FULL WEB STACK DEVELOPMENT {js} CIT 366

Lesson 5 - Assignment

An Angular service contains one or more functions that can be used by any Angular component. In a sense, it provides services (or functionality) to the components. An Angular service is similar to a component in that it has a class, but it does not have an HTML template or any CSS associated with it. Normally, we create services to implement the Model Layer of an Angular application. The Model Layer in an application is responsible for creating, reading, updating and deleting data objects. In this assignment, you will need to create services for the document, message and contact objects used in the `cms` application. Initially, each of these services will contain only a functions to get a single object based on its `id` property and to return the entire list of objects. Later we will add functions to create, update and delete these objects.

Create the Contacts Service

Create an Angular service for managing the `Contact` data objects in your application. This service will initially contain functions to initialize the list of contacts used in the application, to get all of the contacts in the list and to get a single contact in the list. Follow the instruction below.

1. Open the terminal in WebStorm. Change to the `Contacts` directory in your application. Use the `ng` command to create a new service called `contact`. Be sure to use the option to create the service in the same directory).

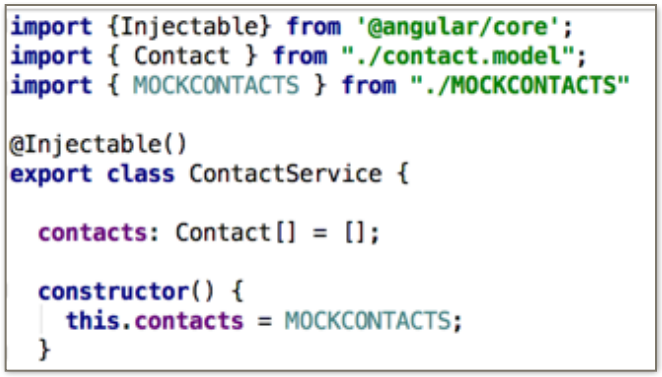
2. Open the `contact.service.ts` file and create a class variable called `contacts` whose datatype is an array of `Contact` objects. Initialize the variable with an empty array (i.e., `[]`) . You will need to import the `Contact` model class.

3. Eventually, the list of `Contacts` will be retrieved from a database running on a server. We will initialize the array with a predefined list of `Contact` objects for now. This list has been already created for you and can be downloaded by clicking on the link below.

[lesson5Files.zip]()

Unzip the file. Locate and open the `lesson5Files` directory. Copy the `MOCKCONTACTS.ts` file located in `lesson5Files` directory to the `contacts` directory in your `cms` project.

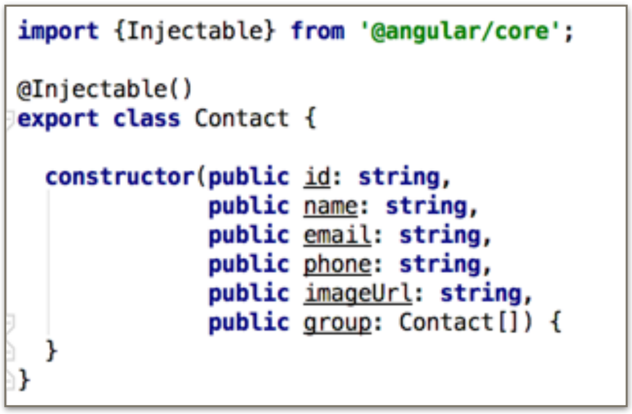
4. Open the `contact.service.ts` file again and import the `MOCKCONTACTS.ts` file. Inside the constructor function, assign the value of the `MOCKCONTACTS` variable defined in the `MOCKCONTACTS.ts` file to the `contacts` class variable in the `ContactService` class.



```
import {Injectable} from '@angular/core';
import { Contact } from "./contact.model";
import { MOCKCONTACTS } from "./MOCKCONTACTS"

@Injectable()
export class ContactService {

  contacts: Contact[] = [];

  constructor() {
    this.contacts = MOCKCONTACTS;
  }
```

5. The contacts defined in the `MOCKCONTACTS.ts` file contains URL references to images in the `assets` directory in your project. The images for each contact can be located in the `lesson5Files` directory that you unzipped. Locate the `images` directory inside the `lesson5Files` directory and copy and paste it to the `assets` directory in you `cms` project.

6. The `contacts.model.ts` file you created earlier may not be compatible with the Contacts objects defined in the `MOCKCONTACTS.ts` file. Open your file and modify your `Contact` class is compatible with the definition below.



```
import {Injectable} from '@angular/core';

@Injectable()
export class Contact {

  constructor(public id: string,
              public name: string,
              public email: string,
              public phone: string,
              public imageUrl: string,
              public group: Contact[]) {
  }
}
```

7. The `ContactService` needs a function to return the list of contacts. Add a new function to the `ContactService` class with the following function signature.

```
  getContacts(): Contact[]
```

   Inside this function return a copy of the contacts array. Use the JavaScript `slice()` function to make a copy of the array. The `getIngredients()` function in the `shopping-list.service.ts` file in the Recipe Book project (`prj-services-final`) illustrates for how to do this.

8. The `ContactService` also needs a function to find and a specific Contact object in the `contacts` array. Add a new function to the `ContactService` class with the following function signature.

```
  getContact(id: string): Contact
```

   This function will search through all of the `Contact` objects in the `contacts` array and returns the `Contact` object whose `id` property is equal to the value if the `id` input parameter. It returns the `Contact` object found if successful; otherwise, it returns a `null` value to indicate that the contact was not found. Implement this function.

   Here is the algorithm you can use for this function:

```
getContact(id: string): Contact {
   FOR each contact in the contacts list
      IF contact.id equals the id THEN
         RETURN contact
      ENDIF
   ENDFOR
   RETURN null
}
```

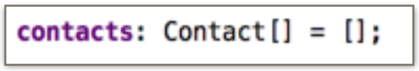
Injecting and using the Contacts service

Previously we hardcoded a dummy list of contacts In the `ContactListComponent` to test our application. A better approach is to get the list of `contacts` stored in the `ContactService.` This requires that we inject the `ContactService` into the `ContactListComponent`. Follow the instructions below to inject the `ContactService` into the `ContactListComponent`, and then call its `getContacts()` function to get the list of `Contacts` to be displayed. A similar of example of how to do this can be found in the `recipe-list.component.ts` file in the Recipe Book project (`prj-services-final`).

1. Open the `contact-list.component.ts` file and modify the `constructor()` function to inject the `ContactsService` into the `ContactListComponent` class. You will need to add a statement to the top of the file to import the `ContactService` class.

2. We no longer need to initialize the contacts array in the `ContactListComponent` class with a list of dummy contacts. Instead we will get the list of contacts from the `ContactService`.

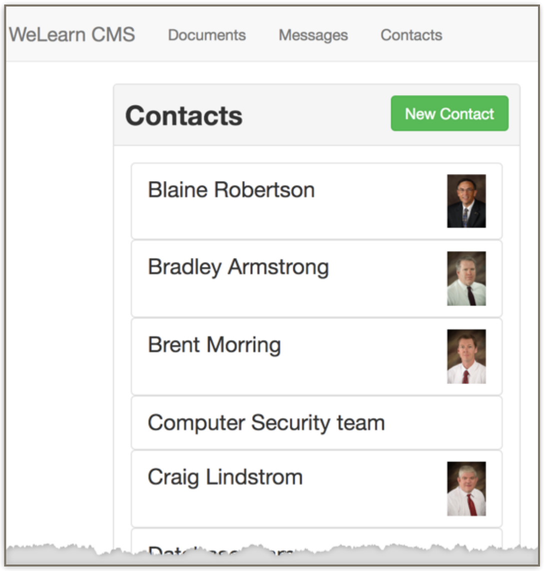
    i. Open the `contact-list.component.ts` file and delete the code that initializes the `contacts` array with the list of dummy `Contact` objects and replace it with empty array instead.



```
contacts: Contact[] = [];
```

b. Modify the `constructor()` function to call the `getContacts()` function in the `ContactsService` and assign the array of contacts returned from the function to the `contacts` class variable in the `ContactListComponent`.

3. Serve up your application and switch to your browser. Select the Contacts feature. You screen should be similar to the figure below.



Contacts

Blaine Robertson

Bradley Armstrong

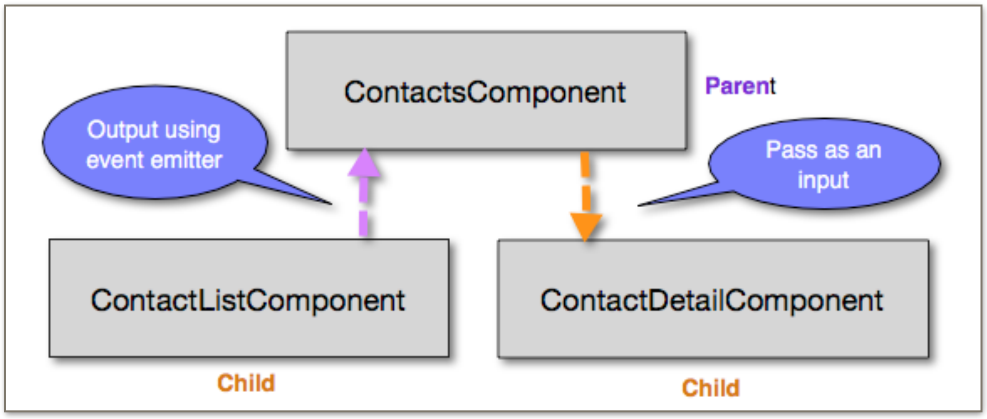
Brent Morring

Computer Security team

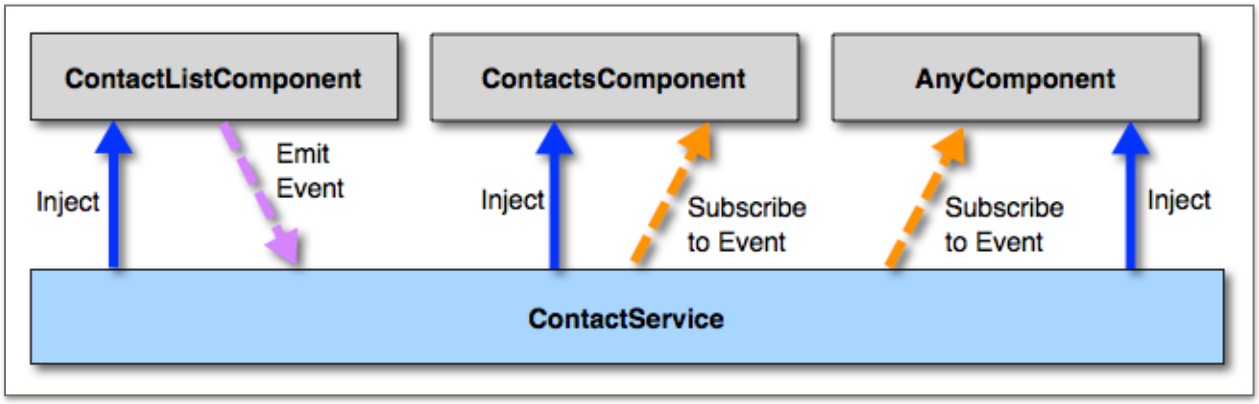
Craig Lindstrom

Using the ContactService for Cross-Component Communication

Currently, when an end user selects a contact in the `ContactListComponent`, the details of the selected `Contact` are displayed in the `ContactDetailComponent`. This is done by outputting and emitting the selected `Contact` back up to its parent, the `ContactsComponent`. The `ContactsComponent` then passed the emitted `Contact` down to the `ContactDetailComponent` where all of the detailed information of the selected `Contact` was displayed.



This approach is somewhat messy and restrictive in that the selected `Contact` object can only be emitted back up to its parent component. A better and simpler approach is to create the `EventEmitter` in the `ContactService` and then inject that service into any component that wants to emit or subscribe to the event. This allows any number of components to be able to pass data to each other directly as shown below. You are no longer restricted to only passing and sharing data from a child component to its parent component.



© 2017 R. Kent Jackson

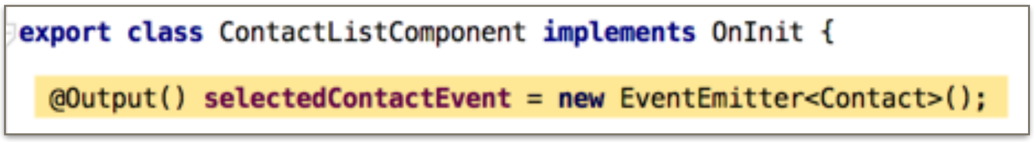
Implement cross-component communication

Follow the directions below to pass the selected contact in the `ContactListComponent` directly to the `ContactsComponent` using cross component communication.

1. Add an `EventEmitter` to the `ContactService`. Open the `contact.service.ts` file and define a new variable called `contactSelectedEvent` and assign a new `EventEmitter` object of the `Contact` datatype to the variable. See a similar of example of how to do this in the `recipe.service.ts` file in the Recipe Book project (`prj-services-final`).

2. Modify the `ContactListComponent` to emit the `contactSelectedEvent` whenever the end user selects a contact in the list. The `recipe-list.component.ts` file in the Recipe Book project (`prj-services-final`) as a similar example of how to do this.

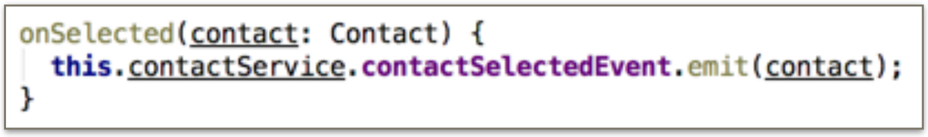
    a. Open the `contact-list.component.ts` file and delete the old `EventEmitter` variable (e.g., `selectedContactEvent`). It is no longer needed because we will be using the new `contactSelectedEvent` emitter you created in the `ContactService` class instead.

    ```
    export class ContactListComponent implements OnInit {

      @Output() selectedContactEvent = new EventEmitter<Contact>();
    ```

    b. Inject the `ContactService` into the `ContactListComponent` so that the new `contactSelectedEvent` emitter can be referenced in the `ContactListComponent`.

    c. Modify the `onSelected(contact:Contact)` function in the `ContactListComponent` class to emit the `contactSelectedEvent` with the `Contact` object passed into the function.

    ```
    onSelected(contact: Contact) {
      this.contactService.contactSelectedEvent.emit(contact);
    }
    ```

3. Finally, we need to modify the `ContactDetailComponent` to subscribe to and watch for the `contactSelectedEvent` event to occur. See the `recipe.component.ts` and `recipe.component.html` files in the Recipe Book project (`prj-services-final`) for a similar example of how to do this.

   a. Open the `contacts.component.html` file. The `<cms-contact-list>` tag no longer needs to detect the `selectedContactEvent`. Delete this code from the `<cms-contact-list>` tag.

   b. Open the `contacts.component.ts` file and inject the `ContactService` into the `ContactsCompnent`.

   c. Implement the `ngOnInit()` function and subscribe to the `contactSelectedEvent` in the `ContactService`. Implement an arrow function to receive the `Contact` object passed with the emitted event and assign it to the `contact` class variable in the `ContactsComponent`.

4. Save all of your changes, serve up your application and open the browser and select the Contacts feature. Select one of the contacts in the list. The contact's detailed information should be displayed the same as it did before you made these changes.

Implement the Document Service

We need to also add a `DocumentService` class to the Model Layer of our application. The `DocumentService` is responsible for creating, reading, updating and deleting documents in our application. Follow the instructions below to create and use the `DocumentService` to get the list of documents to be displayed in your `cms` application. This code will be very similar to `ContactService` except that you will be initializing and get documents instead of contacts.

1. Open the terminal window in WebStorm and change to the `documents` directory. Use the `ng` command to create the `documents` service in the `documents` directory.

2. Open the `documents.service.ts` file and add a class variable called `documents` whose datatype is an array of `Document` objects.

3. Locate and open the `lesson5Files` directory. Copy the `MOCKDOCUMENTS.ts` file located in `lesson5Files` directory to the `documents` directory in your `cms` project.

4. Open the `document.service.ts` file again and import the `MOCKDOCUMENTS.ts` file. Locate the constructor function in the `DocumentService` class and assign the value of the `MOCKDOCUMENTS` variable defined in the `MOCKDOCUMENTS.ts` file to the `documents` class variable in the `DocumentService` class.

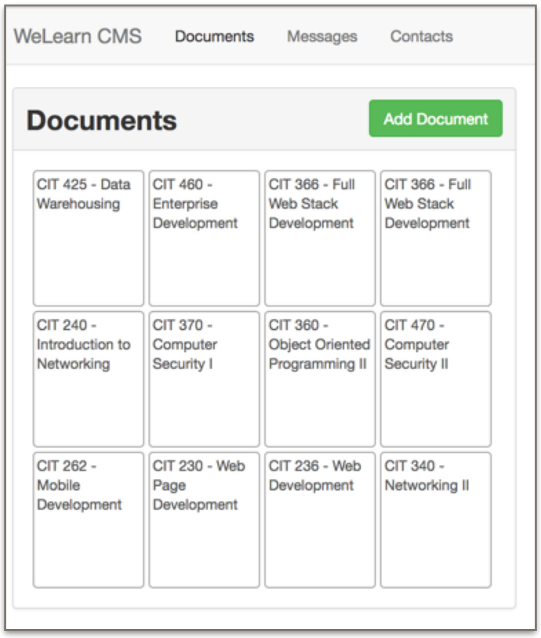
5. Add the `getDocuments()` and `getDocument(id: string)` functions to the `DocumentsService` class. These functions are responsible for getting the list of documents and a single document respectively in the list. These functions are almost identical to the `getContacts()` and `getContact(id: string)` functions you defined in the `ContactService` class.

6. Open the `document-list.component.ts` file and inject the new `DocumentService` into the `DocumentListComponent` class. You will need to add a statement to the top of the file to import the `DocumentService` class.

7. Replace the code that initializes the `documents` array with a list of dummy `Document` objects with an empty array instead.

8. Modify the `constructor()` function located in the `DocumentListComponent` class to call the `getDocuments()` function defined in the `DocumentService`. Assign the array returned from the function to the `documents` class variable.

9. Save your changes and open your browser. The list of Documents should be displayed as shown 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

Implement cross-component communication

Use cross-component communication to pass a contact selected in the contact list directly over to the `ContactDetailComponent` so that it can be displayed in the `ContactDetailComponent`. This is very similar to the code you implemented for cross-component communication between the `ContactListComponent` and the `ContactsComponent`.

Follow the instructions below.

1. Create a new `EventEmitter` in the `DocumentService` called `documentSelectedEvent`.

2. Modify the `DocumenListComponent` to emit the `documentSelectedEvent` in the `DocumentService`.

    a. Open the `document-list.component.ts` file and delete the old `EventEmitter` variable (e.g., `selectedDocumentEvent`). It is no longer needed because we will be using the new `EventEmitter` you created in the `DocumentService` instead.

    b. Inject the `DocumentService` into the `DocumentListComponent` so that we can reference the `contactSelectedEvent` emitter in the `ContactService`.

    c. Modify the `onSelected(document:Document)` function to now emit the `documentSelectedEvent` and pass it the `Document` object selected and passed into the function.

3. Modify the `DocumentDetailComponent` to subscribe to and watch for the `documentSelectedEvent` event.

    a. Open the `documents.component.html` file. The `<cms-document-list>` tag no longer needs to detect the `selectedDocumentEvent`. Delete this code from the `<cms-document-list>` tag.

    b. Open the `documents.component.ts` file and inject the `DocumentService` into the `DocumentsComponent`.

    c. Implement the `ngOnInit()` function and subscribe to the `documentSelectedEvent` in the `DocumentService`. Implement an arrow function to receive the `Document` object passed with the emitted event and assign it to the `document` class variable in the `DocumentsComponent`.

4. Save all of your changes, serve up your application and open the browser and select the Documents feature. Select one of the documents in the list. The document's detailed information should be displayed the same as it did before you made these changes.

Implement the MessageService

Create the `MessageService` for the Model Layer the `cms` application This code will be very similar to `ContactService` except that you will be initializing and get messages instead of contacts.

1. Open the terminal window in WebStorm and change to the `messages` directory. Use the `ng` command to create the `messages` service in the `messages` directory.

2. Open the `messages.service.ts` file and add a class variable called `messages` whose datatype is an array of `Message` objects.

3. Locate and open the `lesson5Files` directory. Copy the `MOCKMESSAGES.ts` file located in `lesson5Files` directory to the `messages` directory in your `cms` project.

4. Open the `message.service.ts` file again and import the `MOCKMESSAGES.ts` file. Locate the constructor function in the `MessageService` class and assign the value of the `MOCKMESSAGES` variable defined in the `MOCKMESSAGES.ts` file to the `messages` class variable in the `MessageService` class.

5. Add the `getMessages()` and `getMessage(id: string)` functions to the `MessagesService` class. These functions are responsible getting the list of messages and a single message respectively. These functions are almost identical to the `getContacts()` and `getContact(id: string)` functions you defined in the `ContactService` class.

6. Open the `message-list.component.ts` file and inject the new `MessageService` into the `MessageListComponent` class. You will need to add a statement to the top of the file to import the `MessageService` class.

7. Replace the code that initializes the `messages` array with a list of dummy messages with an empty array instead.

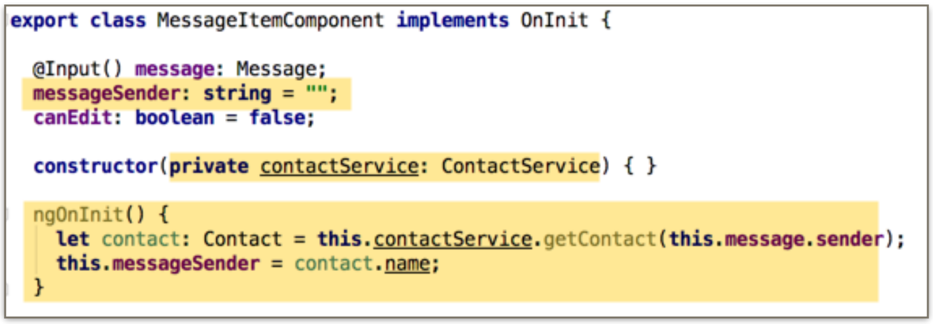
8. Modify the `constructor()` function located in the `MessageListComponent` class to call the `getMessages()` function in the `MessageService`. Assign the array returned from the function to the `messages` class variable.

9. The value of the `sender` property for each `Message` defined in the MOCKMESSAGE.ts file now contains a reference to the `id` property of the `Contact` that sent the message instead of the name of the sender of the message. You will need to modify the `MessageItemComponent` to first get the `Contact` object with the specified `id`, and then get the `name` property from the `Contact` to be displayed as the sender of the message.

    a. Open the `message-item.component.ts` file and inject `ContactService` into the constructor() function of the `MessageItemComponent` class.

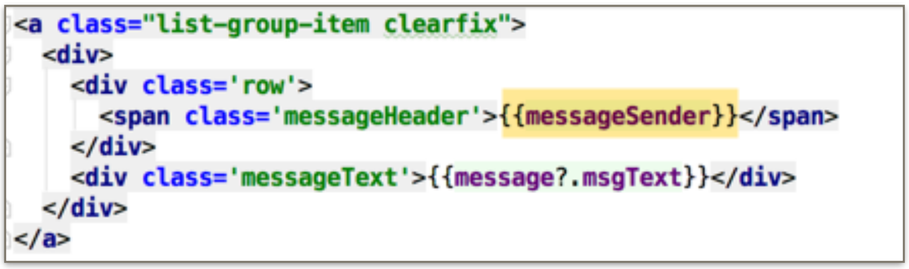
    b. Add a new class variable called `messageSender` of the string data type to the top of the class.

    c. Implement the `ngOnInit()` lifecycle function. Inside the function call the `getContact()` function and pass it the value of the `sender` property of the current message as shown below. Then get the name of the Contact found and assign it to a class variable called `messageSender`.



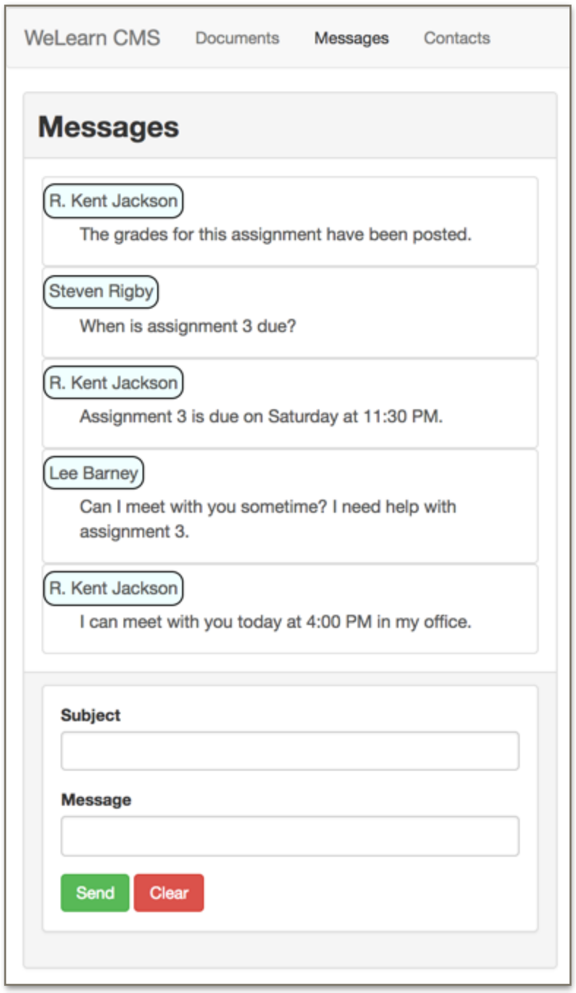
```
export class MessageItemComponent implements OnInit {

  @Input() message: Message;
  messageSender: string = "";
  canEdit: boolean = false;

  constructor(private contactService: ContactService) { }

  ngOnInit() {
    let contact: Contact = this.contactService.getContact(this.message.sender);
    this.messageSender = contact.name;
  }
```

d. Open the message-item.component.html file. In the first string interpolation modify the reference from `message?.sender` to display the value of the new class variable `messageSender`.



```
<a class="list-group-item clearfix">
  <div>
    <div class='row'>
      <span class='messageHeader'>{{messageSender}}</span>
    </div>
    <div class='messageText'>{{message?.msgText}}</div>
  </div>
</a>
```

10. Serve up your application if it is not already running and switch to your browser. You screen should be similar to the figure below.



WeLearn CMS Documents Messages Contacts

Messages

R. Kent Jackson
The grades for this assignment have been posted.

Steven Rigby
When is assignment 3 due?

R. Kent Jackson
Assignment 3 is due on Saturday at 11:30 PM.

Lee Barney
Can I meet with you sometime? I need help with assignment 3.

R. Kent Jackson
I can meet with you today at 4:00 PM in my office.

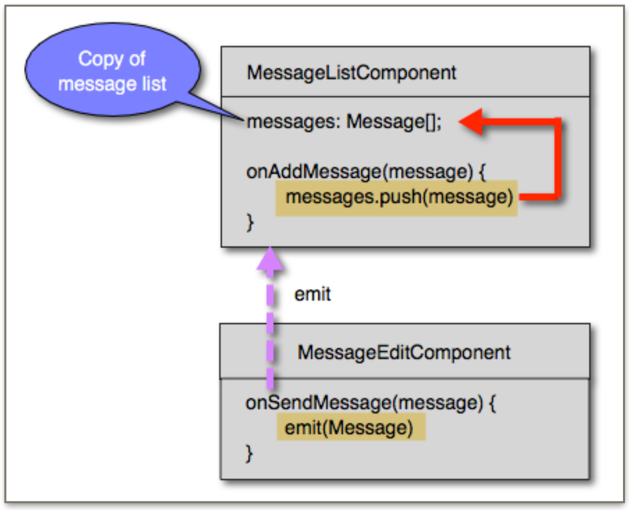
Subject

Message

Send Clear

Adding a new message to the message list using cross-component communication

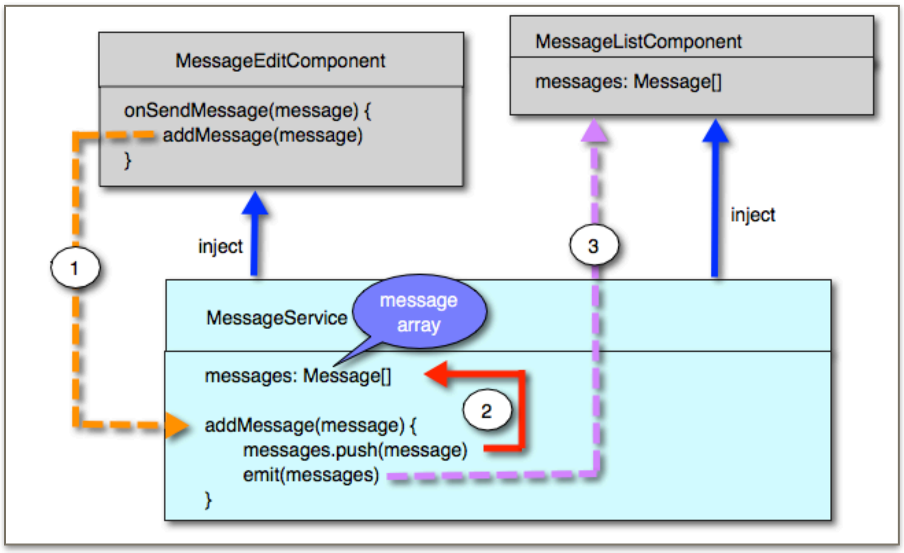
Currently, the `MessageEditComponent` has a function called `onSendMessage()` that gets called when the end user selects the **Send** button. This function emits and passes the new `Message` backup to its parent, the `MessageListCompnent`. The `MessageListComponent` in turn watches for and captures this event and then calls its `onAddMessage(message)` function to pushed the new `Message` on to the `messages` array in the `MessageListComponent`.



Unfortunately, the `messages` array in the `MessageListComponent` only contains a copy of the original messages list stored in the `MessageService` you created. The result is that the new message will not be permanently saved when the `MessageListComponent` is no longer displayed.

The `MessageService`, however, does contains a reference to the actual message list. We need to modify the `MessageEdit` component to save the new message in the permanent `messages` list stored in the `MessageService` and then automatically update the `messages` list stored in the `MessageListComponent` with a new copy of the actual `messages` stored in the `MessageService`.

This can be done by creating a new function in the `MessageService` called `addMessage()` as shown below.



1. The `onSendMessage()` function in the `MessageEditComponent` will call the `addMessage()` function in the `MessageService` and pass it the new `Message` to be added to the `messages` list.

2. The `addMessage()` function in turn pushes the new `Message` on to the `messages` array in the `MessageService`.

3. It then emits a custom event to indicate that the `message` array has been modified and passes a copy of the modified `messages` array with the emitted event. The `MessageListComponent` waits and listens for this event to occur. When this event is detected, the modified copy of the `messages` array passed with event is assigned to the temporary `messages` array defined in the `MessageListComponent` and the updated list of messages will be automatically be displayed on the screen.

Follow the instructions below to implement this functionality..

1. Open the `message.service.ts` file and add a new function to the `MessageService` class with the following function signature.

```
addMessage(message: Message)
```

   See the `addIngredient(ingredient: Ingredient)` function in the `shopping-list.service.ts` file in the Recipe Book project (`prj-services-final`) for a similar example.

    a. Inside the function, push the `Message` passed as an input on to the `messages` array defined in the `MessageService` class.

    b. Create a new `EventEmitter` of the `Message[]` datatype and assign it to a new class variable called `messageChangeEvent` at the top of the `MessageService` class.

    c. At the end of the `addMessage()` function, use the `messageChangeEvent` emitter to emit a copy (e.g., using the `slice()` function.

2. Open the `message-edit.component.ts` file and inject the `MessageService` into the `MessageEditComponent` class. Modify the `onSendMessage()` function to call the new `addMessage()` function in the `MessageService`. Pass the new message to the function. Refer to the `shopping-edit.component.ts` file in the Recipe Book project (`prj-services-final`) for a similar example.

3. Open the `message-list.component.ts` file and inject the `MessageService` into the `MessageListComponent` class. Implement the `ngOnInit()` lifecycle function and subscribe to the `messageChangeEvent` emitter defined in the `MessageService`. Assign the copy of the `messages` array emitted with the `messageChangeEvent` to the `messages` array in the `MessageListComponent`. A corollary example of how to do this can be found in the `shopping-list.component.ts` file in the Recipe Book project (`prj-services-final`).

4. Save all of your changes, serve up your application and then open your browser. Select the Messages feature and try adding a new message to the message list. The new message should appear at the end of the list as it did before. Now try switching to the Documents or Contacts Views and then back to the Messages View. The new message should still appear in the messages list.

This is the end of this assignment