

Your First Angular App – Task List App :

The best way to get started with Angular is to dive in and create a web application. In this Part, I show you how to set up your development environment and take you through the process of creating a basic application, starting with a static mock-up of the functionality and applying Angular features to create a dynamic web application, albeit a simple one. In Parts 7–10, I show you how to create a more complex and realistic Angular application, but for now a simple example will suffice to demonstrate the major components of an Angular app and set the scene for the other Parts in this part of the Ref.

Angular has a steep learning curve, so the purpose of this Part is just to introduce the basic flow of Angular development and give you a sense of how things fit together. It won't all make sense right now, but by the time you have finished reading this Ref., you will understand every step I take in this Part and much more besides.

Preparing the Development Environment

There is some preparation required for Angular development. In the sections that follow, I explain how to get set up and ready to create your first project. There is wide support for Angular in popular development tools, and you can pick your favorites.

Installing Node.js

Many of the tools used for Angular development rely on Node.js—also known as Node—which was created in 2009 as a simple and efficient runtime for server-side applications written in JavaScript. Node.js is based on the JavaScript engine used in the Chrome browser and provides an API for executing JavaScript code outside of the browser environment.

Node.js has enjoyed success as an application server, but for this Ref., it is interesting because it has provided the foundation for a new generation of cross-platform development and build tools. Some smart design decisions by the Node.js team and the cross-platform support provided by the Chrome JavaScript runtime have created an opportunity that has been seized upon by enthusiastic tool writers. In short, Node.js has become essential for web application development.

It is important that you download the same version of Node.js that I use throughout this Ref.. Although Node.js is relatively stable, there are still breaking API changes from time to time that may stop the examples I include in the Parts from working.

The version I have used is the 8.9.1, which is the current Long Term Support release at the time of writing. There may be a later version available by the time you read this but you should stick to the 8.9.1 release for the examples in this Ref.. A complete set of 8.9.1 releases, with installers for Windows and Mac OS and binary packages for other platforms, is available at <https://nodejs.org/dist/v8.9.1>.

When you install Node.js, make sure you select the option to add the Node.js executables to the path. When the installation is complete, run the following command:

```
node -v
```

If the installation has gone as it should, then you will see the following version number displayed:

```
v8.9.1
```

The Node.js installer includes the Node Package Manager (NPM), which is used to manage the packages in a project. Run the following command to ensure that NPM is working:

```
npm -v
```

If everything is working as it should, then you will see the following version number:

```
5.5.1
```

Installing the angular-cli Package

The **angular-cli** package has become the standard way to create and manage Angular projects during development. In the original version of this Ref., I demonstrated how to set up an Angular project from scratch, which is a lengthy and error-prone process that is simplified by **angular-cli**. To install **angular-cli** open a new command prompt and run the following command:

```
npm install --global @angular/cli@1.5.0
```

If you are using Linux or macOS, you may need to use **sudo**, like this:

```
sudo npm install --global @angular/cli@1.5.0
```

Installing Git

The Git revision control tool is required to manage some of the packages required for Angular development. If you are using Windows or macOS, then download and run the installer from <https://git-scm.com/downloads>. (On macOS, you may have to change your security settings in order to open the installer, which has not been signed by the developers).

Git is already installed on most Linux distributions. If you want to install the latest version, then consult the installation instructions for your distribution at <https://git-scm.com/download/linux>. As an example, for Ubuntu, which is the Linux distribution I use, I used the following command:

```
sudo apt-get install git
```

Once you have completed the installation, open a new command prompt and run the following command to check that Git is installed and available:

```
git --version
```

This command prints out the version of the Git package that has been installed. At the time of writing, the latest version of Git for Windows is 2.15.0, the latest version of Git for macOS is 2.15.0 and the latest version of Git for Linux is 2.7.4.

Installing an Editor

Table 2-1. Popular Angular-Enabled Editors

Name	Description
	See
	www.sublimetext.com
	atom.io
	brackets.io
	www.jetbrains.com/webstorm
	code.visualstudio.com

demonstrates how to create a combined Angular and ASP.NET Core MVC project available on the GitHub repository for this Ref..

Installing a Browser

The final choice to make is the browser that you will use to check your work during development. All the current-generation browsers have good developer support and work well with Angular. I have used Google Chrome throughout this Ref. and this is the browser I recommend you use as well.

Creating and Preparing the Project

Once you have Node.js, **angular-cli**, an editor, and a browser, you have enough of a foundation to start the development process.

Creating the Project

To create the project, select a convenient location and use a command prompt to run the following command to create a new project called **todo**.

```
ng new todo
```

The **ng** command is provided by the **angular-cli** package and **ng new** sets up a new project. The installation process creates a folder called **todo** that contains all of the configuration files that are needed to start Angular development, some placeholder files to start development and the NPM packages required for developing, running and deploying Angular applications. (There are a large number of NPM packages, which means that project creation can take a while).

Creating the Package File

NPM uses a file called **package.json** to get a list of the software packages that are required for a project. A **package.json** file is created by **angular-cli** as part of the project setup and it contains all of the packages requires for basic Angular development. The example application in this Part requires the Bootstrap CSS package, which is not part of the basic package set.

Edit the `package.json` file in the `todo` folder to add the Bootstrap package as shown in Listing 2-1.

Listing 2-1. Adding Bootstrap to the package.json File in the todo Folder

```
{
  "name": "todo",
  "version": "0.0.0",
  "license": "MIT",
  "scripts": {
    "ng": "ng",
    "start": "ng serve",
    "build": "ng build",
    "test": "ng test",
    "lint": "ng lint",
    "e2e": "ng e2e"
  },
  "private": true,
  "dependencies": {
    "@angular/animations": "^5.0.0",
    "@angular/common": "^5.0.0",
    "@angular/compiler": "^5.0.0",
    "@angular/core": "^5.0.0",
    "@angular/forms": "^5.0.0",
    "@angular/http": "^5.0.0",
    "@angular/platform-browser": "^5.0.0",
    "@angular/platform-browser-dynamic": "^5.0.0",
    "@angular/router": "^5.0.0",
    "core-js": "^2.4.1",
    "rxjs": "^5.5.2",
    "zone.js": "^0.8.14",
    "bootstrap": "4.0.0-alpha.4"
  },
  "devDependencies": {
    "@angular/cli": "1.5.0",
    "@angular/compiler-cli": "^5.0.0",
    "@angular/language-service": "^5.0.0",
    "@types/jasmine": "~2.5.53",
    "@types/jasminewd2": "~2.0.2",
    "@types/node": "~6.0.60",
    "codelyzer": "~3.2.0",
    "jasmine-core": "~2.6.2",
    "jasmine-spec-reporter": "~4.1.0",
    "karma": "~1.7.0",
    "karma-chrome-launcher": "~2.1.1",
    "karma-cli": "~1.0.1",
    "karma-coverage-istanbul-reporter": "^1.2.1",
    "karma-jasmine": "~1.1.0",
```

```
    "karma-jasmine-html-reporter": "^0.2.2",
    "protractor": "~5.1.2",
    "ts-node": "~3.2.0",
    "tslint": "~5.7.0",
    "typescript": "~2.4.2"
  }
}
```

The **package.json** file lists the packages required to get started with Angular development and some commands to use them. I describe the project configuration in detail in Part 11, but for now, it is enough to understand what each section of the **package.json** file is for, as described in Table 2-2.

Table 2-2. The package.json Sections

Name	Description
scripts	This is a list of scripts that can be run from the command line. The scripts section in the listing defines the commands that are used to compile the source code and run the development HTTP server.
dependencies	This is a list of NPM packages that the web application relies on to run. Each package is specified with a version number. The dependencies section in the listing contains the core Angular packages, libraries that Angular depends on, and the Bootstrap CSS library that I use to style HTML content in this Ref..
devDependencies	This is a list of NPM packages that are relied on for development but that are not required by the application once it has been deployed. This section contains packages that will compile TypeScript files, provide a development HTTP server, and perform testing.

Installing the NPM Package

To process the **package.json** file to download and install the Bootstrap package that it specifies, run the following command inside the **todo** folder:

```
npm install
```

You may see some warnings as NPM grumbles about the packages it processes, but there should be no errors reported.

Starting the Server

The project tools and basic structure are in place, so it is time to test that everything is working. Run the following command from the **todo** folder:

```
ng serve --port 3000 --open
```

This command starts the development HTTP server that **angular-cli** has installed and configured to work with the Angular development tools. The initial startup process takes a moment to prepare the project, and will generate output similar to this:

```
** NG Live Development Server is listening on localhost:3000, open your browser on
http://localhost:3000/ **
Hash: e723156483c6e22fdd79
Time: 7008ms
chunk {inline} inline.bundle.js (inline) 5.79 kB [entry] [rendered]
chunk {main} main.bundle.js (main) 23.6 kB [initial] [rendered]
chunk {polyfills} polyfills.bundle.js (polyfills) 560 kB [initial] [rendered]
chunk {styles} styles.bundle.js (styles) 293 kB [initial] [rendered]
chunk {vendor} vendor.bundle.js (vendor) 7.82 MB [initial] [rendered]
webpack: Compiled successfully.
```

Don't worry if you see slightly different output, just as long as you see the "compiled successfully" message once the preparations are complete. After a few seconds, a new browser window will open, and you will see the output shown in Figure 2-1, which shows that the project startup has been successful and that the placeholder content created by **angular-cli** is being used.



Figure 2-1. The placeholder HTML content

Editing the HTML File

Even though `angular-cli` has added some placeholder content, I am going to strip everything back and start with an HTML file that contains static content that I will later enhance using Angular. The HTML that I am going to use will be styled using the Bootstrap CSS package. To configure the Angular development tools to provide the browser with the Bootstrap CSS file, add the entry shown in Listing 2-2 to the `styles` section of the `.angular-cli.json` file.

Listing 2-2. Configuring CSS in the `.angular-cli.json` File in the `todo` Folder

```
...  
"styles": [  
  "styles.css",  
  "../node_modules/bootstrap/dist/css/bootstrap.min.css"  
],
```

...

Restart the development server by running the following command in the `todo` folder:

```
ng serve --port 3000 --open
```

Next, edit the `index.html` file in the `todo/src` folder to replace the contents with those shown in Listing 2-3.

Listing 2-3. The Contents of the `index.html` File in the `todo/src` Folder

```
<!DOCTYPE html>
<html>
<head>
  <title>ToDo</title>
  <meta charset="utf-8" />
</head>
<body class="m-a-1">

  <h3 class="bg-primary p-a-1">Adam's To Do List</h3>

  <div class="m-t-1 m-b-1">
    <input class="form-control" />
    <button class="btn btn-primary m-t-1">Add</button>
  </div>

  <table class="table table-striped table-bordered">
    <thead>
      <tr>
        <th>Description</th>
        <th>Done</th>
      </tr>
    </thead>
    <tbody>
      <tr><td>Buy Flowers</td><td>No</td></tr>
      <tr><td>Get Shoes</td><td>No</td></tr>
      <tr><td>Collect Tickets</td><td>Yes</td></tr>
      <tr><td>Call Joe</td><td>No</td></tr>
    </tbody>
  </table>
</body>
</html>
```

The `angular-cli` development HTTP server adds a fragment of JavaScript to the HTML content it delivers to the browser. The JavaScript opens a connection back to the server and waits for a signal to reload the page, which is sent when the server detects a change in any of

the files in the `todo` directory. As soon as you save the `index.html` file, the server will detect the change and send the signal, and the browser will reload, reflecting the new content as shown in Figure 2-2.

When you are making changes to a series of files, there may be times when the browser won't be able to load and execute the example application, especially in later Parts where the examples are more complex. For the most part, the development HTTP server will trigger a reload in the browser and everything will be fine, but if it gets stuck, just click the browser's reload button or navigate to `http://localhost:3000` to get going again.

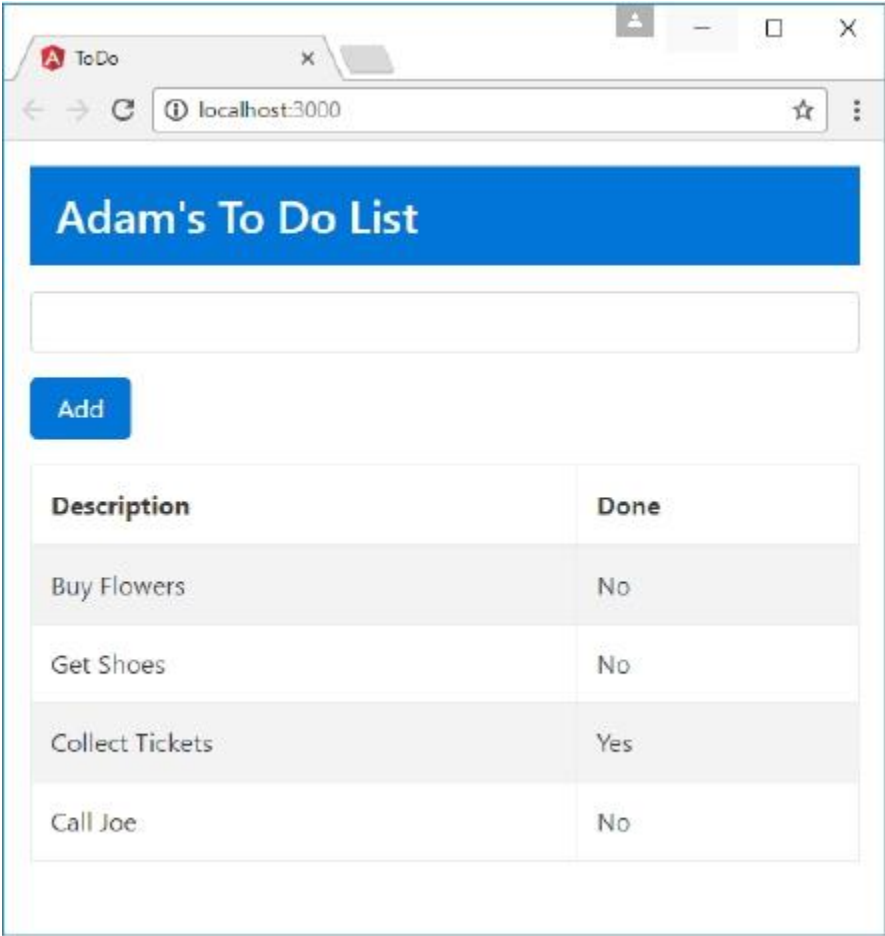


Figure 2-2. Editing the contents of the HTML file

The HTML elements in the `index.html` file show how the simple Angular application I create in this Part will look. The key elements are a banner with the user's name, an `input` element and an Add button that add a new to-do item to the list, and a table that contains all the to-do items and indicates whether they have been completed.

I used the excellent Bootstrap CSS framework to style HTML content. Bootstrap is applied by assigning elements to classes, like this:

```
...  
<h3 class="bg-primary p-a-1">Adam's To Do List</h3>  
...
```

This `h3` element has been assigned to two classes. The `bg-primary` class sets the background color of the element to the primary color of the current Bootstrap theme. I am using the default theme, for which the primary color is dark blue and there are other themed colors available, including `bg-secondary`, `bg-info`, and `bg-danger`. The `p-a-1` class adds a fixed amount of padding to all edges of the element, ensuring that the text has some space around it.

In the next section, I'll remove the HTML from the file, cut it up into smaller pieces, and use it to create a simple Angular application.

Using the Bootstrap Pre-Release

Throughout this Ref., I use a pre-release version of the Bootstrap CSS framework. As I write this, the Bootstrap team is in the process of developing Bootstrap version 4 and has made several early releases. These releases have been labeled as "alpha," but the quality is high and they are stable enough for use in the examples in this Ref..

Given the choice of writing this Ref. using the soon-to-be-obsolete Bootstrap 3 and a pre-release version of Bootstrap 4, I decided to use the new version even though some of the class names that are used to style HTML elements are likely to change before the final release. This means you must use the same version of Bootstrap to get the expected results from the examples, just like the rest of the packages listed in the `package.json` file in Listing 2-1.

Adding Angular Features to the Project

The static HTML in the `index.html` file acts as a placeholder for the basic application. The user should be able to see the list of to-do items, check off items that are complete, and create new

items. In the sections that follow, I add basic Angular features to the project to bring the to-do application to life. To keep the application simple, I assume that there is only one user and that I don't have to worry about preserving the state of the data in the application, which means that changes to the to-do list will be lost if the browser window is closed or reloaded. (Later examples, including the SportsStore application developed in Parts 7–10, demonstrate persistent data storage.)

Preparing the HTML File

The first step toward adding Angular to the application is to prepare the `index.html` file, as shown in Listing 2-4.

Listing 2-4. Preparing for Angular in the index.html File

```
<!DOCTYPE html>
<html>
<head>
  <title>ToDo</title>
  <meta charset="utf-8" />
</head>
<body class="m-a-1">
  <todo-app>Angular placeholder</todo-app>
</body>
</html>
```

This listing replaces the content of the `body` element with a `todo-app` element. There is no `todo-app` element in the HTML specification and the browser will ignore it when parsing the HTML file, but this element will be the entry point into the world of Angular and will be replaced with my application content. When you save the `index.html` file, the browser will reload the file and show the placeholder message, as shown in Figure 2-3.

If you followed the examples in the original edition of this Ref., you may be wondering why I have not added any `script` elements to the HTML file to incorporate the Angular functionality. The project set up by `angular-cli` uses a tool called Web Pack, which generates the JavaScript files for the project automatically and injects them automatically into the HTML files sent to the browser by the development HTTP server.



Figure 2-3. Preparing the HTML file

Creating a Data Model

When I created the static mock-up of the application, the data was distributed across all the HTML elements. The user's name is contained in the header, like this:

```
...  
<h3 class="bg-primary p-a-1">Adam's To Do List</h3>  
...
```

and the details of the to-do items are contained within `td` elements in the table, like this:

```
...  
<tr><td>Buy Flowers</td><td>No</td></tr>  
...
```

The next task is to pull all the data together to create a data model. Separating the data from the way it is presented to the user is one of the key ideas in the MVC pattern, as I explain in Part 3.

I am simplifying here. The model can also contain the logic required to create, load, store, and modify data objects. In an Angular app, this logic is often at the server and is accessed by a web service.

Angular applications are typically written in TypeScript. I introduce TypeScript and explain how it works and why it is useful. TypeScript is a superscript of JavaScript, but one of its main advantages is that it lets you write code using the latest JavaScript language specification with features that are not yet supported in all of the browsers that can run Angular applications. One of the packages that `angular-cli` added to the project in the

previous section was the TypeScript compiler, which is set up to generate browser-friendly JavaScript files automatically when a change to a TypeScript file is detected.

To create a data model for the application, I added a file called `model.ts` to the `todo/src/app` folder (TypeScript files have the `.ts` extension) and added the code shown in Listing 2-5.

Listing 2-5. The Contents of the model.ts File in the todo/src/app Folder

```
var model = {  
  user: "Adam",  
  items: [{ action: "Buy Flowers", done: false },  
    { action: "Get Shoes", done: false },  
    { action: "Collect Tickets", done: true },  
    { action: "Call Joe", done: false }]  
};
```

One of the most important features of TypeScript is that you can just write “normal” JavaScript code as though you were targeting the browser directly. In the Listing, I used the JavaScript object literal syntax to assign a value to a global variable called `model`. The data model object has a `user` property that provides the name of the application’s user and an `items` property, which is set to an array of objects with `action` and `done` properties, each of which represents a task in the to-do list.

This is the most important aspect of using TypeScript: you don’t have to use the features it provides, and you can write entire Angular applications using just the JavaScript features that are supported by all browsers, like the code in Listing 2-5.

But part of the value of TypeScript is that it converts code that uses the latest JavaScript language features into code that will run anywhere, even in browsers that don’t support those features. Listing 2-6 shows the data model rewritten to use JavaScript features that were added in the ECMAScript 6 standard (known as ES6).

Listing 2-6. Using ES6 Features in the model.ts File

```
export class Model {  
  user;  
  items;  
  
  constructor() {  
    this.user = "Adam";  
    this.items = [new TodoItem("Buy Flowers", false),  
      new TodoItem("Get Shoes", false),  
      new TodoItem("Collect Tickets", false),  
      new TodoItem("Call Joe", false)]  
  }  
}
```

```

}

export class TodoItem {
  action;
  done;

  constructor(action, done) {
    this.action = action;
    this.done = done;
  }
}

```

This is still standard JavaScript code, but the **class** keyword was introduced in a later version of the language than most web application developers are familiar with because it is not supported by older browsers. The **class** keyword is used to define types that can be instantiated with the **new** keyword to create objects that have well-defined data and behavior.

Many of the features added in recent versions of the JavaScript language are syntactic sugar to help programmers avoid some of the most common JavaScript pitfalls, such as the unusual type system. The **class** keyword doesn't change the way that JavaScript handles types; it just makes it more familiar and easier to use for programmers experienced in other languages, such as C# or Java. I like the JavaScript type system, which is dynamic and expressive, but I find working with classes more predictable and less error-prone, and they simplify working with Angular, which has been designed around the latest JavaScript features.

The **export** keyword relates to JavaScript modules. When using modules, each TypeScript or JavaScript file is considered to be a self-contained unit of functionality, and the **export** keyword is used to identify data or types that you want to use elsewhere in the application. JavaScript modules are used to manage the dependencies that arise between files in a project and avoid having to manually manage a complex set of **script** elements in the HTML file.

Creating a Template

I need a way to display the data values in the model to the user. In Angular, this is done using a template, which is a fragment of HTML that contains instructions that are performed by Angular. The `angular-cli` setup for the project created a template file called `app.component.html` in the `todo/src/app` folder. I edited this file and added the markup shown in Listing 2-7 to replace the placeholder content. The name of the file follows the standard Angular naming conventions, which I explain later.

Listing 2-7. The Contents of the `app.component.html` File in the `todo/src/app` Folder

```
<h3 class="bg-primary p-a-1">{{getName()}}'s To Do List</h3>
```

I'll add more elements to this file shortly, but a single `h3` element is enough to get started. Including a data value in a template is done using double braces—`{{` and `}}`—and Angular evaluates whatever you put between the double braces to get the value to display.

The `{{` and `}}` characters are an example of a *data binding*, which means that they create a relationship between the template and a data value. Data bindings are an important Angular feature, and you will see more examples of them in this Part as I add features to the example application and as I describe them in detail. In this case, the data binding tells Angular to invoke a function called `getName` and use the result as the contents of the `h3` element. The `getName` function doesn't exist anywhere in the application at the moment, but I'll create it in the next section.

Creating a Component

An Angular component is responsible for managing a template and providing it with the data and logic it needs. If that seems like a broad statement, it is because components are the parts of an Angular application that do most of the heavy lifting. As a consequence, they can be used for all sorts of tasks.

At the moment, I have a data model that contains a `user` property with the name to display, and I have a template that displays the name by invoking a `getName` property. What I need is a component to act as the bridge between them. The `angular-cli` setup created a placeholder component file called `app.component.ts` to the `todo/src/app` folder, which I edited to replace the original content with the code shown in Listing 2-8.

Listing 2-8. The Contents of the `app.component.ts` File in the `todo/src/app` Folder

```
import { Component } from "@angular/core";
```

```

import { Model } from "./model";

@Component({
  selector: "todo-app",
  templateUrl: "app.component.html"
})
export class AppComponent {
  model = new Model();

  getName() {
    return this.model.user;
  }
}

```

This is still JavaScript, but it relies on features that you may not have encountered before but that underpin Angular development. The code in the Listing can be broken into three main sections, as described in the following sections.

Understanding the Imports

The **import** keyword is the counterpart to the **export** keyword and is used to declare a dependency on the contents of a JavaScript module. The **import** keyword is used twice in Listing 2-8, as shown here:

```

...
import { Component } from "@angular/core";
import { Model } from "./model";
...

```

The first **import** statement is used in the Listing to load the **@angular/core** module, which contains the key Angular functionality, including support for components. When working with modules, the **import** statement specifies the types that are imported between curly braces. In this case, the **import** statement is used to load the **Component** type from the module. The **@angular/core** module contains many classes that have been packaged together so that the browser can load them all in a single JavaScript file.

The second **import** statement is used to load the **Model** class from a file in the project. The target for this kind of import starts with **./**, which indicates that the module is defined relative to the current file.

Notice that neither **import** statement includes a file extension. This is because the relationship between the target of an import statement and the file that is loaded by the browser is managed by a module loader, which I configure in the “Putting the Application Together” section.

Understanding the Decorator

The oddest-looking part of the code in the Listing is this:

```
...
@Component({
  selector: "todo-app",
  templateUrl: "app.component.html"
})
...
```

This is an example of a *decorator*, which provides metadata about a class. This is the `@Component` decorator, and, as its name suggests, it tells Angular that this is a component. The decorator provides configuration information through its properties, which in the case of `@Component` includes properties called `selector` and `templateUrl`.

The `selector` property specifies a CSS selector that matches the HTML element to which the component will be applied: in this case, I have specified the `todo-app` element, which I added to the `index.html` file in Listing 2-4. When an Angular application starts, Angular scans the HTML in the current document and looks for elements that correspond to components. It will find the `todo-app` element and know that it should be placed under the control of this component.

The `templateUrl` property is to specify the component's template, which is the `app.component.html` file for this component. I describe the other properties that can be used with the `@Component` decorator and the other decorators that Angular supports.

Understanding the Class

The final part of the Listing defines a class that Angular can instantiate to create the component.

```
...
export class AppComponent {
  model = new Model();

  getName() {
    return this.model.user;
  }
}
...
```

These statements define a class called `AppComponent` that has a `model` property and a `getName` function, which provide the functionality required to support the data binding in the template from Listing 2-7.

When a new instance of the `AppComponent` class is created, the `model` property will be set to a new instance of the `Model` class defined in Listing 2-6. The `getName` function returns the value of the user `property` defined by the `Model` object.

Putting the Application Together

I have the three key pieces of functionality required to build a simple Angular application: a model, a template, and a component. When you saved the change to the `app.component.ts` file, there was enough functionality in place to bring the three pieces together and display the output shown in Figure 2-4.

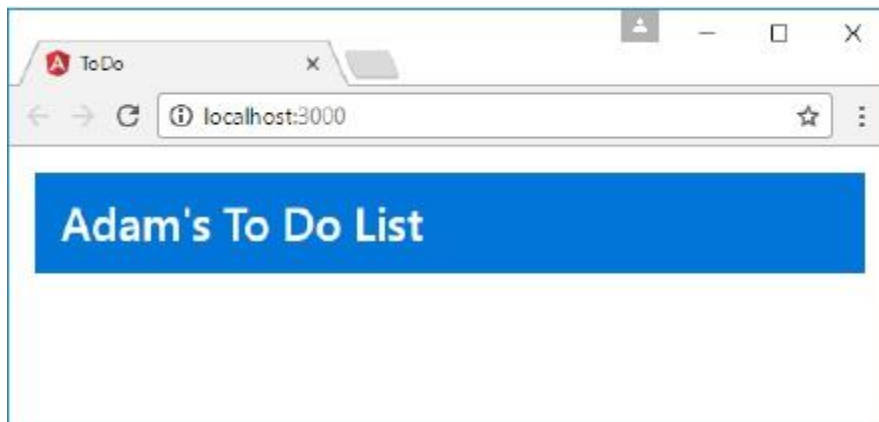


Figure 2-4. Simple Angular functionality in the example application

One advantage of using `angular-cli` to create a project is that you don't have to worry about creating the basic files required by an Angular application. The drawback is that skipping over these files means that you will miss out on some important details that are worth exploring.

Angular applications require a *module*. Through an unfortunate naming choice, there are two types of module used in Angular development. A JavaScript module is a file that contains JavaScript functionality that is used through the `import` keyword. The other type of module is an Angular module, which is used to describe an application or a group of related features. And just to complicate matters, every application has a *root module*, which is the Angular module that provides Angular with the information that it needs to start the application.

When `angular-cli` set up the project, it created a file called `app.module.ts`, which is the conventional file name for the root module, in the `todo/src/app` folder and added the code shown in Listing 2-9.

Listing 2-9. The Contents of the app.module.ts File in the todo/src/app Folder

```
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 { }
```

The purpose of the Angular module is to provide configuration information through the properties defined by the `@NgModule` decorator. The decorator's `imports` property tells Angular that the application depends on features required to run an application in the browser and that the `declarations` and `bootstrap` properties tell Angular about the components in the application and which one should be used to start the application (there is only one component in this simple example application, which is why it is the only value for both properties).

The example application depends on the Angular features for working with form elements, which are defined in the Angular `@angular/forms` module. To enable these features, make the changes shown in Listing 2-10 to the `app.module.ts` file.

Listing 2-10. Enabling Forms Supports in the app.module.ts File in the todo/src/app Folder

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from "@angular/forms";
import { AppComponent } from './app.component';

@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule, FormsModule],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Angular applications also need a bootstrap file, which contains the code required to start the application and load the Angular module. The bootstrap file is called `main.ts` and it is created in the `todo/src` folder with the code shown in Listing 2-11.

Listing 2-11. The Contents of the main.ts File in the todo/src Folder

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';
import { environment } from './environments/environment';

if (environment.production) {
  enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.log(err));
```

Although this Ref. focuses on applications that run in a web browser, Angular is intended to work in a range of environments. The code statements in the bootstrap file select the platform that will be used and load the root module, which is the entry point into the application.

Calling the `platformBrowserDynamic().bootstrapModule` method is for browser-based applications, which is what I focus on in this Ref.. If you are working on different platforms, such as the Ionic mobile development framework, then you will have to use a different bootstrap method specific to the platform you are working with. The developers of each platform that supports Angular provide details of their platform-specific bootstrap method.

The browser executed the code in the bootstrap file, which fired up Angular, which in turn processed the HTML document and discovered the `todo-app` element. The `selector` property used to define the component matches the `todo-app` element, which allowed Angular to remove the placeholder content and replace it with the component's template, which was loaded automatically from the `app.component.html` file. The template was parsed; the `{{` and `}}` data binding was discovered, and the expression it contains was evaluated, calling the `getName` and displaying the result shown in the figure. It may not be that impressive, but it is a good start, and it provides a foundation on which to add more features.

In any Angular project, there is a period where you have to define the main parts of the application and plumb them together. During this period, it can feel like you are doing a lot of work for little return. This period of initial investment will ultimately pay off, I promise. You will

see a larger example of this in Part 7 when I start to build a more complex and realistic Angular application; there is a lot of initial setup and configuration required, but then the features start to quickly snap into place.

Adding Features to the Example Application

Now that the basic structure of the application is in place, I can add the remaining features that I mocked up with static HTML at the start of the Part. In the sections that follow, I add the table containing the list of to-do items and the `input` element and button for creating new items.

Adding the To-Do Table

Angular templates can do more than just display simple data values. I describe the full range of template features in Part 2, but for the example application, I am going to use the feature that allows a set of HTML elements to be added to the DOM for each object in an array. The array, in this case, is the set of to-do items in the data model. To begin, Listing 2-12 adds a method to the component that provides the template with the array of to-do items.

Listing 2-12. Adding a Method in the `app.component.ts` File

```
import { Component } from "@angular/core";
import { Model } from "../model";

@Component({
  selector: "todo-app",
  templateUrl: "app.component.html"
})
export class AppComponent {
  model = new Model();

  getName() {
    return this.model.user;
  }

  getTodoItems() {
    return this.model.items;
  }
}
```

The `getTodoItems` method returns the value of the `items` property from the `Model` object. Listing 2-13 updates the component's template to take advantage of the new method.

Listing 2-13. Displaying the To-Do Items in the `app.component.html` File

```
<h3 class="bg-primary p-a-1">{{getName()}}'s To Do List</h3>

<table class="table table-striped table-bordered">
  <thead>
    <tr><th></th><th>Description</th><th>Done</th></tr>
  </thead>
  <tbody>
    <tr *ngFor="let item of getTodoItems(); let i = index">
      <td>{{ i + 1 }}</td>
      <td>{{ item.action }}</td>
      <td [ngSwitch]="item.done">
        <span *ngSwitchCase="true">Yes</span>
        <span *ngSwitchDefault>No</span>
      </td>
    </tr>
  </tbody>
</table>
```

The additions to the template rely on several different Angular features. The first is the ***ngFor** expression, which is used to repeat a region of content for each item in an array. This is an example of a directive, which I describe in Parts 12–16 (directives are a big part of Angular development). The ***ngFor** expression is applied to an attribute of an element, like this:

```
...
<tr *ngFor="let item of getTodoItems(); let i = index">
...
```

This expression tells Angular to treat the **tr** element to which it has been applied as a template that should be repeated for every object returned by the component's **getTodoItems** method. The **let item** part of the expression specifies that each object should be assigned to a variable called **item** so that it can be referred to within the template.

The **ngFor** expression also keeps track of the index of the current object in the array that is being processed, and this is assigned to a second variable called **i**:

```
...
<tr *ngFor="let item of getTodoItems(); let i = index">
...
```

The result is that the **tr** element and its contents will be duplicated and inserted into the HTML document for each object returned by the **getTodoItems** method; for each iteration, the current to-do object can be accessed through the variable called **item**, and the position of the object in the array can be accessed through the variable called **i**.

Within the `tr` template, there are two data bindings, which can be recognized by the `{{` and `}}` characters, as follows:

```
...
<td>{{ i + 1 }}</td>
<td>{{ item.action }}</td>
...
```

These bindings refer to the variables that are created by the `*ngFor` expression. Bindings are not just used to refer to property and method names; they can also be used to perform simple operations. You can see an example of this in the first binding, where I sum the `i` variable and 1.

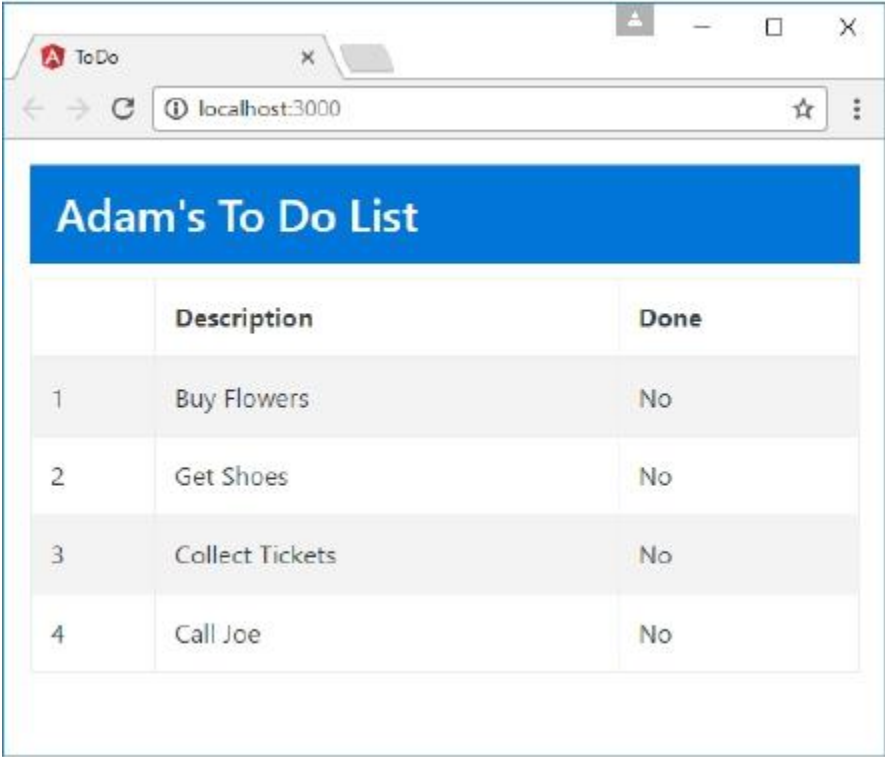
For simple transformations, you can embed your JavaScript expressions directly in bindings like this, but for more complex operations, Angular has a feature called pipes, which I describe in Part 18.

The remaining template expressions in the `tr` template demonstrate how content can be generated selectively.

```
...
<td [ngSwitch]="item.done">
  <span *ngSwitchCase="true">Yes</span>
  <span *ngSwitchDefault>No</span>
</td>
...
```

The `[ngSwitch]` expression is a conditional statement that is used to insert different sets of elements into the document based on a specified value, which is the `item.done` property in this case. Nested within the `td` element are two `span` elements that have been annotated with `*ngSwitchCase` and `*ngSwitchDefault` and that are equivalent to the `case` and `default` keywords of a regular JavaScript `switch` block. I describe `ngSwitch` in detail in Part 13 (and what the square brackets mean in Part 12), but the result is that the first `span` element is added to the document when the value of the `item.done` property is `true` and the second `span` element is added to the document when `item.done` is `false`. The result is that the `true/false`

value of the `item.done` property is transformed into `span` elements containing either `Yes` or `No`. When you save the changes to the template, the browser will reload, and the table of to-do items will be displayed, as shown in Figure 2-5.



The screenshot shows a web browser window with the title 'ToDo' and the address bar displaying 'localhost:3000'. The main content area has a blue header with the text 'Adam's To Do List'. Below the header is a table with three columns: an index column, a 'Description' column, and a 'Done' column. The table contains four rows of data, each representing a to-do item. The 'Done' column for all items shows 'No'.

	Description	Done
1	Buy Flowers	No
2	Get Shoes	No
3	Collect Tickets	No
4	Call Joe	No

Figure 2-5. Displaying the table of to-do items

If you use the browser's F12 developer tools, you will be able to see the HTML content that the template has generated. (You can't do this looking at the page source, which just shows the HTML sent by the server and not the changes made by Angular using the DOM API.)

You can see how each to-do object in the model has produced a row in the table that is populated with the `local` item and `i` variables and how the switch expression shows `Yes` or `No` to indicate whether the task has been completed.

```
...
<tr>
  <td>2</td>
  <td>Get Shoes</td>
  <td><span>No</span></td>
```

```

</tr>
<tr>
  <td>3</td>
  <td>Collect Tickets</td>
  <td><span>Yes</span></td>
</tr>
...

```

Creating a Two-Way Data Binding

At the moment, the template contains only one-way data bindings, which means they are used to display a data value but do nothing to change it. Angular also supports two-way data bindings, which can be used to display a data value and update it, too. Two-way bindings are used with HTML form elements, and Listing 2-14 adds a checkbox `input` element to the template that will let users mark a to-do item as complete.

Listing 2-14. Adding a Two-Way Binding in the `app.component.html` File

```

<h3 class="bg-primary p-a-1">{{getName()}}'s To Do List</h3>

<table class="table table-striped table-bordered">
  <thead>
    <tr><th></th><th>Description</th><th>Done</th></tr>
  </thead>
  <tbody>
    <tr *ngFor="let item of getTodoItems(); let i = index">
      <td>{{i + 1}}</td>
      <td>{{item.action}}</td>
      <td><input type="checkbox" [(ngModel)]="item.done" /></td>
      <td [ngSwitch]="item.done">
        <span *ngSwitchCase="true">Yes</span>
        <span *ngSwitchDefault>No</span>
      </td>
    </tr>
  </tbody>
</table>

```

The `ngModel` template expression creates a two-way binding between a data value (the `item.done` property in this case) and a `form` element. When you save the changes to the template, you will see a new column that contains checkboxes appear in the table. The initial value of the checkbox is set using the `item.done` property, just like a regular one-way binding, but when the user toggles the checkbox, Angular responds by updating the specified model property.

To demonstrate how this works, I have left the column that contains the **Yes/No** display of the **done** property value generated using the **ngSwitch** expression in the template. When you toggle a checkbox, the corresponding **Yes/No** value changes as well, as illustrated in Figure 2-6.

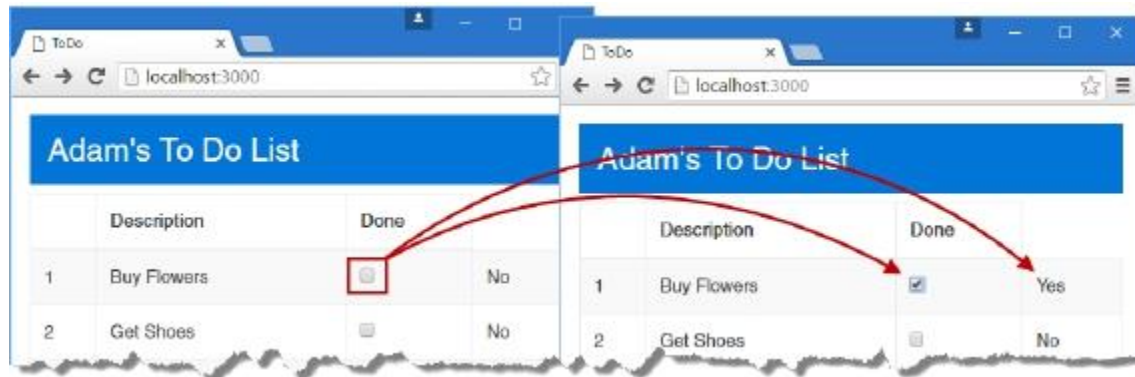


Figure 2-6. Changing a model value using a two-way data binding

This reveals an important Angular feature: the data model is live. This means that data bindings—even one-way data bindings—are updated when the data model is changed. This simplifies web application development because it means you don't have to worry about ensuring that you display updates when the application state changes.

Filtering To-Do Items

The checkboxes allow the data model to be updated, and the next step is to remove to-do items once they have been marked as done. Listing 2-15 changes the component's **getTodoItems** method so that it filters out any items that have been completed.

Listing 2-15. Filtering To-Do Items in the *app.component.ts* File

```
import { Component } from "@angular/core";
import { Model } from "../model";

@Component({
  selector: "todo-app",
  templateUrl: "app.component.html"
})
export class AppComponent {
  model = new Model();

  getName() {
```

```

        return this.model.user;
    }

    getTodoItems() {
        return this.model.items.filter(item => !item.done);
    }
}

```

This is an example of a *lambda function*, also known as a *fat arrow function*, which is a more concise way of expressing a standard JavaScript function. The arrow in the lambda expressions is read as “goes to” such as “**item** goes to not **item.done**.” Lambda expressions are a recent addition to the JavaScript language specification, and they provide an alternative to the conventional way of using functions as arguments like this:

```

...
return this.model.items.filter(function (item) { return !item.done });
...

```

Whichever way you choose to define the expression passed to the **filter** method, the result is that only incomplete to-do items are displayed. Since the data model is live and changes are reflected in data bindings immediately, checking the checkbox for an item removes it from view, as shown in Figure 2-7.

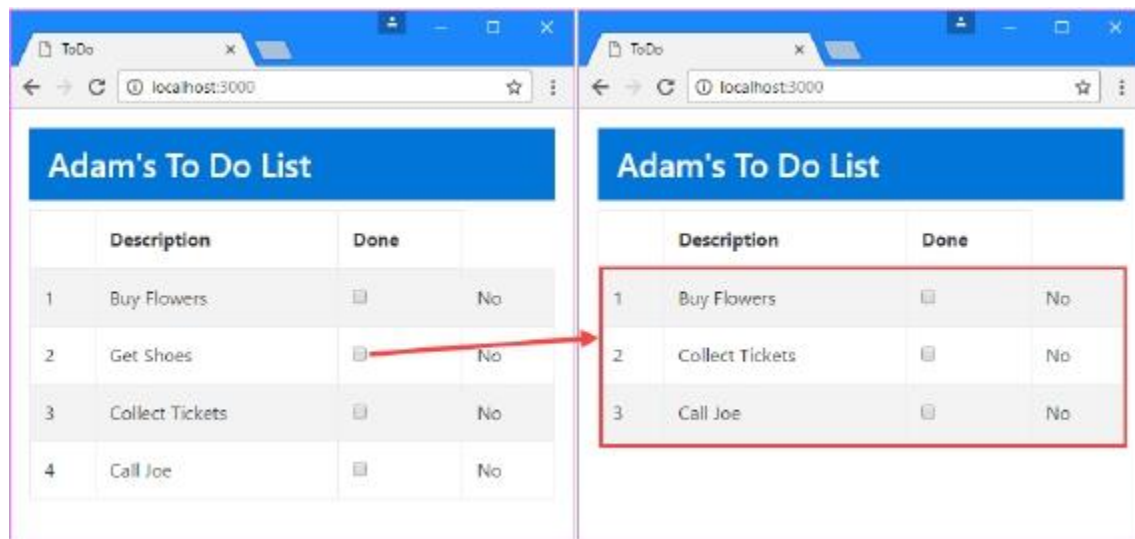


Figure 2-7. Filtering the to-do items

Adding To-Do Items

The next step is to build on the basic functionality to allow the user to create new to-do items and store them in the data model. Listing 2-16 adds new elements to the component's template.

Listing 2-16. Adding Elements in the `app.component.html` File

```
<h3 class="bg-primary p-a-1">{{getName()}}'s To Do List</h3>
<div class="m-t-1 m-b-1">
  <input class="form-control" #todoText />
  <button class="btn btn-primary m-t-1" (click)="addItem(todoText.value)">
    Add
  </button>
</div>
<table class="table table-striped table-bordered">
  <thead>
    <tr><th></th><th>Description</th><th>Done</th></tr>
  </thead>
  <tbody>
    <tr *ngFor="let item of getTodoItems(); let i = index">
      <td>{{i + 1}}</td>
      <td>{{item.action}}</td>
      <td><input type="checkbox" [(ngModel)]="item.done" /></td>
      <td [ngSwitch]="item.done">
        <span *ngSwitchCase="true">Yes</span>
        <span *ngSwitchDefault>No</span>
      </td>
    </tr>
  </tbody>
</table>
```

The `input` element has an attribute whose name starts with the `#` character, which is used to define a variable to refer to the element in the template's data bindings. The variable is called `todoText`, and it is used by the binding that has been applied to the `button` element.

```
...
<button class="btn btn-primary m-t-1" (click)="addItem(todoText.value)">
...
```

This is an example of an event binding, and it tells Angular to invoke a component method called `addItem`, using the `value` property of the `input` element as the method argument. Listing 2-17 implements the `addItem` method in the component.

Don't worry about telling the bindings apart at the moment. I explain the different types of binding that Angular supports in Part 2 and the meaning of the different types of brackets or parentheses that each requires. They are not as complicated as they first appear, especially once you have seen how they fit into the rest of the Angular framework.

Listing 2-17. Adding a Method in the `app.component.ts` File

```
import { Component } from "@angular/core";
import { Model, TodoItem } from "../model";

@Component({
  selector: "todo-app",
  templateUrl: "app.component.html"
})
export class AppComponent {
  model = new Model();

  getName() {
    return this.model.user;
  }

  getTodoItems() {
    return this.model.items.filter(item => !item.done);
  }

  addItem(newItem) {
    if (newItem !== "") {
      this.model.items.push(new TodoItem(newItem, false));
    }
  }
}
```

The **import** keyword can be used to import multiple classes from a module, and one of the **import** statements in the Listing has been updated so that the **TodoItem** class can be used in the component. Within the component class, the **addItem** method receives the text sent by the event binding in the template and uses it to create a new **TodoItem** object and add it to the data model. The result of these changes is that you can create new to-do items by entering text in the **input** element and clicking the Add button, as shown in Figure 2-8.

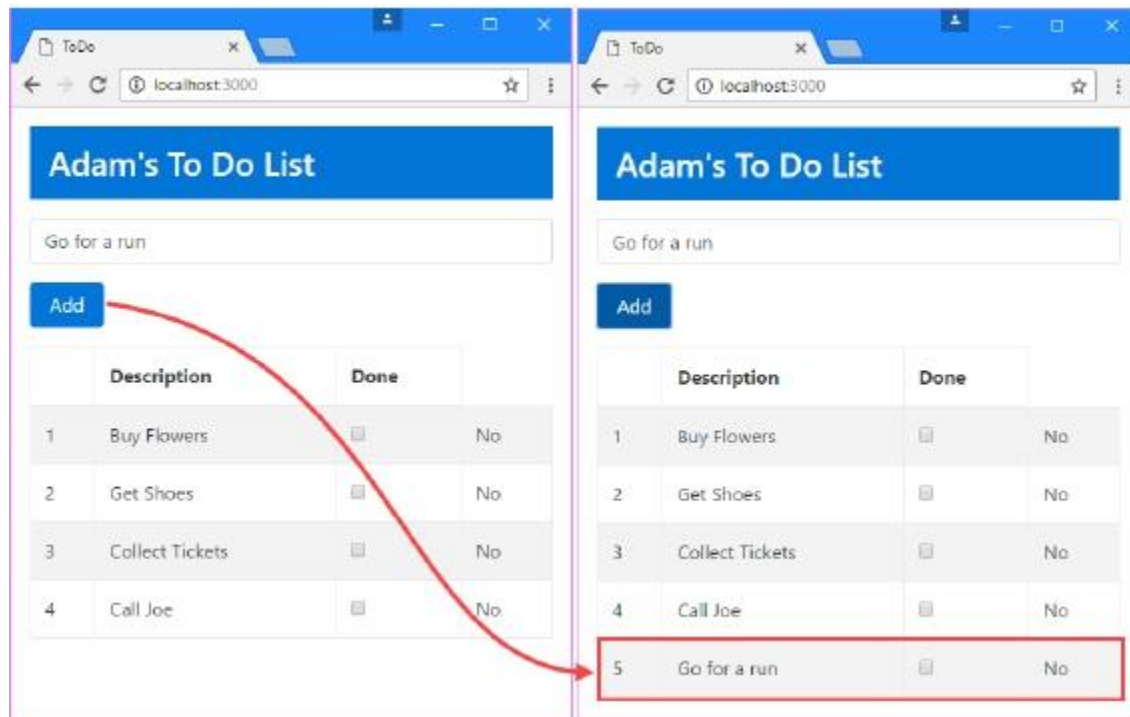


Figure 2-8. Creating a to-do item

In this Part, I showed you how to create your first simple Angular app, moving from an HTML mock-up of the application to a dynamic app that lets the user create new to-do items and mark existing items as complete.

Don't worry if not everything in this Part makes sense. What's important to understand at this stage is the general shape of an Angular application, which is built around a data model, components, and templates. If you keep these three key building blocks in mind, then you will have a context for everything that follows. In the next Part, I put Angular in context by Building an E-Commerce Application.