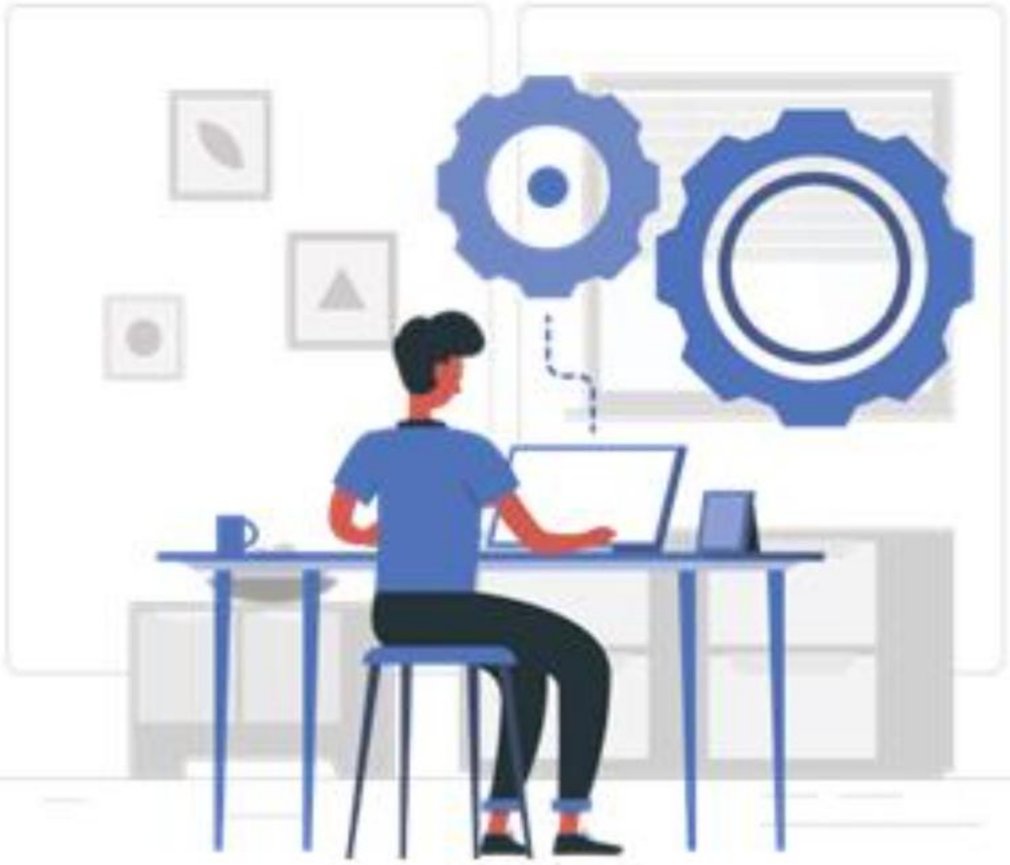


Practice

Implement Interactions Between Angular Components

Practice

- Practice: Create an Angular application with multiple components to add and view data

Points to Remember

- The boilerplate code contains the basic Angular application code with `app` component and `app` module.
    - The components need to be created in the same code.
- Components should be rendered based on the component hierarchy designed.
- Avoid usage of the type `any` to declare variables/constants.
- Unused code should be removed from the solution.

Instructions for Practice

- Click here for the boilerplate.
- Read the README.md file in the boilerplate for further instructions about the practice.
- Fork the boilerplate into your own workspace.
- Clone the boilerplate into your local system.
- Open command terminal and set the path to the folder containing the cloned boilerplate code.
- Run the command `npm install` to install the dependencies.
- Open the folder containing the boilerplate code in VS Code.
- Complete the solution in the given partial code provided in the boilerplate.

Notes:

The solution of this practice will undergo an automated evaluation on `hobbes`.
(Local testing is recommended prior to `hobbes` testing)

The test cases are available in the boilerplate.

PRACTICE

Create an Angular Application with Multiple Components to Add and View data

An application called `Contact-Gallery` needs to be developed that should allow the users to manage their contact details. It should allow its users to add a new contact as well as view the existing contacts.

This application should be developed as a single page application (SPA) using multiple components.

Note: The tasks to develop the solution are given in the upcoming slide.

Tasks

- To develop the solution for the `Contact-Gallery` application, following tasks need to be completed:
  - Task 1: Create components for the `Contact-Gallery` application.
  - Task 2: Display the existing contacts.
  - Task 3: A dd a new contact.

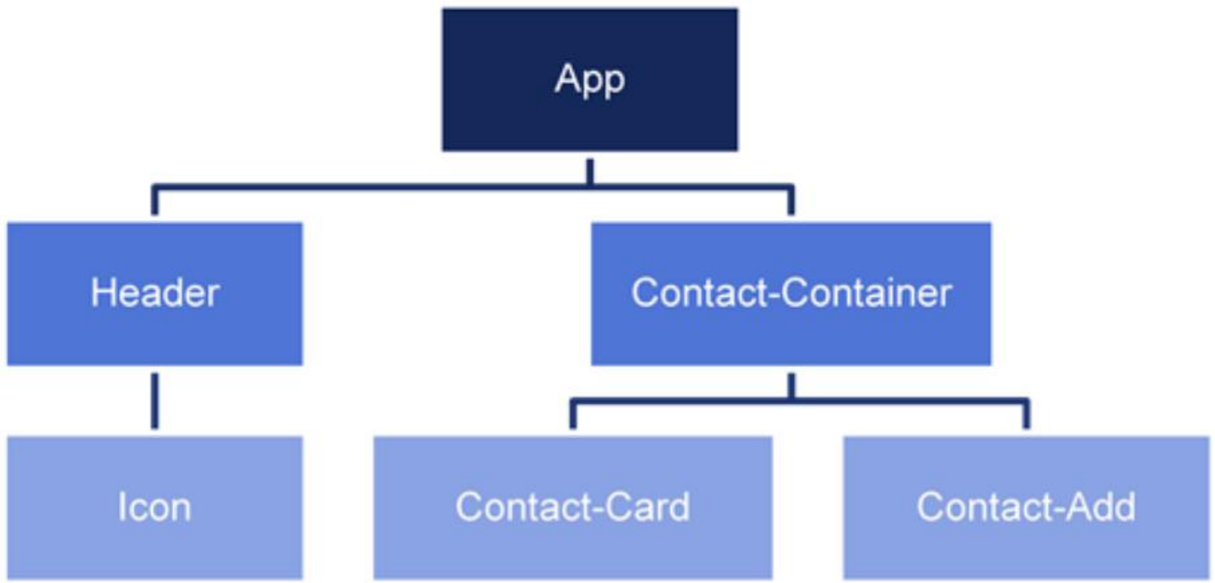
Note: The details of the tasks listed above are provided in the upcoming slides

Task 1: Create Components for the `Contact-Gallery` Application

Note: The boilerplate code contains an Angular application named `Contact-Gallery` created using the Angular CLI command, `ng new contact-gallery.`

Modify the application code as per the steps given below:

- Step 1: Create components as shown in the component hierarchy diagram below:



Task 1: Create Components for the `Contact-Gallery` Application (Cont'd.)

- To create the components, use Angular CLI command:

  `ng generate component <component-name>` or `ng g c <component-name>`

  Note:

  - The names of the components should match with the names displayed in the hierarchy diagram. (this is the requirement of the test code, failing to do so shall lead to test failure)
  - Delete the existing code from the **`app.component.html`** file.

- Step 2: Render the components using component selector as per the hierarchy.
  - `App` component should render `Header` and `Contact-Container` components.
  - `Header` component should render `Icon` component.
  - `Contact-Container` component should render `Contact-Card` and `Contact-Add` components.

Task 1: Create Components for the `Contact-Gallery` Application (Cont'd.)

- Step 3: Design `Header` and `Icon` components to display the application title and a settings icon.
  - The image for `Settings` icon is provided in the `assets` folder.
  - The code snippets to design these components are given below:

```
<input type="text" placeholder="Enter Search Text">
```

Search Component

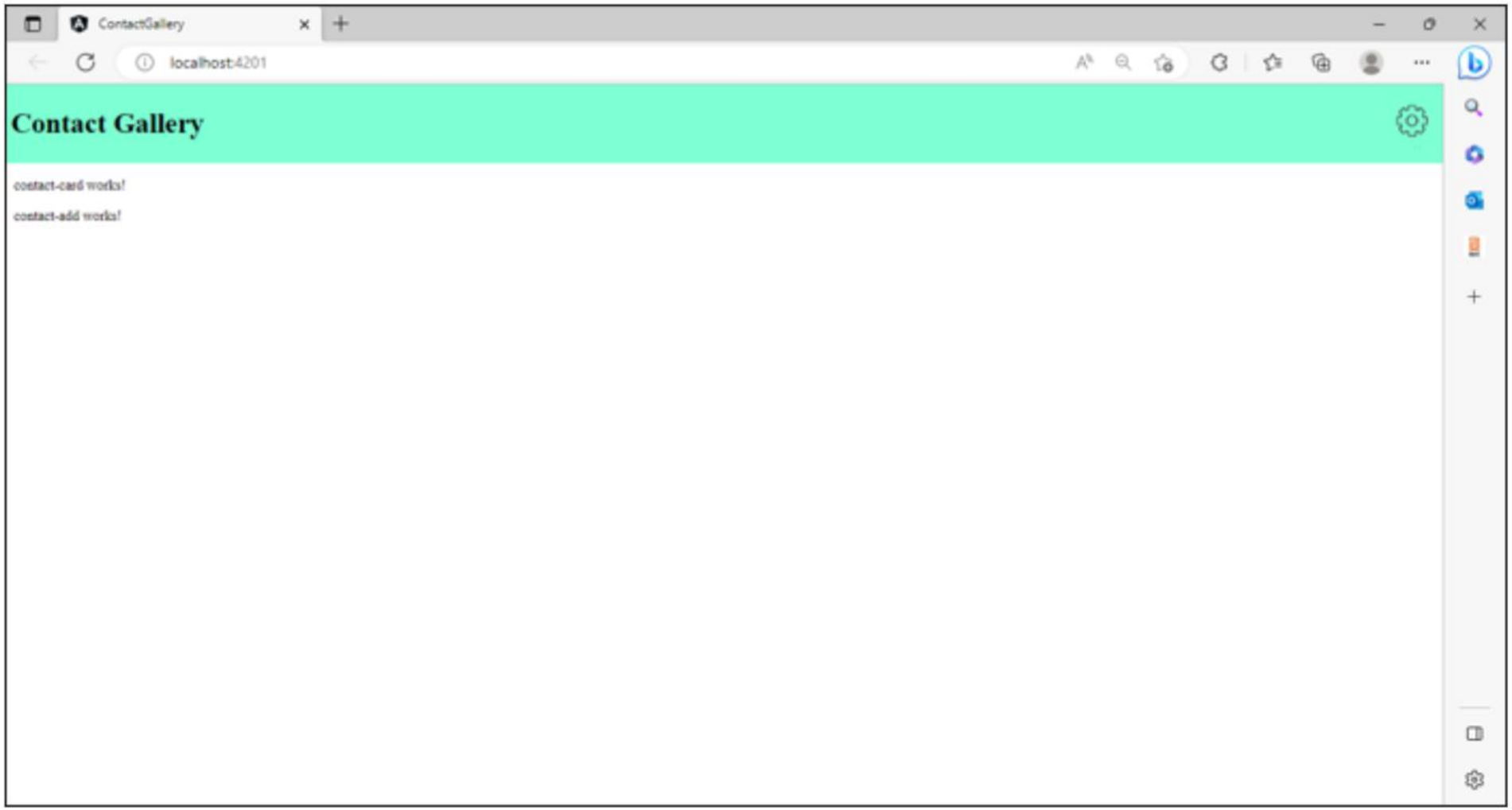
```
<img src="../../assets/settings.png">
```

Icon Component

```
<div id="header">
    <h1>Contact Gallery</h1>
    <app-icon></app-icon>
</div>
```

Header Component

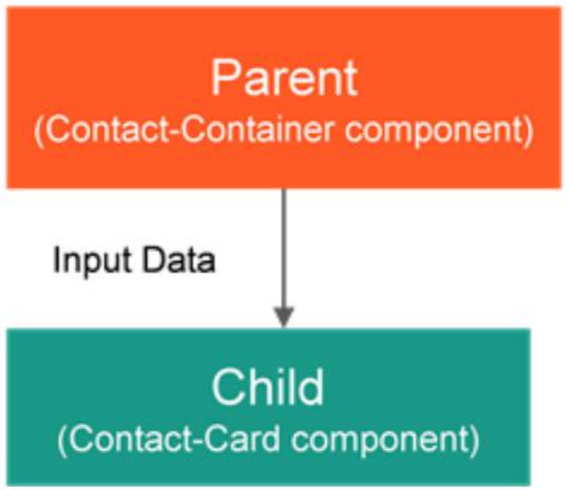
Expected Sample Output



Note: The CSS effects shown in the sample output image can be modified as per learner's preference.

Task 2: Display the Existing Contacts

Modify the `Contact-Gallery` application code to design a view that displays the contacts stored in the **`contacts.ts`** file. The diagram below shows the interaction between the `Contact-Container` and `Contact-Add` components for implementing the display functionality.



- Step 1: Create folder with the name `models` under the **`app`** folder of the `Contact-Gallery` code.
- Step 2: Copy the **`contacts.ts`** and **`contact.ts`** files from the resources folder provided with the boilerplate code.
  - The `CONTACTS` array, defined in the **`contacts.ts`** file under the `models` folder, stores object of type `Contact`. The type `Contact` is defined in the **`contact.ts`** file under the **`models`** folder.

Task 2: Display the Existing Contacts (Cont'd.)

- Step 3: In the `Contact-Card` component, declare a property named `contact` of type `Contact` and adorn it with `@Input()` decorator to mark it as the input property.
  - Make sure that the type `Contact` is imported from the file **`contact.ts`** located inside the **`models`** folder.
  - Also, make sure that the decorator `Input` is imported from the `@angular/core` package.
  - The input property `contact` will receive the contact data passed by the parent component - Contact-Container.

```
...
import { Contact } from '../models/contact';
@Component({
...
})
export class ContactCardComponent implements OnInit {

  @Input() contact?: Contact;
  constructor() { }
  ngOnInit(): void { }
}
```

Task 2: Display the Existing Contacts (Cont'd.)

- Step 4: In the template of the `Contact-Card` component, render the contact details available with the `contact` property.

```
<div class="card">
    <h3>{{contact?.firstName}}.{{contact?.lastName}}</h3>
    <h4>{{contact?.email}}</h4>
    <h4>{{contact?.contactNo}}</h4>
</div>
```

Task 2: Display the Existing Contacts (Cont'd.)

- Step 5: Declare property `contacts` in the **`.ts`** file of the `Contact-Container` component.
  - The `contacts` property should be of the type `Contact` array.
  - Make sure that the type `Contact` is imported from the file **`contact.ts`** located inside the `models` folder.
- Step 6: Assign the data of `CONTACTS` array declared in the **`contacts.ts`** file to the `contacts` property of the `Contact-Container` component.
  - Make sure that the `CONTACTS` array is imported from the file **`contacts.ts`** located inside the **`models`** folder.

```
...
import { Contact } from '../models/contact';
import { CONTACTS } from '../models/contacts';

@Component({
  ...
})
export class ContactContainerComponent {
  contacts: Contact[] = CONTACTS;
  constructor() { }
}
```

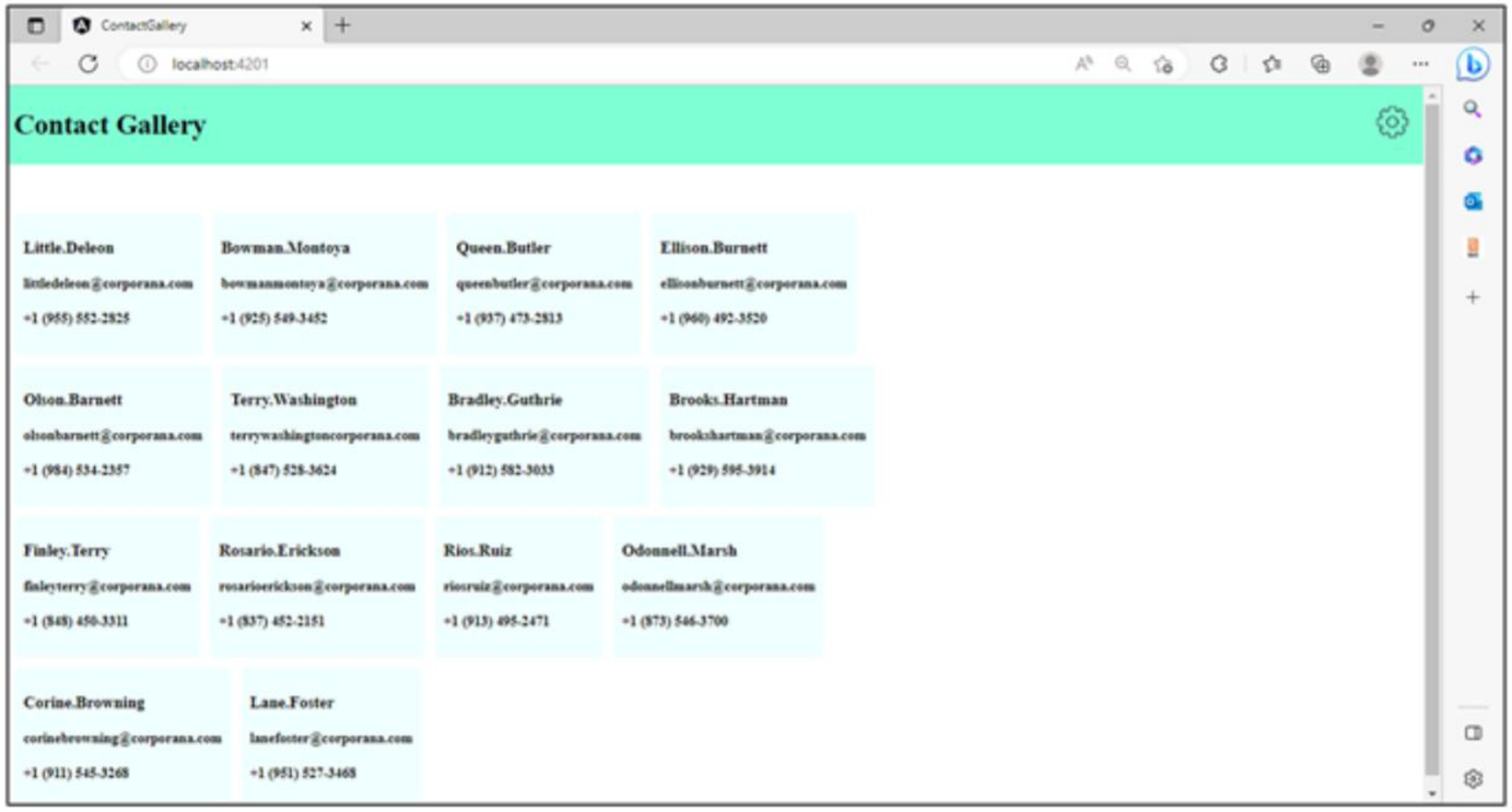
Task 2: Display the Existing Contacts (Cont'd.)

- Step 7: On the template of the `Contact-Container` component (`.html` file), iterate through the `contacts` property using the `*ngFor` directive.
  - At each iteration, the `Contact-Container` component should render the `Contact-Card` component providing a contact object as the input to the `contact` property of the `Contact-Card` component using property binding.

```
<div id="contact-container">
    <div id="contact-cards">
        <app-contact-card class="contact-card"
            *ngFor="let contact of contacts" [contact]="contact">

        </app-contact-card>
    </div>
</div>
```

Expected Sample Output

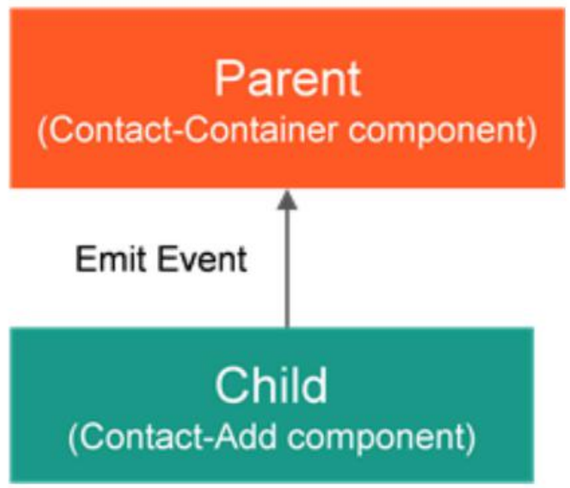
- Display the contact details of each contact data in grid layout.
- Below is a snapshot of the expected output:



Note: Styling the contact data is optional.

Task 3: Add a New Contact

Enhance the `Contact-Gallery` application to add the details of a new contact using the `Contact-Add` component. The diagram below shows the interaction between the `Contact-Add` and the `Contact-Container` components for implementing the add functionality.



The steps to implement the add functionality are given below:

- Step 1: Declare the output property `contactAdded` in the class of the `Contact-Add` component.
- Step 2: Define the method `addContact()` in the class of the `Contact-Add` component.

Task 3: Add a New Contact (Cont'd.)

- Step 3: In the body of the `addContact()` method, the `contactAdded` event should be emitted with the contact data.

```
import { Component, OnInit, Input, Output, EventEmitter } from '@angular/core';
import { Contact } from '../models/contact';

@Component({
  ...
})
export class ContactAddComponent implements OnInit {

  contact: Contact = {};

  @Output()
  contactAdded: EventEmitter<any> = new EventEmitter<any>();

  constructor() { }
  ngOnInit(): void { }

  addContact() {
    this.contactAdded.emit(this.contact);
  }
}
```

The codes in the yellow outlined boxes are to be added for this Task

Task 3: Add a New Contact (Cont'd.)

- Step 4: Add the following in the template of the `Contact-Add` component:
  - Input fields to accept the contact's first name, last name, email and contact number details.
  - A button that should call the `addContact()` method when it is clicked.

```
<div id="contact-input">
    <h2>Add New Contact</h2>
    <input type="text" placeholder="First Name" required [(ngModel)]="contact.firstName"><br/>
    <input type="text" placeholder="Last Name" required [(ngModel)]="contact.lastName"><br/>
    <input type="email" placeholder="Email" required [(ngModel)]="contact.email"><br/>
    <input type="text" placeholder="Contact Number" required [(ngModel)]="contact.contactNo"><br/>
    <br/>
    <button (click)="addContact()">Add</button>
</div>
```

Task 3: Add a New Contact (Cont'd.)

- Step 5: In the class of the `Contact-Container` component, define the method `onContactAdded()` which will be called to handle the `contactAdded` event.
  - The handler method should accept contact data passed by the child component using `$event` argument.
  - The handler method should update the `contacts` property with newly added contact details and raise an alert with the message "`Contact Added`".

```
// import statements
@Component({ ... })
export class ContactContainerComponent implements OnInit {
  contacts: Contact[] = CONTACTS;

  constructor() { }
  ngOnInit(): void { }

  onContactAdded($event:any) {
    this.contacts.push($event);
    alert(`Contact Added`);
  }
}
```

The code in the yellow outlined box is to be added for this Task

Task 3: Add a New Contact (Cont'd.)

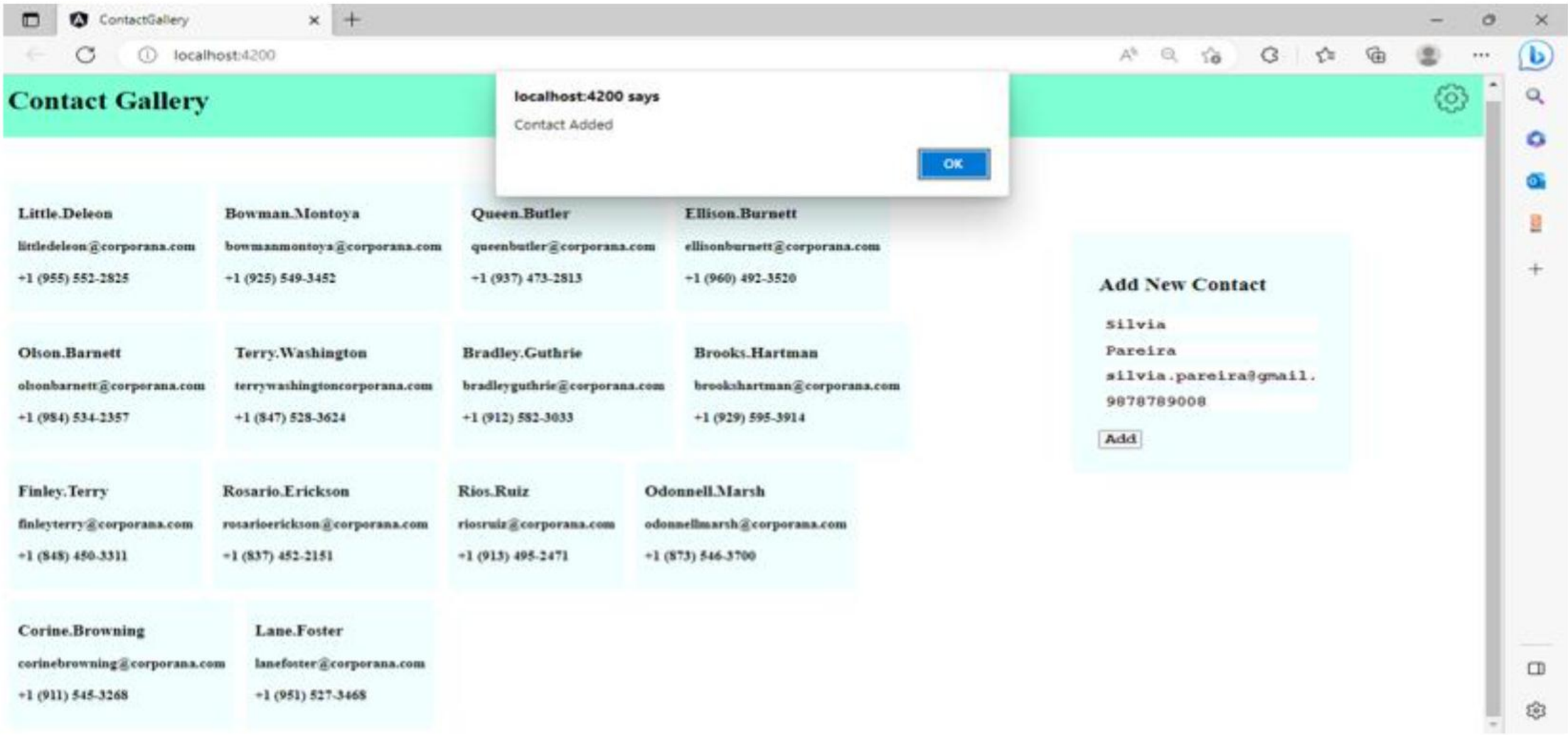
- Step 6: In the template of the `Contact-Container` component, while rendering the `Contact-Add` component, call the `onContactAdded()` method to listen to and handle the `contactAdded` event.

```
<div id="contact-container">
    <div id="contact-cards">
        <app-contact-card class="contact-card"
            *ngFor="let contact of contacts" [contact]="contact">
        </app-contact-card>
    </div>

    <app-contact-add (contactAdded)="onContactAdded($event)">
    </app-contact-add>

</div>
```

The code in the yellow outlined box is to be added for this Task

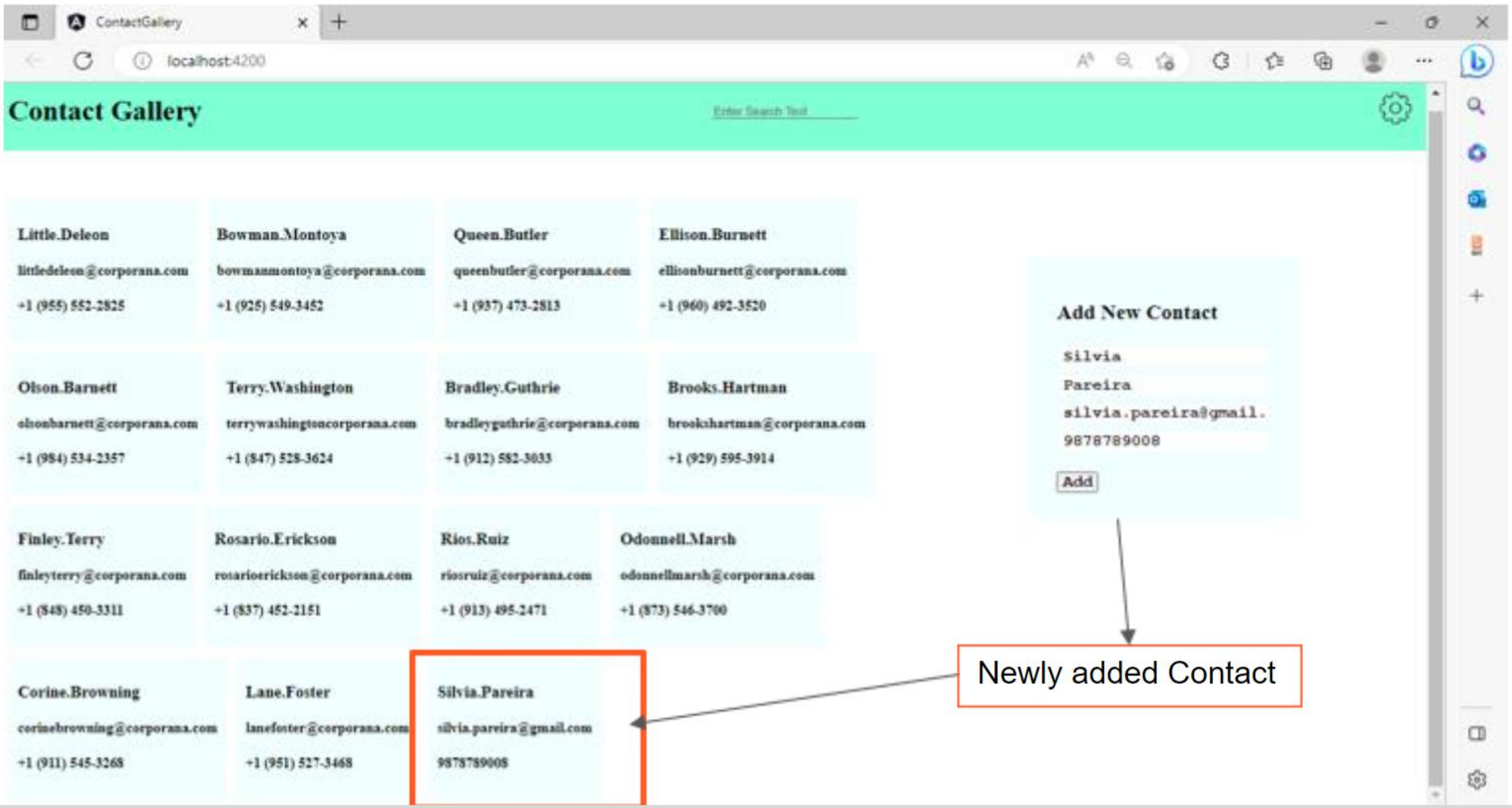
Expected Sample Output with Alert Box

- Below is a snapshot of the expected output:



Expected Sample Output with Added Contact

- Below is a snapshot of the expected output:



Test the Solution Locally

Test the solution first locally and then on `hobbes`. Steps to test the code locally are:

- From the command line terminal, set the path to the folder containing cloned boilerplate code.
- Run the command `ng test` or `npm run test` to test the solution locally and ensure all the test cases pass.
- Refactor the solution code if the test cases are failing and do a re-run.
- Finally, push the solution to git for automated testing on `hobbes`.