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

Eli Elad Elrom Follow
Jan 5, 2020 · 8 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 previous articles, we started developing our dapp. Specifically, we learned about dapp classifications and projects and that you can break your own dapp project into five steps.

We then looked at why to use Angular and its benefits. Next, we created an Angular project, first ensuring the prerequisites were installed and then installing the Angular CLI.

We looked at the pieces that make up Angular such as components, modules, and directives. We also learned how to style a dapp by understanding Angular-style architecture and working with Angular Material.

We started building our own custom components and creating content; we split the app into a footer, header, and body and created a custom transfer component that you will be using in this article. We created the dapp's smart contract utilizing the following

tools the Angular CLI, Truffle, ganache-cli, and MetaMask.

In this article, we will be integrating our smart contract in the dapp's Angular project.

In the last article, **Part IV**; Linking and connecting your dapp to the Ethereum network and testing.

Link with the Ethereum Network

In the previous article, we got the contract working in Terminal; the next step is for our dapp to interact with the contract. This is done via web3.js, which is a collection of libraries allowing you to interact with a local or remote Ethereum node using an HTTP or IPC connection.

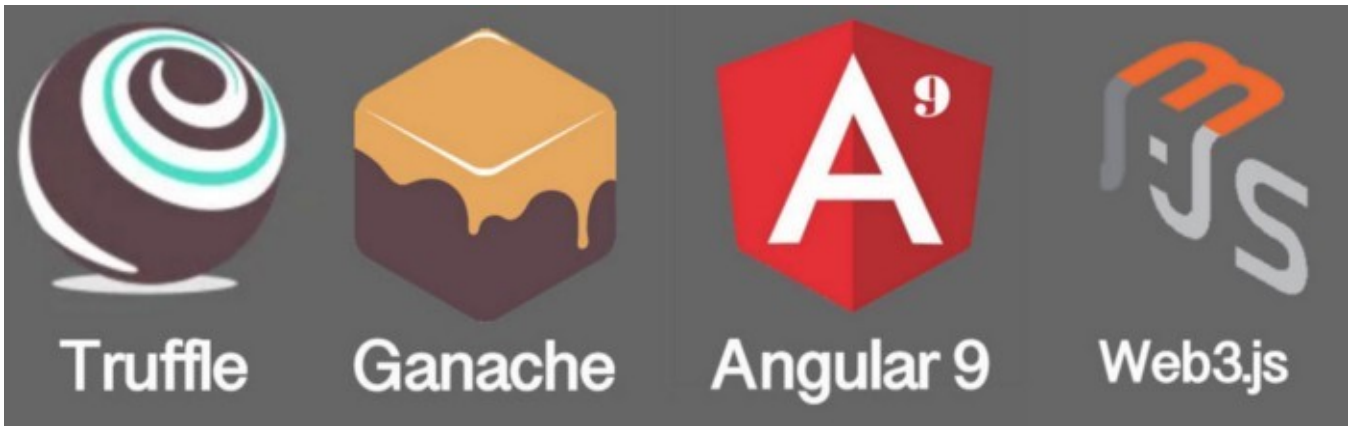

Figure 1. our development include Truffle, Ganache, Angular 9 and Web3.

First, navigate back into your Angular project folder and then install web3.js with the flag — save to save the library you are installing.

```
cd ethdapp/
npm install web3 --save

+ web3@1.2.4
```

If the installation went well, you will see in the output that the version did install. At the time of writing, **web3 is at version 1.2.4**.

You also will be installing @truffle/contract, which provides wrapper code that makes interaction with your contract easier. At the time of writing, the latest is **version 4.1.3** at the time of writing.

```
npm install @truffle/contract --save

+ truffle-contract@4.1.3
```

> ***Tip*** *In case, you run into compatibility issues.*
> *web3 version 1.2.4 and @truffle/contract version*
> *4.0.31 are the latest versions and compatible with Angular 9.x.*
>
> *However, this can change, so watch the version you are installing*
> *to ensure it's compatible and to avoid errors. Re-install with exact*
> *@[version], for instance,*

```
npm install @truffle/contract@4.1.0
```

Transfer Service

Now that you have your libraries installed, you can continue. In this section, you will create and write a service class. A service class is going to be your front-end middle layer for Angular to interact with web3. To get started, you can utilize the ng s flag, which stands for "service."

```
cd ~/Desktop/ethdapp
ng g s services/transfer

CREATE src/app/services/transfer.service.spec.ts (367 bytes)
CREATE src/app/services/transfer.service.ts (137 bytes)
```

Next, we need to replace the service class's initial code with logic to interact

with web3.

To do that, first, we will define the libraries we will be using, which are the Angular core and the truffle-contract and web3 libraries you installed.

Let's review the code of transfer.service.ts. First we will be using the Angular 9 framework as well as the web3 library so it needs to be defined;

```
import { Injectable } from '@angular/core';
const Web3 = require('web3');
```

Next, we need to define three variables we will be using later: require, window, and tokenAbi.

Notice that tokenAbi points to the ABI file we compiled from the contract SOL file in the previous article.

```
declare let require: any;
declare let window: any;
const tokenAbi = require('../../../truffle/build/contracts
/Transfer.json');
```

Next, we need access to root to interact with web3, so we can inject it into our project.

```
@Injectable({
  providedIn: 'root'
})
```

Now we can define the class definition, the account and the web3 variables we will be using, as well as init web3.

```
constructor() {
  if (window.ethereum === undefined) {
```

```
      alert('Non-Ethereum browser detected. Install MetaMask');
    } else {
      if (typeof window.web3 !== 'undefined') {
        this.web3 = window.web3.currentProvider;
      } else {
        this.web3 = new
Web3.providers.HttpProvider('http://localhost:8545');
      }
      console.log('transfer.service :: constructor ::
window.ethereum');
      window.web3 = new Web3(window.ethereum);
      console.log('transfer.service :: constructor :: this.web3');
      console.log(this.web3);
      this.enable = this.enableMetaMaskAccount();
    }
  }
```

The browsers that support MetaMask (if installed) will have the variable "window.ethereum" set for us. In the next chapter, we will install and set MetaMask. In this method set web3 as a global variable so our service class can interact with it, that's why you see "window.web3". Another implementation can only set web3 when needed. I kept it simple.

notice that I am also calling a method "*this*.enable" this method will open MetaMask to enable our app to interact with our wallet. You will see this done in our next article. The code is waiting for the user interaction from MetaMask so I set it as a promise with await. Here is the code;

```
  private async enableMetaMaskAccount(): Promise<any> {
    let enable = false;
    await new Promise((resolve, reject) => {
      enable = window.ethereum.enable();
    });
    return Promise.resolve(enable);
  }
```

> ***Notice*** *that you wrapped console.log messages around the code so you can see the messages in the browser console messages section under developer tool mode to help you understand what's happening.*

> *To do so open the browser in a developer tool mode. For Chrome, select View Developer View ➤ Developer ➤ Developer Tools.*

Next, we need an async method to get the account address and balance, so you can use a promise function. If your account was not retrieved previously, you'll call web3.eth.getAccounts just as you did in Terminal to retrieve the data. You also need error code if something goes wrong.

```
private async getAccount(): Promise<any> {
  console.log('transfer.service :: getAccount :: start');
  if (this.account == null) {
    this.account = await new Promise((resolve, reject) => {
      console.log('transfer.service :: getAccount :: eth');
      console.log(window.web3.eth);
      window.web3.eth.getAccounts((err, retAccount) => {
        console.log('transfer.service :: getAccount: retAccount');
        console.log(retAccount);
        if (retAccount.length > 0) {
          this.account = retAccount[0];
          resolve(this.account);
        } else {
          alert('transfer.service :: getAccount :: no accounts
found.');
          reject('No accounts found.');
        }
        if (err != null) {
          alert('transfer.service :: getAccount :: error retrieving
account');
          reject('Error retrieving account');
        }
      });
    }) as Promise<any>;
  }
  return Promise.resolve(this.account);
}
```

Similarly, you need a service method to interact with and get the balance of the account. You use web3.eth.getBalance just as you did in Terminal and wrap some error checking. You also set this as a promise. The reason you need a promise is that these calls are async, and JavaScript is not.

```
public async getUserBalance(): Promise<any> {
  const account = await this.getAccount();
  console.log('transfer.service :: getUserBalance :: account');
  console.log(account);
  return new Promise((resolve, reject) => {
    window.web3.eth.getBalance(account, function(err, balance) {
      console.log('transfer.service :: getUserBalance ::
getBalance');
      console.log(balance);
      if (!err) {
        const retVal = {
          account: account,
          balance: balance
        };
        console.log('transfer.service :: getUserBalance ::
getBalance :: retVal');
        console.log(retVal);
        resolve(retVal);
      } else {
        reject({account: 'error', balance: 0});
      }
    });
  }) as Promise<any>;
}
```

Lastly, we need to set the method to actually transfer the funds;

```
transferEther(value) {
  const that = this;
  console.log('transfer.service :: transferEther to: ' +
    value.transferAddress + ', from: ' + that.account + ', amount: '
+ value.amount);
  return new Promise((resolve, reject) => {
    console.log('transfer.service :: transferEther :: tokenAbi');
    console.log(tokenAbi);
    const contract = require('@truffle/contract');
    const transferContract = contract(tokenAbi);
    transferContract.setProvider(that.web3);
    console.log('transfer.service :: transferEther ::
transferContract');
    console.log(transferContract);
    transferContract.deployed().then(function(instance) {
      return instance.pay(
        value.transferAddress,
```

```
        {
          from: that.account,
          value: value.amount
        });
    }).then(function(status) {
      if (status) {
        return resolve({status: true});
      }
    }).catch(function(error) {
      console.log(error);
      return reject('transfer.service error');
    });
  });
}
```

The complete code of our **transfer.service.ts** is below;

```
import { Injectable } from '@angular/core';
const Web3 = require('web3');

declare let require: any;
declare let window: any;
const tokenAbi = require('../../../truffle/build/contracts
/Transfer.json');

@Injectable({
  providedIn: 'root'
})
export class TransferService {
  private account: any = null;
  private readonly web3: any;
  private enable: any;

  constructor() {
    if (window.ethereum === undefined) {
      alert('Non-Ethereum browser detected. Install MetaMask');
    } else {
      if (typeof window.web3 !== 'undefined') {
        this.web3 = window.web3.currentProvider;
      } else {
        this.web3 = new
Web3.providers.HttpProvider('http://localhost:8545');
      }
      console.log('transfer.service :: constructor ::
window.ethereum');
      window.web3 = new Web3(window.ethereum);
      console.log('transfer.service :: constructor :: this.web3');
```

```
      console.log(this.web3);
      this.enable = this.enableMetaMaskAccount();
    }
  }

  private async enableMetaMaskAccount(): Promise<any> {
    let enable = false;
    await new Promise((resolve, reject) => {
      enable = window.ethereum.enable();
    });
    return Promise.resolve(enable);
  }

  private async getAccount(): Promise<any> {
    console.log('transfer.service :: getAccount :: start');
    if (this.account == null) {
      this.account = await new Promise((resolve, reject) => {
        console.log('transfer.service :: getAccount :: eth');
        console.log(window.web3.eth);
        window.web3.eth.getAccounts((err, retAccount) => {
          console.log('transfer.service :: getAccount: retAccount');
          console.log(retAccount);
          if (retAccount.length > 0) {
            this.account = retAccount[0];
            resolve(this.account);
          } else {
            alert('transfer.service :: getAccount :: no accounts
found.');
            reject('No accounts found.');
          }
          if (err != null) {
            alert('transfer.service :: getAccount :: error
retrieving account');
            reject('Error retrieving account');
          }
        });
      }) as Promise<any>;
    }
    return Promise.resolve(this.account);
  }
  public async getUserBalance(): Promise<any> {
    const account = await this.getAccount();
    console.log('transfer.service :: getUserBalance :: account');
    console.log(account);
    return new Promise((resolve, reject) => {
      window.web3.eth.getBalance(account, function(err, balance) {
        console.log('transfer.service :: getUserBalance ::
getBalance');
        console.log(balance);
        if (!err) {
```

```
          const retVal = {
            account: account,
            balance: balance
          };
          console.log('transfer.service :: getUserBalance ::
getBalance :: retVal');
          console.log(retVal);
          resolve(retVal);
        } else {
          reject({account: 'error', balance: 0});
        }
      });
    }) as Promise<any>;
  }
  transferEther(value) {
    const that = this;
    console.log('transfer.service :: transferEther to: ' +
      value.transferAddress + ', from: ' + that.account + ', amount:
' + value.amount);
    return new Promise((resolve, reject) => {
      console.log('transfer.service :: transferEther :: tokenAbi');
      console.log(tokenAbi);
      const contract = require('@truffle/contract');
      const transferContract = contract(tokenAbi);
      transferContract.setProvider(that.web3);
      console.log('transfer.service :: transferEther ::
transferContract');
      console.log(transferContract);
      transferContract.deployed().then(function(instance) {
        return instance.pay(
          value.transferAddress,
          {
            from: that.account,
            value: value.amount
          });
      }).then(function(status) {
        if (status) {
          return resolve({status: true});
        }
      }).catch(function(error) {
        console.log(error);
        return reject('transfer.service error');
      });
    });
  }
}
```

Now that we have the transfer service (transfer.service.ts) complete, we can connect

transfer.component to get the user's account address and balance and be able to transfer funds once the form is filled in.

First, we need to define the service component we created.

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

Open and add the import statement at the top of the document.

```
import {TransferService} from '../../services/transfer.service';
```

For the component definition, add TransferService as a provider.

```
@Component({
  selector: 'app-transfer',
  templateUrl: './transfer.component.html',
  styleUrls: ['./transfer.component.css'],
  providers: [TransferService]
})
```

Also, add TransferService to the constructor so you can use it in your class.

```
constructor(private fb: FormBuilder,
private transferService: TransferService) { }
```

Next, update the getAccountAndBalance method to include a call to the service class and retrieve the user's actual account and balance.

```
getAccountAndBalance = () => {
  const that = this;
  this.transferService.getUserBalance().
```

```
    then(function(retAccount: any) {
      that.user.address = retAccount.account;
      that.user.balance = retAccount.balance;
      console.log('transfer.components :: getAccountAndBalance ::
that.user');
      console.log(that.user);
    }).catch(function(error) {
      console.log(error);
    });
  }
```

Lastly, update submitForm to call transferEther to transfer and pay.
Replace the submitForm TODO comments shown here with the call to the service calls:

```
// TODO: service call
```

Then pass the data the user-submitted:

```
// TODO: service call
this.transferService.transferEther(this.userForm.value).
then(function() {}).catch(function(error) {
  console.log(error);
});
```

Now in terminal run ng serve to ensure you don't have any errors;

```
cd ~/Desktop/ethdapp
ng serve

: Compiled successfully.
```

If you navigate to http://localhost:4200 to view our dapp you will get an error message that MetaMask is not installed. This is expected as we did not installed MetaMask yet.
See Figure 2.

Figure 2. The expected error message as we did not install MetaMask.

Let's recap!

In this article, we installed web3.js as well as truffle-contract. We then created our transfer service in Angular and connected our component class to the service class.

Where to go from here

JavaScript Web3 Ethereum Dapps Angular d last article we will be covering;

- Part VI — **Linking** and connecting our dapp to the Ethereum network via MetaMask and test our dapp.

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