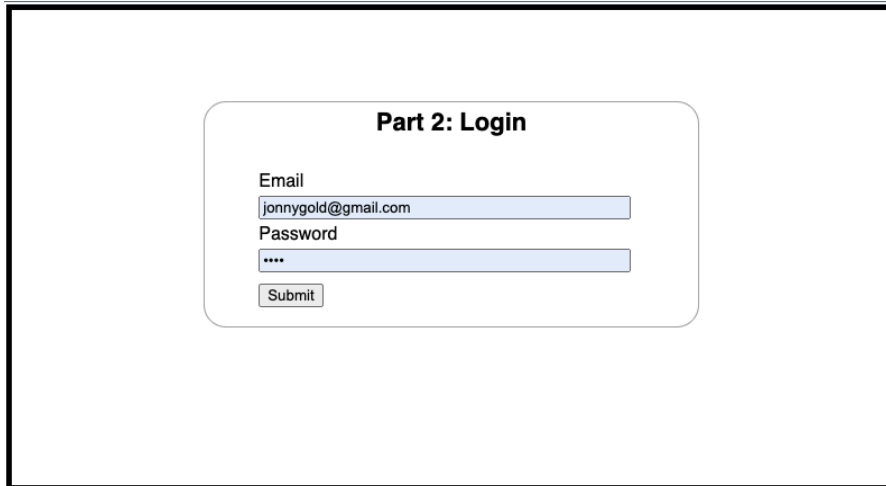


Accelerated Angular Part 3: Forms, User Interaction, and Presentation

In [Part 2](#), we introduced you to the theory and practice of routing in Angular. Next, we removed our apps boilerplate markup and added two-page components: Login and Tasks. Then, we used routing to add links to navigate from one page to the other. Now we have a foundation in place, we can build on it and add some basic functionality. In this article, we will add a form to the Login page.



The image shows a login form titled "Part 2: Login". It contains two input fields: "Email" with the value "jonnygold@gmail.com" and "Password" with four asterisks. Below the inputs is a "Submit" button.

The form enables a user to type an email and password. When the user presses the Submit button, the app checks the submitted credentials and authenticates the user. If the user's credentials are valid, the application displays a message and navigates to the task page. For the sake of simplicity, user credentials are stored in plain text in a Javascript file. In addition, some styles from a CSS template have been applied to the Login and Tasks files.

Key Concepts

As part of its comprehensive approach to web application development, Angular supports user interaction with forms. To complicate matters, [Angular](#) has two different models for implementing forms: [Template-driven](#) and [Reactive](#).

Template-Driven Forms

As the name suggests, template-driven forms let you create forms using templates that resemble generic HTML forms. Template-driven forms are good for creating

Reactive Forms

Reactive uses a combination of Typescript code and Angular Directives to build a form. Without going into details regarding the relative merits of each approach, Template-driven forms are regarded as simpler, and Reactive forms are considered the more complex option.

Despite their added complexity and steeper learning curve, Reactive forms offer a range of powerful and flexible features. This is why Reactive forms are the preferred option for Angular development. For this reason, we will use them to illustrate Angular form management.

Component Setup

Before adding a form to our Login component, we must reference Angular's `ReactiveFormsModule`. Open `src/app/login.component.ts` and add the following.

```
import { ReactiveFormsModule } from "@angular/forms";
```

Next in `@Component`, update imports.

```
@Component({
  selector: 'app-login',
  standalone: true,
  imports: [ ReactiveFormsModule ],
  templateUrl: './login.component.html',
  styleUrls: ['./login.component.scss']
})
```

Declaring and Initializing the Form

In the Login Component class, add a `FormGroup` called `loginForm`. A `FormGroup` contains a group of `FormControls` and manages form data (state) and related events.

```
export class LoginComponent {
  loginForm = new FormGroup({});
}
```

In `loginForm`, add two `FormControls` called `email` and `password`. These controls will collect and manage the data the user types into form fields. In our scenario, the `FormControls` perform a similar role to the `useState` hook. By default, the controls are empty but can be pre-populated with data.

```
loginForm = new FormGroup({
  email: new FormControl(''),
  password: new FormControl(''),
});
```

Before we update the component template, let's add an empty `onSubmit` method. This prevents the Angular compiler from throwing an exception when the method is referenced from our form.

```
export class LoginComponent {
  loginForm = new FormGroup({
    email: new FormControl(''),
    password: new FormControl(''),
  });
  onSubmit() {}
}
```

Add the Form Template

Open `login.component.ts`, delete the current content of the file and add the following. This `div` will contain the Login form.

```
<div class="card">
  <h1>Part 2: Login</h1>
</div>
```

After the header, add an HTML form.

```
<form [formGroup]="loginForm"></form>
```

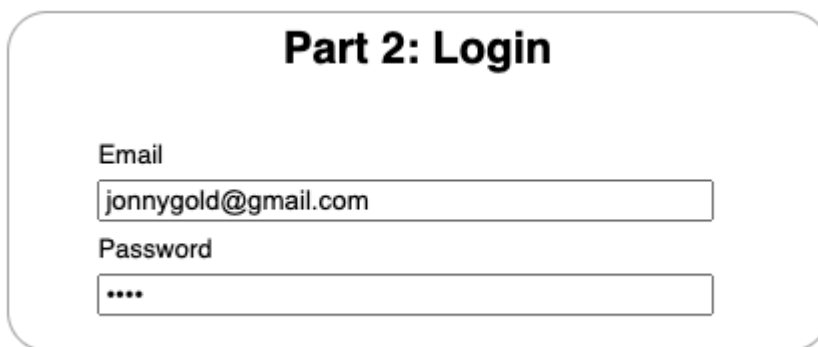
To connect the form to the `FormGroup`, add a reference to `loginForm`.

```
<form [formGroup]="loginForm"></form>
```

Now, we add the controls to the form. For each control, we add a `formControlName` to reference each control's respective `FormControl` in the `FormGroup`.

```
<form [formGroup]="loginForm">
  <label for="email" >Email</label><br/>
  <input formControlName="email" id="email" required /><br/>
  <label for="password" >Password</label><br/>
  <input formControlName="password" type="password" /><br/>
</form>
```

The form should look like this:



Part 2: Login

Email
jonnygold@gmail.com

Password
....

Once the controls are in place, we need to add a button. This lets the user submit their credentials for authentication.

```
<form [formGroup]="loginForm">
  ...
  <button type="submit">Submit</button>
</form>
```

The final step is to ensure that when the user clicks the button, something happens. We do this by adding an `ngSubmit` directive that calls the `LoginComponent`'s `onSubmit()` method.

```
<form [formGroup]="loginForm" (ngSubmit)="onSubmit()"></form>
```

Authenticating the User

At this point, when the user presses Submit, nothing happens. Now, we will add the required functionality to process the user's credentials. Although we added a reference to `User.ts`, `LoginComponent` is unable to read it. This requires declaring a variable and assigning it the contents of the file.

```
export class LoginComponent{
  users:any = Users;
}
```

Once we have authenticated the user, we want to persist and reference their profile information. For this purpose, we declare a user variable.

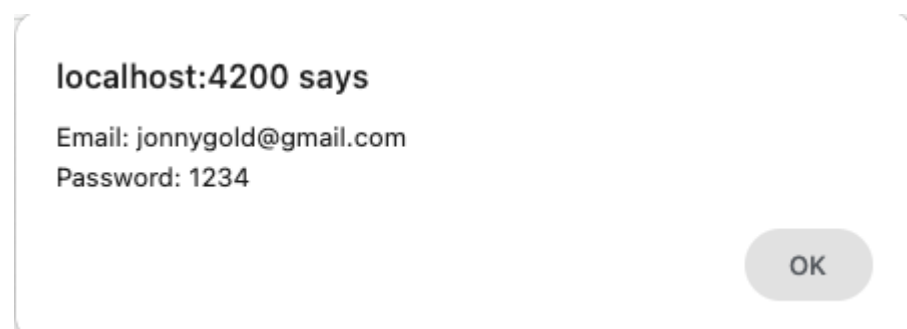
```
users:any = Users;
user:any = null
```

To authenticate the user and preserve their profile, we filter the list of users using the supplied credentials. We match the relevant values in the profile against the value held by the form control. For example, we can extract the submitted email address from `this.loginForm.value.email`.

```
onSubmit() {
  this.user = this.users.filter((user:any) =>
    user.email === this.loginForm.value.email &&
    user.password === this.loginForm.value.password)[0];
}
}
```

The results of the filter are assigned to this.user, by default this.user is null. If a valid user profile is assigned to this.user, a message is displayed and the user's profile data is persisted to the browser's session storage.

```
if(this.user !== null){
  alert(`Email: ${this.user.email}\nPassword: ${this.user.password}`);
  console.log('User found\n\n', this.user);
  sessionStorage.setItem('user', JSON.stringify(this.user));
}
```



Displaying Tasks

After authenticating the user, we want to open the Tasks page. To do this, we will need to reference Angular's Router component.

```
import { Router, RouterLink } from '@angular/router';
```

To use the Router, we need to add a constructor to LoginComponent and declare the Router service. After the loginForm declaration, add the following.

```
constructor( private router: Router) { }
```

Now, once the user is authenticated, we call the router's navigate method and provide the required path.

```
if(this.user !== null){
  ...
  this.router.navigate(['/tasks']);
}
```

Conclusion and What's Next

In this installment, we added a form to the Login page. The form enabled a user to type their email address and password. In the next installment, we will update the Tasks page to display a list of tasks in a table.

