W08 Assignment Instructions

Implement a Form to Add and Edit Documents

You created the *DocumentEditComponent* in the last lesson but it only displayed a simple message when it was loaded into the DOM. This component is needed to add and edit documents. You need to create an HTML form with validation to allow end users to enter the data to be stored in the Document. The basic HTML and CSS for the form of this component has already been created for you. You will need to download these files and modify the form to implement Angular validation in the form.

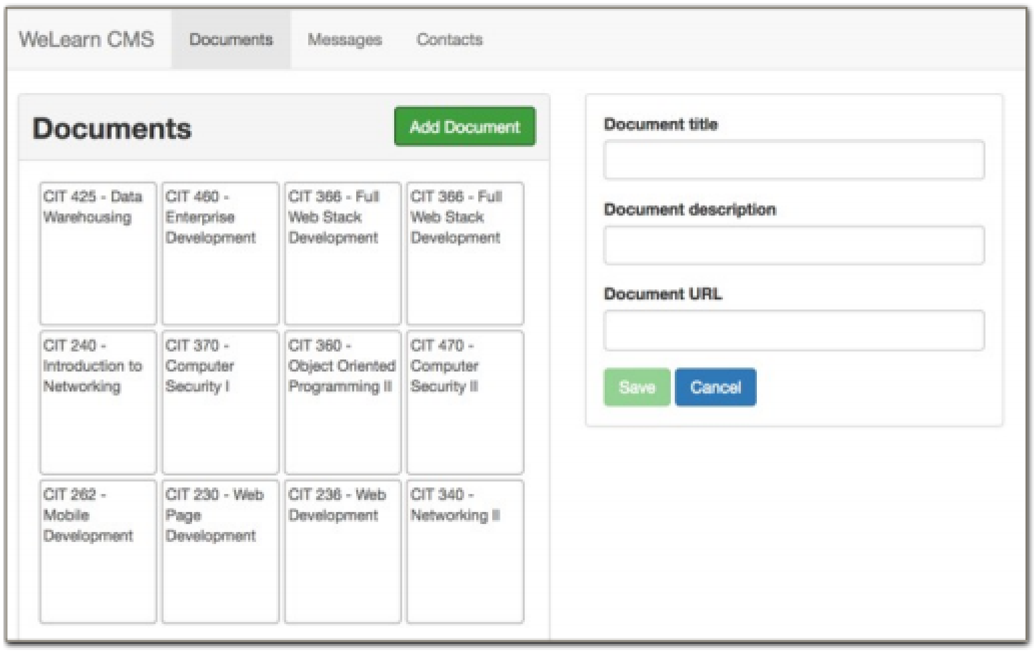
Download the DocumentEditComponent HTML and CSS files

Start by downloading the HTML and CSS files needed for this lesson. Click on the link below to download these files:

Lesson 8 Files (https://byui.instructure.com/courses/146358/files/59077238/preview) (https://byui.instructure.com/courses/146358/files/59077238/download?download_frd=1)

Extract the file you downloaded, change to the created folder, and then copy and paste the corresponding content into the *document-edit.component.html* and *document-edit.component.css* files of your CMS project.

Open the terminal in VS Code and serve up your application by typing the *ng serve* command. Switch to the browser and view your application. Select the *Documents* feature and then select the *Add Document* button in the *DocumentListComponent*. The *DocumentEditComponent* with the new form should appear as below:

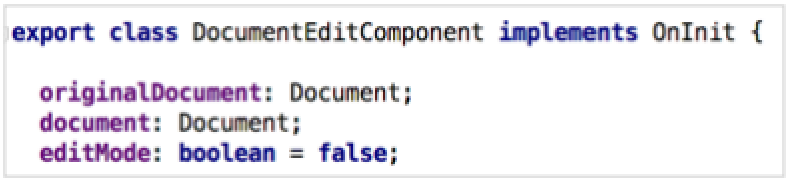

WeLearn CMS
Documents
Messages
Contacts
Documents
Add Document
CIT 425 - Data Warehousing
CIT 460 - Enterprise Development
CIT 366 - Full Web Stack Development
CIT 366 - Full Web Stack Development
CIT 240 - Introduction to Networking
CIT 370 - Computer Security I
CIT 360 - Object Oriented Programming II
CIT 470 - Computer Security II
CIT 262 - Mobile Development
CIT 230 - Web Page Development
CIT 236 - Web Development
CIT 340 - Networking II
Document title
Document description
Document URL
Save
Cancel

Implement Field Validation for the Document Edit Form

The *Document Title* and *Document URL* fields in the form are required. These fields should be highlighted to indicate that they are required, including an error message displayed to indicate as such. The *Save* button should also be disabled until valid values are entered in these two fields. Angular form validation makes it easy to accomplish all of this. Use the *shopping-list.component.html* file in the Recipe Book app as an example of how to do this.

Follow the instructions below:

1. Open the *document-edit.component.ts* file and add the following properties at the top of the class:

   ```
   export class DocumentEditComponent implements OnInit {

     originalDocument: Document;
     document: Document;
     editMode: boolean = false;
   ```

   The *originalDocument* property references the original, unedited version of the document. The *document* property references the edited version of the document displayed in the form. The *editMode* property indicates whether an existing document is to be edited, or a new document is being created. Save your changes.

2. Open the *document-edit.component.html* file and locate the *<form>* element. Place the HTML form under Angular's control.

   a. Define a *local reference* (*#f*) in the *<form>* element and assign a reference to the *FormGroup* (*ngForm*) to it.

   b. Add an *ngSubmit* event binding to the <form> element to call the *onSubmit()* method when the form is submitted. This method will be created later in the *DocumentEditComponent* class. Pass to this method a reference to the *FormGroup* that you saved in the *local reference* (*f*).

3. Map each of the input fields in the form to a corresponding property in a *Document* object. Make the following changes to all of the <input> tags in the form:

   a. Add the *name* attribute to the *<input>* element and assign a value to the attribute that represents the name of the input field (for example, *name="description"* for the *description* <input> element).

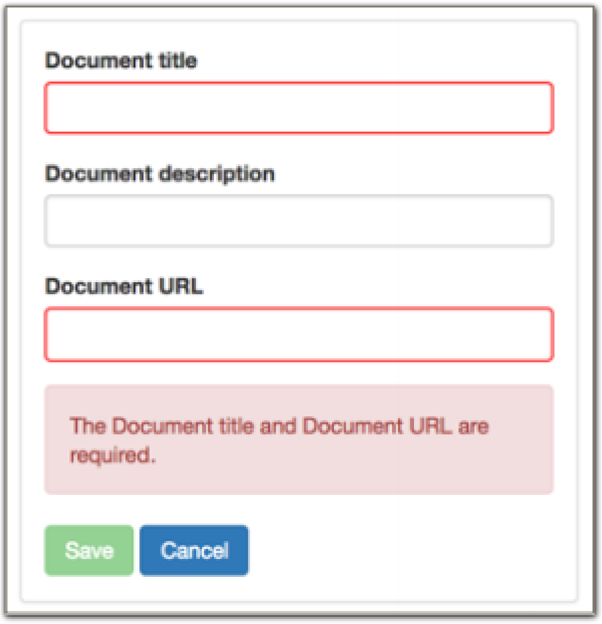
   b. Add the *[ngModel]* directive and assign the corresponding *Document* object property to it (for example, *[ngModel]=document?.description*).

4. The *name* and *URL* fields are required in the form. Make the following additional changes to both the *name* and *URL* <input> elements:

   a. Make the input field *required*.

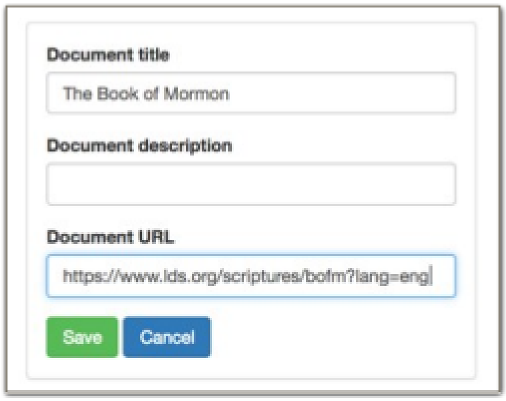
   b. Define a *local reference* (*#name*) and assign *ngModel* to it. An *ngModel* is created for each input element in the form. It contains the value entered into the field and the state of the field (for example, *touched* or *untouched*, *pristine* or *dirty*, *valid,* or *not valid*). This information is used to selectively display error messages and to enable/disable the *Save* button.

5. Add a *<div>* element immediately after the last input field to display the message *“The Document Title and Document URL are required fields.”* Apply the *alert* and *alert-danger* Bootstrap CSS classes to this *<div>*. Add an *ngIf directive to load and display the *<div>* only when either the *Document Title* or *Document URL* fields do not have a valid value.
6. Locate the *Save <button>* element and add an *ngIf directive to disable the *Save <button>* when the form is not valid.
7. Save your changes and open your browser to view your app. Select the *Documents* menu item to display the list of documents. Select the *Add Document* button. The *Save* button should be disabled, the *name* and *URL* fields should be highlighted, and the error message should be displayed at the bottom of the form.



The Document title and Document URL are required.

8. Enter a Document Title and a Document URL. Notice the error message is no longer displayed and the *Save* button is now enabled.



The Book of Mormon

https://www.lds.org/scriptures/bofm?lang=eng

Implementing the Behavior of the Document Edit Form

The behavior associated with the *Save* and *Cancel* buttons is defined in the *DocumentEditComponent* class. When the *Save* button is selected, the form is submitted. If a new *Document* is being created, the *DocumentService addDocument()* method will be called to add the new *Document* to the document list. If an existing *Document* is being edited, the *DocumentService* *updateDocument()* method will be called to update the *Document* in the document list. When the *Cancel* button is called, the form is exited and control is routed back to the *DocumentListComponent*.

Open the *document-edit.component.ts* file and follow the instructions below to implement this behavior:

1. Inject the *DocumentService*, *Router*, and *ActivatedRoute* classes in the class *constructor()* method:

   ```
   constructor(
           private documentService:
   DocumentService,
           private router: Router,
           private route: ActivatedRoute) {

   }
   ```

2. The *DocumentEditComponent* needs to determine if it is being used to edit an existing document or to create a new document. If it is being used to edit an existing document, the document must be retrieved so that the form can be populated with the document's original values. A good place to do this is in the *ngOnInit()* lifecycle method because it is automatically called right after a component is created and before it is loaded into the DOM.

   Here is the algorithm to implement this functionality in the *ngOnInit()* lifecycle method:

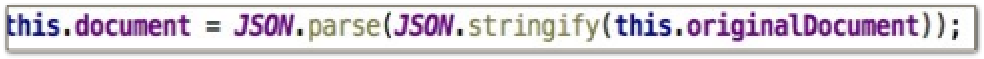
   ```
   ngOnInit() {
     route.subscribe (
      (params: Params) => {
        id = value of id parameter in params list
        if id parameter is undefined or null then
          editMode = false
          return
        endif
        originalDocument = getDocument(id)

        if originalDocument is undefined or null then
          return
        endif
        set editMode to true
        document = clone originalDocument
     })
   }
   ```

   The algorithm subscribes to the currently active route to get the value of the *id* parameter in the router's parameter list. If the *id* parameter is not found in the parameter list, the component is being used to add a new document to the document list and *editMode* is set to false; otherwise, the component is being used to edit an existing *Document* object. The *Document* object is retrieved by calling the *getDocument()* method in the *DocumentService*, and assigned to the *originalDocument* property. The method exits if no document is found. The *editMode* is set to true and a clone (copy) of the *originalDocument* is assigned to the *document* property if the document was found.

Implement the algorithm above in the *ngOnInit()* method in the *DocumentEditComponent* class. You will need to import the *OnInit* interface. Refer to the previous assignment if you have forgotten how to subscribe to the currently active route and get the value of the parameters defined in the route's URL. Note: the *originalDocument* and *document* properties must be referenced with the *this* keyword.

You can easily create a cloned copy of a JavaScript object by using the JavaScript JSON class's *parse()* and *stringify()* methods as illustrated below:



```
this.document = JSON.parse(JSON.stringify(this.originalDocument));
```

The original JSON object stored in the *originalDocument* property is passed to the JSON *stringify()* method. It returns a string representation of the JSON object. Then, we call the JSON *parse()* method and pass it the string returned from the *stringify()* method. The *parse()* method creates and returns a new JSON object based on the string passed to it.

3. Create a new method in the *DocumentEditComponent* class named *onSubmit()*. This method is called when the end user selects the *Save* button and submits the form. It is responsible for either adding a new document to the document list or updating an existing document in the document list.

   Here is the algorithm for the *onSubmit()* method:

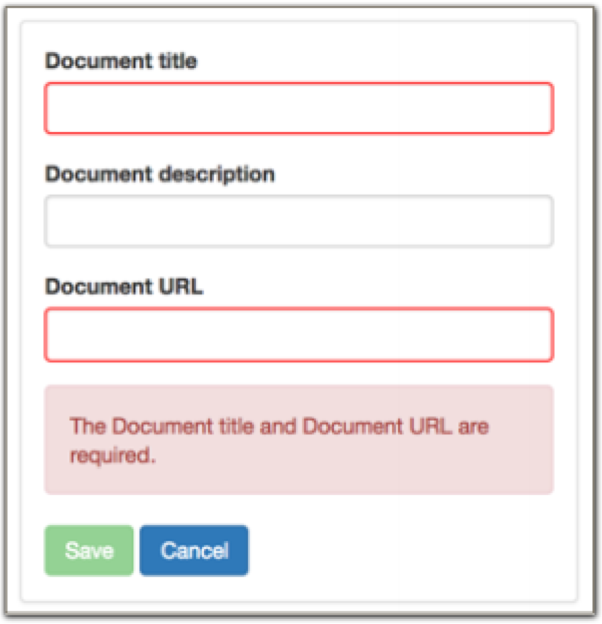
```
onSubmit(form: NgForm) {
  value = form.value // get values from form's fields
  newDocument = new Document()
  Assign the values in the form fields to the
  corresponding properties in the newDocument
  if (editMode = true) then
   documentService.updateDocument(originalDocument,
newDocument)
  else
   documentService.addDocument(newDocument)
  endIf
  route back to the '/documents' URL
}
```

   This method first gets the list of values entered into the fields in the form. It then creates a new *Document* object and assigns the values retrieved from each of the input fields to the corresponding properties in the new *Document* object. If the component is in *edit* mode, the *DocumentService updateDocument()* method is called. Otherwise, the *DocumentService* *addDocument()* method is called. Finally, it routes back to the main *documents* view.

4. Create the *onCancel()* method in the *DocumentEditComponent* class to cancel the form and route back to the main *documents* component. Implement the following algorithm for the *onCancel()* method. Use the router object injected in the *constructor()* method to route to the *'/documents'* URL:

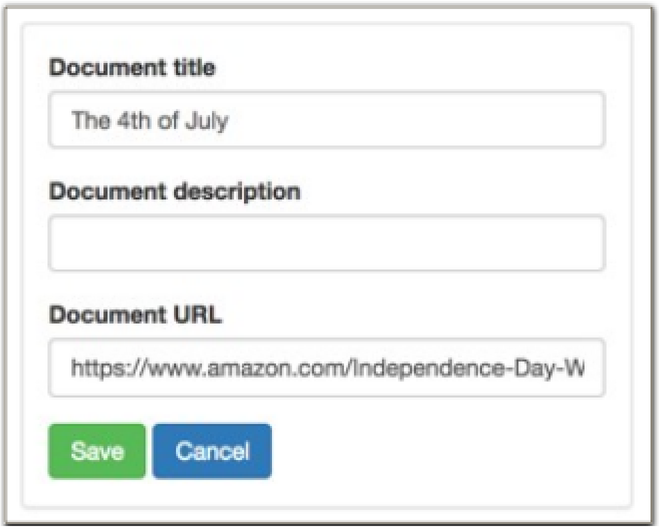
```
onCancel() {
  route back to the '/documents' URL
}
```

5. Save the files and open the browser to view your application. Select the *Documents* feature and select the *Add Document* button in the *DocumentListComponent*. The *DocumentEditComponent* should be loaded and displayed. The required *Document Title* and *Document URL* fields should be highlighted, an error message displayed at the bottom of the form, and the *Save* button disabled.



The Document title and Document URL are required.

Fill in the fields in the form. Notice that the required fields are no longer highlighted, the error message is no longer displayed, and the *Save* button is now enabled. Select the *Save* button. The new document should be added to the document list.


Document title: The 4th of July
Document description:
Document URL: https://www.amazon.com/Independence-Day-W

6. Select one of the existing documents in the document list. The *DocumentDetailComponent* should be displayed. Select the *Edit* button. The *DocumentEditComponent* should be loaded and displayed. The values of the original document should be displayed in the respective fields. Modify the value of the *Document Title* input field and select the *Save* button. The title of the document should be changed for the document you selected in the *DocumentListComponent*.

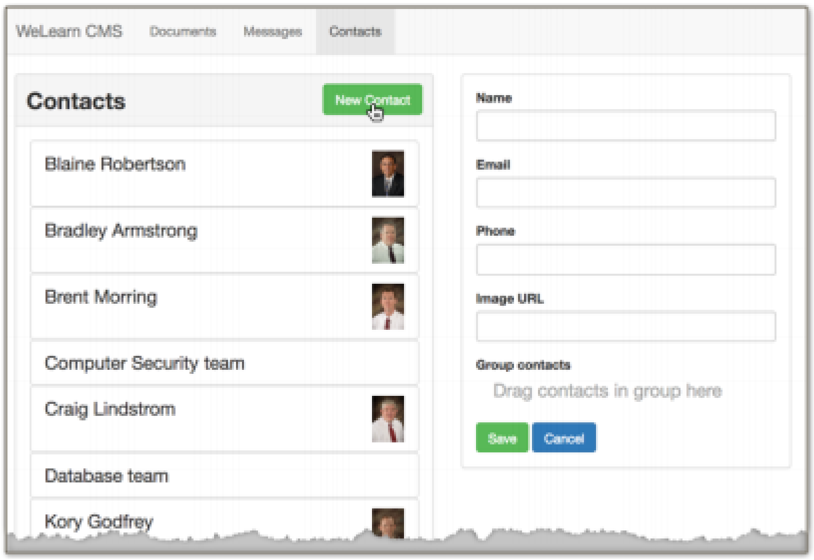
Implement a Form to Add and Edit Contacts

You also created the *ContactEditComponent* in the last lesson. We need to add a form to this component to allow the end user to create and edit contacts.

Create the Form to Add and Edit Contacts

The base HTML and CSS files for the *ContactsEditComponent* are provided for you in the *lesson8Files* folder that you downloaded and uncompressed earlier. Locate this folder, copy and paste the *contact-edit.component.html* and *contact-edit.component.css* files into the corresponding files of your project.

Open a browser to display your application. Select the *Contacts* feature and then select the *New Contact* button in the *ContactListComponent*. The *ContactEditComponent* with the new form should appear as below:



Validate the Fields in the Contact Edit Form

The *Name* and *Email* fields in the form are required. We also want to verify that valid values are entered in the *Email* and *Phone* fields. The end user should not be allowed to submit the form until these conditions are met. Angular gives us an easy way to enforce these restrictions and make the user interface more user-friendly when one of these errors is violated.

Open the *contact-edit.component.html* file and follow the instructions below to implement these changes. The changes that you need to make are very similar to those you made to validate the form in the *document-edit.component.html* file earlier.

1. Locate the *<form>* element and define a local reference (*#f*) in the *<form>* element and assign a

reference to the *FormGroup* (*ngForm*) to it.

2. Add an *ngSubmit* event listener to the *<form>* element to execute a method named *onSubmit()* when the form is submitted. Pass a reference (*f*) to the *FormGroup* assigned to this form to the *onSubmit()* method.
3. Map each of the input fields in the form to a corresponding property of the *Contact* object.
   a. Add the *name* attribute to each input field in the form.
   b. Add the *ngModel* directive to each input field and assign the corresponding property of the *Contact* object (for example, *[ngModel]=contact?.name*).
   c. Add a *local reference* (e.g. *#name*) to the field and assign the *ngModel* for this field. This allows you to reference the *value* and the *state* of the field in other HTML tags in the form.
4. The *name* and *email* input fields are required in the form:
   a. Make these input fields required.
   b. Locate the *<div>* near the bottom of the form containing the input buttons. Add a new *<div>* immediately above this *<div>* to display the following error message: “The Name and Email fields are required.” Apply the *alert* and *alert-danger* Bootstrap CSS classes to this new *<div>.* Add an *ngIf directive to load and display the *<div>* only when the *name* input field is *untouched* and *not valid*, or when the *email* field is *untouched* and *not valid*.
5. The *email* and *phone* fields must be validated against a specific regular expression pattern. Display an error message when the value entered in the fields violates this pattern. You can learn more about how to define your own regular expressions by working through the tutorial at **regexone.com** (https://regexone.com/) if you are interested.
   a. Add the following patterns to these input fields:
      i. The pattern for the *email* input field is the following:

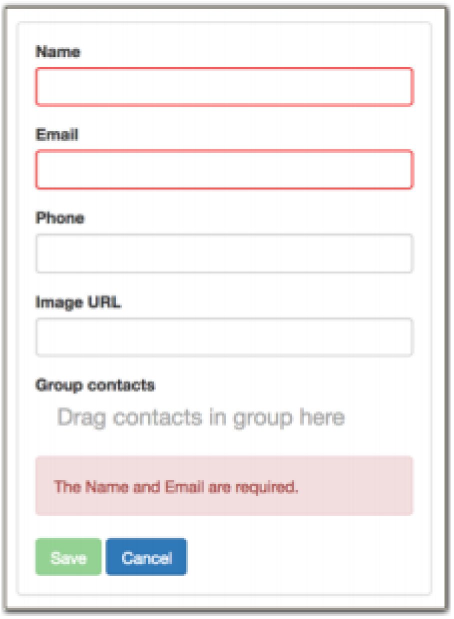
         pattern=“[a-z0-9._%+-]+@[a-z0-9.-]+\.[a-z]{2,3}$”

      ii. The pattern for the *phone* input field is the following:

         pattern=“\D*([2-9]\d{2})(\D*)([2-9]\d{2})(\D*)(\d{4})\D*"

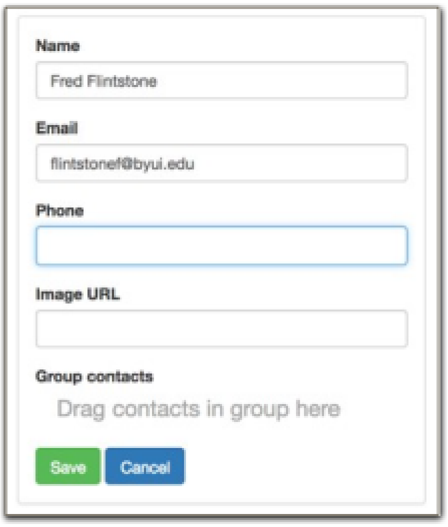
   b. Add a *<div>* immediately after the *email* input field to display an error message (for example, “*Enter a valid Email address*”) when the value of the *email* field is both *touched* and *not valid*.
   c. Add a *<div>* immediately after the *phone* input field to display an error message (for example, “*Enter a valid phone number*”) when the value of the *phone* field is both *touched* and *not valid*.
6. Locate the *Save* <button> and add an *ngIf directive to disable it when the form is not valid.
7. Save your changes and open your browser to view your app. Select the *Contacts* menu item to

display the list of contacts. Click on the *New Contact* button. The *Save* button should be disabled. The *Name* and *Email* fields will be highlighted and the error message displayed at the bottom of the form once the user enters and exits those fields without entering valid values.



The Name and Email are required.

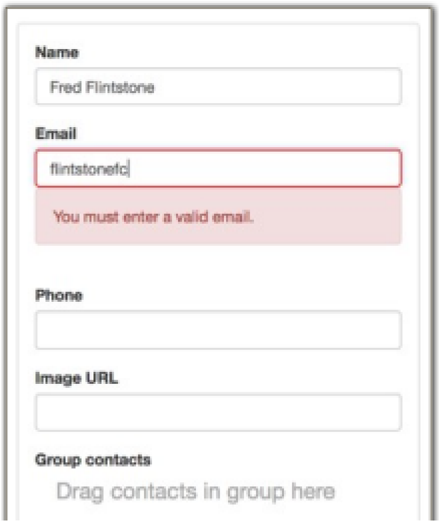
8. Enter valid values for the *Name* and *Email* fields. Notice the error message is no longer displayed and the *Save* button is now enabled.



Fred Flintstone

flintstonef@byui.edu

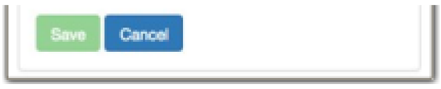
9. Edit the *Email* field and type in an invalid email address. An error message should be displayed under the *Email* field and the *Save* button should again be disabled.



Fred Flintstone

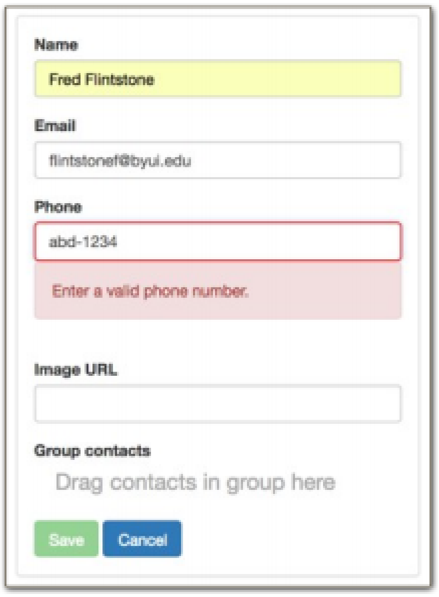
flintstonefc

You must enter a valid email.



10. Type in a valid email address in the *Email* field and then type in an invalid phone number in the *Phone* field. The error message below the *Email* field should no longer be displayed, and an error message below the *Phone* field displayed. The *Save* button should be also disabled.

11. Enter valid values in the *Name*, *Email,* and *Phone* fields. The *Save* button should now be enabled and no error messages should be displayed.



Implementing the Behavior of the Contact Edit Form

The behavior of the *ContactEditComponent* component is very similar to the behavior implemented for the *DocumentEditComponent*. It must determine if a new or existing contact is being added or edited when the component is loaded. If the form is being used to edit a contact, the contact must be retrieved from the contact service. When the end user selects the *Save* button and submits the form, it needs to call the appropriate method to either add a new contact or update an existing contact in the *ContactService*. Use the *DocumentEditComponent* as a guide for the implementation of the behavior in the *ContactEditComponent*.

Open the *contact-edit.component.ts* file and follow the instructions below to implement this behavior:

1. Add the following properties and inject the *ContactService*, *Router,* and *ActivatedRoute* services into the *ContactEditComponent* class as shown below:

```
export class ContactEditComponent implements OnInit {
  originalContact: Contact;
  contact: Contact;
  groupContacts: Contact[] = [];
  editMode: boolean = false;
  id: string;

  constructor(
      private contactService: ContactService,
      private router: Router,
      private route: ActivatedRoute) {
      }
}
```

2. Implement the *ngOnInit()* lifecycle method in the *ContactEditComponent* class. This method needs to subscribe to the current route and get the value of the *id* parameter (if it exists) from the URL. If the *id* has a value, the form is being used to edit an existing contact. Get the contact and make a clone of it.

   Here is the algorithm for the *ngOnInit()* method. It is almost identical to the *ngOnInit()* method implemented in the *DocumentEditComponent*:

```
ngOnInit() {
   route.subscribe (
     (params: Params) => {
        id = value of id parameter in params list
        if id parameter is undefined or null then
           editMode = false
           return
        endif
        originalContact = contactService.getContact(id)
        if originalContact is undefined or null then
            return
        endif
        editMode = true
        contact = clone originalContact

        if the contact has a group then
           groupContacts = clone the contact's group
        endif
   })
 }
```

   The only difference is that the algorithm must determine if the contact has a group (in other words, the group property has a value). If it does, create a cloned copy of the array of *Contact* objects assigned to the *group* property in the *originalContact* and assign the cloned array to the *groupContacts* property. Use the JSON *parse()* and *stringify()* methods to make a clone of the *originalContact.*

3. Create a new method in the *ContactEditComponent* class named *onSubmit()*. This method is called whenever the form is submitted. This method is almost identical to the *onSubmit()* method in the *DocumentEditComponent* class you created earlier.

4. Create the *onCancel()* method in the *ContactEditComponent* class to cancel the form and route back to the *ContactsComponent*. Implement the following algorithm for the onCancel() method:

```
onCancel() {
    route back to the '/contacts' URL
    }
```

Implementing Drag and Drop to Manage Group Contacts

Some contacts in the contact list can have a group of other contacts assigned to them. For example, the Computer Security Team contact has a list of all of the contacts that are assigned to that team. Select this contact in the contact list. You should see the list of contacts assigned to this team in the *ContactDetailComponent*.

We need some way to be able to assign and remove contacts from such a group. A user-friendly way to do this is to drag a contact from the *ContactListComponent* into the *ContactEditComponent*.

Unfortunately, Angular does not have a native capability to implement drag-and-drop, but several people have created their own Angular modules to implement this feature. In this section, you will learn how easy it is to reuse modules that other people have created in your application.

To implement this drag-and-drop functionality in your application, you will need to do the following:

- Install the drag-and-drop module and import it into your app.
- Add directives to each *<cms-contact-item>* tag generated in the *ContactListComponent* to indicate to Angular that these components are draggable elements.
- Add a directive to the target *<div>* in the *ContactEditComponent* to indicate to Angular where the selected contacts may be dragged to.
- Add a method to the *ContactEditComponent* class that will be called automatically whenever a contact is dragged into the target *<div>*.
- Add a method to determine if the contact dragged into the target *<div>* is already in the group.
- Add methods to actually add the contact dragged into the group to the group array in the *ContactEditComponent*.

Install the Drag and Drop Angular 2 Module

Angular components and modules can be installed using the Node Package Manager (*npm*) utility. We will use the *ng2-dnd* module to implement the drag-and-drop functionality needed in our application. This module is implemented by a third-party. View the documentation for this component by selecting the link below.

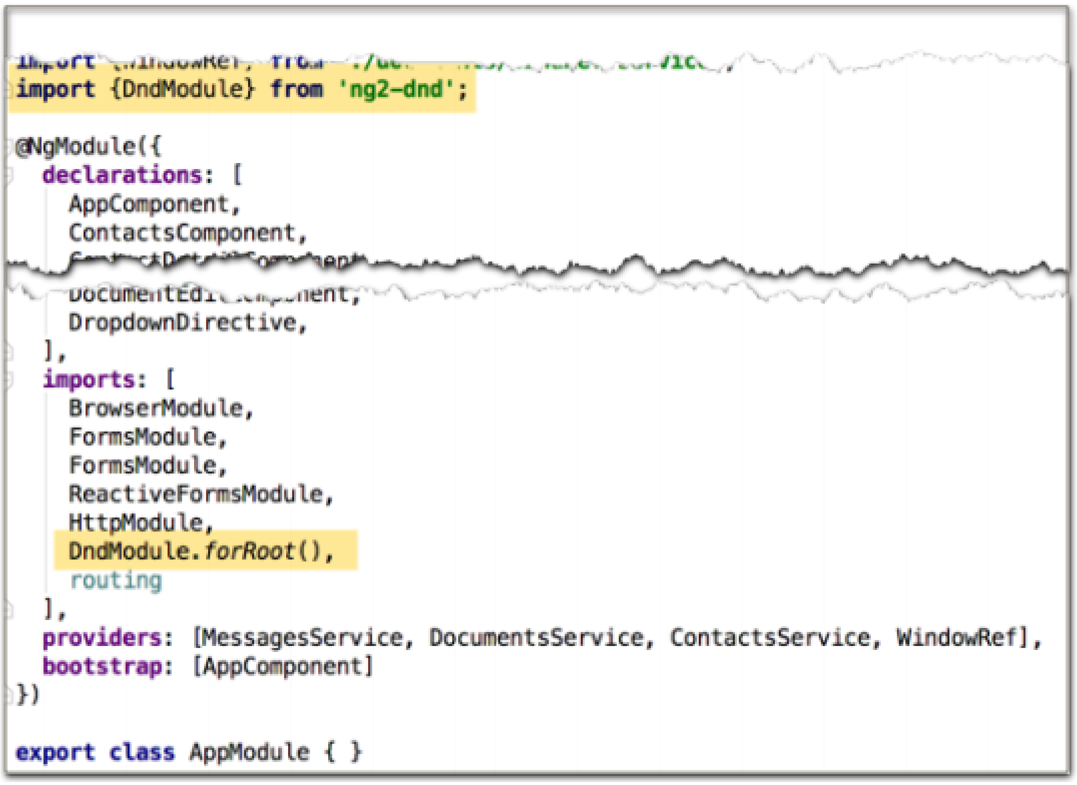
ng2-dnd documentation (https://www.npmjs.com/package/ng2-dnd)

Follow the instructions below to install the *ng2-dnd* module:

1. Open the terminal in the VS Code. Enter the following *npm* command to download and install the *ng2-dnd* module into your application. Ignore any warnings that are reported:

npm install ng2-dnd --save

2. Open the *app.module.ts* file and add an import statement to tell your app where the *ng2-dnd* module is installed. Then add the *DndModule* to the array assigned to the *imports* property as shown below:



```
import {WindowRef} from './[illegible]';
import {DndModule} from 'ng2-dnd';

@NgModule({
  declarations: [
    AppComponent,
    ContactsComponent,
    ContactDetailComponent,
    DocumentEditComponent,
    DropdownDirective,
  ],
  imports: [
    BrowserModule,
    FormsModule,
    FormsModule,
    ReactiveFormsModule,
    HttpModule,
    DndModule.forRoot(),
    routing
  ],
  providers: [MessagesService, DocumentsService, ContactsService, WindowRef],
  bootstrap: [AppComponent]
})

export class AppModule { }
```

Add the Drag and Drop Directives

We need to tell Angular where the items can be dragged to in our application; this is referred to as the *drop zone*. Then, we need to define which items are draggable. This is accomplished by adding the *DndModule* directives to each HTML element that is draggable, and to the HTML container (*<div>*) that will act as the *drop zone* for where the items can be dragged. In the *CMS* application, the end user should be able to drag contacts from the *ContactListComponent* into the group list in the *ContactEditComponent* to add a contact to the group.

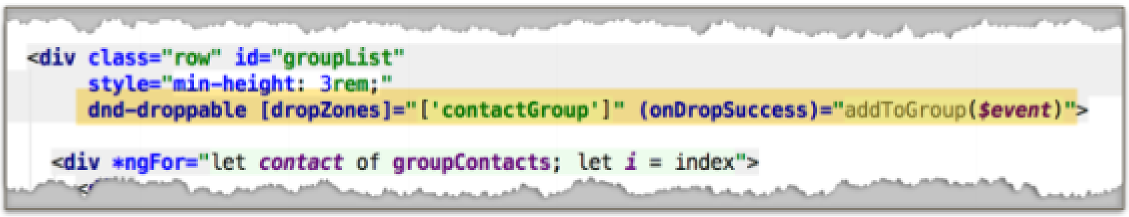
Follow the instructions below to define the *drop zone* in the *ContactEditComponent* and to specify that each contact in the contact list in the *ContactListComponent* is draggable:

1. The *drop zone* will be defined inside the *groupList <div>* in the *ContactEditComponent*. Open the *contact-edit.component.html* file and locate this *<div>*. Add the following *DndModule* directives to this *<div>*:

   a. Add the *dnd-droppable* directive to indicate that this element will receive draggable items as shown below.

   b. Add the *[dropZones]* directive and assign a name (for example, *contactGroup*) to this drop zone. The *[dropZones]* directive requires that you assign an array of names to it. We only defined the *contactGroup* name in the array.

   c. Add the *(onDropSuccess)* directive to specify the method that will be called when a contact is dragged into this *drop zone*.



```
<div class="row" id="groupList"
     style="min-height: 3rem;"
     dnd-droppable [dropZones]="['contactGroup']" (onDropSuccess)="addToGroup($event)">

  <div *ngFor="let contact of groupContacts; let i = index">
```

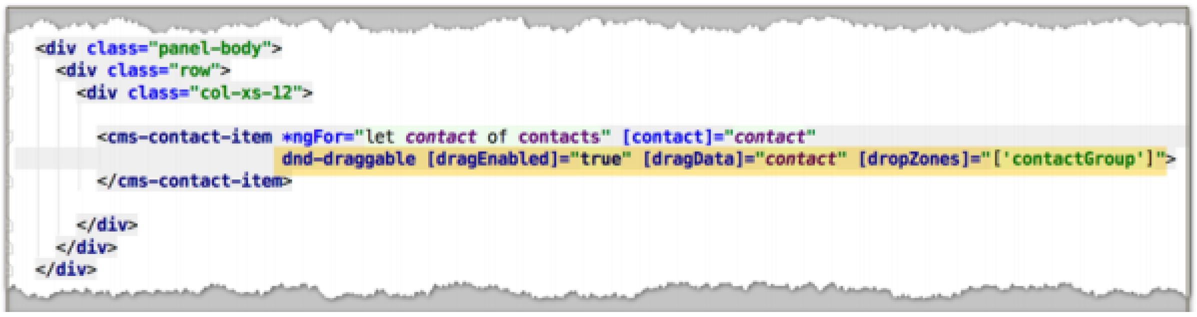
2. All of the *<cms-contact-item>* elements in the *ContactListComponent* need to be identified as *draggable*. Open the *contact-list.component.html* file. Add the following directives to the *<cms-contact-item>* tag:

   a. Add the *dnd-draggable* directive to indicate that this element is draggable as shown below.

   b. Add the *[dragEnabled]* directive to indicate when the element is draggable. The element is draggable when the value of this directive is *true*. It is not draggable when its value is *false*.

   c. Add the *[dragData]* directive to indicate what data is to be associated with the element when it is dragged.

   d. Add the *[dropZones]* directive to indicate the *drop zones* that this element may be dragged to. Again, the value assigned to this directive must be an array of the names of the *drop zones*.



```
<div class="panel-body">
  <div class="row">
    <div class="col-xs-12">

      <cms-contact-item *ngFor="let contact of contacts" [contact]="contact"
                        dnd-draggable [dragEnabled]="true" [dragData]="contact" [dropZones]="['contactGroup']">
      </cms-contact-item>

    </div>
  </div>
</div>
```

Add Methods to Add and Delete Contacts in the Group

We need to add methods to the *ContactEditComponent* class to add a contact to the group, to determine if the contact is already in the group, and to delete a contact from the group.

Open the *contact-edit.component.ts* file and follow the instructions below to implement these three methods.

Implement the Method to See if Contact is Already in the Group

Add a new method in the *ContactEditComponent* class called *isInvalidContact()*. The purpose of this method is to determine if the *newContact* to be added to the group is already in the current contact's group array.

```
isInvalidContact(newContact: Contact) {
       if (!newContact) {// newContact has no value
         return true;
       }
```

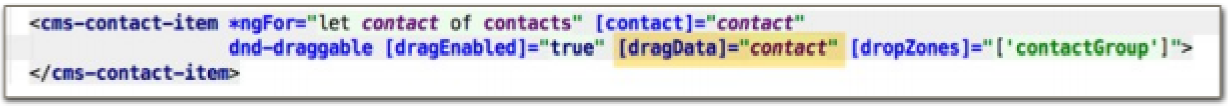
```
      if (this.contact && newContact.id ===
this.contact.id) {
        return true;
      }
      for (let i = 0; i < this.groupContacts.length; i++)
{
        if (newContact.id ===
this.groupContacts[i].id) {
          return true;
        }
      }
      return false;
    }
```

This method first determines if a new contact was passed to the method. If no contact was passed, it exits the method. It then checks to see if the contact being added to the contact list is the same as the current contact being edited (in other words, you cannot add yourself to your own group). It then loops through all of the contacts in the current contact's group array to see if the *newContact.id* property is equal to the *id* property of any *contact* in the group array. Return true if a match is found, else return false.

Implement the Method to Add a New Contact to the Group

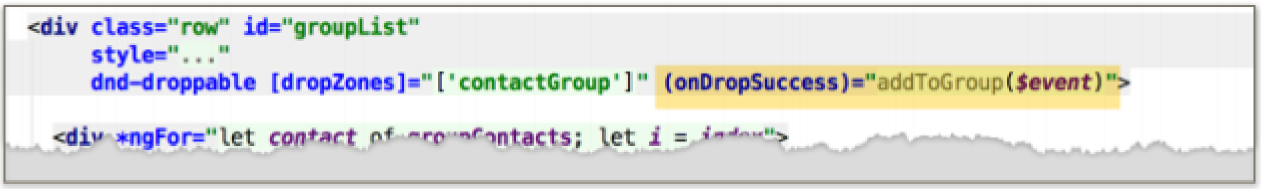
Create a new method to add a contact to the contact's group. You need to understand how the drag-and-drop operation works before we implement this method.

When the end user drags a <cms-contact-item> component in the *ContactListComponent* component into the *drop zone* defined in the *ContactEditComponent*, the selected contact in the *ContactItemComponent* is assigned to the drag event. This is done by assigning the value of the *contact* property in the *ContactItemComponent* to the *[dragData]* directive in the *<cms- contact-item>* tag in the HTML of the *ContactListComponent* as shown below:



```
<cms-contact-item *ngFor="let contact of contacts" [contact]="contact"
                  dnd-draggable [dragEnabled]="true" [dragData]="contact" [dropZones]="['contactGroup']">
</cms-contact-item>
```

The *drop zone* is defined in the *<div>* tag in the *ContactEditComponent* as shown below.



```
<div class="row" id="groupList"
     style="..."
     dnd-droppable [dropZones]="['contactGroup']" (onDropSuccess)="addToGroup($event)">

  <div *ngFor="let contact of groupContacts; let i = index">
```

When a *<cms-contact-item>* tag is dragged and dropped into this *drop zone*, the *onDropSuccess* event is fired and the *addToGroup()* method is called and passed the *$event*. The *$event* argument contains a reference to the *onDropSuccess* event that contains the *dragData* passed with the element dragged into the *drop zone*.

Implement the *addToGroup()* method in the *contact-edit.component.ts* file shown below:

Implement the *addToGroup()* method in the *contact-edit.component.ts* file shown below:

```
addToGroup($event: any) {
  const selectedContact: Contact = $event.dragData;
  const invalidGroupContact =
this.isInvalidContact(selectedContact);
  if (invalidGroupContact){
    return;
  }
  this.groupContacts.push(selectedContact);
}
```

This method first gets the *dragData* from *$event* passed into the method, and assigns it to the *selectedContact* variable. The *isInvalidContact()* method is then called to determine if the *selectedContact* is already in the group. If the contact is already in the group, the method exits. When the *selectedContact* is not in the group, the *selectedContact* is added to the *groupContacts* array.

Implement the Method to Remove a New Contact from the Group

Add a method to the *ContactEditComponent* class to remove the selected contact from the contact's group. Implement the onRemoveItem() method below:

```
onRemoveItem(index: number) {
  if (index < 0 || index >=
this.groupContacts.length) {
    return;
  }
  this.groupContacts.splice(index, 1);
}
```

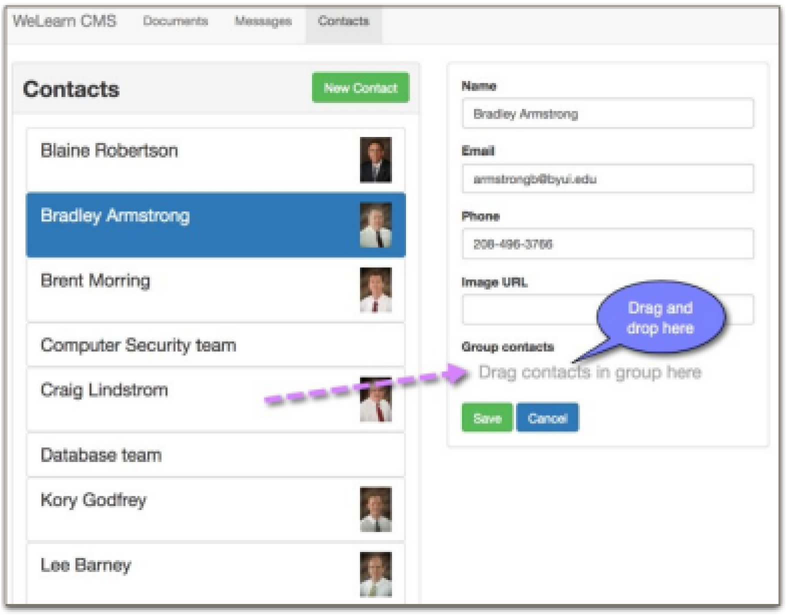
The index (*index*) position of the contact in the *groupContacts* array is passed to the *onRemoveItem()* method. If the index is outside the range of the array, the method exits. The *splice()* method is called to delete the item from the array.

Save your changes.

Test Adding and Removing Contacts from the Contact's Group.

Open your browser and view your app. Choose the *Contacts* feature and then select one of the contacts to load and display the *ContactDetailComponent*. Select the *Edit* button to edit the contact.

Select and drag one or more contacts in the contact list to the *Group Contacts <div>* as shown below:


WeLearn CMS Documents Messages Contacts

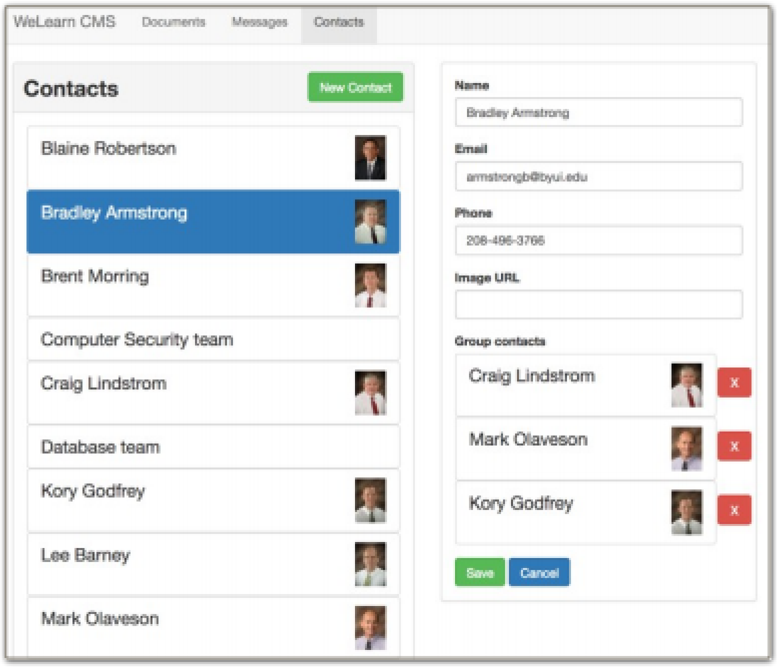
Contacts New Contact

Blaine Robertson
Bradley Armstrong
Brent Morring
Computer Security team
Craig Lindstrom
Database team
Kory Godfrey
Lee Barney

Name: Bradley Armstrong
Email: armstrongb@byui.edu
Phone: 208-496-3766
Image URL:
Group contacts: Drag contacts in group here

Save Cancel

They should be added to the group:


WeLearn CMS Documents Messages Contacts

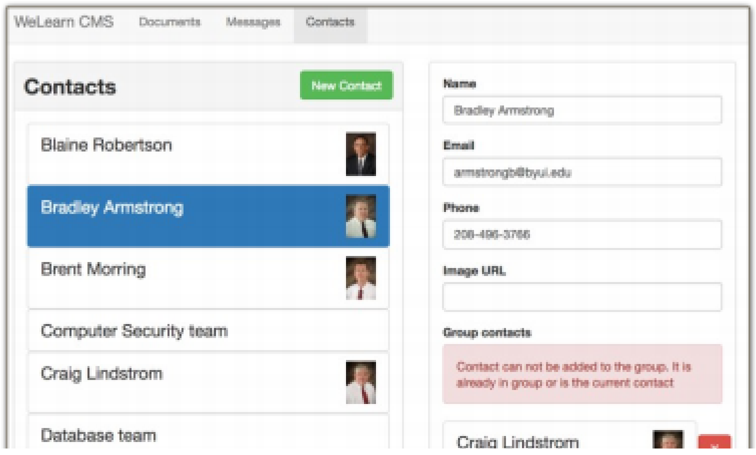
Contacts New Contact

Blaine Robertson
Bradley Armstrong
Brent Morring
Computer Security team
Craig Lindstrom
Database team
Kory Godfrey
Lee Barney
Mark Olaveson

Name: Bradley Armstrong
Email: armstrongb@byui.edu
Phone: 208-496-3766
Image URL:
Group contacts:
Craig Lindstrom x
Mark Olaveson x
Kory Godfrey x

Save Cancel

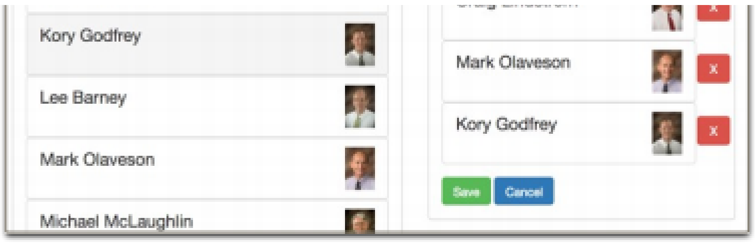
Now, select one of the contacts already in the group, and drag it to the list of contacts in the group. An error message should display and appear as shown below:


WeLearn CMS Documents Messages Contacts

Contacts New Contact

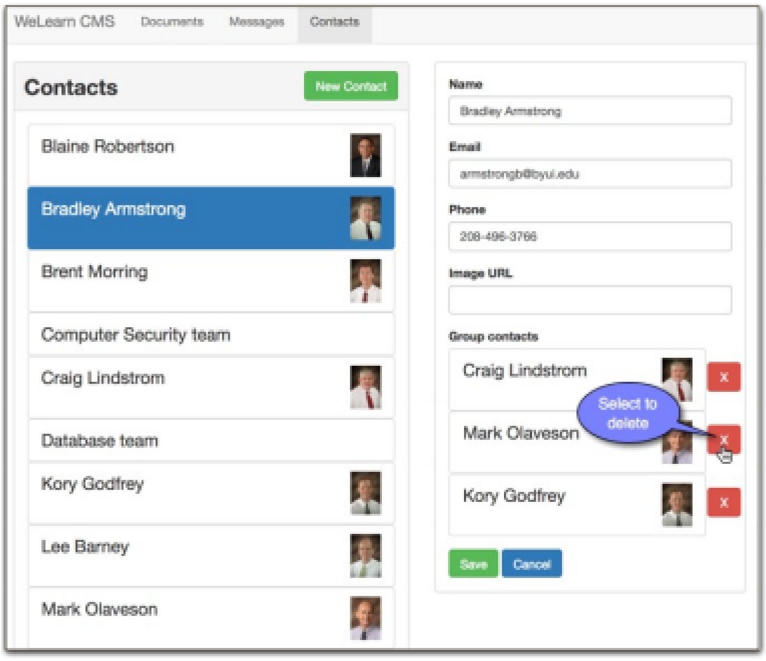
Blaine Robertson
Bradley Armstrong
Brent Morring
Computer Security team
Craig Lindstrom
Database team

Name: Bradley Armstrong
Email: armstrongb@byui.edu
Phone: 208-496-3766
Image URL:
Group contacts:
Contact can not be added to the group. It is already in group or is the current contact
Craig Lindstrom x


Kory Godfrey
Lee Barney
Mark Olaveson
Michael McLaughlin

Mark Olaveson
Kory Godfrey

Delete one or more contacts in the group by selecting the red *X* button:



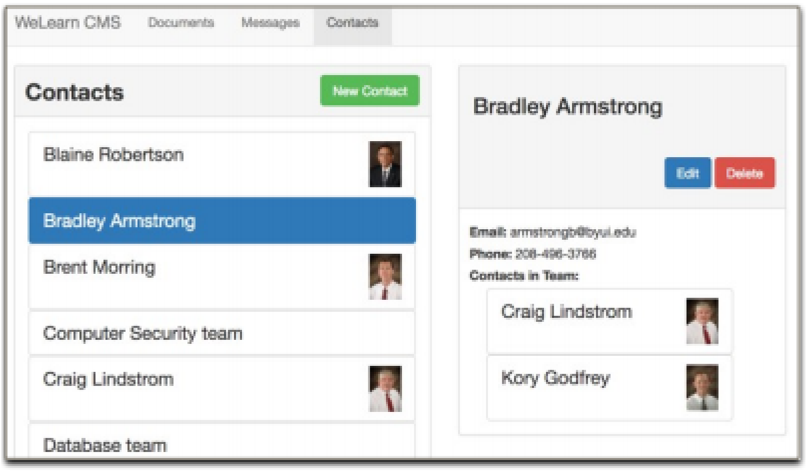
Contacts

Blaine Robertson
Bradley Armstrong
Brent Morring
Computer Security team
Craig Lindstrom
Database team
Kory Godfrey
Lee Barney
Mark Olaveson

Name
Bradley Armstrong
Email
armstrongb@byui.edu
Phone
208-496-3766
Image URL

Group contacts
Craig Lindstrom
Mark Olaveson
Kory Godfrey

Select the *Save* button and then go back and select the same contact in the contact list. The list of group contacts should appear in the *ContactDetailComponent*.



Contacts

Blaine Robertson
Bradley Armstrong
Brent Morring
Computer Security team
Craig Lindstrom
Database team

Bradley Armstrong

Email: armstrongb@byui.edu
Phone: 208-496-3766
Contacts in Team:
Craig Lindstrom
Kory Godfrey

After completing your assignment, push the CMS project to your remote GitHub repository.

In the comment section, list the URL of your remote GitHub repository.

Then, make a short video of your project, host it online, and paste the URL in the Website URL section.