Components and Databinding

In this assignment, you will modify the `cms` application to make it more modular, use data binding to pass input data from a parent component to a child component, use event emitters to output data back up to a parent component, and use the `ngIf` command to selectively load components. You will also use local references and `@ViewChild` to access elements in the Document Object Model (DOM).

Making your application more modular

The `ContactListComponent` currently loops through and displays the name and an image of each contact in the contacts list. The display of this information is tightly coupled with the display of the list itself. Functionally, this component has two highly related but different task: 1) display a list of contacts, and 2) display detailed information about a single contact in the list. This makes the code messy, harder to debug and change. In addition, we need to display the same detailed contact item information in a similar but different list in the `ContactDetailComponent`. A more modular approach is to create two distinct components. One to display the list of contacts and one to display the detailed information for a contact item in the list. This allow changes to be made to either component without affecting the code in the other. In addition, we can reuse the component to display a contact item in the `ContactDetailComponent` as well when we need to display list of contacts belonging to a "team" or group.

Follow the instructions below to make the Contacts View more modular.

1. Open a **terminal** window and change the directory to the `contacts` directory in your `cms` application.

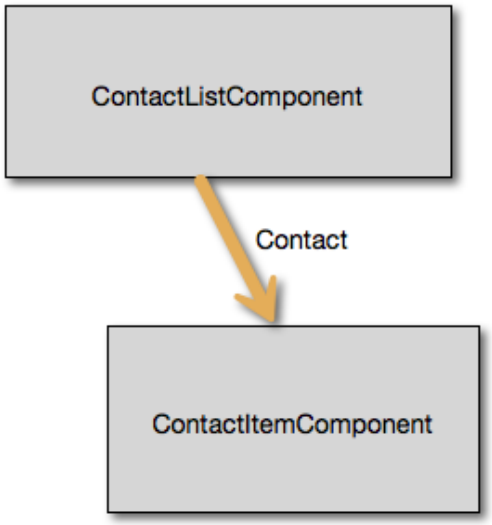
2. Issue the ng command to create a new component called `contact-item` in a new directory.

3. Open up WebStorm. Locate and open the `contacts-list.component.html` file and copy the anchor (`<a>`) tag and all of the tags within in the anchor tag. Locate and open the `contact-item.component.html` file. Delete all of the code in this file and paste the contents of the code that you copied from `contacts-list.component.html` file into this new file. Delete the `ng-for` clause in the anchor tag if there is one.

4. Now, we need to go back and modify the `ContactListComponent` to load and display this contact for every contact in the contact list. Open the `contact-list.component.html` file and replace the anchor tag (`<a>`) and it's contents with a single tag to load and display the new `ContactItemComponent`. You need to add the `ngFor` command to this new tag to load and display the `ContactItemComponent` for every contact in the `contacts` list. See the `recipe-list.component.html` file in the Recipe Book (`prj-cmp-databinding-final`) project for corollary example of how to do this.

Passing data from a parent component to a child component

Components cannot directly share data one with another. A parent component can, however, pass data into one of its child components as an input as shown below. The `ContactListComponent` must pass the current `Contact` object to the `ContactItemComponent` as it loops through all of the contacts in the list.

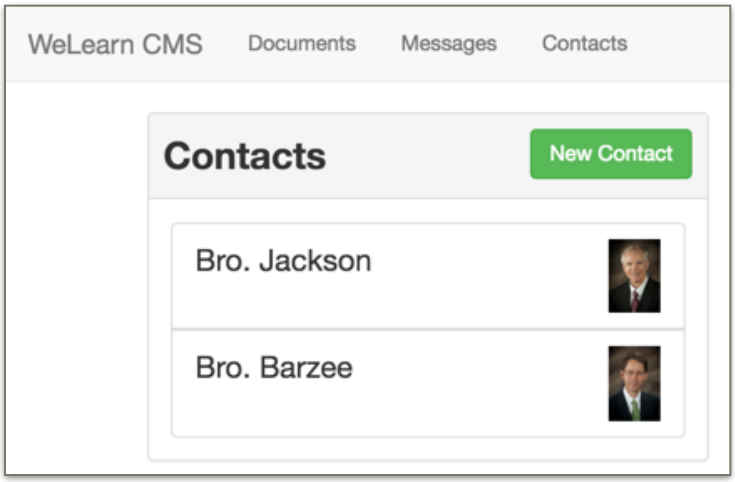


Follow the instruction below to pass the current Contact in the `ContactListComponent` into the `ContactItemComponent`.

1. Open the `contact-item.component.ts` file in WebStorm and add a class input variable called `contact` of the `Contact` datatype to the `ContactItemComponent` class. You will need to import the `Contact` model class as the top of the file.

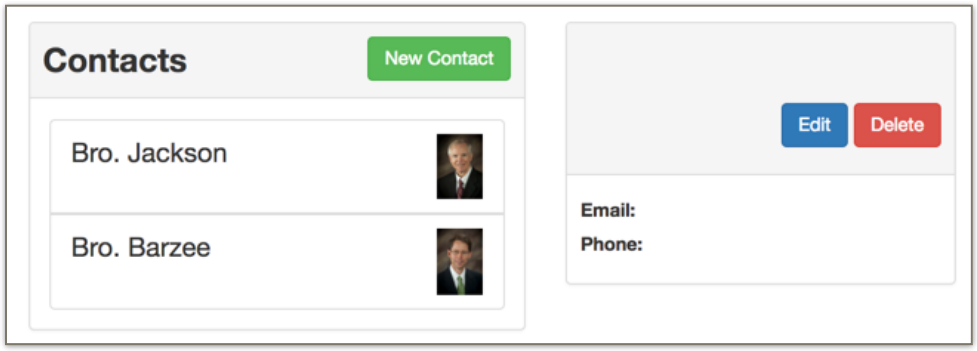
2. Go back and open the `contact.list.component.html` file and modify the `<cms-contact-item>` tag to pass the current `contact` after the `ngFor` statement to the `contact` input variable in the `ContactItemComponent`. Again, see the `recipe-list.component.html` file in the Recipe book (`prj-cmp-databinding-final`) project for an example of how to pass an input into a component.

3. Save all your changes. Open a terminal window, change to the `cms` directory of your project, and issue the `ng serve` command to start your server. Open a browser window and type in the URL: `localhost:4200`. Your application should look as it did before the change except that it is now more modular. In a later lesson, we will see how we can reuse the `ContactItemComponent` in another component.

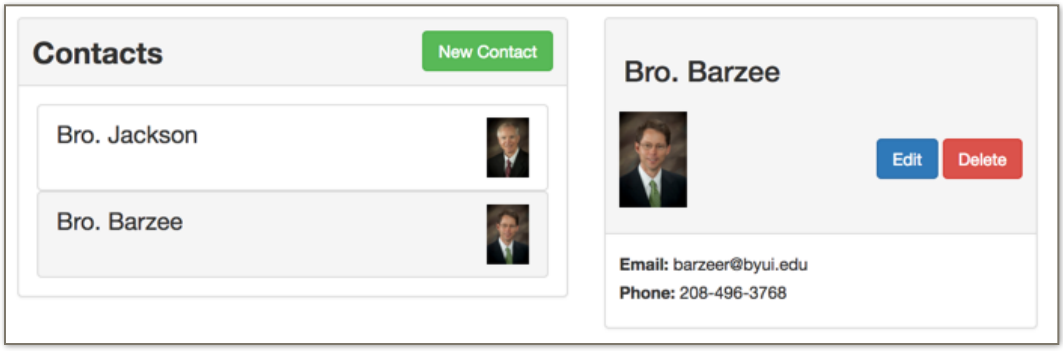


Using Event Emitters to output data

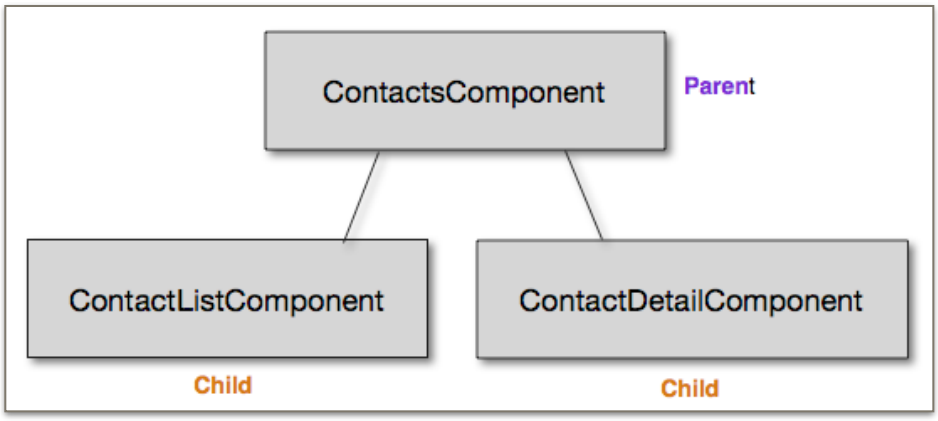
The `ContactComponent` in the `cms` application currently displays the list of contacts in the `ContactListComponent` and an empty `ContactDetailComponent`.



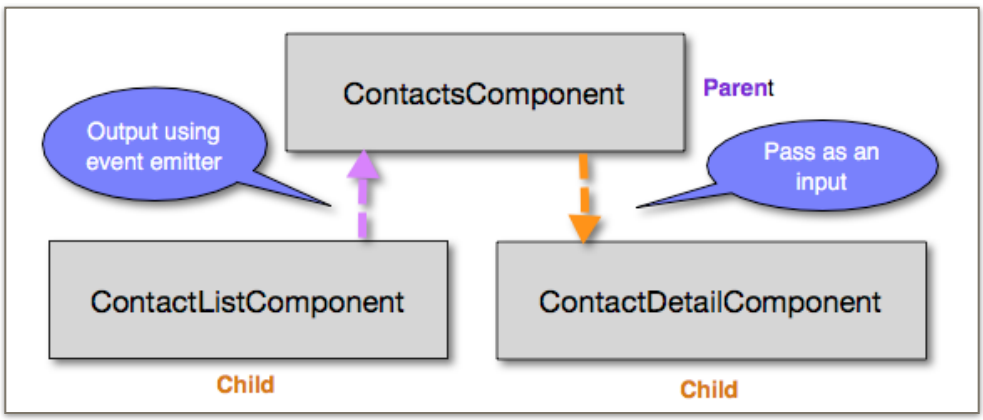
We need to modify the application so that when the end user selects a contact in the contact list, all of the detailed information for the selected contact is displayed in the `ContactDetailComponent` as shown below.



It would be nice if we could pass the selected contact from the `ContactListComponent` directly into the `ContactDetailComponent` as an input as we did when we looped through the list of contacts in the `ContactListComponent` and passed the current contact to each of its child `ContactItemComponents`. Unfortunately, we cannot do this because Angular has a restriction in that a parent component can only pass an input from the parent component to a child component. The `ContactListComponent` and the `ContactDetailComponents` are siblings and children of the `ContactsComponent` as shown below.



Fortunately, Angular provides a way to output data from a child component back ups to its parent component using event emitters. The contact selected in the `ContactListComponent` can now be output backup to its parent, `ContactsComponent`, using an event emitter, and then passed back down to the child `ContactDetailComponent` as an input.



To make this all work, we need to modify the `ContactsListComponent` to detect when an end user clicks on a contact in the contact list and then emit (fire) a custom event containing the selected contact back up to the `ContactsComponent`. The `ContactsComponent` needs to be modified to listen for this custom event, get the contact contained within the event when it occurs and then pass the contact down to the `ContactDetailComponent` as an input.

Following the instructions below to make these changes. Use the `RecipesComponent`, `RecipeListComponent` and the

RecipeDetailComponent in the Recipe Book application (prj-cmp-databinding-final) as examples of how to do this.

1. The ContactListComponent needs to recognize when the end user clicks on a contact in the contacts list and then call a function that will emit a custom event with the selected Contact in it.

   a. Open the contact-list.component.ts file and create a new custom EventEmitter object whose data type is of the Contact data type. Assign the new EventEmitter object to a class output variable called selectedContactEvent. Be sure to add the @Output() annotation in front of the variable name. This tells Angular that this component will emit an event. You will also need to import the EventEmitter and Contact model classes at the top of the file if you have not already done so.

   b. Next, we need to create the function that will emit the event. Add a new function in the ContactListComponent class with the following function signature.

      ```
      onSelected(contact: Contact)
      ```

      Inside this function, call the selectedContactEvent emitter's emit() function and pass it the contact object passed into the onSelected() function as an input.

   c. Now we need to detect when the end user clicks on one of the contact items in the list and then call and pass the selected Contact object to the onSelected() function. Open the contact-list.component.html file and go to the <cms-contact-item> tag and use event binding to detect the click event and call the onSelected() function. Pass the current contact object defined in the ngFor statement to the function.

2. When the end user clicks on a contact in the contact list, the selectedContactEvent will be fired and send the emitted event back up to the parent ContactsComponent. Modify this component to detect when the event occurs, to retrieve the Contact object from the event and finally to pass the contact object down to the ContactDetailComponent as an input when it is loaded.

a. Open the `contacts.component.html` file and bind the `selectedContactEvent` output from the `ContactsListComponent` to a statement that will assign the data passed with the event to a local variable.

```
<div class="row">
  <div class="col-md-5">
    <cms-contact-list (selectedContactEvent)="selectedContact = $event"></cms-contact-list>
  </div>
```

The statement above may seem confusing so let me explain. You have already seen the syntax (e.g., `(event)="expression"`) to detect events such as the `click` event shown below. When an end user clicks on the button element, the click event is emitted (fired) and the expression to the right the equal sign is executed and the `doSomething()` function is called.

```
<button (click)="doSomething()">Click me!</button>
```

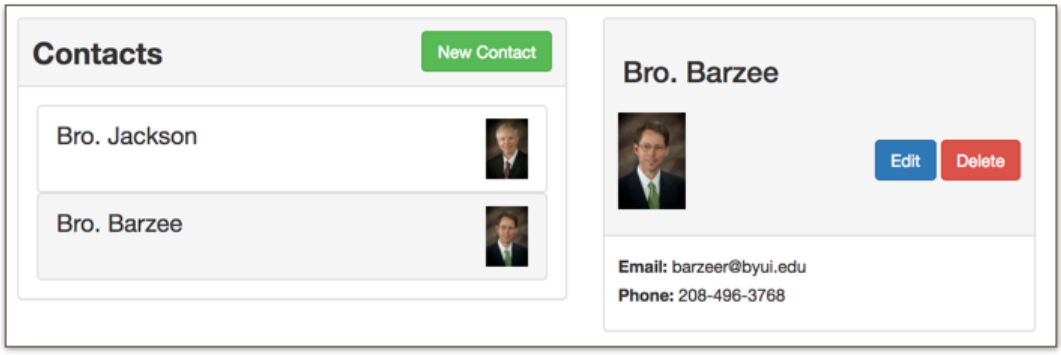
Similarly, when the `selectedContactEvent` in the `<cms-contact-list>` tag above is emitted, it executes the expression to the right of the equal sign and assigns the value of the $event variable to the `selectedContact` class variable defined in the `ContactsComponent` class. The `$event` variable is a reserved Angular variable that always contains the data passed with the event. In this case, it is the `Contact` object that was passed to the `emit()` function in the `ContactsListComponent` class previously.

b. The value of the `selectedContact` variable now needs to be passed down to the `ContactDetailComponent` as an input.

   i. Open the `contact-detail.component.ts` file and define a class input variable called `contact` of the `Contact` data type in the `ContactDetailComponent` class. Now open the `contacts.component.html` file again and modify the `<cms-contact-detail>` tag and assign the value of the `selectedContact` variable to the `contact` class input variable you just defined in the `ContactDetailComponent` class.

3. The last thing we need to do is to selectively load the `ContactDetailComponent` when a contact has been selected in the contact list. This can be easily done using an `ngIf` directive. The `ngIf` directive loads a component only when the specified condition evaluates to true. In the `contacts.component.html` file add an `ngIf` directive to `<cms-contact-detail>` tag to load the component only when the value of the `selectedContact` variable is not null or undefined.

4. Save all of your changes. Start your server using the ng serve command in the browser and display your application (`localhost:4200`). Try selecting each of the contacts in the list. The contact all of their detailed information should display as shown below.



Create documents and messages components

The completed `cms` application will allow us to view and edit a list of documents that belong to the end user and their related messages and contacts. The Documents, Messages and Contacts will each be displayed in separate views. The end user will navigate between the views by selecting the appropriate view in the header component at the top of the screen.

You have already implemented a good deal of the Contacts view of the application. We now need to start working on the Documents and Messages parts of the application. In this section, you will create the basic components for the Documents and Messages views.

Create the documents components

The documents view is responsible for displaying a list of documents that the end user has access to and when a document in the list is selected, it will display the detailed information about the selected document and allow the user to view the document. You will need to create the `DocumentsComponent`, `DocumentListComponent`, `DocumentItemComponent` and `DocumentDetailComponent` components to implement this view.

The `DocumentsComponent` will act as the parent component and display both the `DocumentListComponent`, and `DocumentDetailComponent` components.

The `DocumentListComponent` is responsible for displaying the list of related documents that the end user has access to, and the `DocumentDetailComponent` will display the detailed information about the contact with options to view, edit or delete the document.

The `DocumentItemComponent` is a child component of the `DocumentListComponent` and will display the title of the document.

Follow the instructions below to create these four components. These components will initially have little or no functionality. This will be added in later lessons.

1. Open the terminal in WebStorm. Change to the `app` directory and use the `ng` command to create the `documents` component in a new directory.

2. Change to the `documents` directory and use the `ng` to create the `document-list` component in a new directory.

3. In the `documents` directory, use the `ng` command to create the `document-item` component.

4. In the `documents` directory, use the `ng` command to create the `document-detail` component.

5. We need to also create model class to store the data for a single document. Create the `Document` model class.

a. In the `documents` directory, create a new file called `document.model.ts`. Open the file, define and export the `Document` model class. Add a `constructor()` function to the `Document` model class with the following public class variables.

b. `id` - the document id
c. `name` - the name of the document
d. `description` - a brief description the document
e. `url` - the URL of where the file is located
f. `children` - a list of Document objects that are related to the current document

6. Open the `documents.component.html` file and replace the contents of the file with the following HTML. Replace the two comments with tags to load and to display the `DocumentListComponent` and the `DocumentDetailComponent` respectively.

```
<div class="row">
  <div class="col-md-5">
    <!--Add tag to load the DocumentListComponent here  -->
  </div>

  <div class="col-md-4">
    <!--Add tag to load the DocumentDetailComponent here-->
  </div>
</div>
```

7. Open the `document.list.component.html` file and replace the contents of the file with the HTML shown below. Replace the comment with a tag to display a list of `DocumentItemComponents`.

```
<div class="panel panel-default">
  <div class="panel-heading">
    <div class="row pad-left-right">
      <span class="pull-left title">Documents</span>
      <a class="btn btn-success pull-right">Add Document</a>
    </div>
  </div>
  <div class="panel-body">
    <div class="center-panel">
      <!--Add tag to display a list of DocumentItem components-->
    </div>
  </div>

</div>
```

8. We will not make any further modifications to the other document components at this time. This will be done in a future assignment.

9. Save all your changes.

Create the messages components

The Messages view is very similar to the documents view. It is responsible for displaying a list of messages that the end user has access to. End users will be able to create and add new messages to the message list. You will need to create the `MessageListComponent`, `MessageItemComponent` and `MessageEditComponent` components to implement this view.

The `MessageListComponent` is responsible for displaying the entire list of messages, the `MessageItemComponent` displays the information for a single message in the list, and the `MessageEditComponent` is displays a form to add new messages to the list.

Follow the instructions below to create these three components. These components will initially have little or no functionality. This will be added later lessons.

1. Open the terminal in WebStorm. Change to the `app` directory and create a new directory called `messages`.

2. In the `messages` directory, use the `ng` command to create the `message-item` component in a new directory.

   a. Open the `message-item.component.css` file and define the two following style classes.

```
.messageHeader {
  background-color: azure;
  border: solid thin black;
  border-radius: 10px;
  padding: 0.4rem;
  font-size: 1.5rem;
}


.messageText {
  border-radius: 10px;
  padding: 5px;
  margin-top: .25rem;
  margin-left: 1rem;
  font-size: 1.5rem;

}
```

   b. Open `message-item.component.html` file and replace the current contents of the file with following:

```
<a class="list-group-item clearfix">
  <div class=''>
    <span class='messageHeader'>SendersName</span>
    <div class='messageText'>MessageText</div>
  </div>
</a>
```

3. In the `messages` directory, use the `ng` command to create the `message-edit` component in a new directory. Open `message-edit.component.html` file and replace the current contents of the file with following:

```
<div class="panel panel-default">
  <div class="panel-body">

    <form id="document-edit">
      <div class="row">

        <div class="col-sm-12 form-group">
          <label for="subject">Subject</label>
          <input
            type="text"
            id="subject"
            class="form-control"
            size="120"
            max="120">
        </div>

        <div class="col-sm-12 form-group">
          <label for="message">Message</label>
          <input
            type="text"
            id="message"
            class="form-control"
            size="120"
            max="255">
        </div>
      </div>

      <div class="row">
        <div class="col-xs-12">
          <button class="btn btn-success" type="submit">Send</button>
          <button class="btn btn-danger" type="button">Clear</button>
        </div>
      </div>

    </form>

  </div>
</div>
```

4. Change to the `messages` directory and use the `ng` command to create the `message-list` component in a new directory.

   a. Open the `message-list.component.css` file and define the two following class styles.

```
.title {
  font-size: 2.5rem;
  font-weight: bold;
}

.pad-left-right{
  padding-left:1rem;
  padding-right:1rem;
}
```

   b. Open the `message.list.component.html` file and replace the contents of the file with the HTML shown below. Replace the two comments with tags to load and display a `MessageItemComponent` and the `MessageEditComponent` respectively.

```
<div class="row">
  <div class="col-md-5">
    <div class="panel panel-default">
      <div class="panel-heading">
        <div class="row pad-left-right">
          <span class="title pull-left">Messages</span>
        </div>
      </div>
      <div class="panel-body">
        <div class="row">
          <div class="col-xs-12">
            <!--Add tag to load a MessageItemComponent -->
          </div>
        </div>
      </div>
      <div class="panel-footer">
        <!--Add tag to load the MessageEditCompnent-->
      </div>
    </div>
  </div>
</div>
```

5. You will also need to create model class to store the data associated with a message. Create the `Message` model class.

a. In the `messages` directory, create a new file called `message.model.ts`. Open the file, define and export the `Message` model class. Add a `constructor()` function to the `Message` model class with the following public class variables.

b. `id` - the message id

c. `subject` - the name of the message

d. `msgText` - a brief description the message

e. `sender` - the URL of where the file is located

6. Save all your changes. We will not make any further modifications for now. This will be done later in this assignment.

Switching between views

Now that we have created most of the components for the documents, messages and contacts views we need to add the capability to switch between views. First, we need to modify the `HeaderComponent` to detect when the end user clicks on either the **Documents**, **Messages**, or **Contacts** features. The `HeaderComponent` then needs to emit a custom event backup to the its parent component (`AppComponent`) with a value indicating which feature was selected. The parent `AppComponent` then needs to watch for this custom event. When the event is detected, it must get the value passed with the event and selectively load and display the component that corresponds to the value retrieved (i.e., value = "documents" then load the `DocumentsComponent`, value = "messages" then load the `MessagesComponent`, value = "contacts" then load the `ContactsComponent`).

Follow the instructions below to implement the switching of views. Use the `HeaderComponent` in the Recipe Book (`prj-cmp-databinding-final`) application as template for how this is to be done.

1. Modify the `HeaderComponent` to detect when the end user selects and clicks on either the **Documents**, **Messages**, or **Contacts** feature. The `HeaderComponent` then needs to emit and output a custom event backup to the its parent component (`AppComponent)` with a value indicating which feature was selected.

a. Open the `header.component.ts` file and create a new `EventEmitter` object of the `string` data type and assign this `EventEmitter` object to a class variable called `selectedFeatureEvent`.

b. Create a new function in the `HeaderComponent` class with the following function signature. This function is responsible for emitting or firing the `selectedFeatureEvent`.

```
onSelected(selectedEvent: string)
```

Inside the function, call the `emit()` function for the `selectedFeatureEvent` event emitter. Pass it the value of the `selectedEvent` input parameter variable.

c. Open the `header.component.html` file and locate the three anchor (`<a>`) tabs for the Documents, Messages and Contacts features. Modify each anchor tag and bind a `click` event tag to a function call to the `onSelected()` function. Pass in an appropriate string value that corresponds to the feature clicked on (i.e., "documents" for the Documents feature, "messages" for the Messages feature, "contacts" for the Contacts feature).

2. Modify the `AppComponent` to watch for and detect the `selectedFeatureEvent` being emitted from its child `HeaderComponent`. When this event is detected, call a function to save the string value passed with the event. This value indicates which feature was selected in the `HeaderComponent`.

a. Open the `app.component.html` file and add a class variable called `selectedFeature` of the `string` datatype to the `AppComponent` class. Initialize the variable with the literal value `"documents"`.

b. Create a function in the `AppComponent` class with the following function signature.

```
switchView(selectedFeature: string)
```

Assign the value of the `selectedFeature` input parameter variable to the `selectedFeature` variable in `this` class.

c. Open the `app.component.html` file and locate the `<cms-header>` tag. Add code to this tag to detect the `selectedFeatureEvent` emitted from the `HeaderComponent` and bind a function to the event to call to the `switchView()` function you just created. Pass the value passed up with the event t (e.g., `$event`) to the function as an input.

3. Finally, we need to add the tags to selectively load and display the `DocumentsComponent` , `MessageListComponent`, and `ContactsComponent` based on the value of the `selectedFeature` class variable. For example, the `DocumentsComponent` is to be loaded when the value of the `selectedFeature` class variable equals "documents", the `MessageListComponent` when the value is "messages", and the `ContactsComponent` when the value is "contacts".

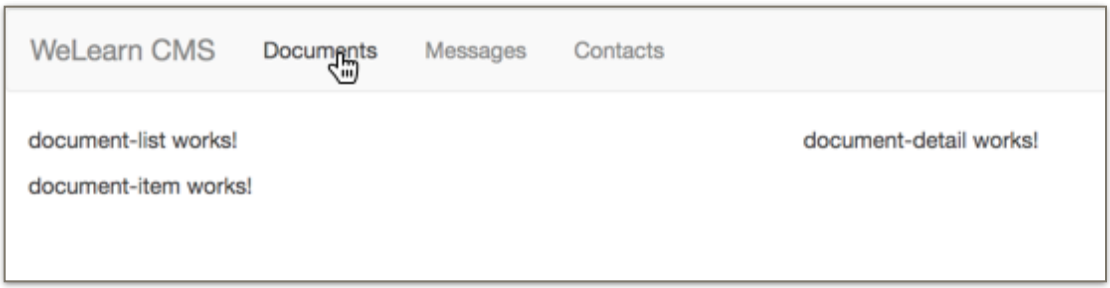
   a. Open the app.component.html file and add tags to display the `DocumentsComponent`, `MessageListComponent` and `ContactsComponent` in the order shown below.

```
<cms-header (selectedFeatureEvent)=switchView($event)></cms-header>
<div class="container  pull-left">
    <cms-documents ></cms-documents>
    <cms-message-list ></cms-message-list>
    <cms-contacts></cms-contacts>
</div>
```

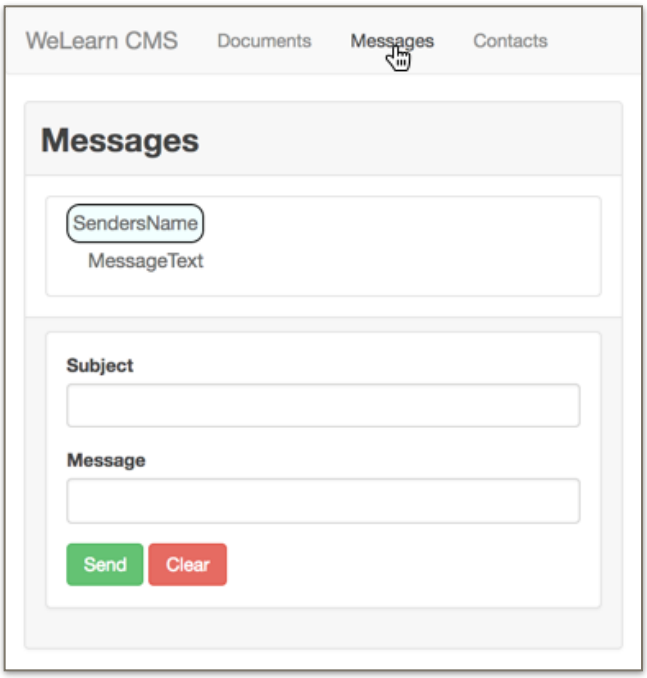
   b. Modify the `<cms-documents>`, `<cms-messages-list>`, and `<cms-contacts>` tags to selectively load and display each of these components based on the value assigned to the `selectedFeature` class variable. Use either the `ngIf` or `ngSwitch` statements to do this.

4. Save all of your changes. Start the server using the ng serve command. Open the browser and view your application. Try selecting each feature in the HeaderComponent. Each of the three different views should be displayed as follows.

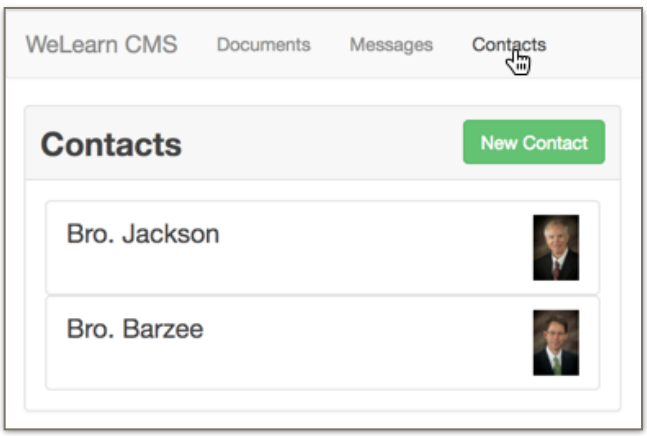
Documents View



Messages View



Contacts View

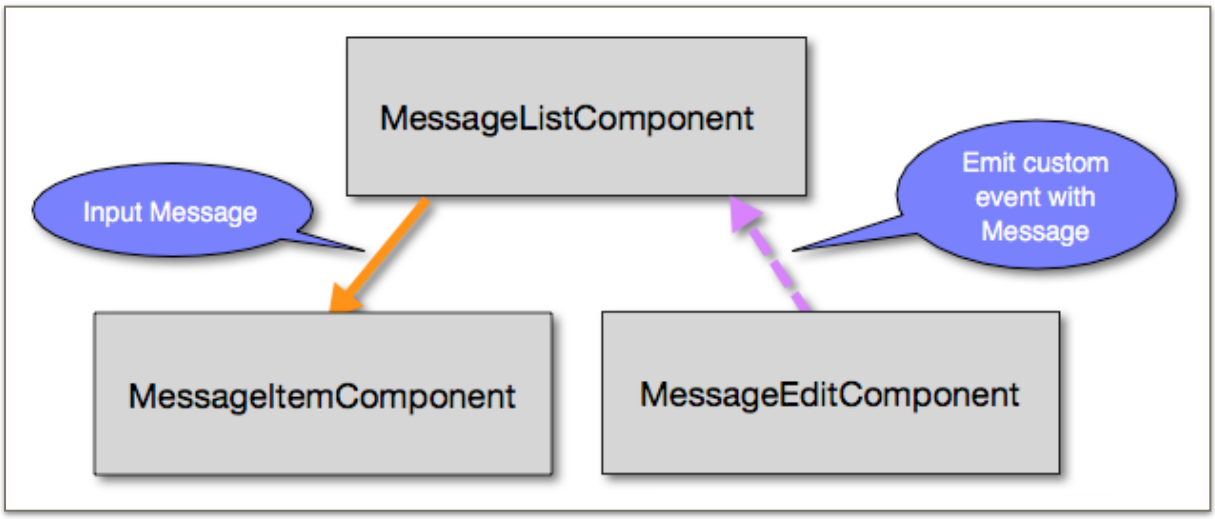


Implement the Message View

We now need to will fully implement the Messages view. The Messages View will display a list of messages and allow the end user to enter and send new messages to the list. This will require that we make changes to the `MessageListComponent`, `MessageItemComponent` and `MessageEditComponent`. The `MessageListComponent` loads and displays the `MessageItemComponent` and `MessageEditComponent`. This is what the screen should look like you have made changes to these three components.

The `MessageListComponent` loads and displays the `MessageItemComponent` and `MessageEditComponent` as shown below.

It loops through all the messages in the message list and loads and displays a `MessageItemComponent` for each message in the list. The current message is passed to the `MessageItemComponent` as an input as shown below. The `MessageEditComponent` allows the end user to create and send a new message to the message list. It must emit a custom event containing the new message backup to the `MessageListComponent` when the **Send** button is pushed.



Receiving and displaying a message in MessageItemComponent

We need to modify the `MessageItemComponent` to receive a message object as an input and then display the senders name and the text of the message when the component is loaded.

Follow the instructions below to implement the changes. These changes are very similar to the changes you made to the `ContactItemComponent` you created earlier. Use that has a guide for the changes you need to make to `MessageItemComponent`.

1. Open the `message-item.component.ts` file and define a class input variable called `message` that is of the `Message` datatype in the `MessageItemComponent` class. You will need to import the `Message` and `Input` classes.

2. Open `message-item.component.html` file and replace the literal text, *SendersName* and the *MessageText* with the actual values stored in the `Message` object passed into the component.

```
<a class="list-group-item clearfix">
  <div class=''>
    <span class='messageHeader'>SendersName</span>
    <div class='messageText'>MessageText</div>
  </div>
</a>
```

   Use string interpolation to get and display actual values of the `sender` and the `msgText` properties in the `Message` object passed as an input to the component.

Modify the MessageEditComponent to send a new message

The `MessageEditComponent` is an HTML form that allows the end user to create and send a new message to the message list. The form contains input fields to enter the subject of the message and the text of the message and two buttons. When the **Sender** button is clicked, the values of the two input fields are retrieved and a new `Message` object is created. The new `Message` object is then emitted back up to the `MessageListComponent` using a custom event emitter. The **Clear** button assigns blank value to the two input fields.

Follow the instructions below to implement this form. See the `ShoppingEditComponent` in the Recipe Book (`prj-cmp-databinding-final`) project for a similar example of the modifications that need to be made.

1. The HTML in the form needs to be first modified to call functions to implement the behavior described above when the **Send** and **Clear** button are clicked on. Open `message-edit.component.html` file and make the following changes.

   a. Add a click listener to the **Send** button element to call a function called `onSendMessage()` when this button is clicked.

   b. Add a click listener to the **Clear** button element to call a function called `onClear()` when this button is clicked.

c. We will also need to define local reference variables for each of the input fields in the form. This will allow us to access the value of the inputs elements when the form is submitted. Define a local reference variable called `subject` for the subject input tag, and a local reference variable called `msgText` for the message input tag..

2. Open the `message-edit-component.ts` file and implement the `onSendMessage()` function in the MessageEditComponent class. This function is called when the **Send** button is clicked in the form. It gets the values entered in the `subject` and `msgText` input elements from the form, create a new `Message` object with the values entered a, and emits a custom event with the new Message object back up to the `MessageListComponent` so that it can be added to the message list.

   a. We need to the values entered in the `subject` and `msgText` input elements from the Document Object Model (DOM). Use `@ViewChild` to create an `ElementRef` for the `subject` and `msgText` input elements in the DOM at the top of the `MessageEditComponent` class.

   b. We also need a custom `EventEmitter` to output the new `Message` object created back up to the `MessageListComponent`. Create a custom `EventEmitter` to emit a `Message` object at the top of the `MessageEditComponent` class and assign it to a variable called `addMessageEvent`.

   c. Create a `string` variable called `currentSender` and initialize it with the value of your name at the top of the class.

   d. At the bottom of the class implement the `onSendMessage()` function. Here is the algorithm.

```
onSendMessage() {

  get the value stored in the subject input element

  get the value stored in the msgText input element

  Create a new Message object
```

```
        Assign a hardcoded number to the id property in the new
        Message object

        Assign the value of the currentSender class variable to
        the sender property in the new Message object.

        Assign the values retrieved from the subject and msgText
        input elements to the corresponding properties in the new
        Message object

        Call the addMessageEvent emitter's emit() function and
        pass it the new Message object just created

    }
```

3. At the bottom of the class implement the `onClear()` function. This function only needs to assign a blank value to the `subject` and `msgText` input elements in the form.

Display and add messages to the MessageList component

The `MessageListComponent` is responsible for displaying a list of `MessageItemComponents`. We need to first create a dummy list of messages so we can test this component. Then we need to modify the HTML to display the list of messages and finally, we need to implement a function to add new messages to the message list.

Follow the instructions below to implement the `MessageListComponent`. The `MessageListComponent` is very similar to the `ContactListComponent` you created earlier. Use that has a guide for the implementation of the `MessageListComponent`.

1. We need to create a sample list of messages to test this component. Open the `message-list.component.ts` file. Define a class variable called `messages` whose datatype is an array of `Message` model objects. Initialize the array with a list of three or more new `Message` objects. Make up the data values for each property in the message objects.

2. We need to also add a new function to the `MessageListComponent` class to add a message to the message list. Create a new function in

signature to the `MessageListComponent` class with the following function signature.

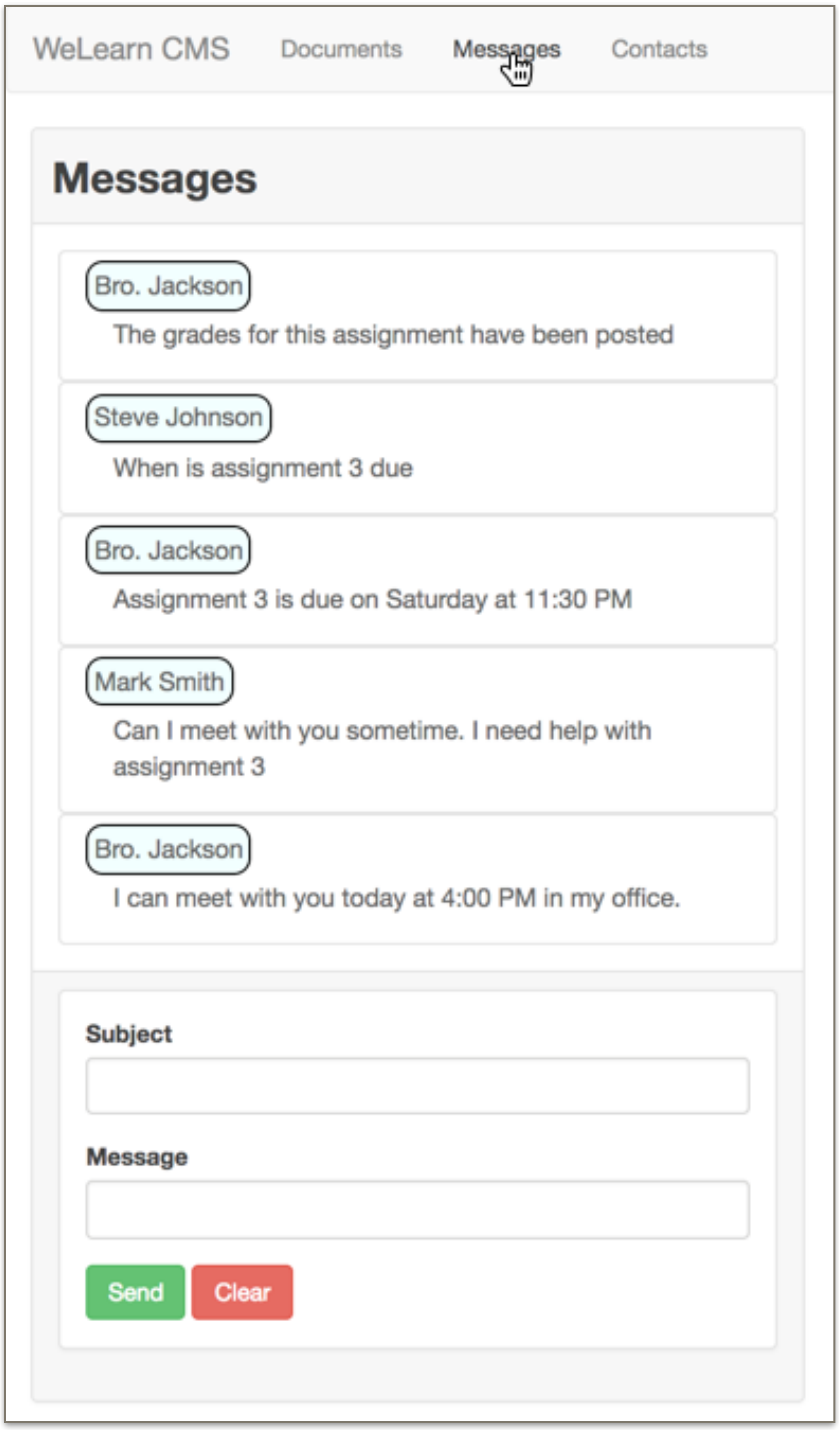
```
onAddMessage(message: Message)
```

Implement the code in this function to push the `Message` object passed as an input into the function to the end of the `messages` list.

3. Open the `message-list.component.html` file and modify the `<cms-message-item>` tag to display the list of all of the messages in the `messages` list. Use an `ngFor` statement and pass the current `message` to the `MessageItemComponent` as an input.

4. We need to also modify the HTML to detect the `addMessageEvent` emitted from the `MessageEditComponent` when the end use clicks the **Send** button in the `MessageEditComponent` and call the `onAddMessage()` function to add the message to the `messages` list. Modify the `<cms-message-edit>` tag to watch for the `addMessageEvent` and call the `onAddMessage()` function when the event occurs. Pass `Message` object passed with the event to the `onAddMessage()` function.

5. Save all of your changes. Start your server using the ng serve command and open your browser and select the Messages view in the HeaderComponent. Your screen should appear similar to the one on the next page. Try creating and sending a new message. The message should be appear at the end of the message list. Select each of the views in the `HeaderComponent` to make sure that you can still switch between the three different views.



This concludes this assignment.