

# A2 in Action

Tuesday, August 13, 2019 5:00 PM

## Example repo

<https://github.com/angular-in-action> [It will have all the code]

<https://github.com/angular-in-action/stocks>

28 (48 of 321)



you can copy the contents from the chapter as it appears.

If you haven't set up the Angular CLI, please go back to chapter 1 and set it up. We're using the CLI version 1.5 in this book, so if you're using an older version you'll want to upgrade.

To the terminal, run `npm install` to install the dependencies and `npm run build` to build the application.

## Keeping examples up-to-date with Angular releases

Because the book was developed to run against Angular 5 versions of code, there may be changes that happen in future versions that could cause an example in this book to fail. Don't worry—you can always visit the chapter project's GitHub page to see any notes that are added to discuss possible changes or errata, as well as view the book forum on Manning's website.

**Table 2.1** Top-level contents of the project generated by the CLI and their roles

Asset	Role
e2e	End-to-end testing folder, contains a basic stub test
node_modules	Standard NPM modules directory, no code should be placed here
src	Source directory for the application
.editorconfig	Editor configuration defaults
.angular-cli.json	Configuration file for the CLI about this project
karma.conf.js	Karma configuration file for unit test runner
package.json	Standard NPM package manifest file
protractor.conf.js	Protractor configuration file for e2e test runner
README.md	Standard readme file, contains starter information
tsconfig.json	Default configuration file for TypeScript compiler
tslint.json	TSLint configuration file for TypeScript linting rules

Table 2.2 Contents of the src directory and their roles

Asset	Role
app	Contains the primary App component and module
assets	Empty directory to store static assets like images
environments	Environment configurations to allow you to build for different targets, like dev or production
favicon.ico	Image displayed as browser favorite icon
index.html	Root HTML file for the application
main.ts	Entry point for the web application code
polyfills.ts	Imports some common polyfills required to run Angular properly on some browsers
styles.css	Global stylesheet
test.ts	Unit test entry point, not part of application
tsconfig.app.json	TypeScript compiler configuration for apps

[www.allitebooks.com](http://www.allitebooks.com)

*How Angular renders the base application*

31

Table 2.2 Contents of the src directory and their roles (continued)

Asset	Role
tsconfig.spec.json	TypeScript compiler configuration for unit tests
typings.d.ts	Typings configuration

32

## CHAPTER 2 Building your first Angular app

That listing might contain some unfamiliar syntax if you're new to TypeScript, so let's take a closer look at each section of the code. First, you import the `Component` annotation. It's used to **decorate the App component** by adding details that are related to the component but aren't part of its **controller logic**, which is the `AppComponent` class. Angular looks at these annotations and uses them with the `AppComponent` **controller class** to create the component at runtime.

is called **interpolation** and is frequently used to display data in a template.

We've looked at the App component, but now we need to look at the App module to see how things get wired up and rendered with Angular.

## 2.4.2 App module

The App module is the **packaging** that helps tell Angular what's available to render. Just as most food items have packaging that describes the various ingredients inside and other important values, a module describes the **various dependencies that are needed to render the module**.

There's at least one module in an application, but it's possible to create multiple modules for different reasons (covered later). In this case, it's the App component from earlier plus additional capabilities that are needed in most applications (such as **routing, forms, and HttpClient**).

### Listing 2.2 App module (src/app/app.module.ts)

Providers are any **services** used in the app.

```
import { BrowserModule } from '@angular/platform-browser';  
import { NgModule } from '@angular/core';  
  
import { AppComponent } from './app.component';  
  
@NgModule({  
  declarations: [  
    AppComponent  
  ],  
  imports: [  
    BrowserModule,  
  ],  
  providers: [],  
  bootstrap: [AppComponent]  
})  
export class AppModule { }
```

Imports Angular dependencies needed

Imports the App component

Uses the NgModule annotation to define a module by passing an object

Declarations are to list any **components** and **directives** used in the app.

Imports are other **modules** that are used in the app.

Bootstrap declares which component to use as the first to bootstrap the application.

Exports an empty class, which gets annotated with configuration from NgModule

Just like a component, a module is an object with an decorator. The object here is called AppModule, and NgModule is the decorator. The first block is to import any Angular dependencies that are common to most apps and the App component.

The NgModule decorator takes an object with a few different properties. The declarations property is to provide a list of any components and directives to make available to the entire application.

The imports property is an **array** of other modules upon which this module

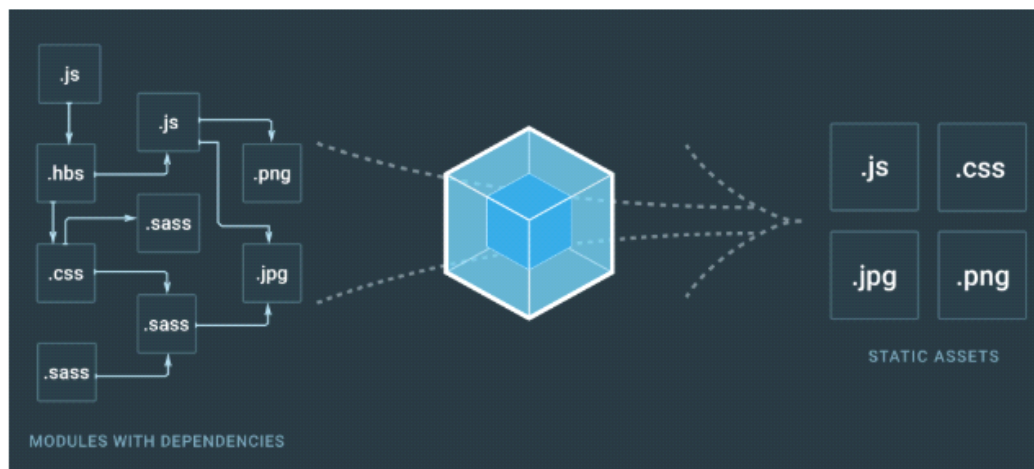
## 2.4.3 Bootstrapping the app

The application must be bootstrapped at runtime to start the process of rendering. So far, we've only declared code, but now we'll see how it gets executed. The CLI takes care of wiring up the **build tooling**, which is based on **webpack**.

Start by taking a look at the **.angular-cli.json** file. You'll see an array of apps, and one of the properties is the main property. By default, it points to the **src/app/main.ts** file. This means that when the application gets built, it will automatically call the contents of the main.ts file as the first set of instructions.

Angular CLI uses **webpack** as a build tool.

## Webpack



### webpack

You could say **webpack** takes a more unified approach than Browserify. Whereas Browserify consists of multiple small tools, webpack comes with a core that provides a lot of functionality out of the box.

Webpack core can be extended using specific *loaders* and *plugins*. It gives control over how it *resolves* the modules, making it possible to adapt your build to match specific situations and workaround packages that don't work correctly out of the box.

Compared to the other tools, webpack comes with initial complexity, but it makes up for this through its broad feature set. It's an advanced tool that requires patience. But once you understand the basic ideas behind it, webpack becomes powerful.

- **Task runners** and **bundlers** solve different problems. You can achieve similar results with both, but often it's best to use them together to complement each other.
- Older tools, such as Make or RequireJS, still have influence even if they aren't as popular in web development as they once were.
- Bundlers like Browserify or **webpack** solve an important problem and help you to manage complex web applications.
- Emerging technologies approach the problem from different angles. Sometimes they build on top of other tools, and at times they can be used together.

typescript compiler can inspect the type definition and determine how to inject the requested object into the class. If you're new to TypeScript, keep in mind that anytime you see a colon after a variable declaration, you're defining the object type that should be assigned to that variable.

```
this.http.get<Array<StockInterface>>(...
```

This is known as a *type variable*, which is a feature of TypeScript that allows you to tell the `http.get()` method what type of object it should expect, and in this case it will expect to get an array of objects that conform to the `StockInterface` (our stock objects). This is optional, but it's very helpful to alert the compiler if you try to access properties that don't exist.

There's one more step we have to do, because the CLI doesn't automatically register the service with the App module, and we need to register `HttpClient` with the application as well. Open the `src/app/app.module.ts` file and near the top add these two

```
1 1
This service is not too complex. It's mostly designed to abstract the modification of the array so it's not directly modified and load the data from the API. While the application runs, the stocks array can be modified, and changes are reflected in both the
```

<https://angular2-in-action-api.herokuapp.com/stocks/snapshot?symbols=aapl,goog,fb>

---

```
1 import { Injectable } from '@angular/core';
2 import { HttpClient } from '@angular/common/http';
3
4 let stocks: Array<string> = ['AAPL', 'GOOG', 'FB', 'AMZN', 'TWTR'];
5 let service: string = 'https://angular2-in-action-api.herokuapp.com';
6
7 export interface StockInterface {
8     symbol: string;
9     lastTradePriceOnly: number;
10    change: number;
11    changeInPercent: number;
12 }
13
14 @Injectable()
15 export class StocksService {
16
17     constructor(private http: HttpClient) {}
18
19     get() {
20         return stocks;
21     }
22
23     add(stock) {
24         stocks.push(stock);
25         return this.get();
26     }
27
28     remove(stock) {
29         stocks.splice(stocks.indexOf(stock), 1);
30         return this.get();
31     }
32
33     load(symbols) {
34         if (symbols) {
35             return this.http.get<Array<StockInterface>>(service + '/stocks/snapshot?symbols=' + symbols.join());
36         }
37     }
38 }
```

---

### 3 App Essentials



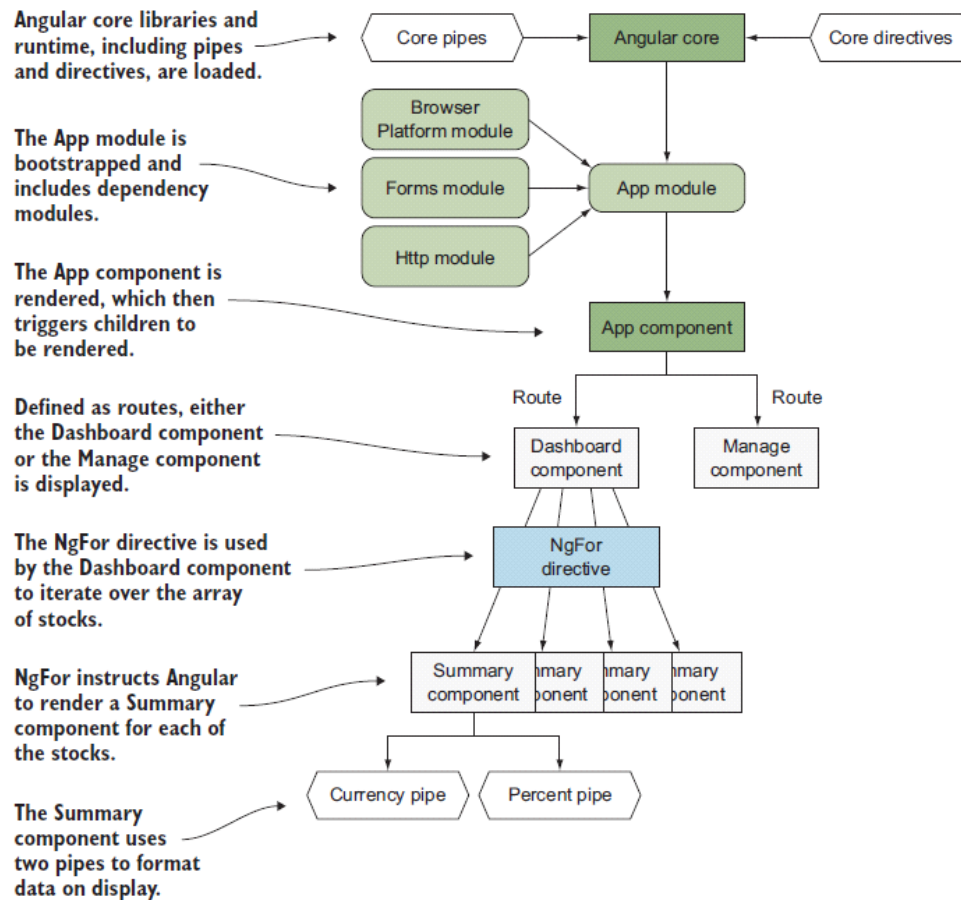


Figure 3.2 The entities and how they are leveraged during the application execution

## Services

- Create class and declare it as *@Injectable*
- Use it as a property in another class/controller
- Add it in the providers array in the *NgModule*

### Listing 6.1 Account Service Basics

```
import { Injectable } from '@angular/core';
import { Stock } from './stocks.model';

const defaultBalance: number = 10000;

@Injectable()
export class AccountService {
  private balance: number = defaultBalance;
```

Imports dependencies

Defines a variable outside of the class

Annotates class with @Injectable()

## Listing 6.2 App Component Consuming the Account Service

```
import { Component, OnInit, OnDestroy } from '@angular/core';
import { AccountService } from '../services/account.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
```

*Creating Angular services*

135

```
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit, OnDestroy {
  constructor(private accountService: AccountService) {}
  // skipping some content
}
```

← Injects service as a private property

Then in the NgModule, add the AccountService to the providers array:

```
providers: [
  LocalStorageService,
  CurrencyPipe,
  AccountService,
]
```

## Routing

### flow

```
index.html - <app-root></app-root>
app.component.ts selector: 'app-root',
app.component.html - <router-outlet></router-outlet>
app.routes
app.module.ts [add to imports]
```



---

```

<div class="mdl-layout mdl-js-layout mdl-layout--fixed-header">
  <header class="mdl-layout__header">
    <div class="mdl-layout__header-row">
      <span class="mdl-layout-title">Stock Tracker</span>
      <div class="mdl-layout-spacer"></div>
      <nav class="mdl-navigation mdl-layout--large-screen-only">
        <a class="mdl-navigation__link" [routerLink]="['/']">Dashboard</a>
        <a class="mdl-navigation__link" [routerLink]="['/manage']">Manage</a>
      </nav>
    </div>
  </header>
  <main class="mdl-layout__content" style="padding: 20px;">
    <router-outlet></router-outlet>
  </main>
</div>

```

---

**Listing 2.14** App routing configuration

```

import { Routes, RouterModule } from '@angular/router';
import { DashboardComponent } from '../components/dashboard/dashboard.component';
import { ManageComponent } from '../components/manage/manage.component';

const routes: Routes = [
  {
    path: '',
    component: DashboardComponent
  },
  {
    path: 'manage',
    component: ManageComponent
  }
];

export const AppRoutes = RouterModule.forRoot(routes);

```

Imports router dependencies

Imports App components that are linked to a route

Defines a route configuration array

Exports the routes for use

```
import { AppRoutes } from './app.routes';

@NgModule({
  declarations: [
    AppComponent,
    SummaryComponent,
    DashboardComponent,
    ManageComponent
  ],
  imports: [
    BrowserModule,
    HttpClientModule,
    FormsModule,
    AppRoutes
  ],
  providers: [
    StocksService
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

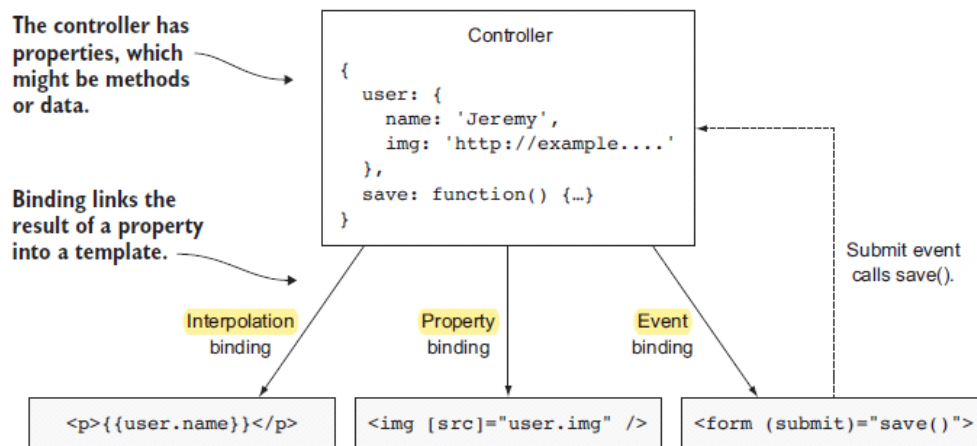


Figure 3.8 How a controller binds data into a template for interpolation, property, and event bindings, each using a different syntax

## Interpolation

It's a kind of binding works by taking an expression, evaluating it, and replacing the binding with the result. This is similar to how a spreadsheet can take a formula interpolation is shorthand for binding to the `textContent` property of an element.

```
document.getElementById("demo").textContent = "Paragraph changed!";
```

```
<img [src]="user.img" />
```

In fact, interpolation is shorthand for binding to the `textContent` property of an element. They can both accomplish the same thing in many situations, so you can choose to use whichever feels most natural.

The syntax for property bindings is to put the property onto the element wrapped in brackets (`[]`). The name should match the property, usually in camel case, like `textContent`. We can rewrite the interpolation template to use property bindings like this:

```
<p [textContent]="user.name"></p>
```

## Property bindings

which allow you to bind values to properties of an element to modify their behavior or appearance.

properties are the DOM element's property

Using the `[]` syntax binds to an element's property, not the attribute. This is an important distinction, because properties are the DOM element's property. That makes it possible to use any valid HTML element property (such as the `img src` property). Instead of binding the data to the attribute, you're binding data directly to the element property, which is quite efficient.

## 4.chap

Templates may include references to other components, which will trigger them to also be rendered as part of the rendering of the parent component. As discussed in chapter 3,

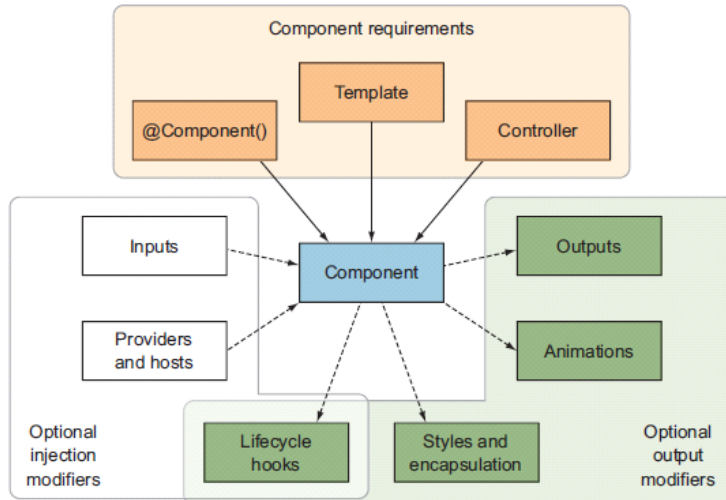


Figure 4.4 Concepts that compose and influence a component's behavior

## 5 Advanced Components

you could create a tabs component and emit an event that describes what tab is currently visible. Another component might be interested in knowing when the tab selection changes so that you can update the state of that component, such as a nearby panel that has contextual help information based on the current tab.

Additionally, we'll look at a couple ways to use other components in this section: using a **View Child** (which gives you access to a child component controller in a component controller) and **using a local variable** (which gives you access to a child component controller in a component's template). Each has a different design and might work in

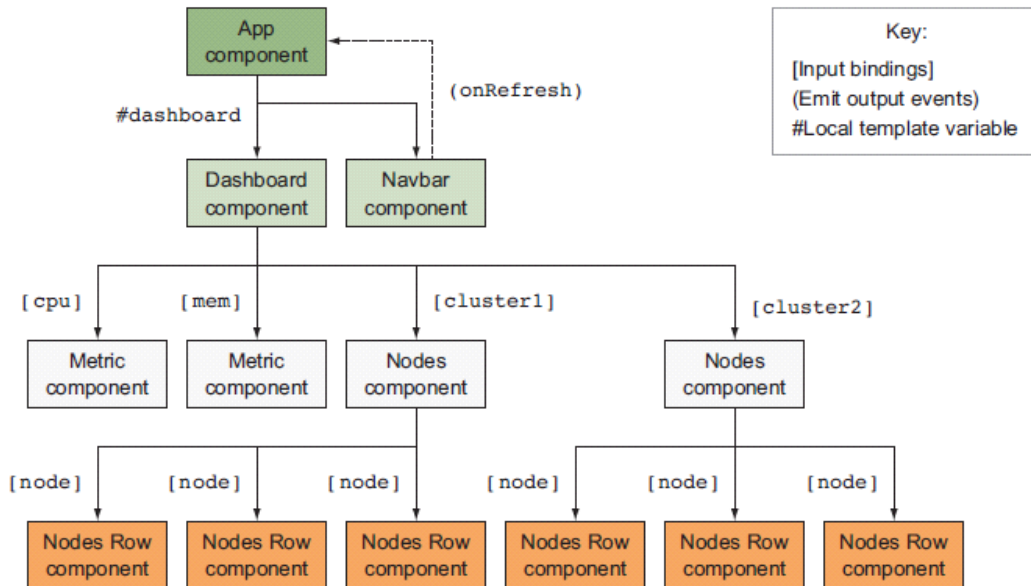


Figure 5.2 Components sharing data, emitting events, and accessing other components via local variables

### Approach-1 - Usingin Local Template Variable

#### Listing 5.3 Navbar component using an output

```
import { Component, Output, EventEmitter } from '@angular/core';

@Component({
  selector: 'app-navbar',
  templateUrl: './navbar.component.html',
  styleUrls: ['./navbar.component.css']
})
export class NavbarComponent {
  @Output() onRefresh: EventEmitter<null> = new EventEmitter<null>();

  refresh() {
    this.onRefresh.emit();
  }
}

<button class="btn btn-success" type="button" (click)="refresh()">Reload
</button>
```

Imports the Output decorator and EventEmitter factory

Creates a property with Output of an EventEmitter type

Implements a method to call on click that emits the event

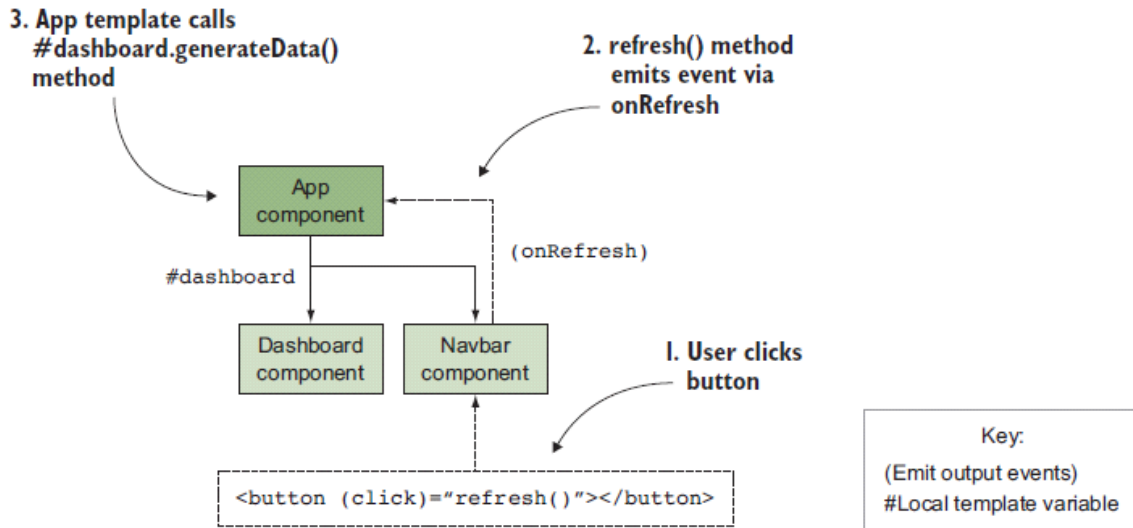


Figure 5.3 Overview of component tree, how user clicks button and it triggers refresh of data via an output event

```

<app-navbar (onRefresh)="dashboard.generateData()"></app-navbar>
<app-dashboard #dashboard></app-dashboard>

```

it from the App component. The second line adds `#dashboard` to the template on the Dashboard component. This denotes a **local template variable**, accessible as the

`generateData()` method from the Dashboard component. This is handy for accessing components that exist in the same template. The major negative is that it only allows you to access the component controller from the template and not from the controller, meaning the

## Approach-2 - *ViewChild*

**ViewChild** is a decorator for a controller property, like `Inject` or `Output`, which tells Angular to fill in that property with a reference to a specific child component controller. It's limited to injecting only children, so if you try to inject a component that isn't a direct descendent, it will provide you with an undefined value.

Listing 5.4 App component controller

```

Imports  → import { Component, ViewChild } from '@angular/core';
ViewChild → import { DashboardComponent } from '../dashboard/dashboard.component';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  @ViewChild(DashboardComponent) dashboard: DashboardComponent;

  refresh() {
    this.dashboard.generateData();
  }
}

```

Imports the Dashboard component

Declares the dashboard property, decorated with ViewChild

Implements refresh method that calls Dashboard component method

```
<app-navbar (onRefresh)="refresh()"></app-navbar>
<app-dashboard></app-dashboard>
```

You may be wondering which approach you should use, and it largely boils down to where you want to store the logic. If you ever need to do anything more than access properties or methods of a child controller, you probably need to use the `ViewChild` approach. This does cause a **coupling between the two components**, which should be avoided when possible. But if you can reference a child controller and call a method directly in the template, it can save code and **reduce coupling**.

5.3 and 5.4 are pending

## Forms

Angular provides two approaches to building forms: **reactive forms** and **template forms**. I'll discuss the differences at length shortly, though they're largely in whether

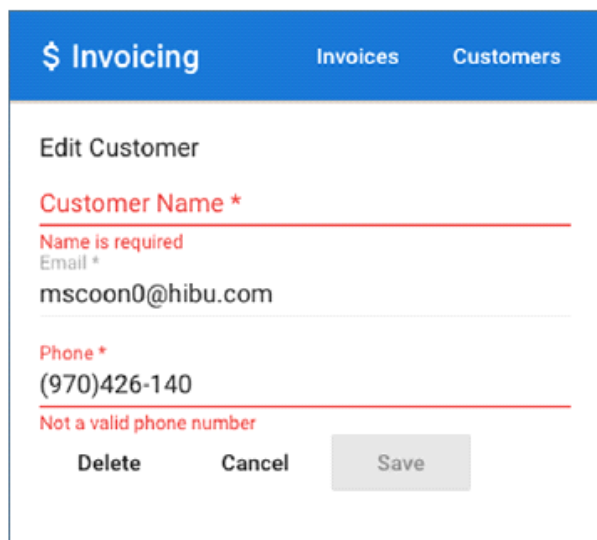
The application is also designed for the **mobile** form factor, which is a nice little twist from our previous examples. It uses the **Covalent UI library**, from **Teradata**, which

Inside of the components you'll see a few new things. The **TdLoading** directive is a feature from the **Covalent** library to display a loading indicator while data is being loaded. The **MdInput** directive will make an input **Material Design**-compliant. There

The **primary goal of a form is to be able to synchronize the data in the view with data in the controller so it can be submitted to be handled**. Secondary goals are to perform **tasks like validation, notify about errors, and handle other events like cancel**.

Because the form is primarily defined in the template, there is the caveat that only processing.

To get started, we need to start working with our form controls and wire them up so that **they bind the model between both the controller and the template using NgModel**.



The screenshot shows a web application interface for managing customers. At the top, there's a blue header with a dollar sign icon and the text "Invoicing". Below the header, there are two tabs: "Invoices" and "Customers". The "Customers" tab is selected. The main content area is titled "Edit Customer". It contains three form fields: "Customer Name \*" with a red error message "Name is required", "Email \*" with the value "mscoon0@hibu.com", and "Phone \*" with the value "(970)426-140" and a red error message "Not a valid phone number". At the bottom of the form, there are three buttons: "Delete", "Cancel", and "Save".

Figure 9.4 Customer form with validation errors



## Listing 9.2 CustomerForm validating fields with NgModel

```

<md-input-container>
  <input name="customer" mdInput placeholder="Customer Name"
    [(ngModel)]="customer.name" required #name="ngModel">
  <md-error *ngIf="name.touched && name.invalid">
    Name is required
  </md-error>
</md-input-container>
<md-input-container>
  <input name="email" mdInput type="email" placeholder="Email"
    [(ngModel)]="customer.email" required #email="ngModel">
  <md-error *ngIf="email.touched && email.invalid">
    A valid email is required
  </md-error>
</md-input-container>
<md-input-container>
  <input name="phone" mdInput type="tel" placeholder="Phone"
    [(ngModel)]="customer.phone" required #phone="ngModel" minlength="7">
  <md-error *ngIf="phone.touched && phone.errors?.required">
    Phone number is required
  </md-error>
  <md-error *ngIf="phone.touched && phone.errors?.minlength">
    Not a valid phone number
  </md-error>
</md-input-container>

```

← Adds validation attributes and template variable to form control

← Uses form control validation to conditionally show error message

← Form control exposes what specific error is found.

Property	Meaning
valid	The form control is valid for all validations.
invalid	The form control has at least one invalid validation.
disabled	The form control is disabled and can't be interacted with.
enabled	The form control is enabled and can be clicked or edited.
errors	An object that either contains keys with validations that are invalid, or null when all are valid.
pristine	The form control has not yet been changed by the user.
dirty	The form control has been changed by the user.
touched	The form control has been in focus, and then focus has left the field.
untouched	The form control has not been in focus.

The first thing we should do is update our form element. Angular does another thing to forms that isn't visible by default. It automatically implements an **NgForm** on a form even if you don't declare a directive (unlike how you have to declare **NgModel**). When it does this, it essentially attaches an **NgForm** controller that then maintains the form controls in the form.

```

<!-- ngForm ngSubmit ngModel :: FormsModule -->
<!-- Template Variables :: #form #name #email -->
<!-- FormControl properties :: invalid touched -->

<form *ngIf="stocks" #form="ngForm" (ngSubmit)="save()">

  <input name="stock" placeholder="Stock Name" required [(ngModel)]="stocks.symbol" #name="ngModel" >
  <span *ngIf="name.invalid && name.touched">
    | Name is required
  </span>

  <br><br>
  <input name="email" placeholder="Email" required [(ngModel)]="stocks.name" #email="ngModel" >
  <span *ngIf="email.invalid && email.touched">
    | Email is required
  </span>

  <br><br>
  <button type="submit" [disabled]="form.invalid">Save</button>

</form>

```

## Angular 2 Vs 4

view engine - generated code reduced 60%

animation package moved out to *@angular/core* package from *@angular/animations*

type script 2.1 and 2.2 compatibility

improved ngIf - ifelse is added

@Component

- selector
- template
- templateUrl
- styleUrls

@NgModule

- imports
- declarations
- providers
- bootstrap

digest lifecycle

ng new <projectName>

npm start

ng generate component <name> [options]

ng g component <name> [options]

## promises Vs Observables

<https://stackoverflow.com/questions/37364973/what-is-the-difference-between-promises-and-observables>

SPA - Single page application

<https://dzone.com/articles/how-single-page-web-applications-actually-work>

fast loading  
less bandwidth needed  
lot of htmls to download  
good for mobile devices and for slower internet connections

\*ngFor  
ngSubmit

<ng-content>

NgModel [directive]

Prototypes

[https://en.wikipedia.org/wiki/Prototype-based\\_programming](https://en.wikipedia.org/wiki/Prototype-based_programming)