Sencha

Building an Email Client from Scratch

PART 1

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 this e-book, we will demonstrate how to lay the foundations for creating a well-structured and maintainable universal Ext JS application. Throughout the series, we will be building an Email Client that will allow users to view messages, perform actions on those messages, navigate message folders, and send new messages.

By the end of this e-book, we will have created a new application and set up the data structures that will support the rest of the application build. We will create simple Message Grid and Message Reader components that will be bound directly to these data structures. Some basic user interaction will be handled to allow navigation between the grid and the reader.

Key Concepts / Learning Points

- Structuring a Universal Application.
- Modelling simple data structures.
- Using data bindings and events.
- Creating a grid, a templated component, and a toolbar.

Code along with Stuart!

Start buddy coding with Stuart on-demand!

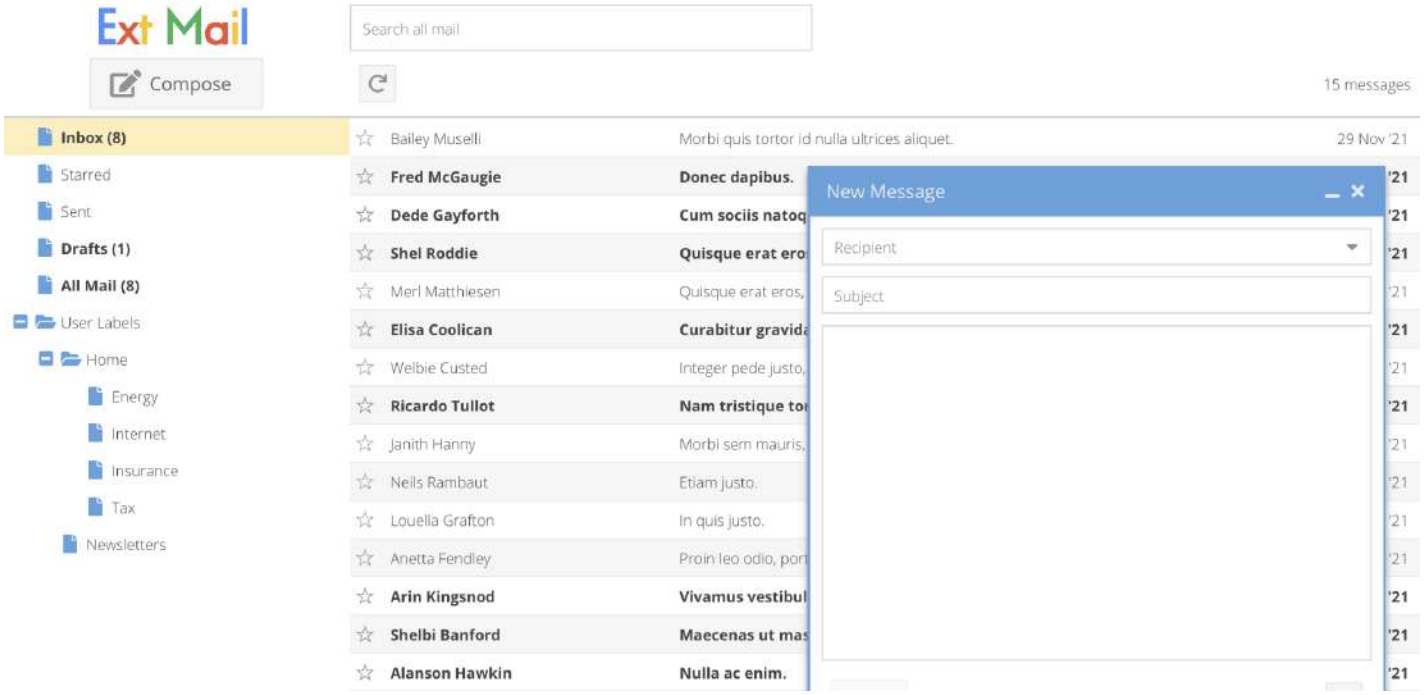
Use the bite-sized video links in this e-book to instantly watch the section you are reading.

Application Home Screen - Final Outcome

We can see how the application will look in the screenshots below, along with the Modern toolkit version of the application which we will develop later in the series.

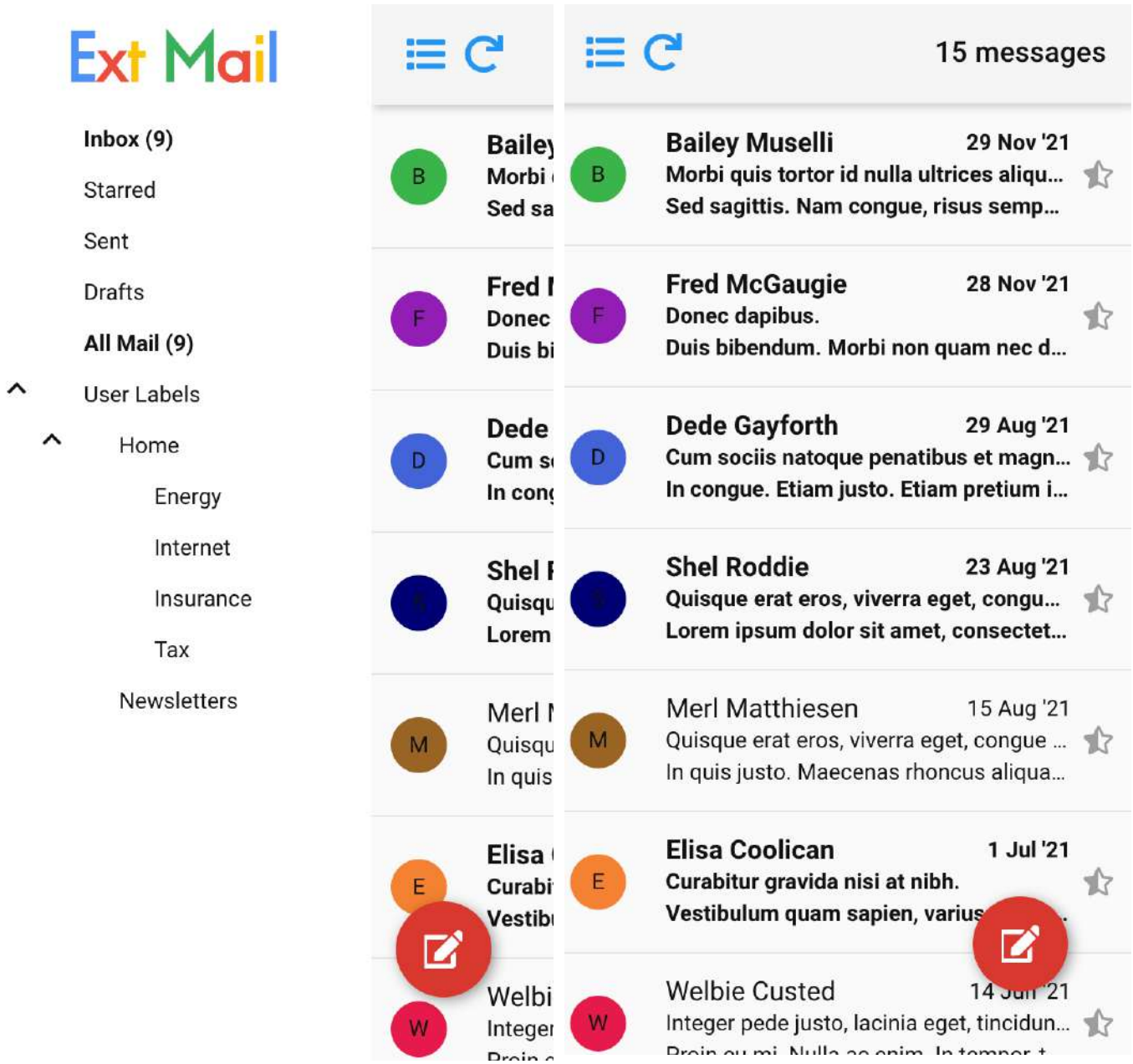
Desktop Version

Ext Mail application's home screen.



Ext Mail's compose window

Mobile Version



Ext Mail's Modern Toolkit interface

Components and data will be linked with bindings and custom events*

Let's get started!

We will start with a completely clean slate for this application but will assume you have the Ext JS framework downloaded and available locally, along with the latest Sencha Cmd installed.

You can download a trial of the Ext JS SDK and tools here:

Download the Free Trial

Application Structure

We start this application build with a barebones Universal Sencha application, created with the following Sencha Cmd command.

```
sencha generate app ExtMail ../ext-mail
```

By creating a Universal application we can build our application with the Classic and Modern toolkits, building on a base of shared, common code.

Shared Code

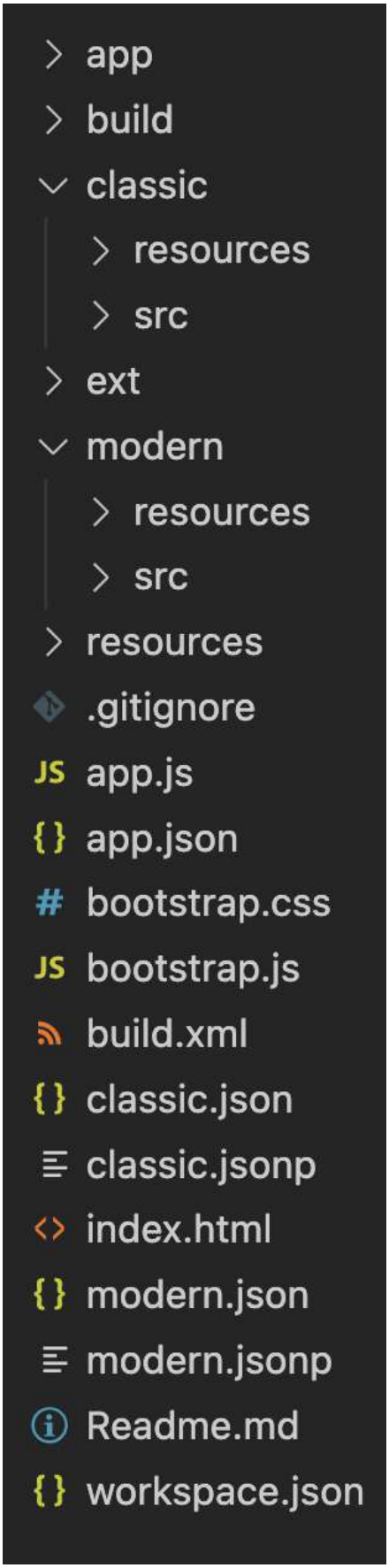
Any code that we want to share between toolkit apps we put into the **'app'**,in the same namespace matching folder structure we use everywhere in an Ext JS application.

For example: a class named **'ExtMail.controller.user.UserGridController'** would be located in **'app/controller/user/UserGridController.js'**

Toolkit Specific Code

All of the Classic toolkit code (i.e. code that references Classic toolkit components) will go in the **'classic/src'** folder, while the Modern toolkit code will be found in the **'modern/src'** folder. Both of these folders should follow the same namespace matching folder structure as the Shared Code's app folder.

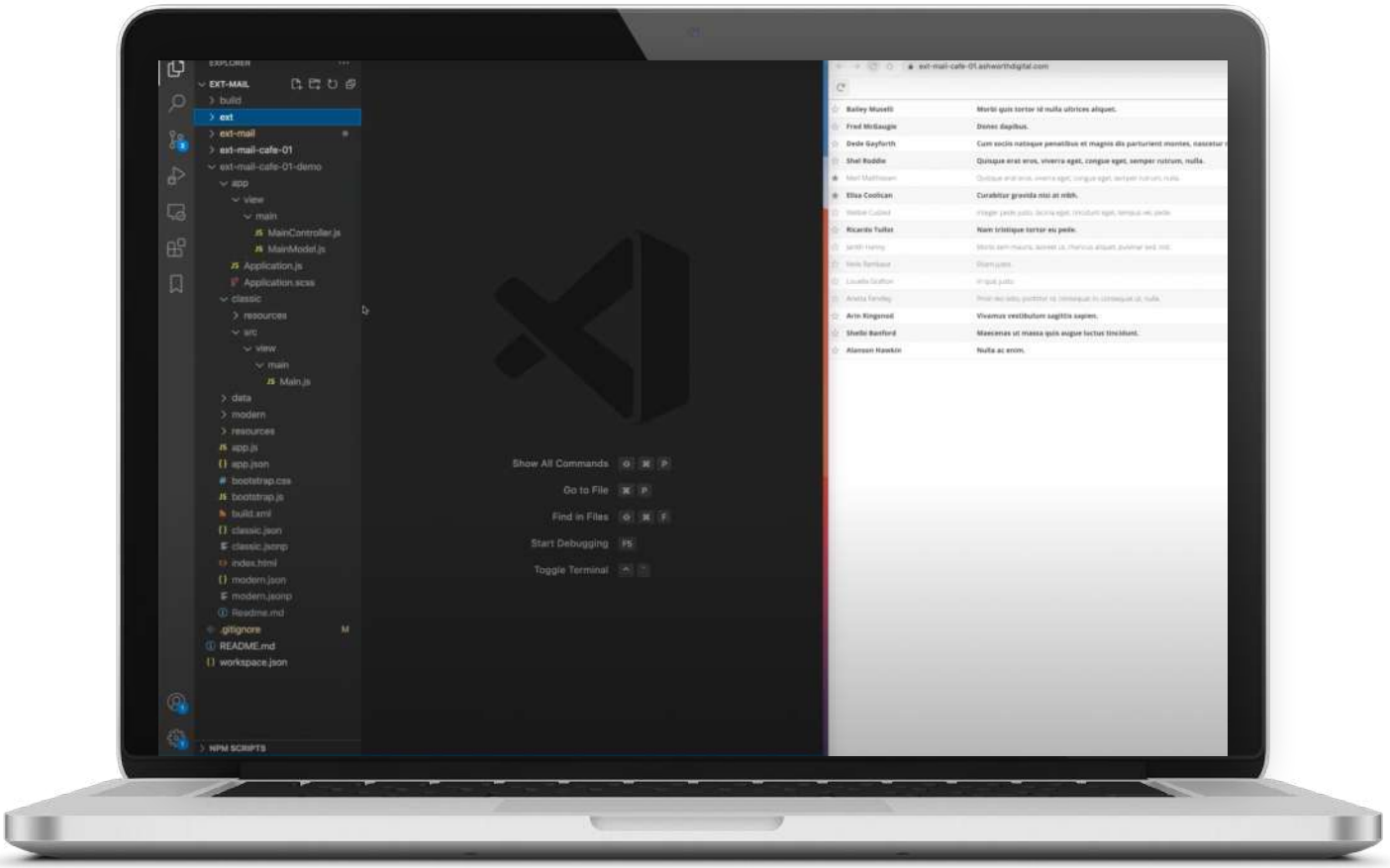
> The loader handles shared code by looking for a matching class in the toolkit folder first, then, if it doesn't find a match, look in the shared code folder for it instead. You can add additional class paths by updating the **'classPath'** array in the **'app.json'**.



The project's folder structure looks like this:

To begin with we will focus on building the Classic Toolkit application but we will architect the app in a way that will let us easily build out the Modern Toolkit app later on by extending our shared code..

To start us off we have a **'Main'** component which will render a Viewport on screen for us, along with a shared **'MainController'** and **'MainModel'** which are attached to the Main component.



Learn how to structure your application

Watch this Section!

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

Setting up Static Data Sources

At this stage we're not going to set up a real backend API but instead, just load in some static dummy data from JSON files stored in a folder named **'data'**. We have 3 data types:

Contacts
contacts.json

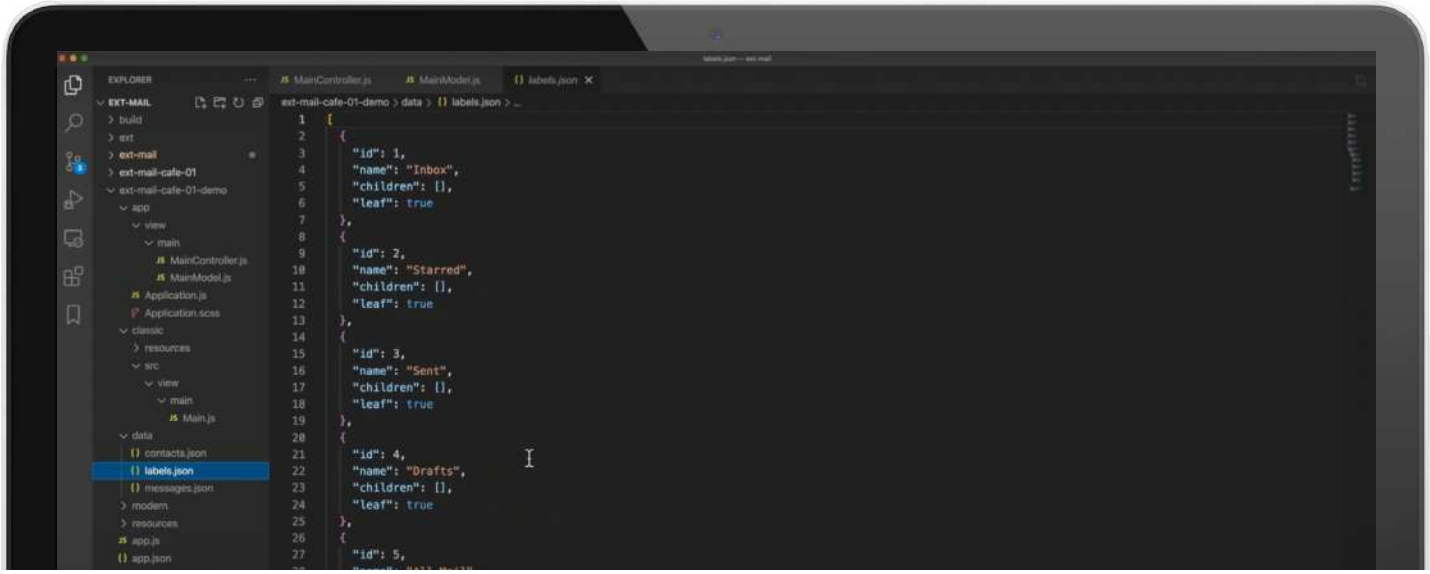
This is the first name, last name, and email address of the contacts that we will use to populate theRecipient field.

Labels
labels.json

This data is in a tree structure and will have the labels/folders that a message can be part of, for example, Inbox, Starred, Drafts, Sent, etc.

Messages
messages.json

Finally, this dataset is all the messages that will be loaded into the app and displayed.



Discover static data source setup techniques now

Watch this Chapter!

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

Data Models

With our static backend data ready we want to create an **'Ext.data.Model'** class to represent a single entity of each. These models will be shared between toolkit apps so will be located under the **'app/model'** folder.

Messages

model/Message.js

We start by creating a basic model class.

```
Ext.define('ExtMail.model.Message', {
    extend: 'Ext.data.Model',
});
```

We can then define the **'fields'** that our model has, mapping them to the fields in our dataset.

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

                return Ext.String.trim(firstName + ' ' + lastName);
            }
        },
        {
```

```
            name: 'email'
        },
        {
            name: 'date',
            type: 'date',
            dateFormat: 'c'
        },
        {
            name: 'subject'
        },
        {
            name: 'message'
        },
        {
            name: 'labels', // an array of ExtMail.enums.Labels
            type: 'auto',
            defaultValue: []
        },
        {
            name: 'unread',
            type: 'boolean'
        },
        {
            name: 'draft',
            type: 'boolean'
        },
        {
            name: 'outgoing',
            type: 'boolean'
        },
        {
            name: 'sent',
            type: 'boolean'
        }
    ]
```

The **'name'** we define will be used to read the property from the source data and for non-string data types, we define the **'type'** property so that we convert the data into the correct type. If this is omitted a type of **'auto'** will be used.

> When just defining the **'name'** of the field then we can simplify it to a single string containing the name. The framework will apply all the defaults for us.

We will want to display the message recipient's full name (i.e. first and last name combined) in various places in the application so we make use of the **'calculate'** config which lets us build a new field based on the others. In this case, we combine the first and last names so we can reference the **'fullName'** field as we would any of the other fixed fields. This **'calculate'** function will be re-executed if any of the dependent fields that are referenced in it are updated.

For our **'labels'** field we are expecting an array of IDs that doesn't have its own type so we explicitly tell it to use **'auto'** and give it a **'defaultValue'** of an empty array so we always have something to work with.

As well as the **'fields'** definition we add the **'hasLabel'**, **'addLabel'**, and **'removeLabels'** helper functions to allow us to easily manipulate the **'labels'** array in a standard way.

```
    hasLabel: function(labelId) {
        var labels = this.get('labels') || [];
        return labels.indexOf(labelId) >= 0
    },
    addLabel: function(labelId) {
        var labels = this.get('labels') || [];
        labels.push(labelId);
        this.set('labels', Ext.clone(labels)); // clone so it triggers an
update on the record
    },
    removeLabel: function(labelId) {
        var labels = this.get('labels') || [];
        labels = Ext.Array.remove(labels, labelId);
```

```
        this.set('labels', Ext.clone(labels)); // clone so it triggers an
update on the record
    }
```

> When we manipulate the array in the **'labels'** field we clone it and then **'set'** it back to the model instance. This is so the operation is recognised as an update and forces any bound UI components to update themselves. The add/remove operations happen in-situ (i.e. on the same array instance) and the model won't recognise deep changes, but it will recognise if an entirely new array instance is given.

Finally, we want all messages created in the browser to have a unique ID so we use the **'Ext.data.identifier.Uuid'** class to generate a Guid type identifier on all new messages.

```
  requires: [
        'Ext.data.identifier.Uuid'
    ],
    identifier: 'uuid',
```

Contact

model/Contact.js

The Contact model is very basic, making use of the simple array of strings syntax for the **'fields'** config.

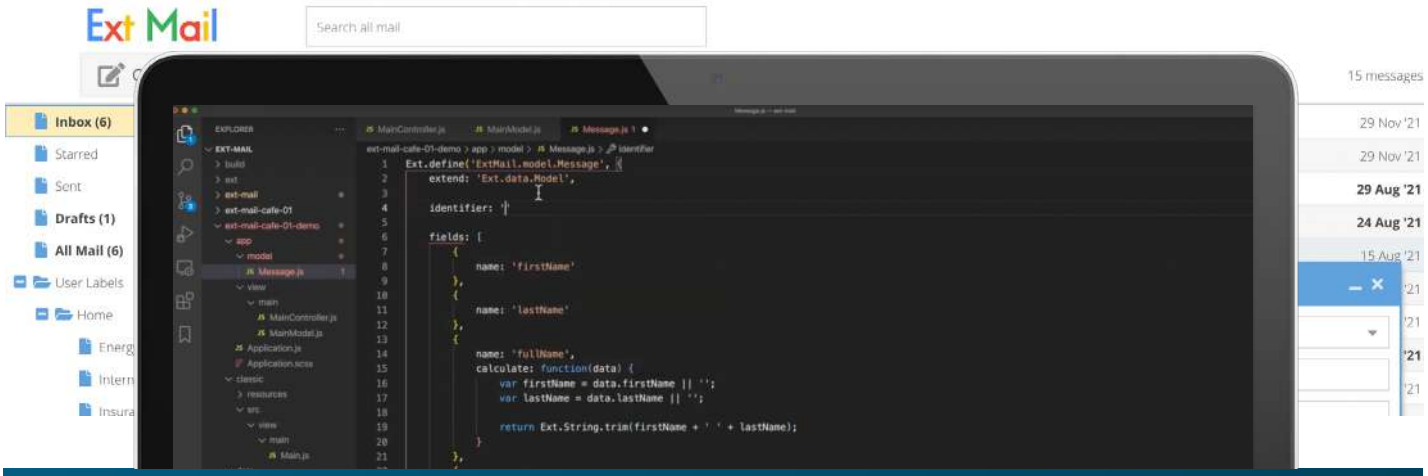
```
Ext.define('ExtMail.model.Contact', {
    extend: 'Ext.data.Model',
    fields: [
        'name', 'email', 'phone'
    ]
```

Label

model/Label.js

The Label entities will form part of a tree structure so instead of the regular **Ext.data.Model'** class we will extend the **'Ext.data.TreeModel'** class which will add some necessary functionality for handling nested model instances. This happens under-the-hood and so all we need to do is define the fields that each entity in the tree structure will have.

```
Ext.define('ExtMail.model.Label', {
    extend: 'Ext.data.TreeModel',
    fields:
        'name',
        { name: 'unreadCount', type: 'int' }
    ]
});
```



Explore how to work with data models

Watch this Part!

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

Data Stores

With the data models in place we can define stores that will hold and manipulate collections of them. These stores will be used to bind to our UI components and ensure the data and UI are kept in sync. Like the models the stores can be shared across toolkits and so will be located in the **'app/store'** folder.

Messages

store/Messages.js

his store extends the **'Ext.data.Store'** base class and defines an **'alias'** of **'store.Messages'**. This alias allows us to reference it from configuration objects - we'll see this in use later.

> Aliases are shorthand names that allow us to reference classes from configuration objects. You will likely be familiar with component aliases known as **'xtype`s'**

When creating your own Component classes you would define its alias by defining alias: **'widget.MyComponent'** or by using the shorthand of xtype: **'MyComponent'.** For non-component classes you would use alias and prefix the value with the category of class, for example "widget", "store", "model" and "proxy".

The store needs to know what type of models it will contain, so we assign the name of the model class to the **'model'** config. We want to have this store load data as soon as it is instantiated so we set the **'autoLoad'** property to true, and we tell the store to always keep the Message model instances in chronological order by specifying a **'sorters'** configuration

Finally, we tell the store how it can load data by defining a **'proxy'**. In this case, we want to load data via AJAX, pointing it to the **'url'** of our dummy data file. We also define a **'reader'** so the proxy knows what format the data will be received in and how to extract the collection of model instances, in this case from the **'rows'** property.

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

    alias: 'store.Messages',

    model: 'ExtMail.model.Message',

    autoLoad: true,

    sorters: [
        {
            property: 'date',
            direction: 'DESC'
        }
    ],

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

Contacts

store/Contacts.js

The Contacts store has an identical setup to the Messages one with an alias, model, and proxy defined. Very often stores will be simple like this but defining them as their own class allows them to extend their functionality with helper functions.

```
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'
        }
    }
});
```

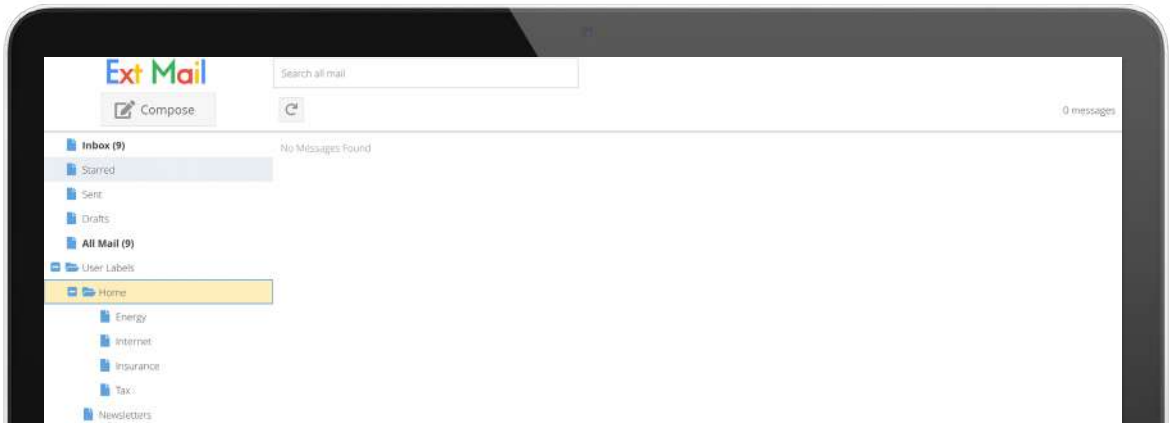
Labels

store/Labels.js

As we mentioned before the Labels are set out in a tree structure. To achieve this we must use the **'Ext.data.TreeStore'** base class which allows the nested data to be read and stored correctly. Like the regular store we include an **'alias'**, **'model'** and **'proxy'.** We also add a **'root'** configuration which defines what the root node of the tree looks like and will be where all the loaded data branches off from.

```
Ext.define('ExtMail.store.Labels', {
    extend: 'Ext.data.TreeStore',
```

```
    alias: 'store.Labels',

    model: 'ExtMail.model.Label',

    root: {
        name: 'Test',
        expanded: true
    },

    proxy: {
        type: 'ajax',
        url: 'data/labels.json'
    }
});
```



Learn to create data stores

Watch this Section!

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

Creating Common View Controller

Next we set up our main View Controller class. We are aiming to share as much code as possible between our Classic and Modern toolkit applications so we start by creating a base View Controller that will be extended by each of the toolkits so toolkit-specific logic can be added when needed.

We already have a **'view/main/MainController.js'** file which we will rename to **'MainControllerBase.js'** and update its contents appropriately:

```
Ext.define('ExtMail.view.main.MainControllerBase', {
    extend: 'Ext.app.ViewController'
});
```

We can then create our toolkit MainController classes in the toolkit folders and define them as extending this base class.

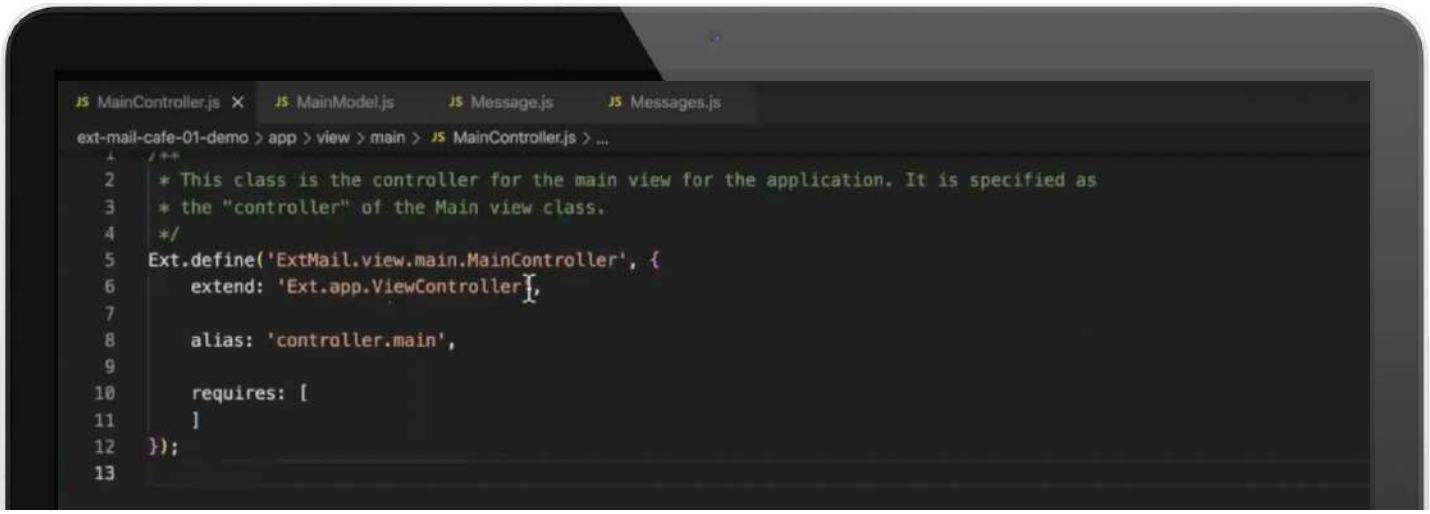
```
Ext.define('ExtMail.view.main.MainController', {
    extend: 'ExtMail.view.main.MainControllerBase',

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

Note that we move the **'alias'** to the sub-class so that we can reference it rather than the base-class. This is used in the **'Main.js'** class to link the two together.

```
Ext.define('ExtMail.view.main.Main', {
    extend: 'Ext.panel.Panel',
    xtype: 'app-main',
```

```
    requires: [
        'ExtMail.view.main.MainController'
    ],

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

> As we said before the loader will initially look for the **'MainControllerBase'** class in the classic toolkit's **'view/main'** folder but when it isn't found it will look in the shared **'app'** folder.



Learn how to create a common view controller

Watch this Step!

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

Creating Common View Model

The main View Model will be used to store the bulk of the application's state. This state will be common between the Classic and Modern toolkits so we can elevate that to the shared **'app'** folder and only define it once.

> In our application there won't be any toolkit specific state to store but if there was you could follow a similar pattern to the ViewController and create a base version which would then be extended in each toolkit.

```
Ext.define('ExtMail.view.main.MainModel', {
    extend: 'Ext.app.ViewModel',

    alias: 'viewmodel.main',

    requires: [
    ],

    data: {
    },

    formulas: {
    },

    stores: {
    },

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

`data´

The data property holds an object with simple key/value pairs of state data. These items can be bound to our views and updated during user interaction. Often this will be small pieces of state data or simple collections of data items.

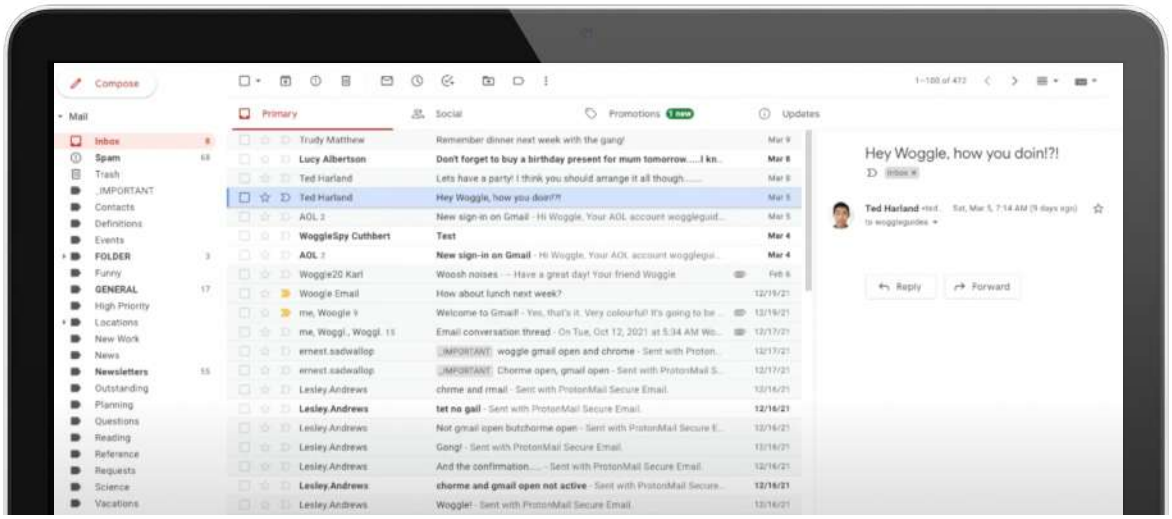
`formulas´

These are calculated fields that are linked to **'data'** properties or other **'formulas'** and are re-evaluated if any of the linked fields' values change.

`stores´

The **'stores'** object has definitions for stores that will be instantiated in the View Model and can be bound to view components.

Finally, our constructor is where we can set up any other bindings and event handlers on our View Model or stores.



Learn how to create a common view model

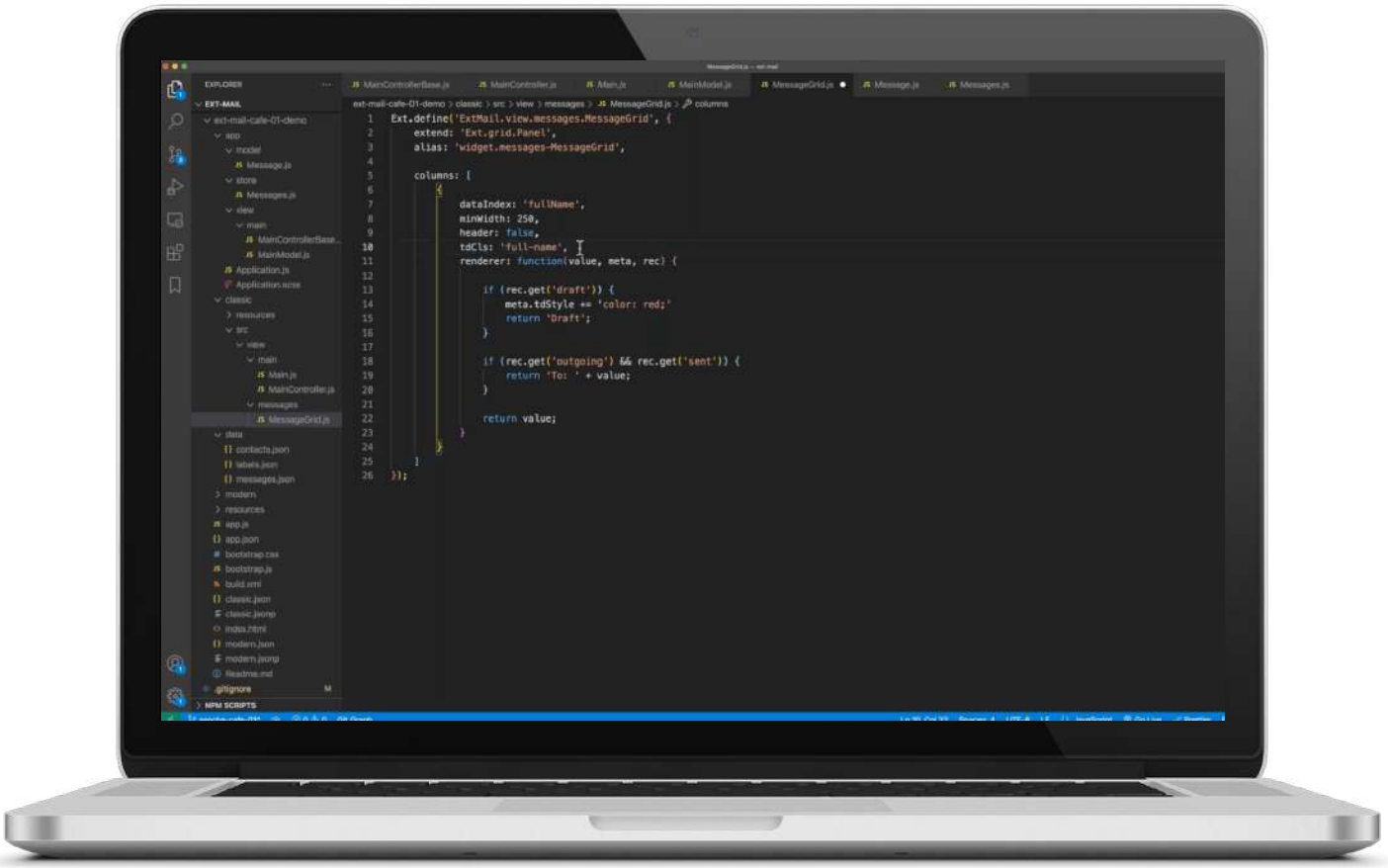
Watch this Section!

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

Messages Store

The main store in our applications is the one that will load and contain our list of messages. We define this by adding a key to our **'stores'** object and giving it a configuration object. This key will be the name we use when binding the store to any view component.

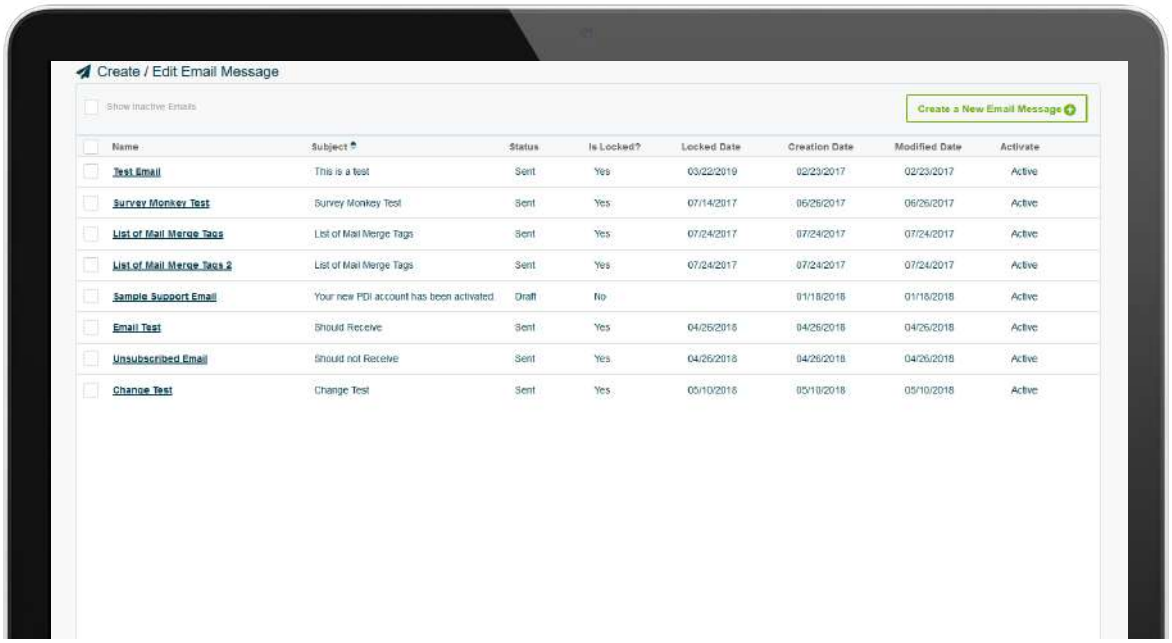
```
...
stores: {
    messages: {
        type: 'Messages'
    },
},
...
```



Creating the Messages Grid

Now that we have the code set up to support our views we can create a grid to render our Messages to the screen. We start by creating a **'ExtMail.view.messages.MessageGrid'** class in our Classic toolkit folder.

```
Ext.define('ExtMail.view.messages.MessageGrid', {
    extend: 'Ext.grid.Panel',
    alias: 'widget.messages-MessageGrid'
});
```



Discover how to create the messages grid

Watch this Phase!

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

Columns

A grid component needs to at least define the columns to show and a data store to render, so we start by defining columns to show the message sender's full name, the subject and the sent date.

```
…
    columns: [
        {
            dataIndex: 'fullName',
            minWidth: 250,
            header: false,
            tdCls: 'full-name'
        },
        {
            dataIndex: 'subject',
            flex: 1,
            header: false
        },
        {
            xtype: 'datecolumn',
            dataIndex: 'date',
            header: 'Received',
            width: 100,
            header: false,
            align: 'end',
            format: 'j M \'y'
        }
    ]
    …
```

If no **'xtype'** is specified then a column will config will instantiate the **`Ext.grid.column.Column'** class, but if we have a data type that needs special formatting we can use one of the various sub-classes available.

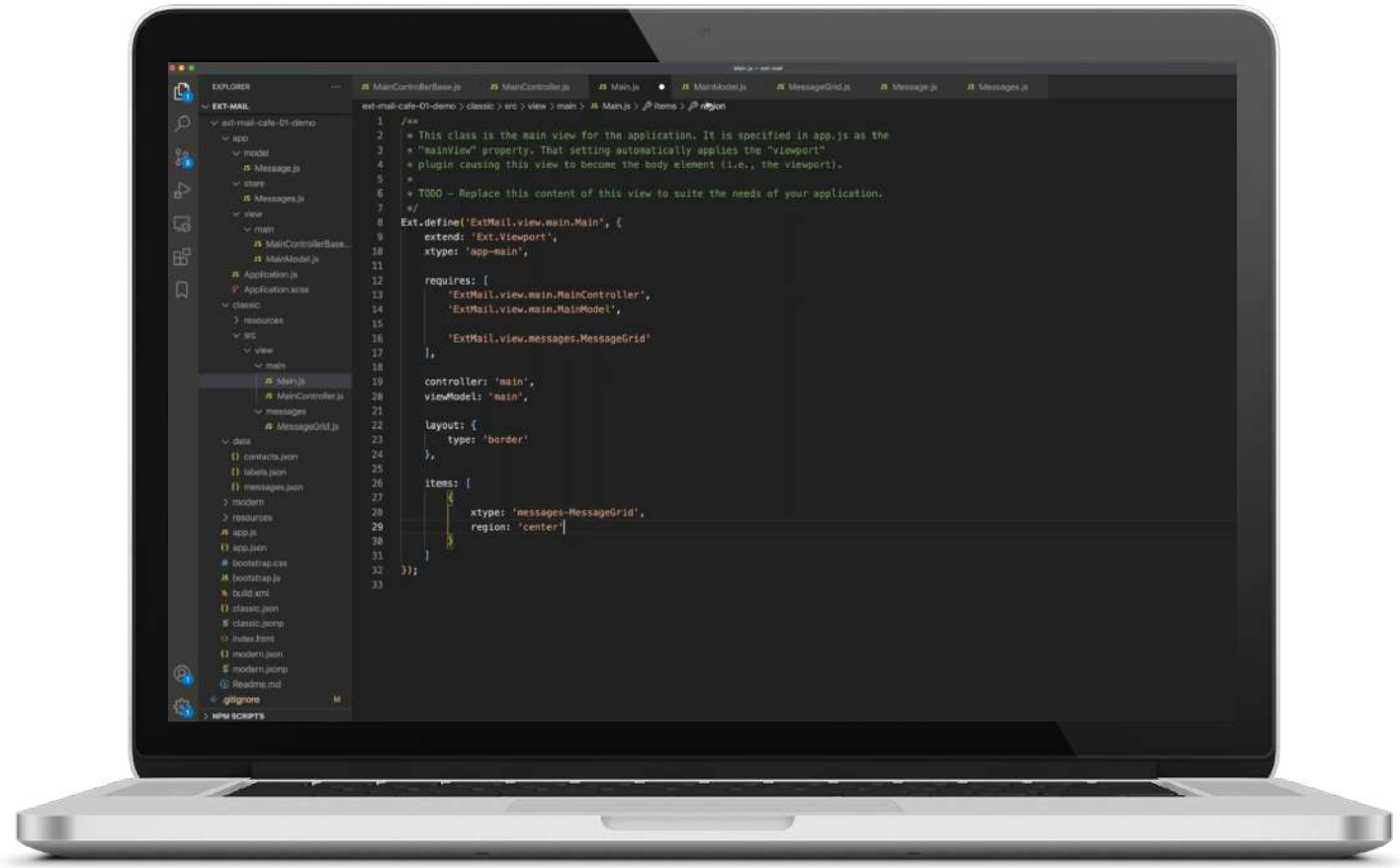
We define the **'dataIndex'** to tell the column which property from the row's record to display in that cell.

The **'datecolumn'** class is one of these sub-classes that allows us to define a **'format'** which will be used to output a formatted date into the grid cell.

With the columns defined we can drop our MessageGrid into our Main component and render it to screen. We use the **'messages-MessageGrid'** alias to add it to our Main component's **'items'** array.

```
Ext.define('ExtMail.view.main.Main', {
    extend: 'Ext.panel.Panel',
    xtype: 'app-main',

    requires: [
        'Ext.plugin.Viewport',

        'ExtMail.view.main.MainController',
        'ExtMail.view.main.MainModel',

        'ExtMail.view.messages.MessageGrid',
    ],

    plugins: 'viewport',

    controller: 'main',
    viewModel: 'main',

    layout: {
```

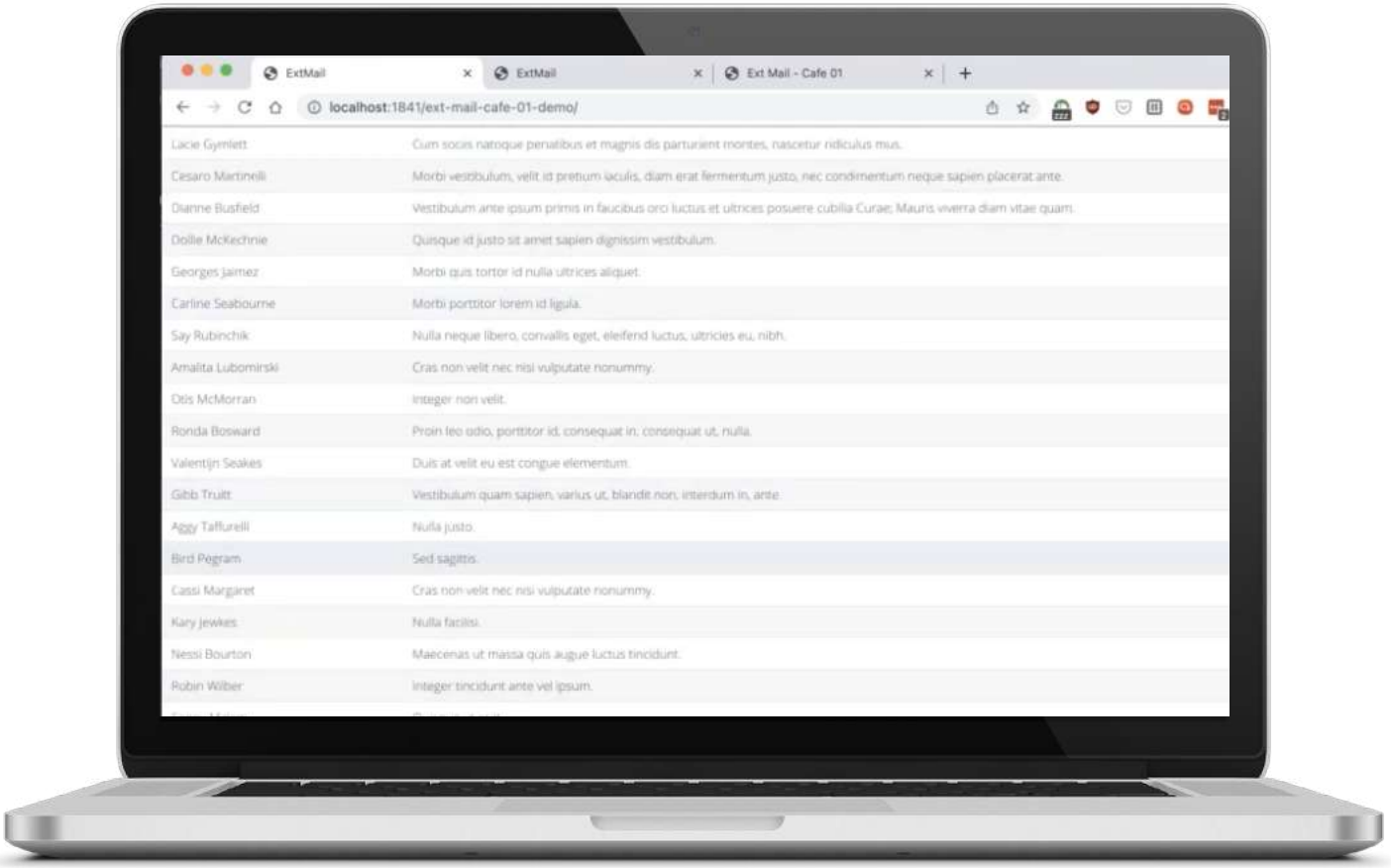
```
        type: 'border'
    },

    items: [
        {
            xtype: 'messages-MessageGrid',
            region: 'center'
        }
    ]
});
```



Binding to a Data Store

All we need to do now is give the grid a data store to render. We do this using the **'bind'** config and pass it the name of the store we created in our View Model earlier.

```
…
    {
        xtype: 'messages-MessageGrid',
        region: 'center',
        bind: {
            store: '{messages}'
        }
    }
    …
```



Styling Specific Rows

At the moment we can't differentiate between read and unread messages in our grid, so we want to make the unread messages bold. To do this we add a **'getRowClass'** method to the **'viewConfig'** configuration object.

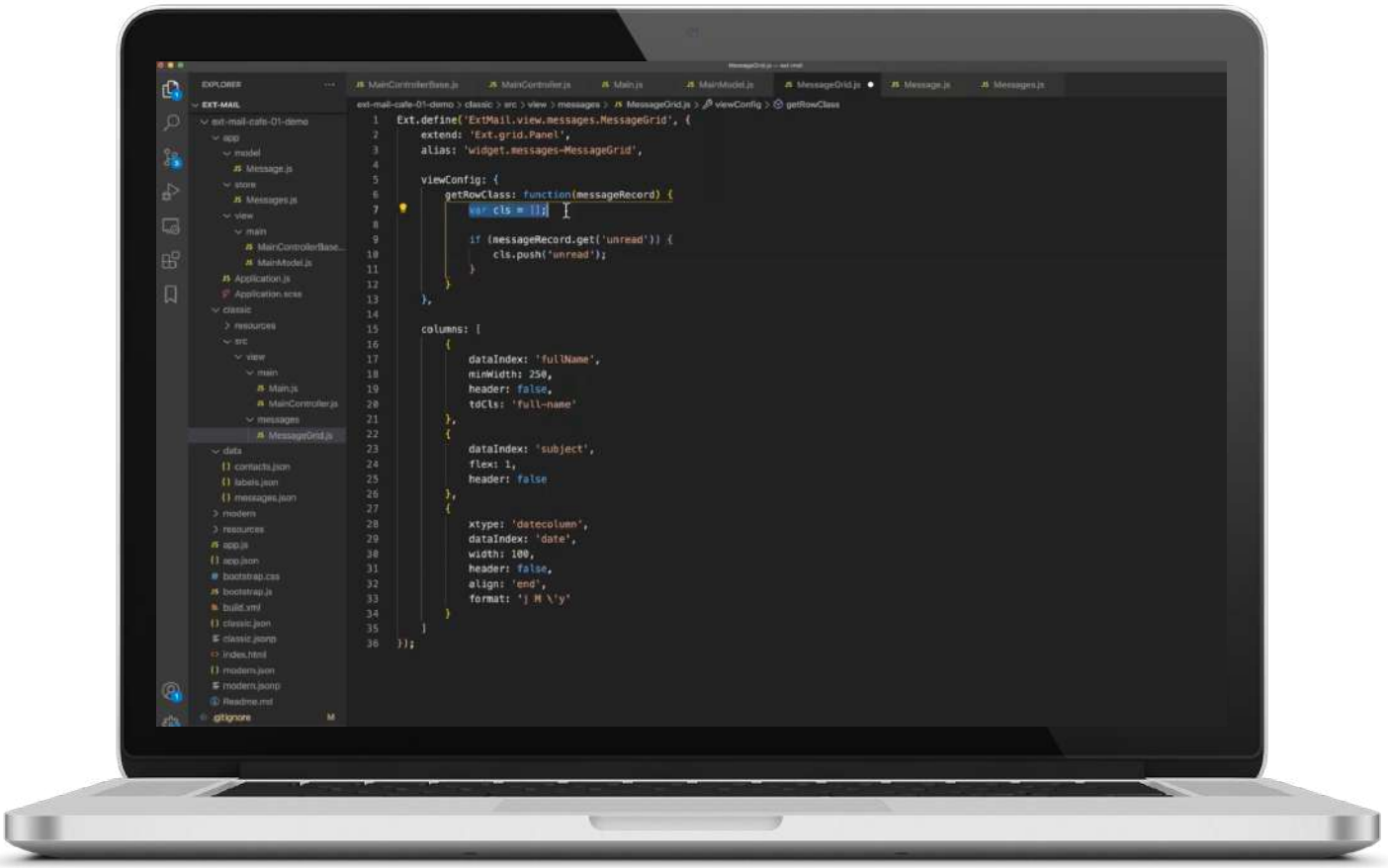
```
viewConfig: {
        getRowClass: function(messageRecord) {
            var cls = [];

            if (messageRecord.get('unread')) {
                cls.push('unread');
            }

            return cls.join('',);
        }
```

This function should return a string containing one or more CSS classes that will be applied to the row's HTML element. In this case, we want to add the **'unread'** class if the record's **'unread'** property is true. We couple this with some custom styling added to an SCSS file located in **'MessageGrid.scss'** as a sibling to the **'MessageGrid.js'** file. Sencha Cmd will automatically detect this file and compile its contents into CSS.

```
.message-grid {
    .unread {
        .x-grid-cell-inner {
            font-weight: bold;
        }
    }
}
```

We also add the **'message-grid'** CSS class to the grid so we can keep our CSS rules scoped to that component.

☆	**Bailey Muselli**	**Morbi quis tortor id nulla ultrices aliquet.**	**29 Nov '21**
☆	**Fred McGaugie**	**Donec dapibus.**	**29 Nov '21**
☆	**Dede Gayforth**	**Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.**	**29 Aug '21**
☆	**Shel Roddie**	**Quisque erat eros, viverra eget, congue eget, semper rutrum, nulla.**	**24 Aug '21**
☆	Merl Matthiesen	Quisque erat eros, viverra eget, congue eget, semper rutrum, nulla.	15 Aug '21
☆	**Elisa Coolican**	**Curabitur gravida nisi at nibh.**	**1 Jul '21**
☆	Welbie Custed	Integer pede justo, lacinia eget, tincidunt eget, tempus vel, pede.	14 Jun '21
☆	**Ricardo Tullot**	**Nam tristique tortor eu pede.**	**21 May '21**
☆	Janith Hanny	Morbi sem mauris, laoreet ut, rhoncus aliquet, pulvinar sed, nisl.	2 May '21
☆	Neils Rambaut	Etiam justo.	5 Apr '21
☆	Louella Grafton	In quis justo.	14 Mar '21
☆	Anetta Fendley	Proin leo odio, porttitor id, consequat in, consequat ut, nulla.	28 Feb '21
☆	**Arin Kingsnod**	**Vivamus vestibulum sagittis sapien.**	**13 Jan '21**
☆	**Shelbi Banford**	**Maecenas ut massa quis augue luctus tincidunt.**	**4 Jan '21**
☆	**Alanson Hawkin**	**Nulla ac enim.**	**3 Jan '21**



Creating the Message Reader

With the Message Grid in place, we want to be able to click a message and be taken to a new screen which will display the message in full.

We create a new class called **'ExtMail.view.reader.MessageReader'** extending the **'Ext.Panel'** class and giving it a basic HTML template string to display the message details.

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

    cls: 'message-reader',
    tpl: [
        '<div class="subject">{subject}</div>',
        '<div class="info">',
        '    <div class="sender">',
        '        <span class="name">{fullName}</span> <span class="email">({e-
mail})</span>',
        '    </div>',
        '    <div class="date">{date:date("D, j M Y, H:i")}</div>',
        '</div>',

        '<div class="message">{message}</div>'
    ],
    data: {}
});
```

> The **'tpl'** config is turned into an **'Ext.XTemplate'** instance and can be defined as a String or an Array of Strings. We can also pass an object as the array's last.

We couple this with an SCSS file alongside it with the same name- **'MessageReader.scss'** - which will give us some nice styling for each of the details:

```
.message-reader {
    padding: 2rem;

    .subject {
        font-size: 1.5rem;
        font-weight: bold;
        margin-bottom: 1.5rem;
    }

    .info {
        display: flex;

        .sender {
            flex: 3;

            .name {
                font-weight: bold;
            }
        }

        .date {
            flex: 1
        }
    }

    .message {
        font-size: 1rem;
        margin: 1.5rem 0;
    }

}
```

Explore how to create the message reader

Watch this Step!

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

Showing the Message Details

Now we can set up our app to switch to the MessageReader component when the user clicks a MessageGrid row.

First, we must create a wrapping component in our Main component allowing us to flip between the Grid and Reader and only have one on-screen at a time. To do this we use the Card layout.

```
{
    xtype: 'panel',
    region: 'center',
    layout: 'card',
    reference: 'messagesWrapper',
    items: [
        {
            xtype: 'messages-MessageGrid',
            bind: {
                store: '{messages}'
            }
        },
        {
            xtype: 'reader-MessageReader',
            bind: {
                data: '{selectedMessage}'
            }
        }
    ]
}
```

We bind the MessageReader's **'data'** property (which is merged with the **'tpl'**) to the **'selectedMessage'** data property. This property is defined on the MainViewModel and will track the Message record that has been clicked on. We default this to **'null'**.

```
// MainModel.js
data: {
    selectedMessage: null
}
```

We can now use the **'itemclick'** event of the grid to handle a user clicking on a message. We attach a handler to the event and attach it to a method that we will define in the MainControllerBase class using a string.

```
{
        xtype: 'messages-MessageGrid',
        bind: {
            store: '{messages}'
        },
        listeners: {
            itemclick: 'onMessageClick'
        }
    }
```

In the **'onMessageClick'** function we're going to set the **'selectedMessage'** with the Message record that was clicked.

```
// MainControllerBase.js
onMessageClick: function(grid, messageRecord, rowEl, index, e) {
    this.getViewModel().set('selectedMessage', messageRecord);
}
```

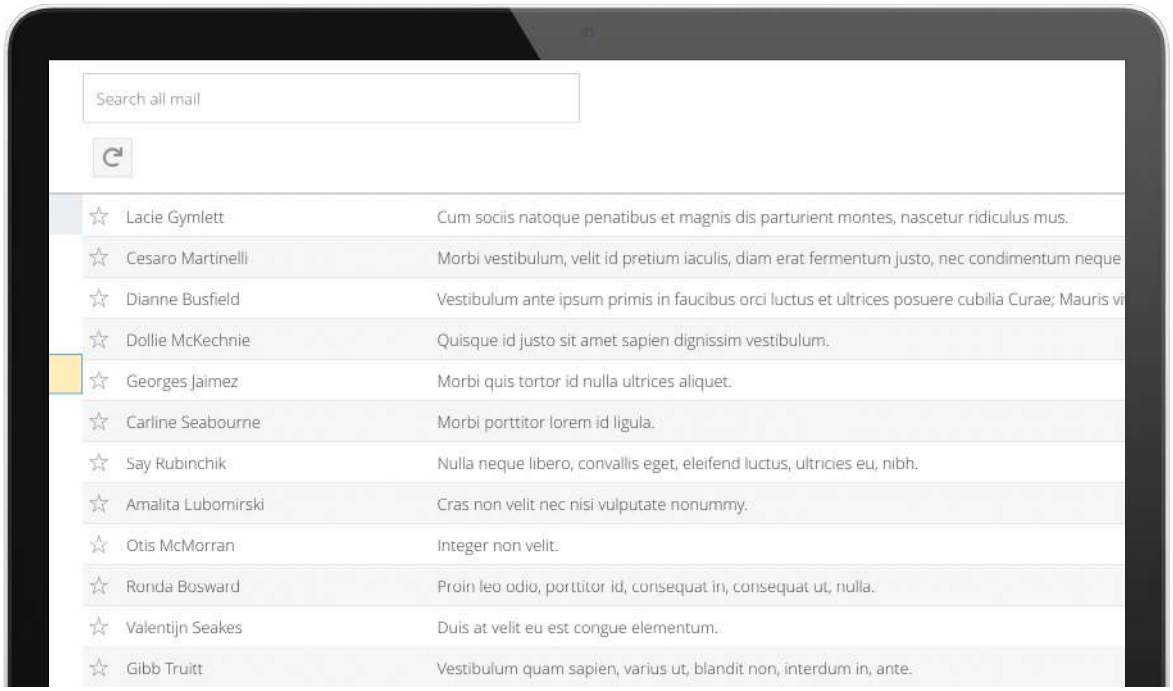
Finally, we need to tell the card layout to switch between the Grid and the Reader when the **'selectedMessage'** property has a value. To do this we define a **'formula'** which will return the index of the card we want to display, i.e. 0 (the Message Grid) when **'selectedMessage'** is empty, and 1 (the Message Reader) when **'selectedMessage'** has a value.

```
// the index of the Message Reader card layout to show. 0 = MessageGrid; 1 =
MessageReader
messageCardIndex: function(get) {
    return get('selectedMessage') ? 1 : 0;
}
```

> By using the passed in **'get'** function to lookup values that we want the formula to react to, the framework will always reevaluate the formula if these data properties change

We can take this formula and bind it to the wrapper component's **'activeItem'** property. This means that the change in the **'selectedMessage'** property will trigger the card layout to change.

```
bind: {
  activeItem: '{messageCardIndex}';
}
```



Creating a Dynamic Toolbar

We're now able to move to a Message and view its contents but we can't yet move back again. To solve this problem we will create a Toolbar whose contents will be dynamic and change based on the current context.

For example: When on the Message Grid we want to show a Refresh button, but when on the Message Reader we want to show a Back button.

We create a new class called **'ExtMail.view.messages.MessageToolbar'** and extend the **'Ext.toolbar.Toolbar'** class.

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

    defaultListenerScope: true,
    items: [
        {
            tooltip: 'Refresh',
            iconCls: 'x-fa fa-redo',
            handler: 'onRefreshClick',
            bind: {
                hidden: '{selectedMessage}'
            }
        },
        {
            tooltip: 'Back',
            iconCls: 'x-fa fa-arrow-left',
            handler: 'onBackClick',
            hidden: true, // hide from start
            bind: {
                hidden: '{!selectedMessage}'
```

```
            }
        }
    ],

    onRefreshClick: function () {
        this.fireEvent('refresh');
    },

    onBackClick: function () {
        this.fireEvent('back');
    }
});
```

We create two buttons - a Refresh and a Back - and bind their **'hidden'** property to the **'selectedMessage'** value. This property will be evaluated as truthy or falsey so we can use this to show and hide the buttons when a message is selected or not. For the Back button, we negate the value using the **'!'** operator.

We define the **'handler'** functions as strings that will reference the methods with these names in the component. Each of these handler functions will raise a custom event that can be bound to our parent component.

> By setting the **'defaultListenerScope'** to true the **'handler'** values will be evaluated within the context of the toolbar component. If this was set to false (or omitted) then the component would look up the tree to a View Controller for methods with matching names. This is useful if you want to try and keep your component self-contained.

Next we add the new toolbar as a **'dockedItem'** of the wrapper component within our Main class. By making it a docked component it will always be at the top of the view and will be visible regardless of which of the cards is visible.

```
{
    xtype: 'panel',
    region: 'center',
    layout: 'card',
    reference: 'messagesWrapper',
    bind: {
        activeItem: '{messageCardIndex}'
    },
    dockedItems: [
        {
            xtype: 'messages-MessagesToolbar',
            dock: 'top',
            height: 56,
            listeners: {
                refresh: 'onRefreshMessages',
                back: 'onBackToMessagesGrid'
            }
        }
    ],
    ...
}
```

We can see our two custom events being referenced in the **'listeners'** object and being given handler functions that we will add in our MainControllerBase class.

```
// MainControllerBase.js
onRefreshMessages: function() {
    this.getViewModel().getStore('messages').reload();
},
```

```
onBackToMessagesGrid: function() {
    this.getViewModel().set('selectedMessage', null);
}// MainControllerBase.js
onRefreshMessages: function() {
    this.getViewModel().getStore('messages').reload();
},

onBackToMessagesGrid: function() {
    this.getViewModel().set('selectedMessage', null);
}
```

The **'onRefreshMessages'** method simply finds the **'messages'** store in the View Model and reloads it. This will automatically refresh the grid and display the original data.

The **'onBackToMessagesGrid'** will reset the **'selectedMessage'** state to null which will cause the **'messageCardIndex'** formula to reevaluate and change the card's **'activeItem'**.

Learn how to create a dynamic toolbar

Watch this Stage!

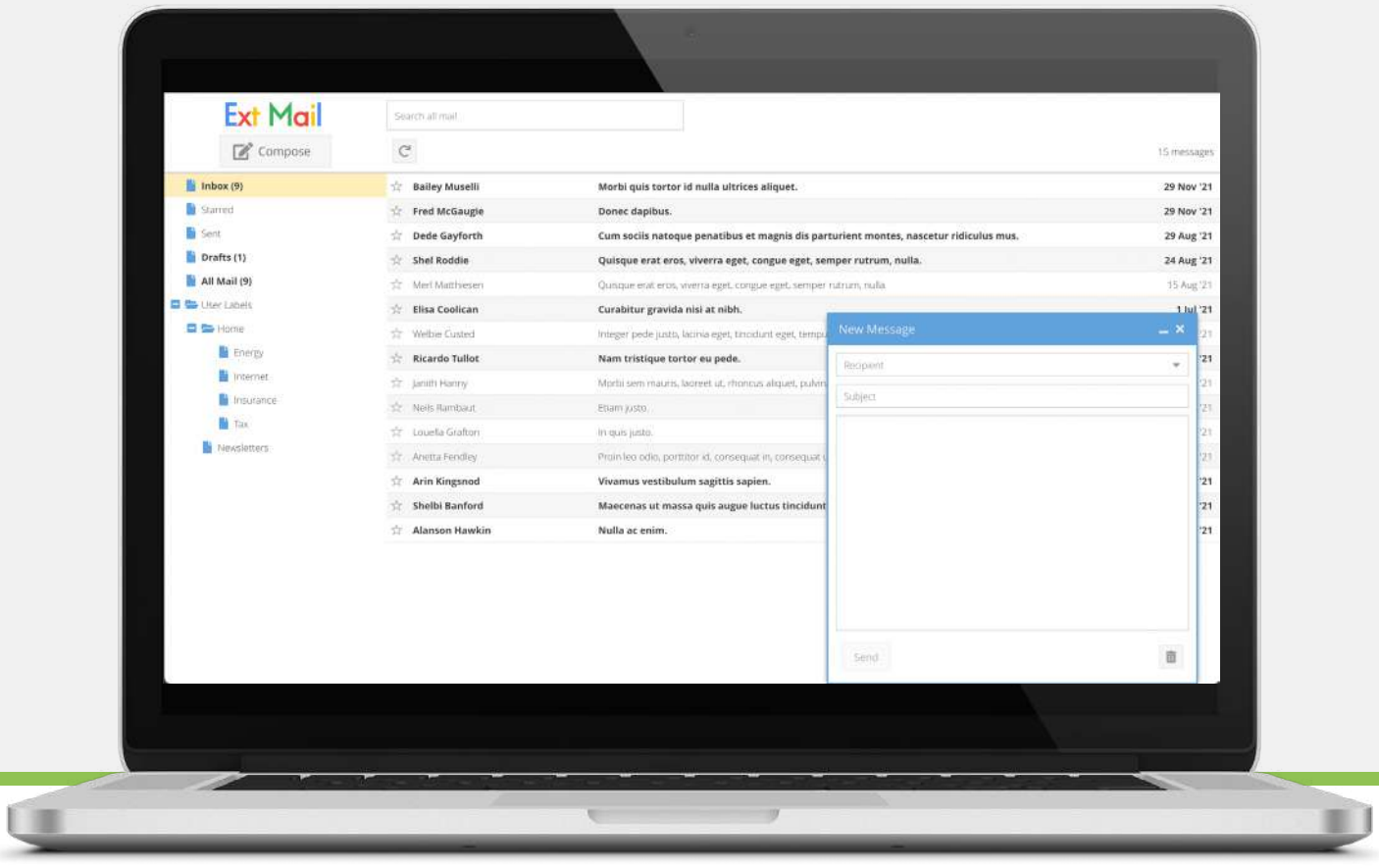
Use the bite-sized video links in this e-book to instantly watch the section you are reading.

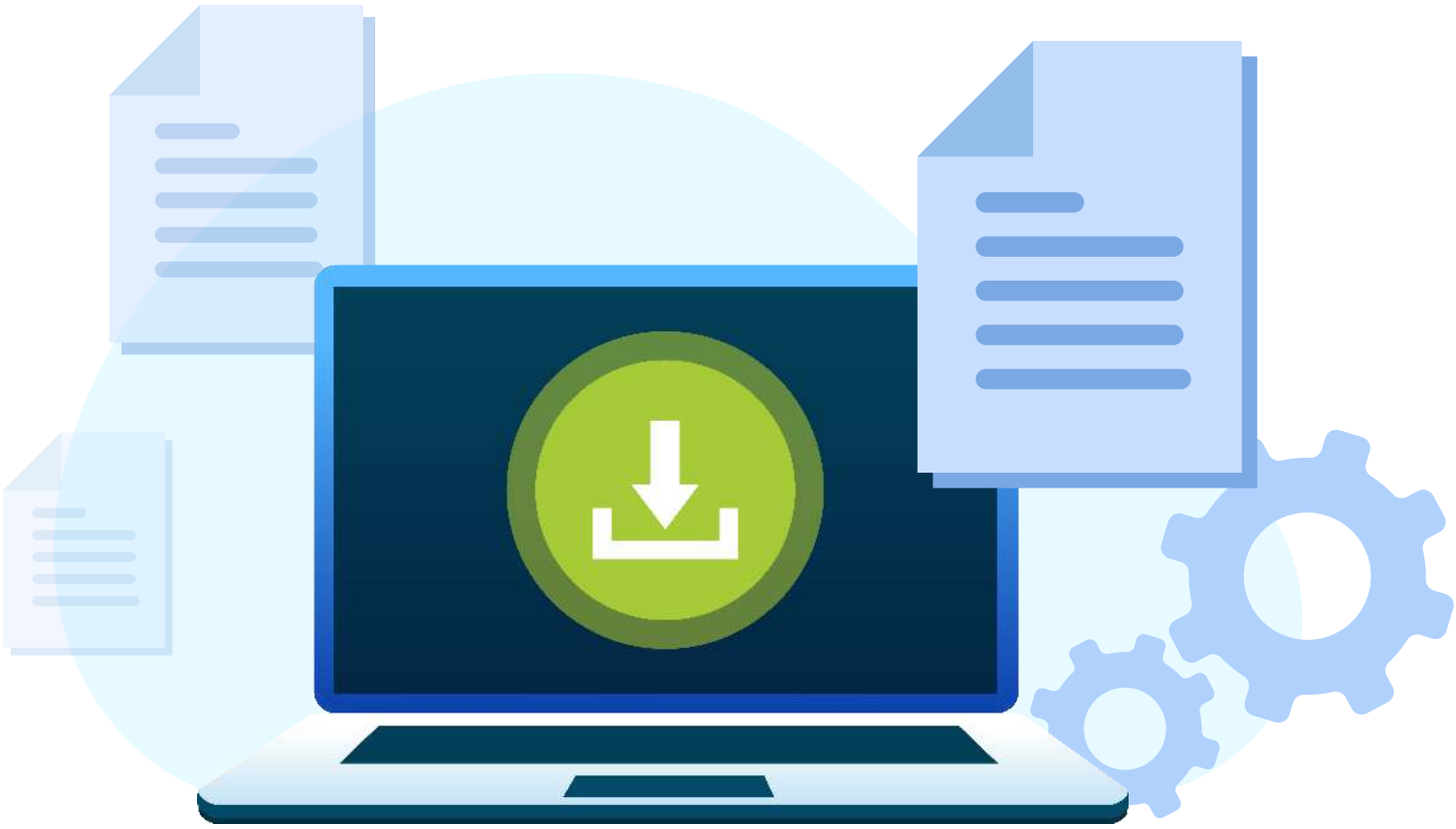
Summary

We have demonstrated a lot of different aspects of Ext JS in this e-book and have built the foundation upon which the rest of our Email Client Application will be built.

We have created and modelled our data structures through Models and Stores and hooking them up to a static backend through Proxies and Readers. Following this, we set up our shared View Controller and View Model to allow a cross-toolkit application to be easily built with the maximum possible amount of code-sharing between the two.

We then started building the Classic toolkit's interface by making a Messages Grid to display all of the loaded messages and a Message Reader to display the full details of a message. These components took advantage of the data binding options of the framework keeping the amount of code needed to a minimum and maintaining a state-driven style of interface design.





Thank you for reading!

Part-1 of Building an Email Client from Scratch

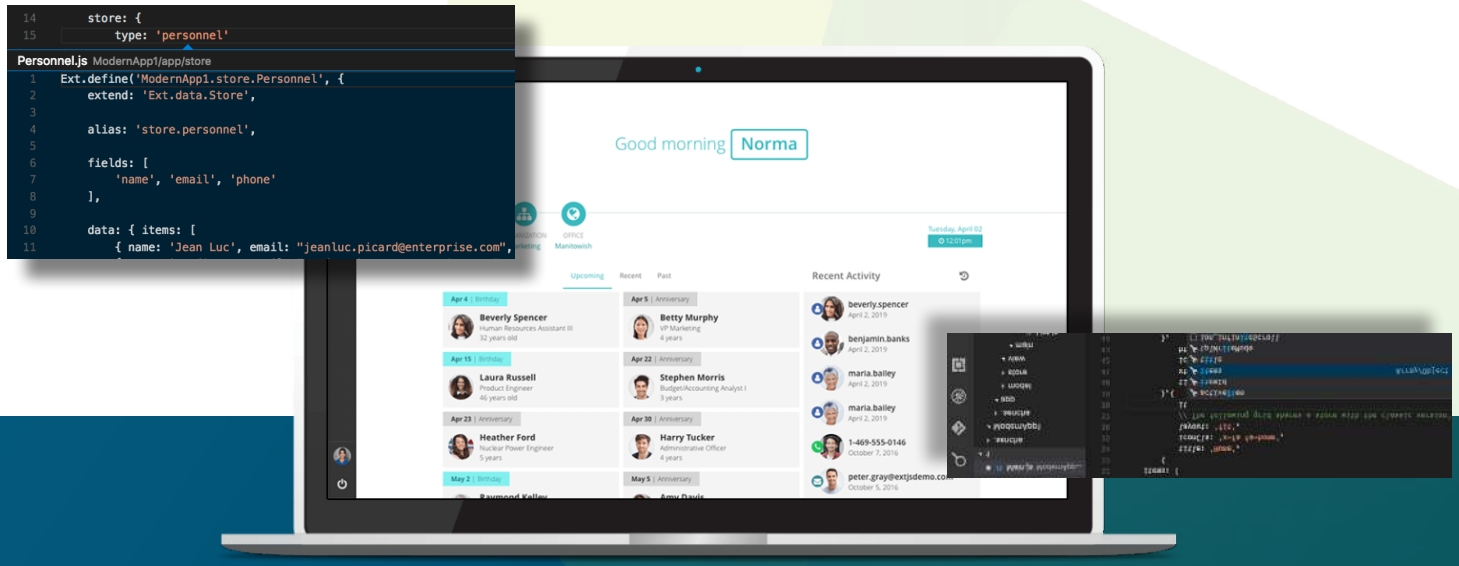
We hope you found it informative and helpful in your development projects. We have 6 more parts lined up to take you through the entire process of building an email client from scratch.

Download the Part-2 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