toolkit.

Since the toolbar will be identical we can refactor it slightly so it can be shared across both toolkits. To do this we will create a `MessagesToolbarBase.js` class in the common `app` folder and create a `buildItems` method which will return an array of the toolbar item configs.

```
Ext.define('ExtMail.view.messages.MessagesToolbarBase', {
    extend: 'Ext.Toolbar',

    buildItems: function() {
        return [
            {
                xtype: 'button',
                tooltip: 'Refresh',
                iconCls: 'x-fa fa-redo',
                handler: this.makeHandler('refresh'),
                scope: this,
                bind: {
                    hidden: '{!visibleMessageButtons.refresh}'
                }
            },
            {
                xtype: 'button',
                tooltip: 'Back',
                iconCls: 'x-fa fa-arrow-left',
                handler: this.makeHandler('back'),
                scope: this,
                hidden: true, // hide from start
                bind: {
                    hidden: '{!visibleMessageButtons.back}'
                }
            },
            {
                xtype: 'button',
                tooltip: 'Archive',
```

```
                iconCls: 'x-fa fa-archive',
                handler: this.makeHandler('archive'),
                scope: this,
                hidden: true, // hide from start
                bind: {
                    hidden: '{!visibleMessageButtons.archive}'
                }
            },
            {
                xtype: 'button',
                tooltip: 'Delete',
                iconCls: 'x-fa fa-trash',
                handler: this.makeHandler('delete'),
                scope: this,
                hidden: true, // hide from start
                bind: {
                    hidden: '{!visibleMessageButtons.delete}'
                }
            },
            {
                xtype: 'button',
                tooltip: 'Mark as Unread',
                iconCls: 'x-fa fa-envelope',
                handler: this.makeHandler('markunread'),
                scope: this,
                hidden: true, // hide from start
                bind: {
                    hidden: '{!visibleMessageButtons.markUnread}'
                }
            },
            '->',
            {
                xtype: 'component',
                tpl: '{count} messages',
                data: {},
                bind: {
                    hidden: '{!visibleMessageButtons.messageCount}',
                    data: {
                        count: '{messages.count}'
                    }
                }
            }
        ];
    },
```

```
        makeHandler: function(event) {
            return function() {
                this.fireEvent(event);
            };
        }
    });
```

To make this fully compatible with the two toolkits we changed the base class from `Ext.toolbar.Toolbar` to `Ext.Toolbar`.

To keep the Classic app working as it is we can refactor its MessagesToolbar component to extend this new base class and call the `buildItems` method in its `initComponent` method.

> We couldn't share this toolbar in its entirety because the Modern toolkit doesn't use the `initComponent` lifecycle hook.

Creating the Modern Messages Toolbar

With the base class in place we can create the Modern toolkit version which makes use of the `initialize` lifecycle hook that is a close equivalent to the `initComponent` method.

```
Ext.define('ExtMail.view.messages.MessagesToolbar', {
    extend: 'ExtMail.view.messages.MessagesToolbarBase',
    alias: 'widget.messages-MessagesToolbar',

    initialize: function () {
        this.setItems(this.buildItems());

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

We pass the output of the `buildItems` method into the `setItems` call which will set the component up with those child components.

With the component defined, we can add it to the Main component. To make the component docked to the top we use the `docked: 'top'` config and add it to the `items` array, rather than the explicit `dockedItems` array we used in the Classic toolkit.

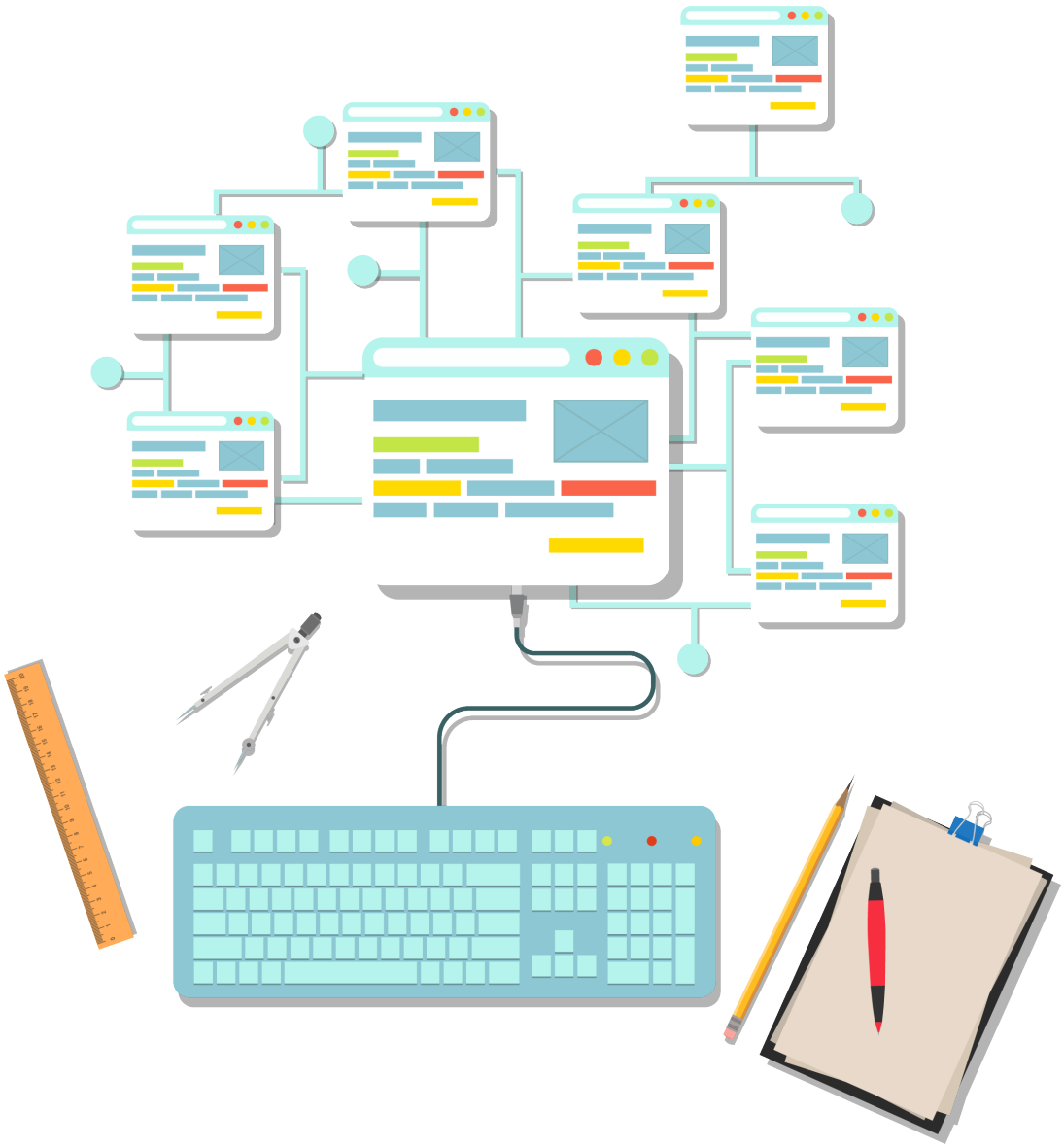
```
{
    xtype: 'messages-MessagesToolbar',
    docked: 'top'
}
```

Hooking up the Actions

Finally, we can hook up the toolbar events to the common handler functions:

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

All of these handlers exist in the `MainControllerBase` class and because they are acted purely on the data (and not making UI changes) they can be safely shared and give us all that existing functionality for free.



Discover How to Add an Actions Toolbar

Watch this Part!

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

Adding a Row Action

The final piece of functionality we want to add is the ability to “star” and “unstar” messages using a button on the grid row. To do this we will make use of the `tools` config of a column.

Tools can be given similar configs to a Button (e.g. handler and iconCls) and can be positioned to the start or the end of the column using the `zone` config.

In our case, we create two tools - star and unstar - which will be shown/hidden depending on the `starred` state of the message row.

```
{
    dataIndex: 'subject',
    flex: 1,
    cell: {
        encodeHtml: false,
        tools: {
            star: {
                handler: 'onUnStarMessage',
                iconCls: 'x-fa fa-star',
                zone: 'end',
                bind: {
                    hidden: '{!record.starred}'
                }
            },
            unstar: {
                handler: 'onStarMessage',
                iconCls: 'x-fa fa-star-half-alt',
                zone: 'end',
                bind: {
                    hidden: '{record.starred}'
                }
            }
        }
    },
    tpl: [...]
}
```

There are a couple of configs we need to set on the grid itself to allow the tool configurations to work as desired. First, in order to have the `handler` strings resolve to methods defined on the grid itself, rather than searching for a View Controller method, we must add the `defaultListenerScope: true` config.

Next, the `bind` configuration on the tools will, by default, try to bind to a property called "record" on an attached View Model. However, in this case, we want it to bind to the current grid row's record. To make this happen we need to add the following to the grid's configuration:

```
...
itemConfig: {
    viewModel: true;
}
...
```

This tells the column items to act as their own view model rather than looking for one on the grid itself.

Adding Tool Handler Functions

With those configs in place we can add the `onStarMessage` and `onUnStarMessage` functions to the grid:

```
...
onStarMessage: function(grid, info) {
    this.fireEvent('starmessage', info.record);
},

onUnStarMessage: function(grid, info) {
    this.fireEvent('unstarmessage', info.record);
}
```

We simply fire custom events from the grid which will be handled in the Main component and tied to our common code.

Listening for Star & Unstar Events

Now that we have the `starmessage` and `unstarmessage` events coming out of the MessagesGrid we can add a listener for them on the MessagesGrid config. These listeners can use the functions defined in the MainControllerBase class which flips the starred flag as required.

```
...
{
    xtype: 'messages-MessagesGrid',
    ...
    listeners: {
        childtap: 'onMessageTap',
        starmessage: 'onStarMessage',
        unstarmessage: 'onUnStarMessage'
    }
}
...
```

Explore How to Add a Row Action

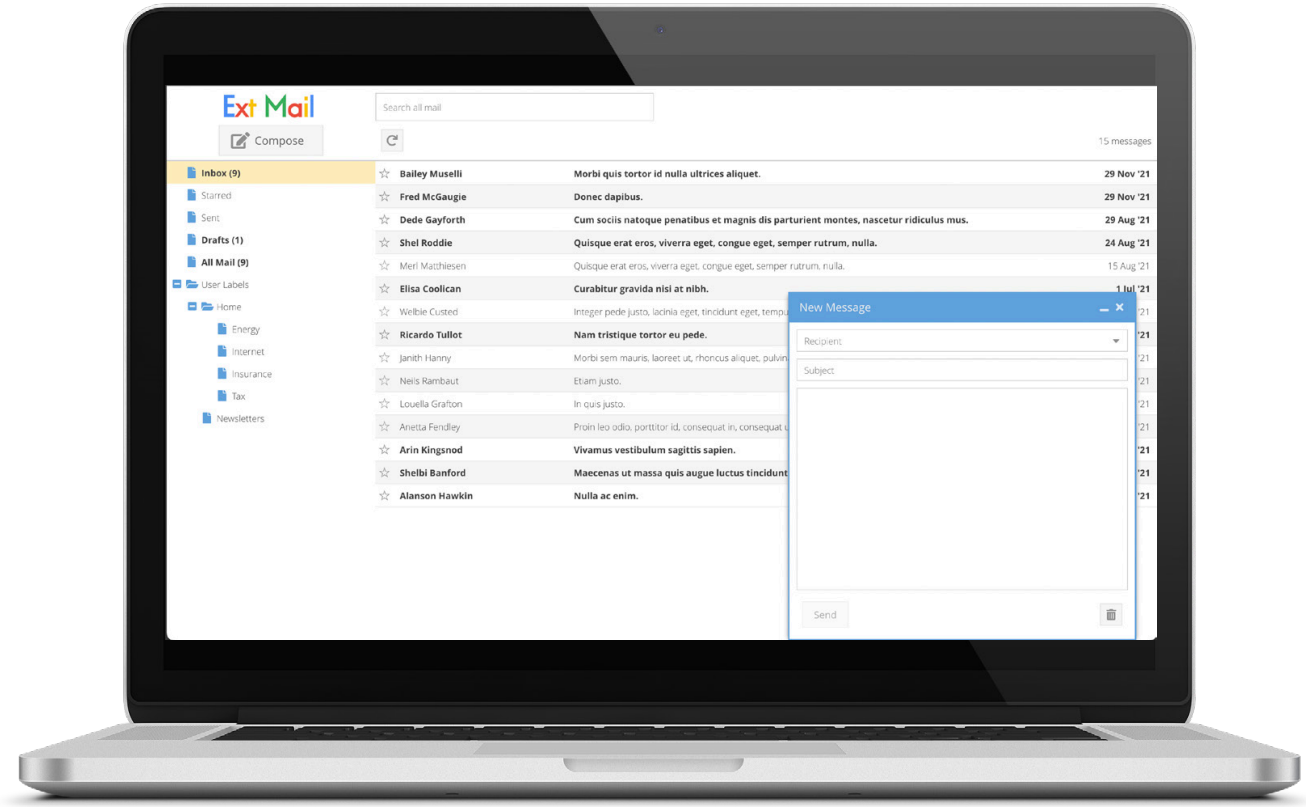
Watch this Phase!

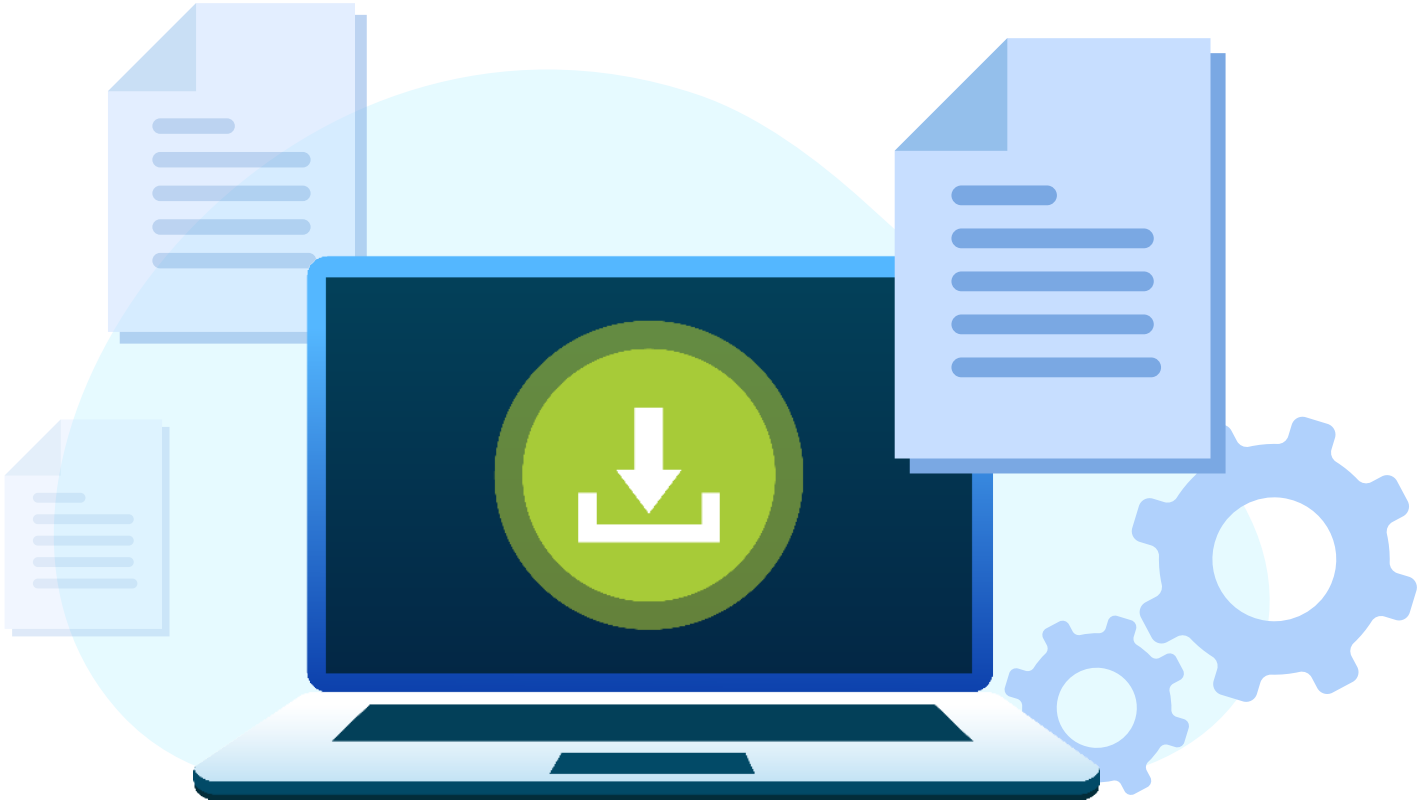
Use the bite-sized video links in this e-book to instantly watch the section you are reading.

SUMMARY

We now have our Modern toolkit application well on the way to feature parity with the existing Classic application. We have created a touch-friendly Messages grid that builds upon the existing data models and business logic. We have allowed users to tap a message and view its details and integrated common actions for messages.

Next, we will look into adding a mobile-friendly labels menu and implementing the interface for composing messages with the Modern toolkit.





Thank you for reading!

Part-4 of Building an Email Client from Scratch

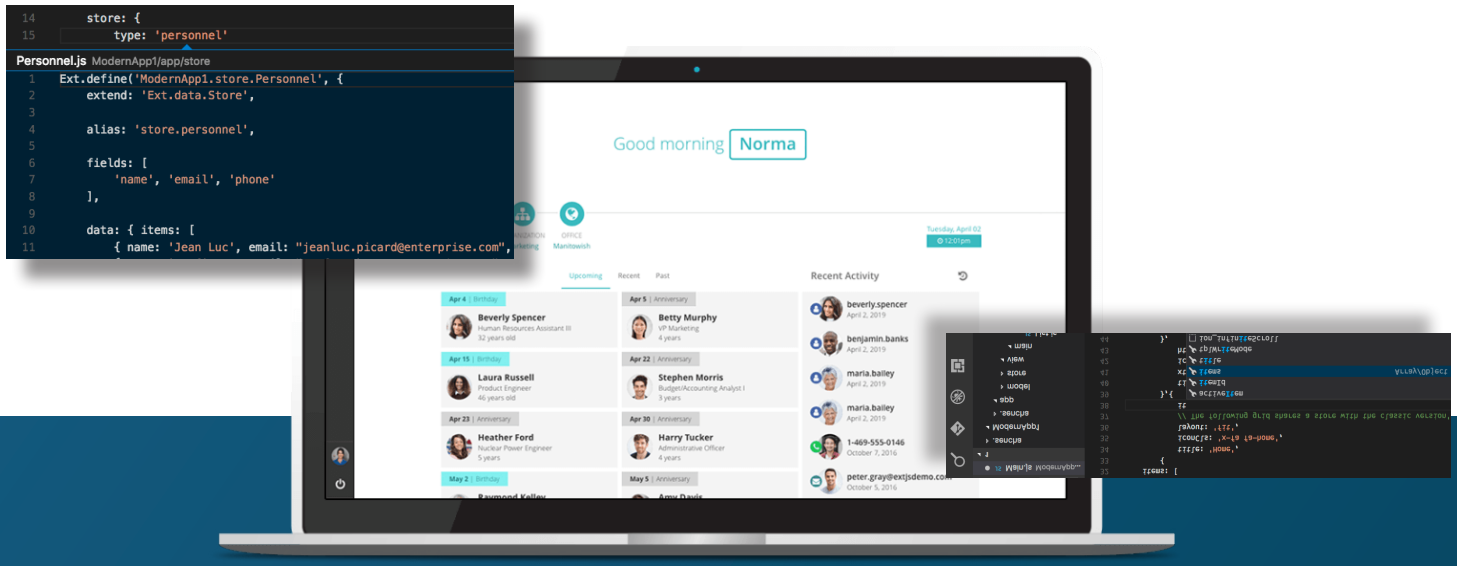
We hope you found it informative and helpful in your development projects. We have 3 more parts lined up to take you through the entire process of building an email client from scratch.

Download the Part-5 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