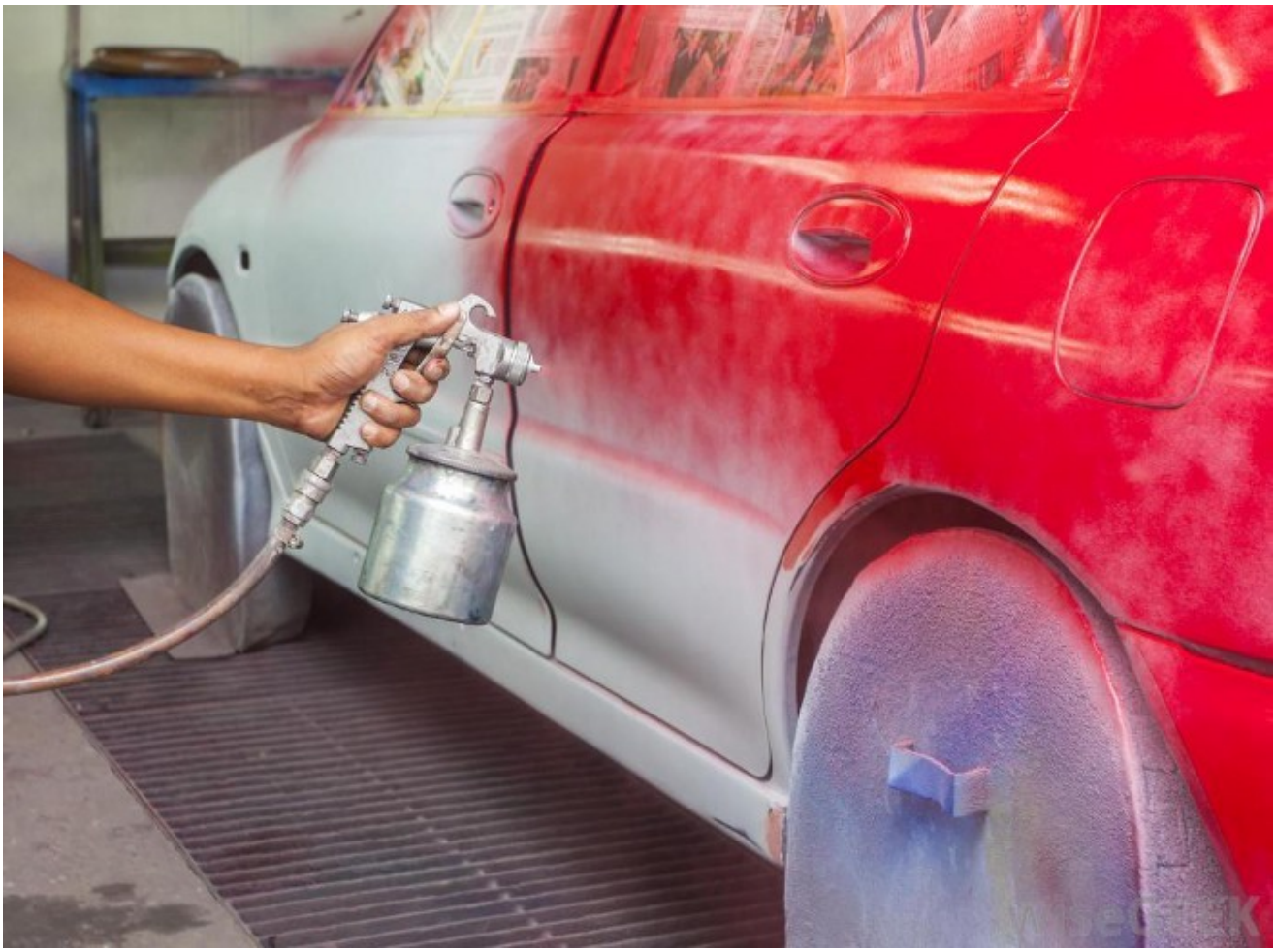
Learn how to create your own Dapp with Angular 9 — Part III.

Eli Elad Elrom Follow
Jan 2, 2020 · 14 min read ★

In this six-part article, we will cover how to create a Dapp with Angular. In Part I, which served as an introduction we covered general information regarding developing the dapp benefits and classification. Using Angular, Angular architecture, benefits. You should start there.

In the second part, we covered creating an Angular Dapp — prerequisites, and creating an Angular skeleton app.

In this part, Part III, we will cover Styling Angular — including custom components.



Styling an Angular App

Our app from the previous article is not styled yet and only shows text with a header, the page, and a footer; however, before we start styling, it's helpful to understand the Angular style architecture to ensure we don't end up with a Cascading Style Sheets (CSS) file that is too big to manage.

You can style your app on a global level with styles that you need across your entire app as well as a specific style unique to only one component.

Additionally, it would be neat to **sprint from zero to a styled app quickly**. This can be done with Angular Material. Angular Material gives you a shortcut to get a consistent "look" to your app without all the hassle of thinking about cross-browser, cross-device programming. Let's take a look.

Angular-Style Architecture

Angular is set up to have a global CSS file. That CSS file is called style.css, and you can find it in the root of the project.

```
src/style.css
```

The file holds the styles that you want to use for your entire app, such as fonts, themes, styles for all the components, and so on.

As you have seen, each component also includes a private CSS file. The the specific component CSS file is where you put styles that are unique and used only for that component.

For instance,

```
/src/app/components/footer/footer.component.css
```

holds the styles specific for the footer component.

Angular Material

Right now, your starter application is fast because it includes minimal code; however, there is a potential performance issue as you add more and more components, assets, and style to your app. You can get your app bloated easily, and every millisecond dealy counts.

The other potential issue is testing. All the different browsers, versions of browsers, screen sizes, and devices need to be tested, and creating your pages from scratch will require rigid testing and quality assurance (QA) team to ensure it works consistency across devices.

Angular Material solves all these issues plus provides accessibility and internationalization. That is because Angular Material is optimized for Angular and built by the Angular team, so it integrates seamlessly with Angular. It has already passed all these compatibility tests.

For more information, check the Angular Material getting started page: https://material.angular.io/guide/getting-started.

Install Angular Material

There are a few ways to install Material. Because you have installed the Angular DevKit, you are able to just run the ng add the command to get the Angular Material library. You need to first install cdk because it's a dependency.

```
ng add @angular/cdk
```

Next, install Material.

```
ng add @angular/material
```

> ***Notice** that the output asks you which theme color you would like with links. I will cover themes in the next section of this article, but for now, select the first or any color you prefer.*

The installation wizard allow you to select what you want to install, I have been using the following settings; (you can learn more about these features on Angular Material docs).

```
? Choose a prebuilt theme name, or "custom" for a custom theme:
Indigo/Pink [Preview: https://material.angular.io?theme=indigo-pink]
? Set up HammerJS for gesture recognition? Yes
? Set up browser animations for Angular Material? Yes
```

The expected output should be showing the files that were updated:

```
UPDATE package.json (1434 bytes)
✓ Packages installed successfully.
```

```
UPDATE src/main.ts (391 bytes)
UPDATE src/app/app.module.ts (878 bytes)
UPDATE angular.json (3740 bytes)
UPDATE src/index.html (487 bytes)
UPDATE src/styles.css (181 bytes)
```

Next, you want to modify your app to have Angular Material include animations, Material icons, gesture support, and component modules.

In your project, you will only be using component modules and not all the features that Angular Material has to offer; what you need to do is import NgModule for each component you want to use. Open;

```
src/app/app.module.ts
```

Add the import statements;

```
import {
  MatButtonModule,
  MatCheckboxModule,
  MatInputModule,
  MatSelectModule,
  MatDatepickerModule,
  MatNativeDateModule
} from '@angular/material';
```

Next, update the import statements of @NgModule to include the Material modules you imported.

```
imports: [
BrowserModule,
AppRoutingModule,
BrowserAnimationsModule,
MatButtonModule,
MatInputModule,
MatDatepickerModule,
MatNativeDateModule,
MatCheckboxModule,
MatSelectModule
```

```
]
```

That's it. You can now have access to the Angular Material components you included.

Theme Your Angular Material App

Now that you have access to the Angular Material components, you can use themes to style them. A theme is a set of colors that will be used on your Angular Material components.

In Angular Material, a theme is created by creating multiple palettes.

— **Primary palette**: These are the colors most used across all screens and components.

— **Accent palette**: These are the colors used for the button and interactive elements.

— **Warn palette**: These are the colors for errors.

— **Foreground palette**: These are the colors for text and icons.

— **Background palette**: These are the colors for an element's background.

In Angular Material, all theme styles are generated statically at build time to avoid slowing the app on startup.

Angular Material comes prepackaged with several prebuilt theme CSS files. As you probably recall, you had the option of selecting a theme to use when you installed Material.

These theme files also include all of the styles for the core (styles common to all components), so you have to include only a single CSS file for Angular Material in your app. You can include a theme file directly into your application from @angular/material/prebuilt-themes.
These are the available prebuilt themes:

```
— deeppurple-amber.css
— indigo-pink.css
— pink-bluegrey.css
— purple-green.css
```

You are using Angular CLI here, so you can simply include the style you want in the global

```
src/styles.css
```

The original content;

```
/* You can add global styles to this file, and also import other
style files */

html, body { height: 100%; }
body { margin: 0; font-family: Roboto, "Helvetica Neue", sans-serif;
}
```

Add the following import statement at the top of the document:

```
@import "~@angular/material/prebuilt-themes/indigo-pink.css";
```

While you have the src/app.component.css file open, you can also create a style for a container, a paragraph, and a button that you can use across your app for your pages.

```
p {
  padding-left: 20px;
  font-size: 12px;
}
.container {
  margin-right: auto;
  margin-left: auto;
  padding: 20px 15px 30px;
```

```
  width: 750px;
}
button {
  color: #ffffff;
  background-color: #611BBD;
  border-color: #130269;
  display: inline-block;
  margin-bottom: 0;
  font-weight: normal;
  text-align: center;
  vertical-align: middle;
  touch-action: manipulation;
  cursor: pointer;
  white-space: nowrap;
  padding: 6px 12px;
  font-size: 12px;
  line-height: 1.42857143;
  border-radius: 4px;
  -webkit-user-select: none;
  -moz-user-select: none;
  -ms-user-select: none;
  user-select: none;
}
```

Creating Content

At this point, you have a skeleton app with a header, body, and footer. The body can be switched between your start page and the transfer page by changing the URL in the browser. You also imported and injected Material modules and set up global styles for your app. The next step is to create actual content to replace the temporary text message you placed in your header, footer, and start components.

Footer Component

For the footer component, you will just replace the message for your company copyright. To do so, all you need to do is open;

```
src/app/components/footer/footer.component.html
```

And replace the default code.

```
<p>
footer works!
</p>
```

Replace the code by creating a div container with the style you added to the global CSS file.

```
<div class="ng-scope">
  <div class="container">
    <p>Copyright © 2019 Company Name. All Rights Reserved.</p>
  </div>
</div>
```

You are also going to create a specific style for the footer component, so every time you use the p tag, your font will be size 12px with no padding on the left. Open

```
src/app/components/footer/footer.component.css
```

And insert the following:

```
p {
  padding-left: 0;
  font-size: 11px;
}
```

Notice that you defined the <p> tag twice, once in the global CSS file and one at the component level. What's going to happen is that the global <p> tag will be overwritten by the component <p>, so you can use the <p> tag for your footer and a different <p> tag for other components such as the start and transfer pages while keeping your HTML code free of CSS code.

Header Component

For the header component, you will create a navigation menu to be able to switch between the start page and the transfer page. For styles specific to the header component, open

```
src/app/components/header/header.component.css
```

and add the nav list styles.

```
.nav {
  margin-bottom: 0;
  padding-left: 0;
  list-style: none;
}
li {
  display: block;
  float: left;
  width: 100px;
  height: 25px;
  padding: 5px;
}
.nav>li>a {
  margin-bottom: 0;
  padding-left: 0;
  font-weight: 500;
  font-size: 12px;
  text-transform: uppercase;
  position: relative;
}
```

For;

```
src/app/components/header/header.component.html
```

You create a container and a list of the two links to the pages start and transfer. To do so replace the initial code:

```
<p>
```

```
header works!
</p>
```

with the following;

```
<div class="ng-scope">
  <div class="container">
    <ul class="nav">
      <li>
        <a routerLink='/start'>home</a>
      </li>
      <li>
        <a routerLink='/transfer'>transfer</a>
      </li>
    </ul>
  </div>
</div>
```

The working dapp now includes basic styling and functional navigation, as shown in Figure 1.

Figure 1. Ethdapp with basic styling and working navigation

As you recall, run ng serve, if it’s not running already;

```
ng serve
```

Transfer Component

The transfer component will hold a form that you will submit to transfer Ethereum coins from one account address to another. You will be using the forms module to expedite creating your form. To do so, you need to include the Material FormsModule and ReactiveFormsModule form modules in app.module.ts just as you did with other Material modules. Open;

```
src/app/app.module.ts
```

And add the following import statement:

```
import { FormsModule, ReactiveFormsModule } from '@angular/forms';
```

You also want to update the import statement.

```
imports: [
  FormsModule,
  ReactiveFormsModule,
  ...
  ...

],
```

You will be using the <mat-form-field> tag, which represents a component that wraps several Angular Material components together and applies common text field styles such as the underline, floating label, and hint messages. This will expedite development as you won't need to implement all of these and test them on multiple devices/browsers.

The form field is the wrapper component named <mat-form-field>.

You can use any of the form field controls (such as input, textarea, list, etc.). You can find information about mat-forms here: https://material.angular.io/components/form-field/overview.

For;

```
src/app/components/transfer/transfer.component.ts
```

You will update the initial code. First, you need to import the components you will be using; in this case, you need to initialize the class and use form, form control, and validators.

```
import {FormBuilder, FormControl, FormGroup, Validators} from
'@angular/forms';
```

Then you need to update the component definition to implement the OnInit method.

```
export class TransferComponent implements OnInit {
```

You will be using a flag to indicate whether the form was submitted and to create an instance of a form group, as well as an object called user, to hold the user's information.

```
formSubmitted = false;
userForm: FormGroup;
user: any;
```

To validate your form, you will define the messages in case the form is not filled incorrectly. Each "form" control needs to be defined with the required fields and messages.

```
accountValidationMessages = {
  transferAddress: [
    { type: 'required', message: 'Transfer Address is required' },
    { type: 'minLength', message: 'Transfer Address must be 42
characters long' },
    { type: 'maxLength', message: 'Transfer Address must be 42
characters long' }
  ],
  amount: [
    { type: 'required', message: 'Amount is required' },
    { type: 'pattern', message: 'Amount must be a positive number' }
  ],
  remarks: [
    { type: 'required', message: 'Remarks are required' }
  ]
};
```

When you create the constructor, you need to include the FormBuilder component to be able to generate the form.

```
constructor(private fb: FormBuilder) { }
```

When your component gets init, you will set the “formSubmitted” flag to false and set default values for the user’s information. You then will call a method to go fetch the user’s account and balance, which you will implement later. Lastly, you will call the “createForms” method that will generate the form.

```
ngOnInit() {
  this.formSubmitted = false;
  this.user = {address: '', transferAddress: '', balance: '', amount:
'', remarks: ''};
  this.getAccountAndBalance();
  this.createForms();
}
```

The createForms method will generate the form controls bypassing

the validators and data.

```
createForms() {
  this.userForm = this.fb.group({
    transferAddress: new FormControl(this.user.transferAddress,
Validators.compose([
      Validators.required,
      Validators.minLength(42),
      Validators.maxLength(42)
    ])),
    amount: new FormControl(this.user.amount, Validators.compose([
      Validators.required,
      Validators.pattern('^[+]?([.]\\d+|\\d+[.]?\\d*)$')
    ])),
    remarks: new FormControl(this.user.remarks, Validators.compose([
      Validators.required
    ]))
  });
}
```

The “getAccountAndBalance” method will set the user account’s address and balance; for now you are using dummy data, but you will implement the actual service later in this article.

```
getAccountAndBalance = () => {
  const that = this;
  that.user.address = '0xd8d0101f83e79fb4e8d21134f5325e64816bd6a0';
  that.user.balance = 0;
  // TODO: fetch data
}
```

Lastly, once you submit your form, you need a method to handle the data and call the service. submitForm will be used by checking whether the form is valid, and then later you will call the service component you will create.

```
submitForm() {
  if (this.userForm.invalid) {
    alert('transfer.components :: submitForm :: Form invalid');
    return;
  } else {
```

```
    console.log('transfer.components :: submitForm :: this.
    userForm.value');
    console.log(this.userForm.value);
    // TODO: service call
  }
}
```

Here is the complete code for transfer.component.ts;

```
import { Component, OnInit } from '@angular/core';
import {FormBuilder, FormControl, FormGroup, Validators} from
'@angular/forms';

@Component({
  selector: 'app-transfer',
  templateUrl: './transfer.component.html',
  styleUrls: ['./transfer.component.css']
})
export class TransferComponent implements OnInit {

  formSubmitted = false;
  userForm: FormGroup;
  user: any;

  accountValidationMessages = {
    transferAddress: [
      { type: 'required', message: 'Transfer Address is required' },
      { type: 'minLength', message: 'Transfer Address must be 42
characters long' },
      { type: 'maxLength', message: 'Transfer Address must be 42
characters long' }
    ],
    amount: [
      { type: 'required', message: 'Amount is required' },
      { type: 'pattern', message: 'Amount must be a positive number'
}
    ],
    remarks: [
      { type: 'required', message: 'Remarks are required' }
    ]
  };

  constructor(private fb: FormBuilder) { }

  ngOnInit() {
    this.formSubmitted = false;
    this.user = {address: '', transferAddress: '', balance: '',
amount: '', remarks: ''};
    this.getAccountAndBalance();
```

```
    this.createForms();
  }

  createForms() {
    this.userForm = this.fb.group({
      transferAddress: new FormControl(this.user.transferAddress,
Validators.compose([
        Validators.required,
        Validators.minLength(42),
        Validators.maxLength(42)
      ])),
      amount: new FormControl(this.user.amount, Validators.compose([
        Validators.required,
        Validators.pattern('^[+]?([.]\\d+|\\d+[.]?\\d*)$')
      ])),
      remarks: new FormControl(this.user.remarks,
Validators.compose([
        Validators.required
      ]))
    });
  }
  getAccountAndBalance = () => {
    const that = this;
    that.user.address = '0xd8d0101f83e79fb4e8d21134f5325e64816bd6a0';
    that.user.balance = 0;
    // TODO: fetch data
  }
  submitForm() {
    if (this.userForm.invalid) {
      alert('transfer.components :: submitForm :: Form invalid');
      return;
    } else {
      console.log('transfer.components :: submitForm ::
this.userForm.value');
      console.log(this.userForm.value);
      // TODO: service call
    }
  }
}
```

For;

```
transfer.component.html
```

We will set the form tag to call the submitForm method once the form is submitted.

```
<form [formGroup]="userForm" (ngSubmit)="submitForm()"
      novalidate autocomplete="off">
  <div class="container">
    <div class="transfer-container">
      <div>
        Address: {{user.address}} <br/>
        Balance: {{user.balance}} Eth
      </div>
```

> ***Notice*** *that you have used the transfer-container style, which you have not yet defined; you will define it in your CSS file, and it will be used to format your form.*

For form controls, you need input boxes for the account you are transferring the funds to, the amount, and a message. You also need to set up your validations.

```
<mat-form-field>
  <input matInput placeholder="Transfer Address"
name="transferAddress" formControlName="transferAddress"
         maxlength="42" minlength="42" required>
  <mat-error *ngFor="let validation of
accountValidationMessages.transferAddress">
    <mat-error
*ngIf="userForm.get('transferAddress').hasError(validation.type) &&
(userForm.get('transferAddress').dirty ||
userForm.get('transferAddress').touched)">{{validation.message}}
</mat-error>
  </mat-error>
</mat-form-field>
<mat-form-field>
  <input matInput placeholder="Amount" name="amount"
formControlName="amount" required>
  <mat-error *ngFor="let validation of
accountValidationMessages.amount">
    <mat-error
*ngIf="userForm.get('amount').hasError(validation.type) &&
(userForm.get('amount').dirty || userForm.get('amount').touched)">
{{validation.message}}</mat-error>
  </mat-error>
</mat-form-field>
<mat-form-field>
  <input matInput placeholder="Remarks" name="remarks"
```

```
formControlName="remarks"
         maxlength="42" required>
  <mat-error *ngFor="let validation of
accountValidationMessages.remarks">
    <mat-error
*ngIf="userForm.get('remarks').hasError(validation.type) &&
(userForm.get('remarks').dirty || userForm.get('remarks').touched)">
{{validation.message}}</mat-error>
  </mat-error>
</mat-form-field>
```

Lastly, remember to close the divs and form, as well as include a submit button.

```
<div style="width: 100px">
        <button type="submit">Transfer Ether</button>
      </div>
    </div>
  </div>
</form>
```

For transfer.component.css, we will be using the transfer-container div to format your form horizontally.

```
.transfer-container {
  display: flex;
  flex-direction: column;
}
.transfer-container > * {
  width: 100%;
}
```

That's it. Now you can check your dapp in the browser, and you should be able to see the user's default data, test the form, validate it, and submit the form. See Figure 2.

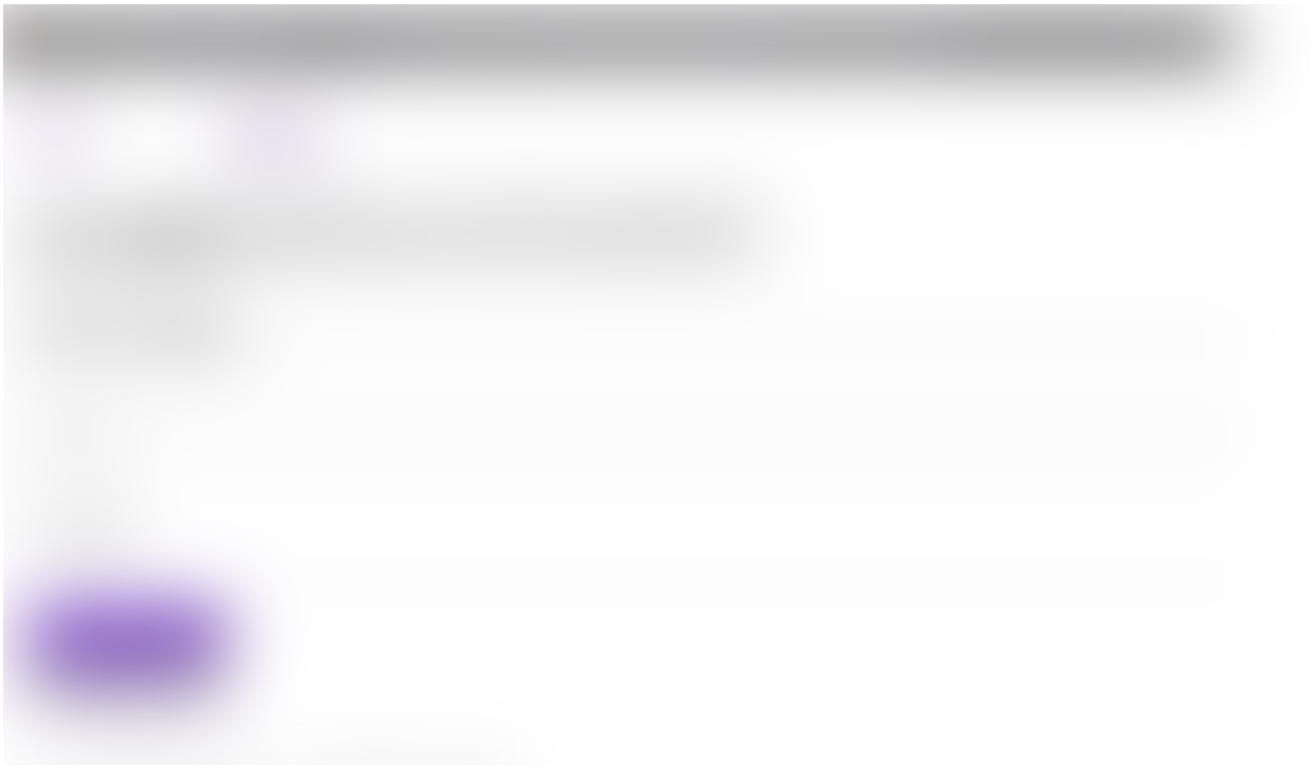


Figure 2. Ethdapp transfer page including user's info, validators, and submit button

Here is the complete transfer.component.html file;

```
<form [formGroup]="userForm" (ngSubmit)="submitForm()"
      novalidate autocomplete="off">
  <div class="container">
    <div class="transfer-container">
      <div>
        Address: {{user.address}} <br/>
        Balance: {{user.balance}} Eth
      </div>
      <mat-form-field>
        <input matInput placeholder="Transfer Address"
name="transferAddress" formControlName="transferAddress"
               maxlength="42" minlength="42" required>
        <mat-error *ngFor="let validation of
accountValidationMessages.transferAddress">
          <mat-error
*ngIf="userForm.get('transferAddress').hasError(validation.type) &&
(userForm.get('transferAddress').dirty ||
userForm.get('transferAddress').touched)">{{validation.message}}
</mat-error>
        </mat-error>
      </mat-form-field>
```

```
      <mat-form-field>
        <input matInput placeholder="Amount" name="amount"
formControlName="amount" required>
        <mat-error *ngFor="let validation of
accountValidationMessages.amount">
          <mat-error
*ngIf="userForm.get('amount').hasError(validation.type) &&
(userForm.get('amount').dirty || userForm.get('amount').touched)">
{{validation.message}}</mat-error>
        </mat-error>
      </mat-form-field>
      <mat-form-field>
        <input matInput placeholder="Remarks" name="remarks"
formControlName="remarks"
               maxlength="42" required>
        <mat-error *ngFor="let validation of
accountValidationMessages.remarks">
          <mat-error
*ngIf="userForm.get('remarks').hasError(validation.type) &&
(userForm.get('remarks').dirty || userForm.get('remarks').touched)">
{{validation.message}}</mat-error>
        </mat-error>
      </mat-form-field>
      <div style="width: 100px">
        <button type="submit">Transfer Ether</button>
      </div>
    </div>
  </div>
</form>
```

Angular Directives

Creating directives in Angular gives you the ability to create your own custom HTML tags with just a few lines of code, just as you saw in the Material form. You were able to include custom tags that wrap many components. At a high level, directives are markers on a DOM element. These markers can point to any DOM component, from an attribute to an element name or even a comment or CSS class. These markers then tell the AngularJS's HTML compiler to attach a specified behavior or to transform the entire DOM element and its children based on specific logic.

Angular comes with many of these directives built-in. However, during development, it's a good chance you will be creating your own directives. Your dapp is simple now, so you don't need to create any directive, and it's beyond the scope of this article to explain this. When you do need to generate a

skeleton directive, use the Angular CLI just as you generated other components.

```
ng generate directive {directive-name}
```

Although you are not creating a directive in your app, I wanted to introduce you to the concept as it's an integral part of creating an Angular project.

JavaScript Angular Dapps Development Metamask

Let's recap!

So far, we took a deep dive into what a dapp is and looked at dapp classifications and projects. We learned how to start your own dapp project by breaking the process into five steps: writing a white paper, launching an ICO, developing the dapp, launching it, and marketing your dapp.

We then looked at why to use Angular. Next, you created an Angular dapp, first ensuring the prerequisites were installed and installing the Angular CLI. Then you created an Angular project and served the application.

Next, you learned how to import your Angular project to WebStorm or create a new project. You looked at the pieces that make Angular such as components, modules, and directives. You also learned how to style the dapp by understanding Angular-style architecture and working with Angular Material.

You started building components and created content; you split your app into a footer, header, and body and created a custom component called transfer that includes a form to be able to later transfer tokens.

In the next articles, you will create a transfer smart contract and a Truffle development project as well as connect to the Ganache development network.

You will learn how to work with the Ethereum network via Truffle and test your smart contract. You also will link your dapp with the Ethereum Network's web3 library and connect via MetaMask.

Where to go from here

Continue following our articles as in the next articles we will be covering the following;

- Part IV — Creating a smart contract with Truffle

To learn more about what's possible with Blockchain as well as develop your own project chec
The Blockchain Developer.