



Community Experience Distilled

Learning Web Development with Bootstrap and Angular

Second Edition

**It takes two to tango: Combine the power
of Bootstrap 4 and Angular 2 to build
cutting-edge web apps**

Sergey Akopkokhyants

[PACKT] open source*
PUBLISHING community experience distilled

Table of Contents

Chapter 1: Saying Hello!	1
Development Environment Setup	1
Defining Shell	1
Installing Node.js	2
Setting up Node Package Manager	3
Installing Git	4
Code Editor	4
The TypeScript Crash Course	4
Types	5
Arrow Function	6
Block Scope Variables	7
Template Literals	8
The for-of loop	9
Default Value, Optional and Rest parameters	10
Interfaces	11
Classes	11
Modules	14
Generics	15
What are Promises?	16
Event Loop	16
Angular 2 Concepts	19
Building blocks of Angular 2	20
Module	20
The Metadata	20
The Directive	20
The Attribute Directives	21
The Structural Directives	21
The Component	21
The Template	21
The Data Binding	21
The Service	22
The Dependency Injection	22
SystemJS loader and JSPM package manager	22
SystemJS Loader	22

JSPM package manager	23
Writing your first application	24
TypeScript Compile Configuration	25
Task Automation and Dependency Resolution	26
Creating and bootstrapping an Angular component	27
Compile and Run	30
Integrating Bootstrap 4	32
Summary	36
Chapter 2: Working with Bootstrap Components	37
<hr/>	
The Bootstrap 4	37
Introduction to Sass	38
Setup of Ruby	39
Setup of Sass	39
Sass Crash Course	40
Variables	40
Mathematical Expressions	41
Functions	42
Nesting	42
Imports	43
Extends	43
Placeholders	43
Mixins	44
Function Directives	45
The Example Project	46
The Scenario	46
Gathering Customer Requirements	47
Preparing the Use-Cases	47
The Welcome Page	48
The Products Page	49
The Product Page	50
The Cart Page	51
The Checkout Page	52
Designing layouts with grids and containers	53
Using Images	57
Using Cards	58
Using Buttons	59
The Button General Styles	59
The Button with Outline Styles	60
The Sizes of Button	61
The Button with Block Level Styles	61

The Button with Active Style	62
The Button with Inactive State	62
The Radio Buttons and Checkboxes	62
Navs	63
The Base Nav	63
Inline Navigation	64
Tabs	65
Pills	66
Stacked Pills	66
Navigation with Dropdowns	67
Navbars	68
The Content	68
The Colors	69
Containers	70
Responsive Navbar	72
The Navbar Content Alignment	73
Summary	74
Chapter 3: Advanced Bootstrap Components and Customization	75
How to Capture Customers Attention?	75
Display content with Jumbotron	75
Typography	76
Headings	76
Sub-Headings	77
Display Headings	78
Lead	79
Inline Text Elements	80
Abbreviations	80
Blockquotes	81
Address	82
Display content with Carousel	82
Carousel Container	83
Carousel Inner	83
Carousel Item	83
Carousel Indicators	83
Carousel Controls	84
Products Page Layout	85
Quick Shop Component	86
Input Group	86
Text Addons	87
Sizing	88

Checkboxes and Radio Options Addons	88
Button Addons	89
Dropdown Menu Addons	89
Segmented Buttons	90
Categories Component	92
List Group	92
List with Labels	92
Linked List Group	93
Button List Group	94
Contextual Classes	95
Custom Content	96
Grid of Products	98
Nested Rows	98
Product Component	99
Image	100
Responsive Images	100
Image Shapes	101
Image Alignment	101
Label	102
Button Group	103
Sizing	104
Button Toolbar	105
Nesting Dropdown	105
Vertical Button Group	106
Dropdown Menu	107
Dropdown Container	107
Dropdown Trigger	107
Dropdown Menu with Items	107
Menu Alignment	108
Menu Headers and Dividers	109
Menu Divider	110
Disabled Menu Items	111
Table	111
Inverse Table	112
Striped Rows	113
Bordered Table	113
Hoverable Rows	114
Table Head Options	114
Small Table	115
Contextual Classes	115
Responsive Tables	116
Reflow Tables	116

Shopping Cart Component	117
Summary	119
Chapter 4: Creating the Template	120
Dive deeper in Angular 2	120
Welcome Page Analysis	121
Single Responsibility Principle	121
Naming conventions	122
Barrels	122
Application Structure	122
Folders-by-Feature Structure	122
Shared Folder	123
Navigation Component	123
Decorators	124
Tree of Components	125
NavItem Object	126
Template Expressions	126
Expression Context	127
Template Reference Variable	127
Expression Guidelines	128
Expression Operators	128
The Elvis Operator	128
The Pipe Operator	129
The Custom Pipes	131
Template Statements	131
Statement Context	132
Statement Guidelines	132
Data Binding	132
HTML Attributes vs. DOM Properties	133
Interpolation	133
Property Binding	134
Attribute Binding	134
Class Binding	135
Style Binding	135
Event Binding	136
Custom Events	136
Two-way Data Binding	138
Built-in Directives	139
NgClass	139
NgStyle	139

NgIf	140
NgSwitch	141
NgFor	142
Structural Directives	143
Custom Structural Directive	144
Category Product Component	146
Summary	150
Chapter 5: Routing	151
<hr/>	
The Modern Web Applications	151
Routing	152
Routing Path	152
Installing the Router	153
The Base URL	153
The Component Router	153
The Router Configuration	154
Creating Basic Routes	154
Query Parameters	154
Router Parameters	155
Route vs. Query Parameters	155
Register Routing in bootstrap	155
Redirecting Routes	156
Router Outlet	156
Welcome View	157
Category Card View	158
Products View	159
Quick Shop Component	160
List of Categories Component	161
Router Links	162
Product Card	162
Products Grid Component	163
Card Groups	163
Card Columns	164
Card Desks	165
Router Change Events	167
Routing Strategies	170
Summary	172
Chapter 6: Dependency Injection	173
<hr/>	
What is Dependency Injection	173

The Real Life Example	173
The Dependency Injection	175
Constructor Injection	176
Another Injection methods	176
Components vs. Services	176
The Injector	177
Injectable Decorator	177
Inject Decorator	178
Optional Decorator	178
Configuring the Injector	179
Class Providers	180
Aliased class providers	180
Value Providers	181
Multiple Values	182
Factory Providers	183
The Hierarchy of Injectors	184
Category Service	184
Injector Provider for Category Service	186
Product Service	187
Injector Provider for Product Service	188
The Shopping Cart	189
The Cart Model and Cart Item	190
The Cart Service	190
The Cart Menu Component	192
Update the Cart via Service	195
Summary	196

1

Saying Hello!

Let's follow several steps establish a development environment for the simplest application possible, to show you how easy it is to get up and running the web application with Angular 2 and Bootstrap 4. At the end of the chapter, you will have a solid understanding:

- How to setup your development environment
- How Typescript can change your development life
- Core concepts of Angular and Bootstrap
- How create a simple Angular component with Bootstrap
- How display some data through it

Development Environment Setup

It's time to set up your development environment. This process is one of the most overlooked and often frustrating parts of learning to program because developers don't want to think about it. The developers must know nuances how to install and configure many different programs before they start real development. Everyone's computers are different as a result; the same setup may not work on your computer. We will expose and eliminate all of these problems by defining the various pieces of environment you need to setup.

Defining Shell

The **Shell** is a required part of your software development environment. We will use the shell to install software, run commands to build and start the web server to bring the life to your web project. If your computer has installed Linux operating system then you will use the shell called **Terminal**. There are many Linux-based distributions out there that use

diverse desktop environments, but most of them use the equivalent keyboard shortcut to open the Terminal.



Use keyboard shortcut *Ctrl + Alt + T* to open Terminal in Linux.

If you have a Mac computer with installed OS X, then you will use the Terminal shell as well.



Use keyboard shortcut *Command + Space* to open the *Spotlight*, type Terminal to search and run.

If you have a computer with installed Windows operation system, you can use the standard **Command Prompt**, but we can do better. In a minute later I will show you how can you install the Git on your computer, and you will have Git Bash free.



You can open a Terminal with Git Bash shell program on Windows.

I will use the shell bash for all exercises in this book whenever I need to work in the Terminal.

Installing Node.js

The **Node.js** is technology we will use as a cross-platform runtime environment for running server-side Web applications. It is a combination of native, platform independent runtime based on Google's V8 JavaScript engine and a huge number of modules written in JavaScript. Node.js ships with different connectors and libraries help you use HTTP, TLS, compression, file system access, raw TCP and UDP, and more. You as a developer can write own modules on JavaScript and run them inside Node.js engine. The Node.js runtime makes ease build a network, event-driven application servers.



The terms package and library are synonymous in JavaScript so that we will use them interchangeably.

Node.js is utilizing **JSON** (JavaScript Object Notation) format widely in data exchange between server and client sides because it readily expressed in several parse diagrams, notably without complexities of XML, SOAP, and other data exchange formats.

You can use Node.js for the development of the service-oriented applications, doing something different than web servers. One of the most popular service-oriented application is NPM (Node Package Manager) we will use to manage library dependencies, deployment systems, and underlies the many **PaaS** (platform-as-a-service) providers for Node.js.

If you do not have Node.js installed on your computer, you shall download the pre-build installer from <https://nodejs.org/en/download>. You can start to use the Node.js immediately after installation. Open the Terminal and type:

```
node --version
```

The Node.js must respond with version number of installed runtime:

```
v4.4.3
```

Setting up Node Package Manager

The Node Package Manager (or **NPM**) is a package manager for JavaScript. You can use it to find, share, and reuse packages of code from many developers across the world. The number of packages dramatically grows every day and now is more than 250K. NPM is a Node.js package manager and utilizes it to run itself. NPM is included in setup bundle of Node.js and available just after installation. Open the Terminal and type:

```
npm --version
```

The NPM must answer on your command with version number:

```
2.15.1
```

The following command gives us information about Node.js and NPM install:

```
npm config list
```

There are two ways to install NPM packages: locally or globally. In cases when you would like to use the package as a tool better install it globally:

```
npm install --global <package_name>
```

If you need to find the folder with globally installed packages you can use the next command:

```
npm config get prefix
```

Installation global packages are important, but best avoid if not needed. Mostly you will install packages locally.

```
npm install <package_name>
```

You may find locally installed packages in a `node_modules` folder of your project.

Installing Git

You missed a lot if you are not familiar with **Git**. Git is a distributed version control system and each Git working directory is a full-fledged repository. It keeps the complete history of changes and has full version tracking capabilities. Each repository is entirely independent of network access or a central server.

You can install Git on your computer via a set of pre-build installers available on official website <https://git-scm.com/downloads>. After installation, you can open the Terminal and type

```
git --version
```

Git must respond with version number

```
git version 2.8.1.windows.1
```

As I said for developers who use computers with installed Windows operation system now, you have Git Bash free on your system.

Code Editor

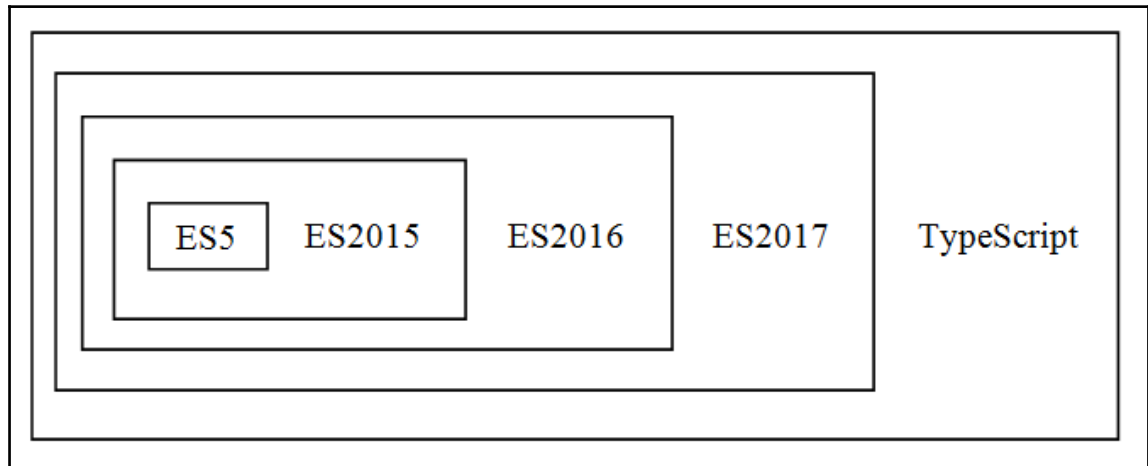
You can imagine how many programs for code editing exists but we will talk today only about free, open source and runs everywhere Visual Studio Code from Microsoft. You can use any program you prefer for development, but I use only Visual Studio Code in our future exercises, so please install it from <http://code.visualstudio.com/Download>.

The TypeScript Crash Course

The **TypeScript** is an open source programming language that is developed and maintained by Microsoft. Its initial public release was in October 2012 and presented by Anders Hejlsberg, the lead architect of C# and creator of Delphi and Turbo Pascal.

TypeScript is a typed superset of JavaScript that compiles to plain JavaScript. Any existing JavaScript is already TypeScript. It gives you type checking, explicit interfaces, and easier module exports. For now, it includes ES5, ES2015, ES2016 and in fact; it's a little like getting some of the tomorrow's ECMAScripts early so that we can play with some of those features today.

Here is relationship between ECMAScripts and TypeScript:



Writing code using TypeScript is relatively straightforward if you already have a background in JavaScript language. There are following features TypeScript brings into JavaScript ES5:

Types

TypeScript provides static type checking operation that allows caught early many bugs in the development cycle. TypeScript enables type checking at compile time via type annotations. Types in TypeScript always optional, so you can ignore them if you prefer regular dynamic typing of JavaScript. It supports `number`, `boolean`, and `string` type annotations for the primitive types and `any` for dynamically-typed structures. On the following example, I add type annotations to return and parameters of the function:

```
function add(first: number, second: number): number {  
    return first + second;  
}
```

In a moment of compilation, a TypeScript compiler can generate a declaration file contains

only signatures of the exported types. The resulting declaration file with the extension `.d.ts` along with a JavaScript library or module can be consumed later by a third-party developer. You can find a vast collection of declaration files for many popular JavaScript libraries on:

- The **DefinitelyTyped** (<https://github.com/DefinitelyTyped/DefinitelyTyped>)
- The **Typings Registry** (<https://github.com/typings/registry>)

Arrow Function

Functions in JavaScript are first class citizens, which mean they can be passed around like any other values:

```
var result = [1, 2, 3]
  .reduce(function (total, current) {
    return total + current;
  }, 0); // 6
```

The first parameter in `reduce` is anonymous function. Anonymous functions are very useful in many scenarios but too verbose. Typescript introduced new, less verbose syntax to define anonymous functions called **arrow function** syntax:

```
var result = [1, 2, 3]
  .reduce( (total, current) => {
    return total + current;
  }, 0); // 6
```

Or even less:

```
var result = [1, 2, 3]
  .reduce( (total, current) => total + current, 0); // 6
```

When defining parameters, you can even omit parentheses if the parameters are just a single identifier. So the regular `map` method of `Array`:

```
var result = [1, 2, 3].map(function (x) {
  return x * x
});
```

Could be much more concise:

```
var result = [1, 2, 3].map(x => x * x);
```

Both syntaxes `(x) => x * x` and `x => x * x` are allowed.

Another important feature of arrow function is that it doesn't shadow `this` and pick it up from the lexical scope. Let's assume we have a constructor function `Counter` which increments the value of internal variable `age` in timeout and prints it out:

```
function Counter() {
  this.age = 30;
  setTimeout(() => {
    this.age += 1;
    console.log(this.age);
  }, 100);
}
new Counter(); // 31
```

As result of usage of arrow function the `age` from the scope of `Counter` available inside callback function of `setTimeout`. The following variables are all lexical inside arrow functions:

- `arguments`
- `super`
- `this`
- `new.target`

Block Scope Variables

All variables in ES5 declared with `var` statement are function-scoped, and their scope belongs to enclosing functions. The result of the following code can confuse because returns `undefined`:

```
var x = 3;
function random(randomize) {
  if (randomize) {
    var x = Math.random(); // x initialised as reference on function
    return x;
  }
  return x; // x is not defined
}
random(false); // undefined
```

The `x` is an inner variable of `random` function and does not have relations to variable defined at the first line. The result of call the `random` function at the last line return `undefined`, because the JavaScript interprets the code in `random` function like that:

```
function random(randomize) {
```

```
    var x; // x is undefined
    if (randomize) {
        // x initialized as reference on function
        x = Math.random();
        return x;
    }
    return x; // x is not defined
}
```

This confusing code can be fixed in TypeScript with new block-scope variable declarations:

- The `let` is a block-scope version of `var`.
- The `const` is similar `let` but allows initialize variable only once.

The TypeScript compiler throws more errors with new block-scope variable declarations and prevents writing complicated and damaged code. Let's change `var` to `let` in the previous example:

```
let x = 3;
function random(randomize) {
    if (randomize) {
        let x = Math.random();
        return x;
    }
    return x;
}
random(false); // 3
```

And now our code works as expected.



I recommend use `const` and `let` and avoid `var` in your code.

Template Literals

If we need string interpolation, we usually combine the values of variables and string fragments such as:

```
let out: string = '(' + x + ', ' + y + ')';
```

With TypeScript we can do that better if use **string interpolation** via **template literals**:

```
let out: string = `(${x}, ${y})`;
```


If you need multiline string the template literals can help again:

```
let x = 1, y = 2;
let out: string = `
Coordinates
  x: ${x},
  y: ${y}`;
console.log(out);
```

The last line prints result as follow:

```
Coordinates
x: 1,
y: 2
```



I recommend use template literals as safer way of string interpolation.

The for-of loop

We usually use `for` statement or `forEach` method of `Array` to iterate over elements in JavaScript ES5:

```
let arr = [1, 2, 3];
// The for statement usage
for (let i = 0; i < arr.length; i++) {
  let element = arr[i];
  console.log(element);
}
// The usage of forEachmethod
arr.forEach(element => console.log(element));
```

Each of this methods has its benefit:

- We can interrupt the `for` statement via `break` or `continue`
- The `forEach` method less verbose

The TypeScript has `for-of` loop as combination of both of them:

```
const arr = [1, 2, 3];
for (const element of arr) {
  console.log(element);
}
```

The for-of loop supports break and continue and can use index and value of each array via new Array method entries:

```
const arr = [1, 2, 3];
for (const [index, element] of arr.entries()) {
  console.log(`${index}: ${element}`);
}
```

Default Value, Optional and Rest parameters

We quite often need to check the input parameters of function and assign the default values to them:

```
function square(x, y) {
  x = x || 0;
  y = y || 0;
  return x * y;
}
let result = square(4, 5); // Out 20
```

The TypeScript has syntax to handle default values of parameters to make previous function shorter and safer:

```
function square(x: number = 0, y: number = 0) {
  return x * y;
}
let result = square(4, 5); // Out 20
```



A default value of a parameter assigned only by the undefined value of it.

In JavaScript ES5 every parameter is optional, so omitted one is equal undefined. To make it strict, the TypeScript expects a question mark to the end of parameters we want to be optional. We can mark the last parameter of the square function as optional and call the function with one or two parameters:

```
function square(x: number = 0, y?: number) {
  if (y) {
    return x * y;
  } else {
    return x * x;
  }
}
```

```
let result = square(4); // Out 16
let result = square(4, 5); // Out 20
```



Any optional parameters must follow required parameters.

In some scenarios, we need to work with multiple parameters as a group, or you may not know how many parameters a function takes. The JavaScript ES5 provides `arguments` variable in the scope of function to work with them. In TypeScript, we can use a formal variable to keep the rest of parameters. The compiler builds an array of the arguments passed in with the name given after the ellipses so that we can use it in our function:

```
function print(name: number, ...restOfName: number[]) {
    return name + " " + restOfName.join(" ");
}
let name = print("Joseph", "Samuel", "Lucas");
// Out: Joseph Samuel Lucas
```

Interfaces

The TypeScript defines an interfaces via `interface` keyword only for one reason – to describe a type, the shape of data to helps us keep our code error-free. Let's define a type `Greetable`:

```
interface Greetable {
    greetings(message: string): void;
}
```

It has a member function called `greetings` that takes a string argument. Here is how we can use it as a type of parameter:

```
function hello(greeter: Greetable) {
    greeter.greetings('Hi there');
}
```

Classes

The JavaScript has a prototype-based, object-oriented programming model. We can instantiate objects using the object literal syntax or constructor function. The prototype-based inheritance implemented on prototype chains. If you come from an object-oriented approach, you may feel uncomfortable when you try to create classes and inheritance based

on prototypes. The TypeScript allows writing the code based on the object-oriented class-based approach. The compiler translates the class down to JavaScript that works across all major web browsers and platforms. Here is the class `Greeter`. It has a property called `greeting`, a constructor, and a method `greet`:

```
class Greeter {
  greeting: string;
  constructor(message: string) {
    this.greeting = message;
  }
  greet() {
    return "Hello, " + this.greeting;
  }
}
```

To refer to any member of the class we use prepend `this`. To create an instance of the class we using `new` keyword:

```
let greeter = new Greeter("world");
```

We can extend existing class to create new ones via inheritance:

```
class EmailGreeter extends Greeter {
  private email: string;
  constructor(emailAddr: string, message: string) {
    super(message);
    this.email = emailAddr;
  }
  mailto() {
    return "mailto:${this.email}?subject=${this.greet()}";
  }
}
```

In the class `EmailGreeter`, we demonstrate several features of the inheritance in TypeScript:

- We use `extends` to create a subclass
- We must call `super` in the first line of constructor to pass values into base class
- We call the `greet` method of the base class to create a subject for `mailto`.

Each member of the `EmailGreeter` class is implicitly public, but you can mark them explicitly if you want. The `email` member is `private`. The containing class does not have access to private members even via inheritance. Use `protected` modifier if you need members accessed by instances of derived classes.

If you look at constructors of `EmailGreeter`, we had to declare a private member `email`

and a constructor parameter `emailAddr`. Instead, we can use a parameter properties to let us create and initialize a member in one place:

```
class EmailGreeter extends Greeter {
  constructor(private email: string, message: string) {
    super(message);
  }
  mailto() {
    return "mailto:${this.email}?subject=${this.greet()}";
  }
}
```

You can use any modifier in parameter properties.



Use parameter properties to consolidate the declaration and assignment in one place.

The TypeScript supports getters and setters to organize intercepting access to members of an object. We can change original `Greeter` class with the following code:

```
class Greeter {
  private _greeting: string;
  get greeting(): string {
    return this._greeting;
  }
  set greeting(value: string) {
    this._greeting = value || "";
  }
  constructor(message: string) {
    this.greeting = message;
  }
  greet() {
    return "Hello, " + this.greeting;
  }
}
```

We check the `value` parameter inside the setter of `greeting` and modify it if necessary to empty string before assigning to the private member.

TypeScript supports the class members via the `static` modifier as well. Here is the class `Types` contains only static members:

```
class Types {
  static GENERIC: string = "";
  static EMAIL: string = "email";
}
```

```
}
```

We can access those values through prepending the name of the class:

```
console.log(Types.GENERIC);
```

TypeScript gives us supreme flexibility via abstract classes. We cannot create instances of them, but we can use them to organize base classes from which each distinct classes may be derived. We can convert the `Greeting` class into abstract with only one keyword:

```
abstract class BaseGreeter {
  private _greeting: string;
  get greeting(): string {
    return this._greeting;
  }
  set greeting(value: string) {
    this._greeting = value || "";
  }
  abstract greet();
}
```

The method `greet` is marked as `abstract`. It doesn't contain an implementation and must be implemented in derived classes.

Modules

When we are writing the code, we usually divide it into functions and the blocks inside those functions. The size of a program can increase very quickly, and individual functions start to blend into the background. We can make such a program more readable if split them into large units of an organization like modules. At the beginning of writing the program, you may not know how to structure it, and you can use structureless principles. When your code becomes stable you can put pieces of functionality into separate modules to make them easy to track, update, and share. We store modules of TypeScript in files, exactly one module per file and one file per module.

The JavaScript ES5 doesn't have built-in support for modules and we used AMD or CommonJS syntax to work with them. TypeScript supports a concept of modules.

How scope and module depend on each other? The global scope of JavaScript program doesn't have access to the scope of executing modules. It creates own scope per each execution module, so everything declared inside the module is not visible from outside. We need to explicitly export them to make visible and, import them to consume. The relationship between modules is defined at the file level regarding exports and imports. Any file defines a top-level `export` or `import` is considered a module. Here is a string-

validator.ts file contains exported declaration:

```
export interface StringValidator {  
  isAcceptable(s: string): boolean;  
}
```

I have created another file zip-validator.ts with several members, but export only one of them to hide another one from outside:

```
const numberRegexp = /^[0-9]+$/;  
export class ZipCodeValidator implements StringValidator {  
  isAcceptable(s: string) {  
    return s.length === 5 && numberRegexp.test(s);  
  }  
}
```

You can re-export declaration if your module extends other modules. Here is validators.ts contains module, wraps other validators modules and combines all their exports in one place:

```
export * from "./string-validator";  
export * from "./zip-validator";
```

Now we can import validators modules using one of the import forms. Here is a single export from a module:

```
import { StringValidator } from "./validators";  
let strValidator = new StringValidator();
```

To preventing a naming conflict we can rename imported declaration:

```
import { ZipCodeValidator as ZCV } from "./validators";  
let zipValidator = new ZCV();
```

At the end, we can import entire module into single variable, and use it to access to module exports:

```
import * as validator from "./validators";  
let strValidator = new validator.StringValidator();  
let zipValidator = new validator.ZipCodeValidator();
```

Generics

The authors of TypeScript put maximum effort to help us write reusable code. One of the tools to create code that can work with a variety of types rather than a single one is generics. Benefits of generics:

- Allows you to write code/use methods which are type-safe. An `Array<string>` is guaranteed to be an array of strings.
- The compiler can perform a compile-time check on code for type safety. Any try to assign the `number` into an array of strings causes an error.
- Faster than using any type to avoid casting into a required reference type.
- Allow you to write code which applicable to many types with the same underlying behavior.

Here is the class I have created to show you how useful can be generics:

```
class Box<T> {      private _value : T;      set value(val : T) {
  this._value = val;      }      get value() : T {      return this._value;
} }
```

This class keeps the single value of particular type. To set or return it we can use corresponding getter and setter methods:

```
var box1 = new Box<string>(); box1.setValue("Hello World");
console.log(box1.getValue()); var box2 = new Box<number>();
box2.setValue(1); console.log(box2.getValue());
var box3 = new Box<boolean>(); box3.setValue(true);
console.log(box3.getValue());
// Out: Hello World
// Out: 1
// Out: true
```

What are Promises?

The promise represents the final result of an asynchronous operation. But before start talking about it let's talk a bit about browser environment which executes your JavaScript code.

Event Loop

Each browser tab has an event loop, uses different tasks to coordinate events, user interaction, run scripts, rendering, networking, etc. It has a one or more queues keep an ordered list of tasks. Other processes are running around the **Event Loop** and communicate with it by adding tasks to its queue such as:

- The Timer waits after given period and then adds a task to the queue.

- We can call a `requestAnimationFrame` function to coordinate DOM updates.
- DOM elements can call event handlers
- The browser can request the parsing of HTML page
- JavaScript can load external program and performs computation on it

Many of items in the list above are JavaScript code. They usually small enough, but if we run any long-running computation it could block execution of other tasks, as a result – freezes the user interface. To avoid blocking the Event Loop we can:

- Use **Web Worker API** to execute a long-running computation in different process of browser
- Do not wait for the result of a long-running computation synchronously and allow the task to inform us about results via events or callbacks asynchronously.

Asynchronous results via events

The following code uses event-driven approach to convince us adds event listeners to execute small code snippets inside:

```
var request = new XMLHttpRequest();
request.open('GET', url);

request.onload = () => {
  if (req.status == 200) {
    processData(request.response);
  } else {
    console.log('ERROR', request.statusText);
  }
};

request.onerror = () => {
  console.log('Network Error');
};

request.send(); // Add request to task queue
```

The method `send` in the last line of code just adds another task to the queue. This approach useful if you receive results multiple times, but this code is quite verbose for a single result.

Asynchronous results via callbacks

To manage asynchronous results via callbacks, we need to pass a callback function as a

parameter into asynchronous function calls.

```
readFileFunctional('myfile.txt', { encoding: 'utf8' },
  (text) => { // success
    console.log(text);
  },
  (error) => { // failure
    // ...
  }
);
```

This approach is very easy to understand, but it has disadvantages:

- It is mixing up input and output parameters
- It is complicated to handle errors especially in the code combined many callbacks
- It is more complicated to return result from combined asynchronous functions

Asynchronous results via promises

As I mentioned earlier, the promise represents the final result of an asynchronous operation happened in the future. Promises have following advantages:

- You write cleaner code without callback parameters
- You do not adapt the code of the underlying architecture for delivery results
- Your code handle errors with easy

The promise may be in one of the following states:

- The pending state. The asynchronous operation hasn't completed yet.
- The resolved state. The asynchronous operation has completed and the promise has a value.
- The rejected state. The asynchronous operation failed and the promise has a reason indicates why it failed

The promise becomes immutable after resolving or rejecting.

Usually, you write the code to return promise from functions or methods:

```
function readFile(filename, encode){
  return new Promise((resolve, reject) => {
```

```
    fs.readFile(filename, encode, (error, result) => {
      if (error) {
        reject(error);
      } else {
        resolve(result);
      }
    });
  });
}
```

We use `new` keyword with function constructor to create the `Promise`. We give a factory function with two parameters into the constructor which does the actual work. The both parameters are callback functions. Once the operation has successfully completed the factory function calls the first callback with the result. If operation is fails it calls the second function with the reason.

The returned promise has several methods like `.then` and `.catch` to inform us about the result of execution so that we can act accordingly:

```
function readJSON(filename){
  return readFile(filename, 'utf8').then((result) => {
    console.log(result);
  }, (error) => {
    console.log(error);
  })
}
```

We can call another operation to quickly transform the result of original one:

```
function readJSON(filename){
  return readFile(filename, 'utf8').then((result) => {
    return JSON.parse(result);
  }, (error) => {
    console.log(error);
  })
}
```

Angular 2 Concepts

The **Angular 2** is a development platform for building the web, mobile, and desktop applications. It based on web standards to make web development simpler and efficient, and entirely different from the Angular JS 1.x. The architecture of Angular 2 builds on top of **Web Components** standard so that we can define custom HTML selectors and program behavior to them. The Angular team develops the Angular 2 to use in ECMAScript 2015,

TypeScript and Dart languages.

Building blocks of Angular 2

Any web application built on Angular 2 is composition of:

- HTML templates with Angular-specific markup
- Directives and Components managing the HTML templates
- Services contain application logic
- Especial bootstrap function helps to load and start the Angular Application

Module

The Angular 2 application is an assembly of many modules. The Angular 2 itself is a set of modules with names begin with `@angular` prefix, combined into libraries:

- The `@angular/core` is the primary Angular 2 library contains all core public APIs.
- The `@angular/common` is the library keeps API to reusable components, directives, and form building.
- The `@angular/router` is the library supports navigation.
- The `@angular/http` is the library helps work asynchronously via HTTP.

The Metadata

The Metadata is information we can attach to underlying definitions via TypeScript decorators to tell Angular how to modify them. Decorators are playing a significant role in Angular 2.

The Directive

The Directive is the fundamental building block of Angular 2 allows you to connect behavior to an element in the DOM. There are three kinds of directives:

- Attribute Directives
- Structural Directives
- Components

A Directive is a class with assigned `@Directive` decorator.

The Attribute Directives

The Attribute Directive usually changes the appearance or behavior of an element. We can change several styles, or use it to render text bold or italic by binding to property.

The Structural Directives

The Structural Directive changes the DOM layout by adding and removing other elements.

The Component

The Component is a Directive with a template. Every Component constitute of two parts:

- The class, where we define the application logic
- The view controlled by component and interacts with it through an API of properties and methods

A Component is a class with assigned `@Component` decorator.

The Template

The Component uses the **Template** to render the view. It is a regular HTML with custom defined selectors and Angular-specific markups.

The Data Binding

The Angular 2 supports **data binding** to update parts of template via properties or methods of a Component. The **binding markup** is part of data binding; we use it on the template to connect both sides.

The Service

The Angular 2 has no definition of a **Service**. Any value, function, and the feature can be a service, but usually, it is a class created for a distinct purpose with assigned `@Injectable` decorator.

The Dependency Injection

Dependency Injection is a design pattern helps configure objects by an external entity and resolve dependencies between them. It separates the creation of objects from their behavior in **Loosely Coupled** design.

SystemJS loader and JSPM package manager

We spoke about TypeScript modules, so it's time to talk about tools we can use for loading modules in our scripts.

SystemJS Loader

The **SystemJS** is universal dynamic module loader. It hosts the source code on GitHub on the following address <https://github.com/systemjs/systemjs>. It can load modules in the web browser and Node.js of the following formats:

- ECMAScript 2015 (ES6) or TypeScript
- AMD,
- CommonJS
- Global scripts

The SystemJS loads modules with exact circle reference, binding support, and assets through the module naming system such as CSS, JSON or images. Developers can easily extend the functionality of the loader via plugins.

We can add SystemJS loader to our future project:

- Via direct link to Content Delivery Network (CDN)
- To install via NPM manager

In both scenarios, we include reference to SystemJS library into our code and configure it via `config` method:

```
<!DOCTYPE html>
<html>
  <head>
    <script src="https://jspm.io/system.js"></script>
    <script type="text/javascript">
      System.config({
        transpiler: 'babel',
        packages: {
          './': {
            defaultExtension: false
          }
        }
      });
    </script>
    <script type="text/javascript">
      System.import('./app.js');
    </script>
  </head>
  <body>
    <div id="main"></div>
  </body>
</html>
```

We will speak about installation via NPM manager a bit later in this chapter.

JSPM package manager

The developers of the SystemJS follow the single responsibility principle and implement a loader for doing only one thing – load the modules. To get modules available in your project, we need to use the package manager. We spoke about NPM package manager at the beginning, so now we talk about the JSPM package manager sitting on the top of SystemJS. What it can do:

- It can download modules from any registry such as NPM and GitHub
- It can compile modules into simple, layered, and self-executing bundles with a single command

The JSPM package manager looks like an NPM package manager, but it puts the browser loader first. It helps you organize the seamless workflow for installing and using libraries in the browser with small effort.

Writing your first application

Now, when we have everything in place, it's time to create our first project, actually NPM module. Open the Terminal and create folder `hello-world`. I intentionally follow the NPM package name conventions:

- The package name length should be greater than zero and cannot exceed 214
- All the characters in the package name must be lowercase
- The package name can consist of hyphens
- The package name must contain any URL safe characters (since name ends up being part of a URL)
- The package name should not start with dot or underscore letters
- The package name should not contain any leading or trailing spaces
- The package name cannot be the same as a node.js/io.js core module nor a reserved/blacklisted name like `http`, `stream`, `node_modules`, etc.

Move in the folder and run command:

```
npm init
```

NPM will ask you several question to create a `package.json` file. This file keeps important information about your package in JSON format:

- Project information like name, version, author, and license
- Set of packages the project depends on

- Set of pre-configured commands to build and test the project

Here is how the `package.js` could look like:

```
{
  "name": "hello-world",
  "version": "1.0.0",
  "description": "The Hello World",
  "author": " Put Your Name Here",
  "license": "MIT"
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  }
}
```

We are ready to configure our project.

TypeScript Compile Configuration

Run the Visual Studio Code and open the folder of the project. We need to create the configuration file guides the TypeScript compiler where to find source folder, required libraries and how to compile the project. From the File menu create `tsconfig.json` file, copy/paste the following:

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "moduleResolution": "node",
    "sourceMap": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "removeComments": false,
    "noImplicitAny": false
  },
  "exclude": [
    "node_modules",
    "typings/main",
    "typings/main.d.ts"
  ]
}
```

The TypeScript compiler needs type definition files of JavaScript libraries from `node_modules` of our project because it doesn't recognize them natively. We help him with `typings.json` file. You shall create the file and copy/paste the following:

```
{
  "ambientDependencies": {
    "es6-shim": "registry:dt/es6-shim#0.31.2+20160317120654"
  }
}
```

Task Automation and Dependency Resolution

Now, it's time to add libraries into the `package.json` file the application require. Please update it accordingly:

```
{
  "name": "hello-world",
  "version": "1.0.0",
  "description": "The Hello World",
  "author": "Put Your Name Here",
  "license": "MIT",
  "scripts": {
    "start": "tsc && concurrently \"npm run tsc:w\" \"npm run lite\" ",
    "lite": "lite-server",
    "postinstall": "typings install",
    "tsc": "tsc",
    "tsc:w": "tsc -w",
    "typings": "typings"
  },
  "dependencies": {
    "@angular/common": "2.0.0-rc.1",
    "@angular/compiler": "2.0.0-rc.1",
    "@angular/core": "2.0.0-rc.1",
    "@angular/http": "2.0.0-rc.1",
    "@angular/platform-browser": "2.0.0-rc.1",
    "@angular/platform-browser-dynamic": "2.0.0-rc.1",
    "@angular/router": "2.0.0-rc.1",
    "@angular/router-deprecated": "2.0.0-rc.1",
    "@angular/upgrade": "2.0.0-rc.1",

    "systemjs": "0.19.27",
    "es6-shim": "^0.35.0",
    "reflect-metadata": "^0.1.3",
    "rxjs": "5.0.0-beta.6",
    "zone.js": "^0.6.12",

    "angular2-in-memory-web-api": "0.0.7",
    "bootstrap": "^ 4.0.0-alpha.2"
  },
  "devDependencies": {
```

```
    "concurrently": "^2.0.0",
    "lite-server": "^2.2.0",
    "typescript": "^1.8.10",
    "typings": "^0.8.1"
  }
}
```

Our configuration includes `scripts` to handle common development tasks. Configuration was finished, so let's run NPM manager to install the packages require. Go back to Terminal and enter the following command:

```
npm install
```

During the installation, you may see warning messages in red starting with

```
npm WARN
```

You shall ignore them if the installation finishes successfully. After installation, the NPM executes the `postinstall` script to run `typings` installation.

Creating and bootstrapping an Angular component

The Angular 2 application must always have a top-level component, where all another components and logic lies. Let's create it. Go to the Visual Studio Code and create a sub-folder `app` of the root directory where we will keep the source code. Create file `app.component.ts` under `app` folder, copy/paste the following:

```
// Import the decorator class for Component
import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  template: '<h1> Hello, World</h1>'
})
export class AppComponent { }
```

As you see, we added metadata via `@Component` decorator to class `AppComponent`. This decorator tells to Angular how to process the class via configuration with the following options:

- The `selector` defines the name of HTML tag which our component will link

- We pass in any service in the `providers` property. Any service registered on application level becomes available globally.
- We pass in any directives or components in the `directives` property.
- We give away any number of style files to `styles` a particular component
- The `template` property will hold the template of the component

We need export the class `AppComponent` to make it visible from other modules and Angular can instantiate it.

Now we need bootstrap the Application Component. Let's create `main.ts` file under `app` folder, copy/paste the following:

```
// Import bootstrap method to load our application component
import { bootstrap } from '@angular/platform-browser-dynamic';
// Load App Component class
import { AppComponent } from './app.component';
// Bootstrap out application
bootstrap(AppComponent);
```

And last but not list, we rely on the bootstrap function to load top-level components. We import it from `'@angular/platform-browser-dynamic'`. The Angular has a different kind of bootstrap function for:

- Web Workers
- Development on mobile devices
- Render the first page of application on server

Angular doing several tasks after instantiation of any component:

- It creates a shadow DOM for it
- It loads the selected template into the shadow DOM
- It creates all the injectable objects configured with `'providers'` and `'viewProviders'`.

In the end, the Angular 2 evaluates all template expressions and statements against the component instance.

Now, create `index.html` file in Microsoft Visual Studio code under the root folder with the following content:

```
<html>
  <head>
    <title>Angular 2 First Project</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="stylesheet" href="styles.css">

    <!-- 1. Load libraries -->
    <!-- Polyfill(s) for older browsers -->
    <script src="node_modules/es6-shim/es6-shim.min.js"></script>
    <script src="node_modules/zone.js/dist/zone.js"></script>
    <script src="node_modules/reflect-metadata/Reflect.js"></script>
    <script src="node_modules/systemjs/dist/system.src.js"></script>

    <!-- 2. Configure SystemJS -->
    <script src="systemjs.config.js"></script>
    <script>
      System.import('app').catch(function(err){ console.error(err); });
    </script>
  </head>

  <!-- 3. Display the application -->
  <body>
    <my-app>Loading...</my-app>
  </body>
</html>
```

Because we are referencing on `systemjs.config.js` file, let's create it in the root folder with the code:

```
(function(global) {

  // map tells the System loader where to look for things
  var map = {
    'app': 'app', // 'dist',
    'rxjs': 'node_modules/rxjs',
    'angular2-in-memory-web-api': 'node_modules/angular2-in-memory-web-api',
    '@angular': 'node_modules/@angular'
  };

  // The packages tells the System loader how to load when no filename
  and/or no extension
  var packages = {
    'app': { main: 'main.js', defaultExtension:
```

```
'js' },
  'rxjs': { defaultExtension: 'js' },
  'angular2-in-memory-web-api': { defaultExtension: 'js' },
};

var packageNames = [
  '@angular/common',
  '@angular/compiler',
  '@angular/core',
  '@angular/http',
  '@angular/platform-browser',
  '@angular/platform-browser-dynamic',
  '@angular/router',
  '@angular/router-deprecated',
  '@angular/testing',
  '@angular/upgrade',
];

// add package entries for angular packages in the form
'@angular/common': { main: 'index.js', defaultExtension: 'js' }
packageNames.forEach(function(pkgName) {
  packages[pkgName] = { main: 'index.js', defaultExtension: 'js' };
});

var config = {
  map: map,
  packages: packages
}

// filterSystemConfig - index.html's chance to modify config before we
register it.
if (global.filterSystemConfig) { global.filterSystemConfig(config); }

System.config(config);

})(this);
```

Compile and Run

We are to run our first application. Go back to Terminal and type

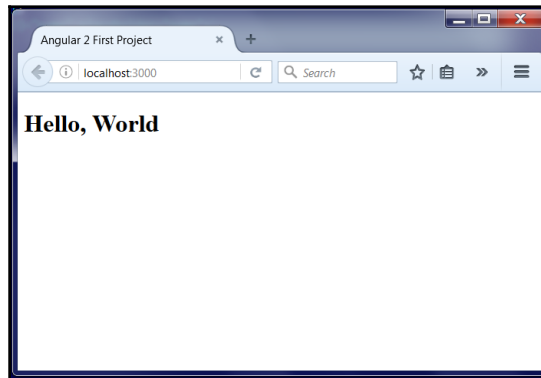
```
npm start
```

This script runs two parallel Node.js processes:

- The TypeScript compiler in watch mode

- The `staticlite-server` loads `index.html` and refreshes browser when application file changes.

In your browser you must see the following picture:



We now need to include our text input and also, specify the model we want to use. When a user types in the text input, our application shows the changed value in the title. Updated version of `app.component.ts`:

```
// Import the decorator class for Component
import { Component } from '@angular/core';
// Import the bootstrap method to load application component
import {bootstrap} from '@angular/platform-browser-dynamic';

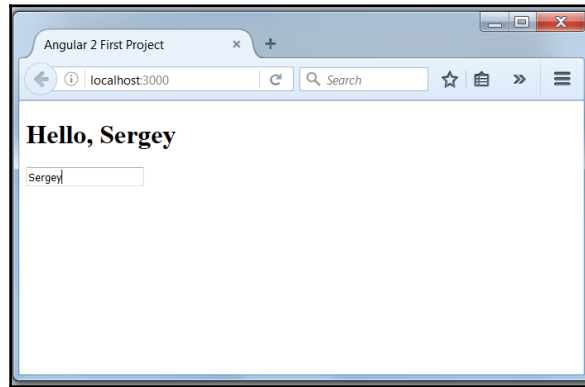
@Component({
  selector: 'my-app',
  template: `<h1>Hello, {{name}}</h1>
    <input type="text" [(ngModel)]="name" placeholder="name">`
})
export class AppComponent {
  let name: string = 'World';
}
```

The `ngModel` attribute declares a model binding on that element, and anything we type into the input box will be automatically bound to it by Angular. Obviously, this isn't going to be displayed on our page by magic; we need to tell the framework where we want it echoed. To show our model on the page, we just need to wrap the name of it in double curly braces:

```
{{name}}
```

I popped this in place of `World` in our `<h1>` tag and refreshed the page in my browser. If

you pop your name in the input field, you'll notice that it's automatically displayed in your heading in real time. Angular is doing all of this for us, and we haven't written a single line of code:



Now, while that's great, it would be nice if we could have a default in place so it doesn't look broken before a user has entered their name. What's awesome is that everything in between those curly braces is parsed as an Angular expression, so we can check and see if the model has a value, and if not, it can echo World. Angular calls this an expression and it's just a case of adding two pipe symbols as we would in TypeScript:

```
{{name || 'World'}}
```

It's good to remember that this is TypeScript, and that's why we need to include the quotation marks here, to let it know that this is a string and not the name of a model. Remove them and you'll notice that Angular displays nothing again. That's because both the name and World models are undefined.

Integrating Bootstrap 4

Now that we've created our Hello World application, and everything is working as expected, it's time to get involved with Bootstrap and add a bit of style and structure to our app.

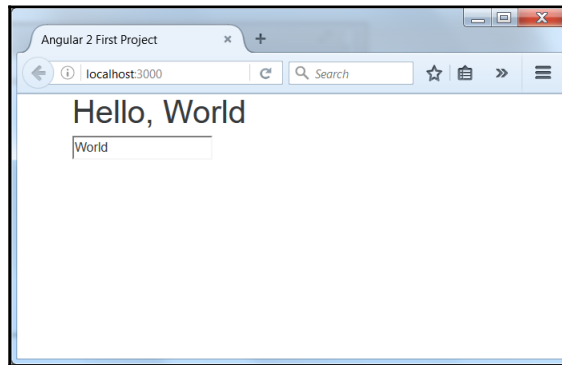
The application is currently misaligned to the left, and everything is looking cramped so let's sort that out first with a bit of scaffolding. Bootstrap comes with a great **mobile first** responsive grid system that we can utilize with the inclusion of a few `div` elements and classes. First, though, let's get a container around our content to clean it up immediately:



Mobile first is a way of designing/developing for the smallest screens first and adding to the design rather than taking elements away.

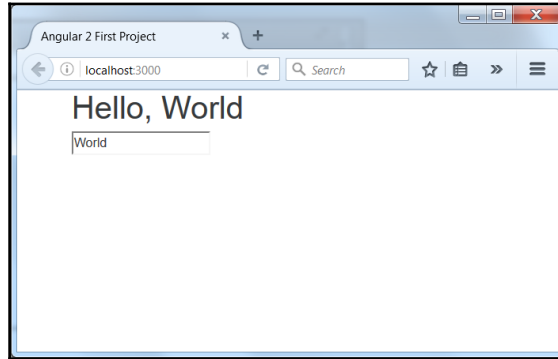
```
<div class="container">
  <h1>Hello, {{name || 'World'}}</h1>
  <input type="text" [(ngModel)="name">
</div>
```

If you resize your browser window, you should start to notice some of the responsiveness of the framework coming through and see it collapsing:



Now, I think it's a good idea to wrap this in what Bootstrap calls a Jumbotron (in previous versions of Bootstrap this was a Hero Unit). It'll make our headline stand out a lot more. We can do this by wrapping our `H1` and `input` tags in a new `div` with the `jumbotron` class:

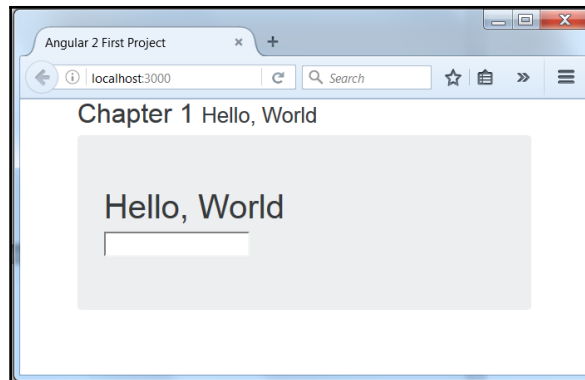
```
<div class="container">
  <div class="jumbotron">
    <h1>Hello, {{name || 'World'}}</h1>
    <input type="text" ng-model="name">
  </div>
</div>
```



It's starting to look a lot better, but I'm not too happy about our content touching the top of the browser like that. We can make it look a lot nicer with a page header, but that input field still looks out of place to me.

First, let's sort out that page header:

```
<div class="container">
  <div class="page-header">
    <h2>Chapter 1 <small>Hello, World</small></h2>
  </div>
  <div class="jumbotron">
    <h1>Hello, {{name || 'World'}}</h1>
    <input type="text" ng-model="name">
  </div>
</div>
```

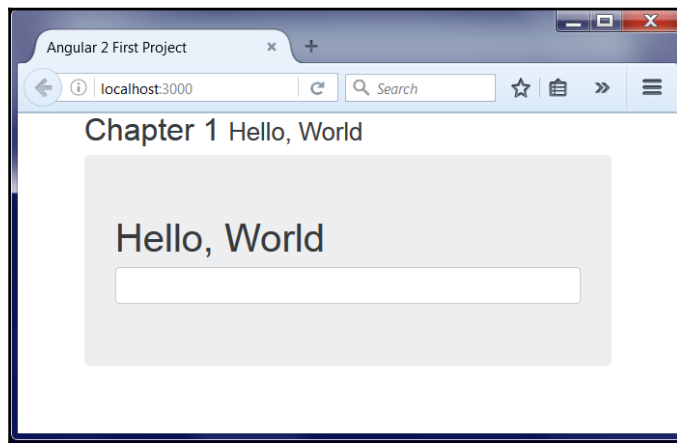


I've included the chapter number and title here. The `<small>` tag within our `<h2>` tag

gives us a nice differentiation between the chapter number and the title. The page-header class itself just gives us some additional margin and padding as well as a subtle border along the bottom.

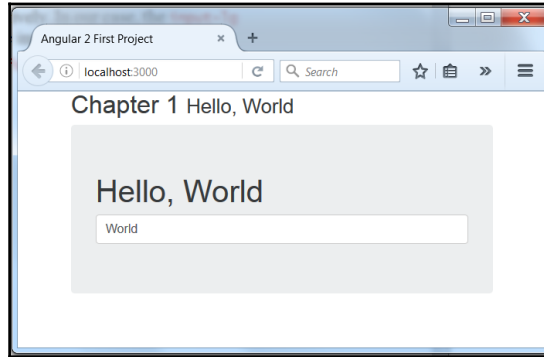
The utmost thing I think we could improve upon is that input box. Bootstrap comes with some cool input styles so let's include those. First, we need to add the class of `form-control` to the text input. This will set the width to 100% and also bring out some beautiful styling such as rounded corners and glow when we focus on the element:

```
<input type="text" [(ngModel)]="name" class="form-control">
```



Much better, but to me it looks a little small when you compare it with the heading. Bootstrap provides two additional classes we can include that will either make the element smaller or larger: `input-lg` and `input-sm` respectively. In our case, the `input-lg` class is the one we want, so go ahead and add that to the input.

```
<input type="text" [(ngModel)]="name"
      class="form-control input-lg">
```



Summary

Our app's looking great and working exactly how it should, so let's recap what we've learnt in the first chapter.

To begin with, we saw just how to setup working environment and finish the TypeScript Crash Course.

The Hello, World app we've created, while being very basic, demonstrates some of Angular's core features:

- @Component directive
- Application bootstrapping
- Two-way data binding

All of this was possible without writing a single line of TypeScript, as the component we created was just to demonstrate two-way.

With Bootstrap, we utilized a few of the many available components such as the jumbotron and the page-header classes to give our application some style and substance. We also saw the framework's new mobile first responsive design in action without cluttering up our markup with unnecessary classes or elements.

In *Chapter 2, Working with Bootstrap Components* we're going to explore more Bootstrap fundamentals and introduce the project we're going to be building over the course of this book.

2

Working with Bootstrap Components

In the world of web designing and development, we heard a lot about Twitter Bootstrap 3. The hero of our days is Bootstrap 4 – a CSS framework that helps in designing web applications easier and faster.

In this chapter, I will explain how you can start using Bootstrap 4 by showcasing a demo layout page, and how you can explore the framework and customize it to your requirements. At the end of the chapter, you will have a solid understanding of:

- How use Syntactically Awesome Style Sheets (**Sass**)
- How add Bootstrap 4 in your project
- How design layouts with grids and containers
- How add navigation elements
- How to customize of selected components

The Bootstrap 4

In the first chapter, we spoke about the Twitter Bootstrap 4 in a couple of words, but it's time to look at this CSS framework closer. But before delved deeper into Bootstrap 4 let's talk about all of the newly introduced features:

- The source CSS files of Bootstrap 4 based on Sass
- The **rem** is primary CSS unit instead of **px**
- Global font size increased from 14px to 16px
- Added new grid tiers for small devices (from ~480px and below)

- The Bootstrap 4 optionally supports **Flex Box Grid**
- It adds the Improved **Media Queries**
- The new **Card** component replaces the Panel, Well, and Thumbnail
- There is the new Reset Component called **Reboot.css**
- Everything is customizable with Sass variables
- Dropped support for IE 8 and iOS 6
- It is no longer support non-responsive usage

Introduction to Sass

If you are not familiar with Sass, I think, it is the right moment to introduce to you that fantastic CSS preprocessing framework. There is not a secret that preprocessing of CSS file allows you write more concise and less verbose stylesheets. The syntax of the first version of Sass used indentations, didn't require semi-colons, has shorthand operators and used `sass` file extension. It was so different from CSS that Sass version 3 start to support new format of syntax with brackets, semicolons, and `scss` file extension. Let's compare the various forms to each other.

Here is a vanilla CSS style:

```
#container {
    width:100px;
    padding:0;
}
#container p {
    color: red;
}
```

In files with `sass` extension, we should use only indentation, and it is heavily dependents on white spaces:

```
$red: #ff0000
#container
    width:100px
    padding: 0
    p
        color:$red
```

In files with `scss` extension, we will use brackets and semicolons:

```
$red: #ff0000;
#container {
```

```
width:100px;
padding:0;
p {
    color :$red;
}
}
```

It is ultimately up to you which style you prefer, but I will use the newest one in this book.

Setup of Ruby

Before you start using Sass, you will need to install Ruby. Please follow the recommendations of how to install Ruby on you PC on official Ruby website <https://www.ruby-lang.org/en/documentation/installation>.

Setup of Sass

After finishing installation of Ruby open the Terminal and type following command for Windows:

```
gem install sass
```

or for Linux and Mac:

```
sudo gem install sass
```

This command will install Sass and necessary dependencies for you. Run the command below to check installation of Saas on your PC:

```
sass -v
```

The Sass must answer with version number:

```
Sass 3.4.22 (Selective Steve)
```

Now, that we have installed Sass, we can explore it files and output them into CSS. You can use CLI or GUI to get you started with Sass. If you prefer GUI style of development, please use one from the list below:

- CodeKit (Mac, Paid) – <http://incident57.com/codekit>
- Compass.app (Windows, Mac, Linux, Paid, Open Source) – <http://compass.hondlino.com>
- Ghostlab (Web-based, Paid) – <http://www.vanamco.com/ghostlab>

- Hammer (Mac, Paid) – <http://hammerformac.com>
- Koala (Windows, Mac, Linux, Open Source) – <http://koala-app.com>
- LiveReload (Mac, Paid, Open Source) – <http://livereload.com>
- Prepros (Windows, Mac, Linux, Paid) – <https://prepros.io>
- Scout (Windows, Mac, Open Source) – <http://mhs.github.io/scout-app>

Personally, I prefer the Scout GUI that runs Sass and Compass in a self-contained Ruby environment, does all of the heavy liftings, so we will never have to worry about technical issues like Ruby setup.

Another interesting option I would recommend is a web-based Sass playground **SassMeister** you can find by the following address <http://www.sassmeister.com>. We will use it for a minute in our Sass crash course.

Sass Crash Course

The main idea laying behind the Sass is that – create reusable, less verbose code easy to read and understand. Let's see what the features make that happens. Please open SassMeister website and prepare to our exercises.

Variables

We can create variables in Sass, especially to reuse them throughout our document. Acceptable values for variables include:

- number
- string
- color
- null
- list
- map

We use the `$` symbol to define a variable. Switch to SassMeister and create our first variables:

```
$my-pad : 2em;  
$color-primary : red;  
$color-secondary : #ff00ed;
```

The SassMeister compiles them but without output any CSS. We just define variables in the

scope, and that is it. We need to use them in CSS declaration to see the result of compilation:

```
body {  
  background-color: $color-primary;  
}  
  
.container {  
  padding: $my-pad;  
  color: $color-secondary;  
}
```

Here is the result of compilation from Sass to CSS:

```
body {  
  background-color: red;  
}  
  
.container {  
  padding: 2em;  
  color: #ff00ed;  
}
```

Mathematical Expressions

The Sass allows us to use following mathematical operations in expressions:

- Addition (+)
- Subtraction (-)
- Division (/)
- Multiplication (*)
- Modulo (%)
- Equality (==)
- Inequality (!=)

Jump to [SassMeister](#) and play with introduced mathematical operations:

```
$container-width: 100%;  
$font-size: 16px;  
  
.container {  
  width: $container-width;  
}  
  
.col-4 {  
  width: $container-width / 4;
```

```
font-size: $font-size - 2;
}
```

Here is compiler CSS code:

```
.container {
  width: 100%;
}

.col-4 {
  width: 25%;
  font-size: 14px;
}
```



You should avoid to use different units in your expressions.

Functions

The Sass has a reach set of built-in functions and here is the address where you can find all of them: <http://sass-lang.com/documentation/Sass/Script/Functions.html>.

Here is the simplest example of usage the `rgb($red, $green, $blue)` function. It creates a color from red, green, and blue values:

```
$color-secondary : rgb(ff,00,ed);
```

Nesting

Sass allows us to have a declaration inside of another declaration. In the following vanilla CSS code we defining two statements:

```
.container {
  width: 100px;
}

.container h1 {
  color: green;
}
```

We have a container class and header within a container styles declarations. In Sass we can create the compact code:

```
.container {  
  width: 100px;  
  h1 {  
    color: green;  
  }  
}
```

Nesting makes code more readable and less verbose.

Imports

The Sass allows to break styles into separate files and import them into one another. We can use `@import` directive with or without the file extensions. There are two lines of code giving the same result:

```
@import "components.scss";  
@import "components";
```

Extends

If you need to inherit style from existing one, the Sass has `@extend` directive to help you:

```
.input {  
  color: #555;  
  font-size: 17px;  
}  
  
.error-input {  
  @extend .input;  
  color: red;  
}
```

Here is the result of how Sass compiler proper handled the compiled code:

```
.input, .error-input {  
  color: #555;  
  font-size: 17px;  
}  
  
.error-input {  
  color: red;  
}
```

Placeholders

In the case when you want to extend declaration with set of styles that doesn't exist, Sass helps with placeholder selector:

```
%input-style {
  font-size: 14px;
}

.input {
  @extend %input-style;
  color: #555;
}
```

We use % sign to prefix a class name and with the help of @extend, all magic happens. Sass doesn't render the placeholder. It renders only the result of its extending elements. Here is the compiled code:

```
.input {
  font-size: 14px;
}

.input {
  color: #555;
}
```

Mixins

We can create reusable chunks of CSS styles with mixins. Mixins always return markup code. We use the @mixin directive to define mixins and @include to use them in the document. You may see the following code quite often before:

```
a:link { color: white; }
a:visited { color: blue; }
a:hover { color: green; }
a:active { color: red; }
```

Indede, we change the color of element depends on states. Usually we writing this code over and over again, but with Sass we can done it like this:

```
@mixin link ($link, $visit, $hover, $active) {
  a {
    color: $link;
    &:visited {
      color: $visit;
    }
  }
}
```

```
    &:hover {
      color: $hover;
    }
    &:active {
      color: $active;
    }
  }
}
```

The & symbol here points to the parent element, i.e., to the anchor element. Let's use this mixin in the following example:

```
.component {
  @include link(white, blue, green, red);
}
```

Here is the compiled to CSS code:

```
.component a {
  color: white;
}
.component a:visited {
  color: blue;
}
.component a:hover {
  color: green;
}
.component a:active {
  color: red;
}
```

Function Directives

The function directives is another feature of Sass helps create reusable chunks of CSS styles returns values via the @return directive. We use @function directive to define it:

```
@function getTableWidth($columnWidth, $numColumns, $margin) {
  @return $columnWidth * $numColumns + $margin * 2;
}
```

In this function we calculate the width of the table depends on individual column width, number of columns and margin values:

```
$column-width: 50px;
$column-count: 4;
$margin: 2px;
```

```
.column {
  width: $column-width;
}

.table {
  background: #1abc9c;
  height: 200px;
  width: getTableWidth($column-width, $column-count, $margin);
  margin: 0 $margin;
}
```

Resulting CSS code looks like this:

```
.column {
  width: 50px;
}

.table {
  background: #1abc9c;
  height: 200px;
  width: 204px;
  margin: 0 2px;
}
```

I think it's time to handover out Sass crash course, but please don't think that you know everything about it. The Sass is big and incredibly powerful, so if you decided to continue the journey we started here, please follow to the next address to get more: http://sass-lang.com/documentation/file.SASS_REFERENCE.html.

The Example Project

Let's talk about what the web application we will develop in the moment of reading this book. I decided that the **e-commerce** application is the best candidate to demonstrate the full flavor of different Bootstrap 4 components tight in one place.

The term e-commerce, as we think of it today, refers to the buying and selling of goods or services over the Internet, so we design web application based on a real-world scenario. After the introduction, we consolidate a high-level list of customer requirements. We then prepare a series of mockups which help you get a clearer picture of how the final application will look to an end-user. Finally, we break down the customer requirements into a set of implementation tasks and structure the application so that the responsibilities and interactions among functional components are clearly defined.

The Scenario

The Dream Bean is a small grocery store collaborates with several local farms to supply organic produce and foods. The store has a long-standing customer base and increasing profit to the area. The store has decided to investigate the possibility of providing an online delivery service to customers because a recent survey has indicated that 95% of its regular clientele has continuous Internet access, and 83% percent would be interested in using this service.

The manager of the grocery asked you, to create a website that will enable their customers to shop online on a broad range of devices includes cell phones, tablets, and desktop computers.

Gathering Customer Requirements

Before making any design or implementation decisions, you need to collect information from the client, so after direct communications with client, we have the following conditions:

- The customer can buy products available in the physical store. There are following categories of the products:
 - Meat
 - Seafood
 - Bakery
 - Dairy
 - Fruit and Vegetables
 - Takeaway
- The customer can browse all the goods or filter them by category
- The customer has a virtual shopping cart
- The customer can add, remove, update item quantities in the shopping cart
- The customer can view a summary of all things
- The customer can place an order and make payment through security checkout process

Preparing the Use-Cases

Now, when the requirements are in place, it is time to work with managers from the Dream Bean to gain how website to look and behave. We create a set of use-cases that describe how

the customer will use the web application:

- The customer visits the welcome page and selects a product by category.
- The customer browses products within the selected category page, then adds a product to shopping cart.
- The customer clicks on 'Info' button to open the separate sheet contains the full information about the product and then adds a product to shopping cart.
- The customer continues shopping and selects a different category.
- The customer adds several products from this class to the shopping cart.
- The customer selects 'View Cart' option and updates quantities for products in the cart.
- The client verifies shopping cart contents and proceeds to checkout.
- In the checkout page, customer views the cost of the order and other information, can update quantities of the goodies, fills in personal data, then submits the details.

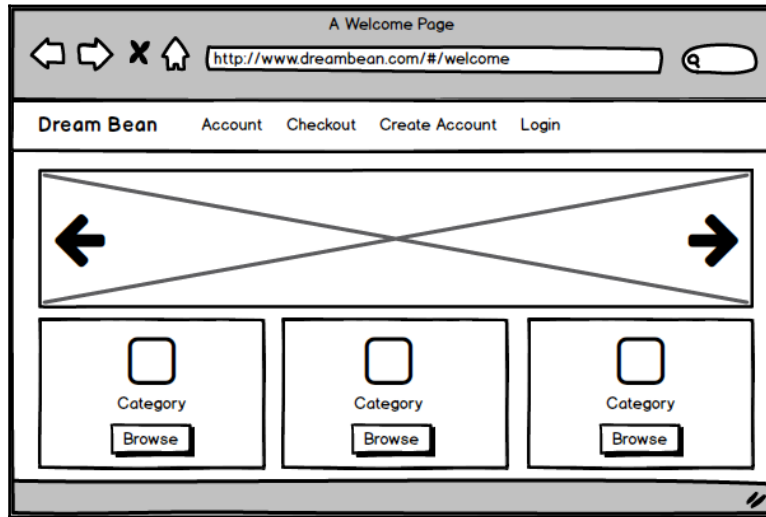
We continue to work with staff of the Dream Bean company and need to create mockups in one of the following ways:

- Use storyboard software
- Create set of wireframes
- Use paper prototyping

I use Balsamiq Mockups helps me quickly create wireframes. The fully functional trial version of Balsamiq Mockups working for 30 days available on official website: <https://balsamiq.com>.

The Welcome Page

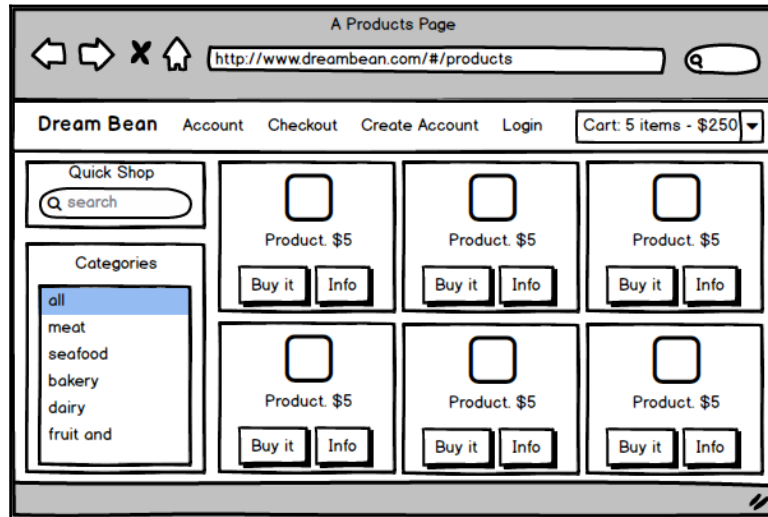
The welcome page is an entry point for the application. It introduces the business and service to the customer and enables him or here to navigate to any of the product categories. We add a slideshow in the middle of the welcome page.



The wireframe of the Welcome Page

The Products Page

The products page provides a listing of all goodies within the chosen category. From this page, a customer can view all product information, and add any of the listed products to his or her shopping cart. A user can also navigate to any of the provided categories or use **Quick Shop** feature to search products by name.

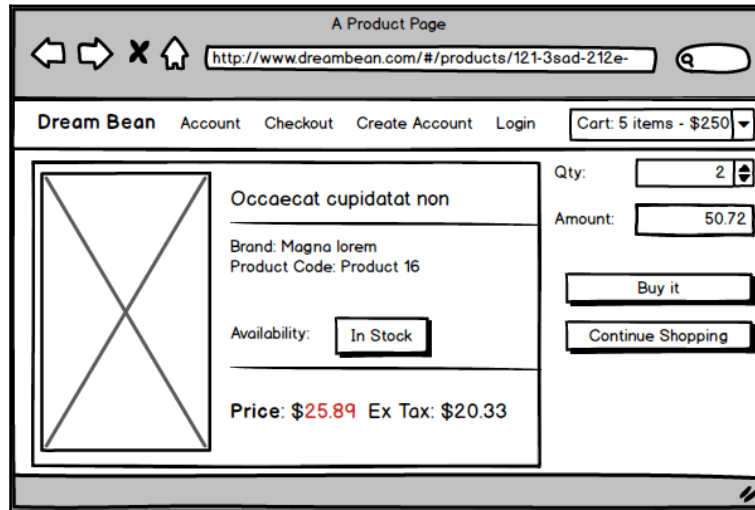


The wireframe of the Products Page

The Product Page

The product page displays information about the product. On this page the customer can:

- Check availability of the product
- Update the quantity of the product
- Add the product to the cart by clicking **But it**.
- Return to products list by clicking on **Continue Shopping**.

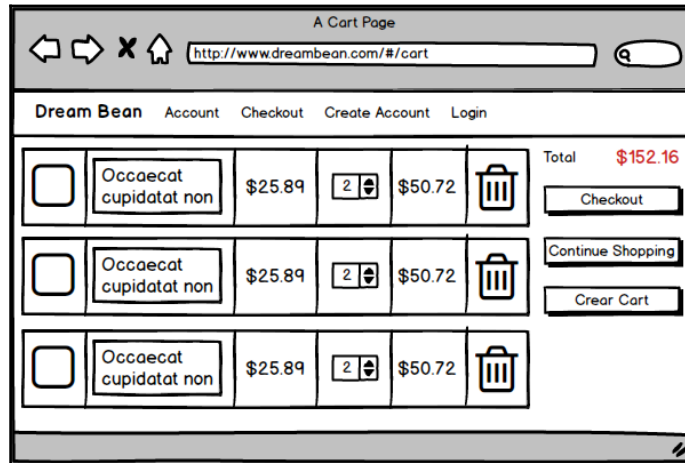


The wireframe of the Product Page

The Cart Page

The cart page lists all items held in the user's shopping cart. It displays product details for each item and from this page, a user can:

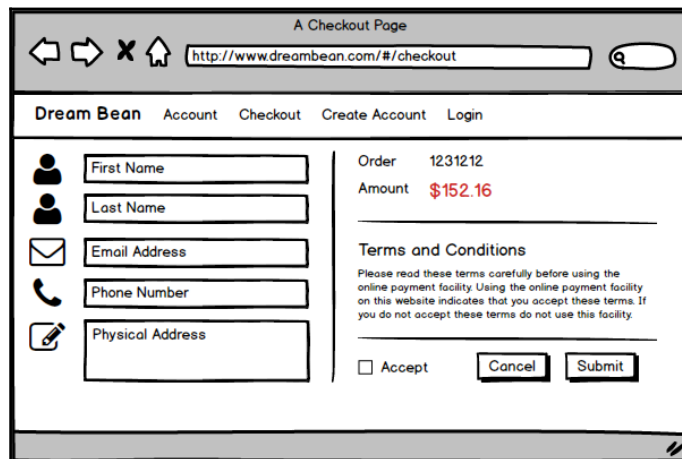
- Remove all goodies from his or her cart by clicking **Clear Cart**.
- Update the quantity for any listed item
- Return to products list by clicking on **Continue Shopping**.
- Proceed to checkout by clicking **Checkout**.



The wireframe of the Cart Page

The Checkout Page

The checkout page displays the customer details form, purchase conditions, and the order information. The customer should fill in the form, accept payment and click on **Submit** button to start the payment process.



The wireframe of the Checkout Page

We have everything to initiate the journey with Angular 2 and Bootstrap. We projected the

business requirements on mockups, and now we need to do next:

- Open Terminal, create folder 'ecommerce' and move in
- Copy the content of the project from the folder 'ecommerce-seed.' into the new project
- Run the following script to install NPM modules:

```
npm install
```

- Start the TypeScript watcher and lite server with next command:

```
npm run start
```

This script opens the web browser and navigates to welcome page of the project. We are ready to start development.

Designing layouts with grids and containers

The Bootstrap includes a powerful mobile-first grid system for building designs of all shapes and sizes, and that sounds very promising because we need to create several pages for our project. We will use the grid systems for creating the page layouts through a series of rows and columns. Since Bootstrap is developed to be mobile first, we use a handful of `media queries` to create sensible breakpoints for our layouts and interfaces. These breakpoints are mostly based on minimum viewport widths and allow us to scale up elements as the viewport changes. There are three main components of the grid system:

- Container
- Row
- Column

The container is the core and required layout element in Bootstrap. There are two classes to create the containers for all other items:

- You can create a responsive, fixed-width container with a `container` class. This one doesn't have extra space for both sides of hosting element and its `max-width` property changes at each breakpoint.
- You can use the full-width container with a `container-fluid` class. This one always has 100% width of a viewport.

To create a simple layout for our project open `app.component.html` file, and insert a `div` element with a `container` class inside:

```
<div class="container">
</div>
```

We can nest containers, but most layouts do not require that. The container is just a placeholder for rows, so let's add the row inside:

```
<div class="container">
  <div class="row">
  </div>
</div>
```

The row has a `row` class, and the container can contain as many rows as you need.



I recommend using one or several containers with all of the rows inside to wrap the page content and center elements on the screen.

A row is a horizontal group of columns. It exists only for one purpose to keep columns lined up correctly. We must put the substance of the page only inside columns and indicates the number of columns to use. Each row can contain up to 12 of them.

We can add the column to the row as a combination of a `col` class, and it prefixes size:

```
<div class="col-md-12">
```

The Bootstrap 4 supports five different sizes of displays, and the columns classes names depend on them:

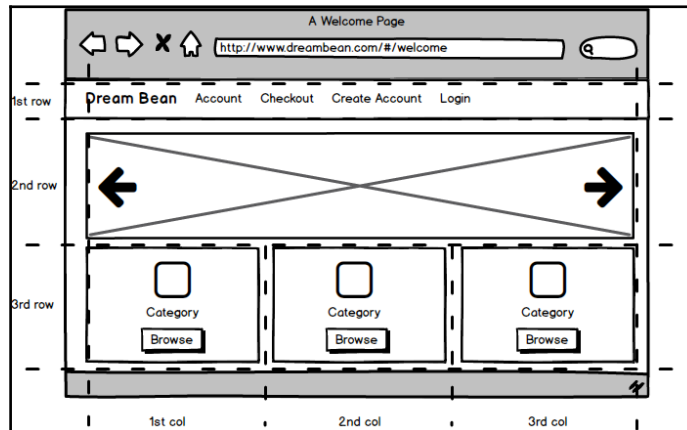
- `col-xs` for extra small display (screen width less than 34em or 544px)
- `col-sm` for smaller display (screen width 34em or 544px and up)
- `col-md` for medium display (screen width 48em or 768px and up)
- `col-lg` for larger display (screen width 62em or 992px and up)
- `col-xl` for extra large display (screen width 75em or 1200px and up)

The columns classes names are always applying to devices with screen widths greater than or equal to the breakpoint sizes.

The width of a column sets in percentage, so it is always fluid and sized about parent element. Each column has a horizontal padding to create a space between individual columns. The first and last columns have negative margins, and this is why the content within the grid lines up with the substance outside. Here is an example of grid for extra small devices:

.col-xs-1	.col-xs-1	.col-xs-1	.col-xs-1	.col-xs-1	.col-xs-1	.col-xs-1	.col-xs-1	.col-xs-1	.col-xs-1	.col-xs-1	.col-xs-1	.col-xs-1
.col-xs-2		.col-xs-3			.col-xs-7							
.col-xs-4				.col-xs-4				.col-xs-4				
.col-xs-5					.col-xs-7							
.col-xs-6						.col-xs-6						
.col-xs-12												

Look at the welcome page mockup of our project and imagine splitting it into rows and columns:



The wireframe of the Welcome Page

Our markup has three rows. The first has a header with company logo and menu. It spans 12 middle size columns marked with `col-md-12`. I use grid for now, but later I will change it to more suitable component:

```
<div class="container">
  <div class="row">
    <div class="col-md-12 table-bordered">
      <div class="product-menu">Logo and Menu</div>
    </div>
  </div>
  <!-- /.row -->
</div>
```

The second one has a single column contains an image 1110x480px and spans all 12 middle size columns marked with `col-md-12` like previous one:

```
<div class="container">
  <div class="row">
    <div class="col-md-12 table-bordered">
      
    </div>
  </div>
<!-- /.row -->
```

The last one includes the plates with six product categories, and each of them occupies the different number of columns depends on the size of layout:

- Four columns for middle size marked with `col-md-4`
- Six columns for small size marked as `col-sm-6`
- Twelve extra small columns marked with `col-xs-12`

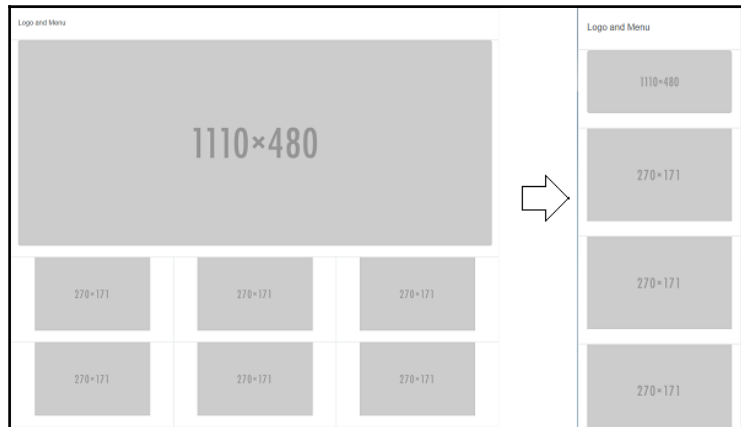
The size of each image is 270x171px. The markup of the bottom part of the screen quite long, so I cut it off:

```
<div class="row">
  <div class="col-xs-12 col-sm-6 col-md-4 table-bordered">
    <a href="#">
      
    </a>
  </div>
<!-- /.col -->
<div class="col-xs-12 col-sm-6 col-md-4 table-bordered">
  <a href="#">
    
  </a>
</div>
<!-- /.col -->
...
<div class="col-xs-12 col-sm-6 col-md-4 table-bordered">
  <a href="#">
    
  </a>
</div>
<!-- /.col -->
</div>
<!-- /.row -->
```



```
</div>
<!-- /.container -->
```

I intentionally added the Bootstrap class `table-bordered` to display the boundaries of columns. I will remove it later. Here is the result of how the website looks like:



If I change the viewport to smaller size, the Bootstrap immediately transforms columns into the rows as you see on the picture above. I didn't use real images on the page but pointed on <http://placeholder.it>. It is a service in the web generates placeholder images of specified size on fly. The link like that <https://placeholder.it/imgix.net/~text?txtsize=25&txt=270%C3%97171&w=270&h=171> returns the placeholder image with 270x171px size.

Using Images

In our markup I used images, so pay attention to `img-fluid` class opts the image into responsive behavior:

```

```

The logic behind the class will never allow the image become larger than parent element. The same time, it adds the lightweight styles management via classes. You can easily design the shape of the picture as follow:

- Rounded it with `img-rounded` class. The border radius is 0.3rem.
- Circle it with the help of `img-circle`, so border radius became to 50%.

- Transform it with `img-thumbnail`.

In our example, the `center-block` centered the image, but you can align it with helper float or text alignment classes:

- The class `pull-sm-left` float left on small or wider devices
- The class `pull-lg-right` float right on large and bigger devices
- The class `pull-xs-none` prevents floating on all viewport sizes.

Now, I would like to create the plates and change them with images at the bottom of the page. The best one, we can use for this purposes is a **Card** component.

Using Cards

A Card component is very flexible, and extensible content container required a small amount of markup and classes to make fantastic things. The Cards replace the following elements exists in Bootstrap 3:

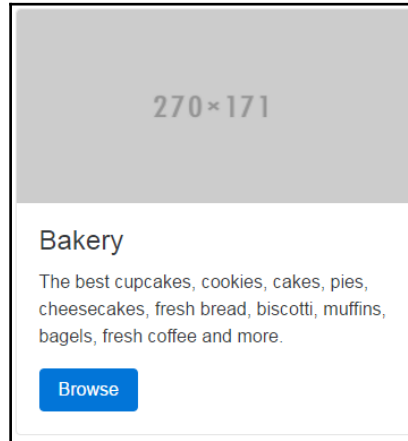
- Panels
- Wells
- Thumbnails

The simplest way to create it is to add the `card` and `card-block` classes to the element:

```
<div class="col-xs-12 col-sm-6 col-md-4">
  <div class="card">
    
    <div class="card-block">
      <h4 class="card-title">Bakery</h4>
      <p class="card-text">The best cupcakes, cookies, cakes,
                          pies, cheesecakes, fresh bread,
                          biscotti, muffins, bagels, fresh coffee
                          and more.</p>
      <a href="#" class="btn btn-primary">Browse</a>
    </div>
  </div>
</div>
```

The `card-block` class adds a padding space between the content and the card border. In my example, I moved it inside to allow the card header to line up flush with the card edge. If you need, you can create a header with `card-header` and footer with `card-footer`

classes. As you see, it includes a broad range of components into the Card like images, texts, list groups, and more. Here is how our Card component looks like:



But this is not only a single place where we use Card components. We will use them a lot in the following chapters.

Using Buttons

I added the button to the Card component, and I want to talk about it. You can apply the button style to following elements:

- The standard `button` works correctly across all browsers
- To the `input` element with `type="button"`
- To an anchor element, only behave like a button with `role="button"`. Use it only to trigger in-page functionality rather than linking to new a page or section within the current one.
- To the label when working with checkboxes and radio buttons

The Button General Styles

In Bootstrap 4 we can find seven styles for buttons and each of them for a different semantic purpose. The class `btn` add style for contextual variations, sizes, states of buttons placed standalone, in forms, or dialogs:



The primary actions style provides with an extra visual weight:

```
<button type="button" class="btn btn-primary">Primary</button>
```

The secondary, less important than primary actions style provides with less background color:

```
<button type="button" class="btn btn-secondary">Secondary</button>
```

This one indicates of any success operations or position actions:

```
<button type="button" class="btn btn-success">Success</button>
```

This is to guide users for informational actions or alerts:

```
<button type="button" class="btn btn-info">Info</button>
```

This one warning with cautions actions:

```
<button type="button" class="btn btn-warning">Warning</button>
```

This is indicates dangerous or potentially negative actions:

```
<button type="button" class="btn btn-danger">Danger</button>
```

This one presents button as a link:

```
<button type="button" class="btn btn-link">Link</button>
```

The Button with Outline Styles

You can remove hefty background images and colors on any button of any predefines styles by replacing the default modified classes with the `.btn-*-outline` ones.



```
<button type="button"
  class="btn btn-primary-outline">Primary</button>
<button type="button"
```

```
        class="btn btn-secondary-outline">Secondary</button>
<button type="button"
        class="btn btn-success-outline">Success</button>
<button type="button"
        class="btn btn-info-outline">Info</button>
<button type="button"
        class="btn btn-warning-outline">Warning</button>
<button type="button"
        class="btn btn-danger-outline">Danger</button>
```



There is no outline for link buttons (i.e. there is no `btn-link-outline` class).

The Sizes of Button

Buttons may have small and big sizes:



Use `btn-sm` and `btn-lg` classes to make that happens:

```
<button type="button"
        class="btn btn-primary btn-lg">Large button</button>
<button type="button"
        class="btn btn-primary btn-sm">Small button</button>
```

The Button with Block Level Styles

If you planning to create block level buttons that span the full width of parent element just add `btn-block` class:

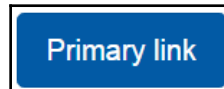
```
<button type="button"
        class="btn btn-primary btn-lg btn-block">Block</button>
```



The Button with Active Style

The pseudo-classes in button style update the visual state of element accordingly with user actions, but if you need to change the states manually use active class:

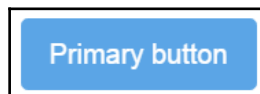
```
<a href="#" class="btn btn-primary btn-lg active"
  role="button">Primary link</a>
```



The Button with Inactive State

We can make button looks inactive with disabled property:

```
<button type="button" disabled
  class="btn btn-lg btn-primary">Primary button</button>
```

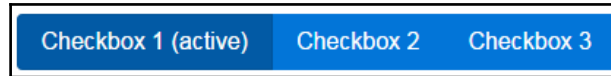


The Radio Buttons and Checkboxes

Bootstrap 4 provides button styles with toggle features to input elements similar to radio buttons and checkboxes. To achieve that you need to create the massive construction includes group element, label and input element itself:

```
<div class="btn-group" data-toggle="buttons">
  <label class="btn btn-primary active">
    <input type="checkbox" checked autocomplete="off">
    Checkbox 1 (active)
```

```
</label>
<label class="btn btn-primary">
  <input type="checkbox" autocomplete="off"> Checkbox 2
</label>
<label class="btn btn-primary">
  <input type="checkbox" autocomplete="off"> Checkbox 3
</label>
</div>
```



```
<div class="btn-group" data-toggle="buttons">
  <label class="btn btn-primary active">
    <input type="radio" name="options" id="option1"
      autocomplete="off" checked> Radio 1 (preselected)
  </label>
  <label class="btn btn-primary">
    <input type="radio" name="options" id="option2"
      autocomplete="off"> Radio 2
  </label>
  <label class="btn btn-primary">
    <input type="radio" name="options" id="option3"
      autocomplete="off"> Radio 3
  </label>
</div>
```



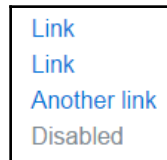
Navs

The Bootstrap 4 provides a base style for navigation elements. It exposes the base `nav` class that shares general markup and styles by extending it. All navigation components built on top of this by specifying additional styles. It doesn't have styles for the active state. By the way, you can use methods for disabled one.

The Base Nav

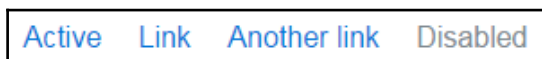
Any **Nav** component must have the outer navigation element based on `ul` or `nav` elements. Here are a list based approach display navigation elements vertically:

```
<ul class="nav">
  <li class="nav-item">
    <a class="nav-link" href="#">Link</a>
  </li>
  <li class="nav-item">
    <a class="nav-link" href="#">Link</a>
  </li>
  <li class="nav-item">
    <a class="nav-link" href="#">Another link</a>
  </li>
  <li class="nav-item">
    <a class="nav-link disabled" href="#">Disabled</a>
  </li>
</ul>
```



Our markup can be very flexible because all components based on classes. We can use `nav` with regular anchor elements to layout navigation horizontally:

```
<nav class="nav">
  <a class="nav-link active" href="#">Active</a>
  <a class="nav-link" href="#">Link</a>
  <a class="nav-link" href="#">Another link</a>
  <a class="nav-link disabled" href="#">Disabled</a>
</nav>
```



I like this approach because it is less verbose than the list-based one.

Inline Navigation

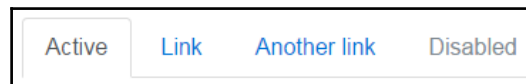
You can easily inline navigation elements with spacing horizontally as shown in the previous example with help of `nav-inline` class:

```
<ul class="nav nav-inline">
  <li class="nav-item">
    <a class="nav-link" href="#">Link</a>
  </li>
  <li class="nav-item">
    <a class="nav-link" href="#">Link</a>
  </li>
  <li class="nav-item">
    <a class="nav-link" href="#">Another link</a>
  </li>
  <li class="nav-item">
    <a class="nav-link disabled" href="#">Disabled</a>
  </li>
</ul>
```

Tabs

We can quickly transform Nav component from above to generate a tabbed interface with `nav-tabs` class:

```
<ul class="nav nav-tabs">
  <li class="nav-item">
    <a class="nav-link active" href="#">Active</a>
  </li>
  <li class="nav-item">
    <a class="nav-link" href="#">Link</a>
  </li>
  <li class="nav-item">
    <a class="nav-link" href="#">Another link</a>
  </li>
  <li class="nav-item">
    <a class="nav-link disabled" href="#">Disabled</a>
  </li>
</ul>
```



Pills

Just change nav-tabs to nav-pills to display the **Pills** instead:

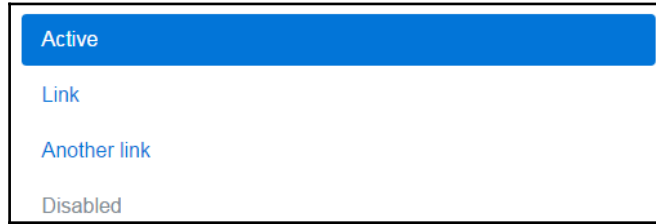
```
<ul class="nav nav-pills">
  <li class="nav-item">
    <a class="nav-link active" href="#">Active</a>
  </li>
  <li class="nav-item">
    <a class="nav-link" href="#">Link</a>
  </li>
  <li class="nav-item">
    <a class="nav-link" href="#">Another link</a>
  </li>
  <li class="nav-item">
    <a class="nav-link disabled" href="#">Disabled</a>
  </li>
</ul>
```



Stacked Pills

If you need layout Pills vertically use nav-stacked class:

```
<ul class="nav nav-pills nav-stacked">
  <li class="nav-item">
    <a class="nav-link active" href="#">Active</a>
  </li>
  <li class="nav-item">
    <a class="nav-link" href="#">Link</a>
  </li>
  <li class="nav-item">
    <a class="nav-link" href="#">Another link</a>
  </li>
  <li class="nav-item">
    <a class="nav-link disabled" href="#">Disabled</a>
  </li>
</ul>
```



Navigation with Dropdowns

You can add a dropdown menu to inline navigation, tabs, or pills via applying a dropdown class to the list item and with a little extra HTML and dropdown JavaScript plugins:

```
<ul class="nav nav-tabs">
  <li class="nav-item">
    <a class="nav-link active" href="#">Active</a>
  </li>
  <li class="nav-item dropdown">
    <a class="nav-link dropdown-toggle" data-toggle="dropdown"
      href="#" role="button" aria-haspopup="true"
      aria-expanded="false">Dropdown</a>
    <div class="dropdown-menu">
      <a class="dropdown-item" href="#">Action</a>
      <a class="dropdown-item" href="#">Another action</a>
      <a class="dropdown-item" href="#">Something else here</a>
      <div class="dropdown-divider"></div>
      <a class="dropdown-item" href="#">Separated link</a>
    </div>
  </li>
  <li class="nav-item">
    <a class="nav-link" href="#">Another link</a>
  </li>
  <li class="nav-item">
    <a class="nav-link disabled" href="#">Disabled</a>
  </li>
</ul>
```



Navbars

I mentioned a bit early about the header with company logo and menu, temporary implemented as a grid. Now we change this construction to the suitable component. Please welcoming the **Navbars**.

The Navbar it just a simple wrapper helping to position containing elements. Usually, it displays as a horizontal bar, but you can configure it to collapse on smaller layouts.

Like many other components of Bootstrap the Navbar container required a small amount of markup and classes to make it works:

- To create one, you must use a `navbar` class with conjunction of a color scheme
- The topmost must be `nav` or `div` element with `role="navigation"`

The Content

We can include built-in sub-components to add the placeholders as we need:

- Use `navbar-brand` class for your company, product, or project name
- Use `navbar-nav` class for full-height and lightweight navigation. It includes support for dropdowns as well.
- Use `navbar-toggler` class to organize collapsible behave

Let's use what we know about Navbar to build our header. First of all, I use `nav` to create topmost element:

```
<nav class="navbar navbar-light bg-faded">
```

Then, I need `navbar-brand` class for the company name. We can apply this class to most

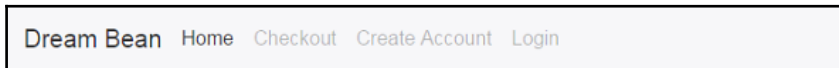
elements, but an anchor works best:

```
<a class="navbar-brand" href="#">Dream Bean</a>
```

At the end, I add set of navigation links with first active:

```
<ul class="nav navbar-nav">
  <li class="nav-item active">
    <a class="nav-link" href="#">
      Home <span class="sr-only">(current)</span>
    </a>
  </li>
  <li class="nav-item">
    <a class="nav-link" href="#">Checkout</a>
  </li>
  <li class="nav-item">
    <a class="nav-link" href="#">Create Account</a>
  </li>
  <li class="nav-item">
    <a class="nav-link" href="#">Login</a>
  </li>
</ul>
</nav>
<!-- /.navbar -->
```

Here is our header with branding and set of links:



With help of `nav` classes we can make navigation simple if avoid the list-based approach entirely:

```
<nav class="navbar navbar-light bg-faded">
  <a class="navbar-brand" href="#">Dream Bean</a>
  <div class="nav navbar-nav">
    <a class="nav-item nav-link active" href="#">
      Home <span class="sr-only">(current)</span>
    </a>
    <a class="nav-item nav-link" href="#">Checkout</a>
    <a class="nav-item nav-link" href="#">Create Account</a>
    <a class="nav-item nav-link" href="#">Login</a>
  </div>
</nav>
```

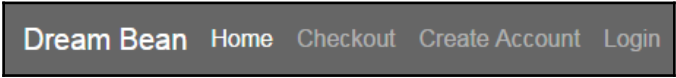
The Colors

You can manage colors of Navbar very elegant:

- Specify the scheme with `navbar-light` or `navbar-dark` classes
- Add color value via one of the Bootstrap color classes or by you own color with CSS

In my example I used light scheme and Bootstrap background faded color. Let's change it to the dark scheme and custom color:

```
<nav class="navbar navbar-dark" style="background-color: #666">
  <a class="navbar-brand" href="#">Dream Bean</a>
  <div class="nav navbar-nav">
    <a class="nav-item nav-link active" href="#">
      Home <span class="sr-only">(current)</span>
    </a>
    <a class="nav-item nav-link" href="#">Checkout</a>
    <a class="nav-item nav-link" href="#">Create Account</a>
    <a class="nav-item nav-link" href="#">Login</a>
  </div>
</nav>
```



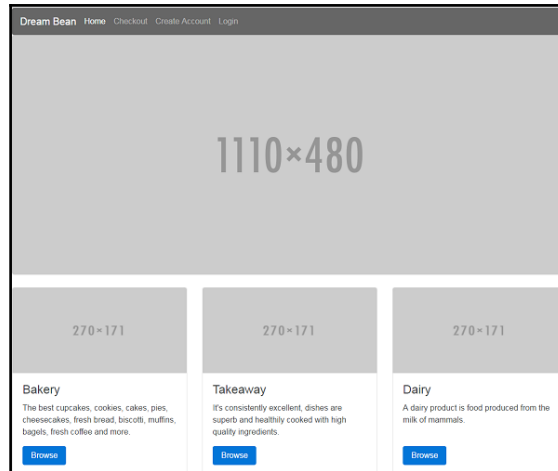
It looks nice, but Navbar is spanning the full width of the viewport. This is not, what managers from the Dream Bean want. The header must be centered and has a specific size.

Containers

We will wrap our Navbar in a container class to center it on the page:

```
<div class="container">
  <nav class="navbar navbar-dark" style="background-color: #666">
    <a class="navbar-brand" href="#">Dream Bean</a>
    <div class="nav navbar-nav">
      <a class="nav-item nav-link active" href="#">
        Home <span class="sr-only">(current)</span>
      </a>
      <a class="nav-item nav-link" href="#">Checkout</a>
      <a class="nav-item nav-link" href="#">Create Account</a>
      <a class="nav-item nav-link" href="#">Login</a>
    </div>
  </nav>
</div>
```

```
    </div>
  </nav>
</div>
```



Another correction they would like to have is that the header must be statically placed to top of the page. I used `navbar-fixed-top` class to place it to the top of the viewport:

```
<div class="container">
  <nav class="navbar navbar-fixed-top navbar-dark"
    style="background-color: #666">
    <a class="navbar-brand" href="#">Dream Bean</a>
    <div class="nav navbar-nav">
      <a class="nav-item nav-link active" href="#">
        Home <span class="sr-only">(current)</span>
      </a>
      <a class="nav-item nav-link" href="#">Checkout</a>
      <a class="nav-item nav-link" href="#">Create Account</a>
      <a class="nav-item nav-link" href="#">Login</a>
    </div>
  </nav>
</div>
```

You can use `navbar-fixed-bottom` class to reach out the same effect but on the ground of the page.

With last changes, the header spans the full width of viewport again. To fix that issue, we need to move `container` inside `navbar` to wrap up its content:

```
<nav class="navbar navbar-fixed-top navbar-dark"
```

```
        style="background-color: #666">
<div class="container">
  <a class="navbar-brand" href="#">Dream Bean</a>
  <div class="nav navbar-nav">
    <a class="nav-item nav-link active" href="#">
      Home <span class="sr-only">(current)</span>
    </a>
    <a class="nav-item nav-link" href="#">Checkout</a>
    <a class="nav-item nav-link" href="#">Create Account</a>
    <a class="nav-item nav-link" href="#">Login</a>
  </div>
</div>
</nav>
```

Our Navbar hides the part of viewport underneath, so we need to add a padding to compensate this issue:

```
body {
  padding-top: 51px;
}
```

If you Navbar was fixed at the bottom add padding for it as well:

```
body {
  padding-bottom: 51px;
}
```

Responsive Navbar

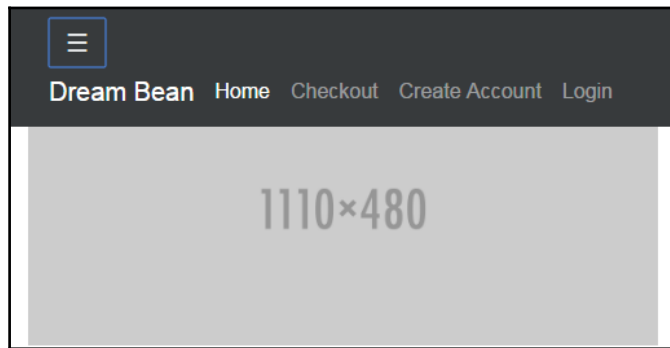
Another issue the staff of Dream Bean would like to fix is that the content must collapse at the given viewport width. Let's do it with `navbar-toggler` class along with `navbar-toggleable` classes, and their prefix sizes:

```
<nav class="navbar navbar-fixed-top navbar-dark bg-inverse">
  <div class="container">
    <button class="navbar-toggler hidden-sm-up"
      type="button" data-toggle="collapse"
      data-target="#exCollapsingNavbar">≡
    </button>
    <div class="collapse navbar-toggleable-xs"
      id="exCollapsingNavbar">
      <a class="navbar-brand" href="#">Dream Bean</a>
      <div class="nav navbar-nav">
        <a class="nav-item nav-link active" href="#">
          Home <span class="sr-only">(current)</span>
        </a>
```



```
    <a class="nav-item nav-link" href="#">Checkout</a>
    <a class="nav-item nav-link" href="#">Create Account</a>
    <a class="nav-item nav-link" href="#">Login</a>
  </div>
</div>
</div>
</nav>
```

As I mentioned earlier the `navbar-toggler` class helps to organize collapsible behave. The collapsible plugin uses information from `data-toggle` property to trigger the action and on element defined in `data-target`. The `data-target` keeps the id of an element contains with `navbar-toggleable` classes, and it prefixes size. Only with combination of all of them responsively collapsible header will work:

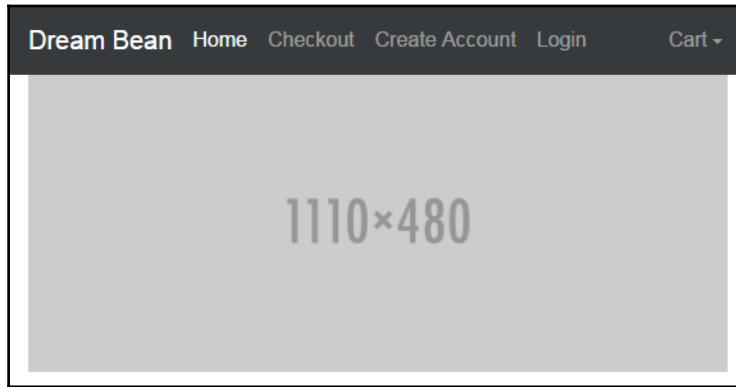


The Navbar Content Alignment

The last thing we need to fix is the placement of menu in Navbar. We can use any of `pull-left` or `pull-right` classes to align the menu and all other components in Navbar. The managers of Dream Bean want to add the Cart item with dropdown as last item of menu and aligned it to the right side:

```
<ul class="nav navbar-nav pull-xs-right">
  <li class="nav-item dropdown">
    <a class="nav-link dropdown-toggle"
      data-toggle="dropdown" href="#" role="button"
      aria-haspopup="true" aria-expanded="false">Cart</a>
    <div class="dropdown-menu">
      <span>The Cart Placeholder</span>
    </div>
  </li>
</ul>
```

I created separate menu group based and align it to the right with `pull-xs-right` on all sizes of layout:



Summary

In this chapter, we introduced the project we're going to be building over the course of this book. This information includes important aspects how to start project development from scratch.

We explore the most fundamental Grid component helping us to layout all other elements across the page.

We introduced the flexible Card component and built the plates contain categories of products from the building project.

We know how to use Nav and Navbar components to organize responsively collapsible header with the menu and how to customize it.

In *Chapter 3, Advanced Bootstrap Components and Customization*, we're going to explore more Bootstrap fundamentals and continue to build the project we started develop in this chapter.

3

Advanced Bootstrap Components and Customization

In this chapter, we continue to discover the world of Bootstrap 4. You meet new components, and we will continue to demonstrate usage of Bootstrap 4 by showcasing a project we started the build in the previous chapter. At the end of the chapter, you will have a solid understanding:

- How to displaying content with Jumbotron
- How create slideshow with Bootstrap
- How use Typography in text
- How create Input, Button, and List Groups
- Getting attention with Images and Labels
- Using Dropdown Menus and Tables

How to Capture Customers Attention?

The welcome page presents the core marketing messages to website users, and it needs to take extra attention from them. We can use two different components make that happens.

Display content with Jumbotron

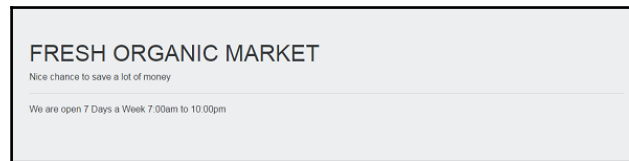
We can use the Jumbotron component to draw significant attention to marketing messages. It is a lightweight component, styled with large text and dense padding. We need to show:

- The marketing message.

- The slogan.
- The essential information for customers.

Here is a part of the welcome page includes the Jumbotron component:

```
<div class="jumbotron">
  <h1>FRESH ORGANIC MARKET</h1>
  <p>Nice chance to save a lot of money</p>
  <hr>
  <p>We are open 7 Days a Week 7:00am to 10:00pm</p>
</div>
```



You can force Jumbotron to take the full width of the page with the help of `jumbotron-fluid` class and `container` or `container-fluid` class within.

I use the standard HTML markup elements inside the Jumbotron, but it can look better with a different style.

Typography

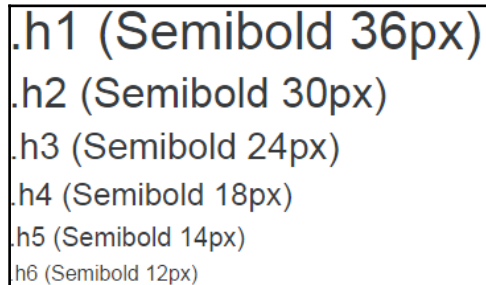
In the previous code, we used text elements without any classes, and see how Bootstrap renders them on the page. It uses the global default `font-size` of 16px and `line-height` 1.5. The “Helvetica Neue, Helvetica, Arial, Sans Serif” is default font-family in Bootstrap 4. Every elements has a `box-sizing` to prevent exceeding width due to padding or borders. The paragraph element has a bottom margin of 1rem. The body has a declared white background-color. Any page linked to Bootstrap 4 style sheets renders with those page-wide defaults.

Headings

All heading elements, `<h1>` through `<h6>`, have the weight of 500 and a line-height of 1.1. The Bootstrap developers remove `margin-top` from them, but add the `margin-bottom` of 0.5rem for easy spacing.

In cases when you need to display some inline text you can use `h1` through `h6` classes to style element who mimic headings.

```
<p class="h1">.h1 (Semibold 36px)</p>
<p class="h2">.h2 (Semibold 30px)</p>
<p class="h3">.h3 (Semibold 24px)</p>
<p class="h4">.h4 (Semibold 18px)</p>
<p class="h5">.h5 (Semibold 14px)</p>
<p class="h6">.h6 (Semibold 12px)</p>
```



.h1 (Semibold 36px)
.h2 (Semibold 30px)
.h3 (Semibold 24px)
.h4 (Semibold 18px)
.h5 (Semibold 14px)
.h6 (Semibold 12px)

Sub-Headings

If you require include sub-heading or secondary text smaller than original one you may use `<small>` tag.

```
<h1>Heading 1 <small>Sub-heading</small></h1>
<h2>Heading 2 <small>Sub-heading</small></h2>
<h3>Heading 3 <small>Sub-heading</small></h3>
<h4>Heading 4 <small>Sub-heading</small></h4>
<h5>Heading 5 <small>Sub-heading</small></h5>
<h6>Heading 6 <small>Sub-heading</small></h6>
```

Heading 1 Sub-heading

Heading 2 Sub-heading

Heading 3 Sub-heading

Heading 4 Sub-heading

Heading 5 Sub-heading

Heading 6 Sub-heading

We can show the faded smaller text with the help of `text-muted` class.

```
<h3>
  The heading
  <small class="text-muted">with faded secondary text</small>
</h3>
```

The heading with faded secondary text

Display Headings

When the standard heading is not enough and, you need to accent the users attention on something special I recommend to use the **display heading** classes. There are four different sizes of them, and that means you can render `<h1>` element with four different styles.

```
<h1 class="display-1">Display 1</h1>
<h1 class="display-2">Display 2</h1>
<h1 class="display-3">Display 3</h1>
<h1 class="display-4">Display 4</h1>
```

Display 1
Display 2
Display 3
Display 4

Lead

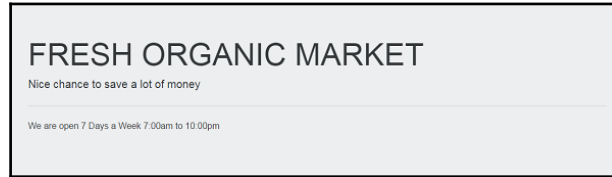
We need to add the `lead` class to any paragraph to stand out it of other text.

```
<p class="lead">  
This is the article lead text.  
</p>  
<p>  
This is the normal size text.  
</p>
```

This is the article lead text.
This is the normal size text.

Let's update our to make it looks better:

```
<div class="jumbotron">  
  <h1 class="display-3">FRESH ORGANIC MARKET</h1>  
  <p class="lead">Nice chance to save a lot of money</p>  
  <hr class="m-y-2">  
  <p>We are open 7 Days a Week 7:00am to 10:00pm</p>  
</div>
```

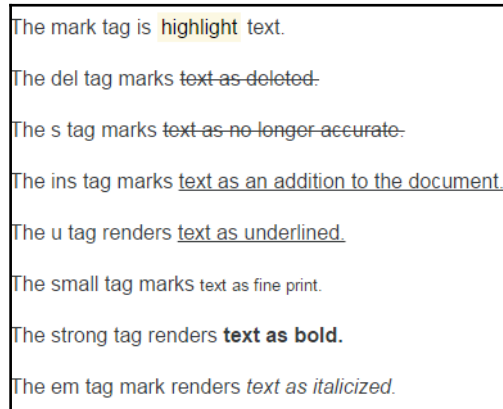


The marketing message looks gorgeous, the slogan in place, so we left untouched only the essential information for customers because we don't need to change it.

Inline Text Elements

Here is the set of different styles we can use as inline text:

```
<p>The mark tag is <mark>highlight</mark> text.</p>
<p>The del tag marks <del>text as deleted.</del></p>
<p>The s tag marks <s> text as no longer accurate.</s></p>
<p>The ins tag marks <ins>text as an addition to the document.</ins></p>
<p>The u tag renders <u>text as underlined.</u></p>
<p>The small tag marks <small>text as fine print.</small></p>
<p>The strong tag renders <strong>text as bold.</strong></p>
<p>The em tag mark renders <em>text as italicized.</em></p>
```



Abbreviations

To mark any text as an abbreviation or acronym, we can use `<abbr>` tag. It shows standing out of another text and provides the expanded version on hover with the help of the `title`

attribute.

```
<p>The Ubuntu is <abbr title="Operation System">OS</abbr>.</p>
```



The class `initialism` makes an abbreviation for a slightly smaller font size.

Blockquotes

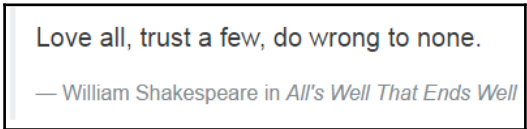
We can quote the content from another source within our document with the help of `blockquote` tag and class.

```
<blockquote class="blockquote">
  <p>Love all, trust a few, do wrong to none.</p>
</blockquote>
```

A screenshot of a blockquote rendered in Bootstrap. The text "Love all, trust a few, do wrong to none." is centered within a box with a light gray background and a thin border.


As well, we can add the author at the bottom of blockquote with nested `footer` and `cite` tags.

```
<blockquote class="blockquote">
  <p>Love all, trust a few, do wrong to none.</p>
  <footer class="blockquote-footer">William Shakespeare in
    <cite>All's Well That Ends Well</cite>
  </footer>
</blockquote>
```

A screenshot of a blockquote with a footer. The main text "Love all, trust a few, do wrong to none." is centered. Below it, the footer text "— William Shakespeare in All's Well That Ends Well" is also centered and rendered in a smaller, lighter font.

Do you prefer blockquotes aligned to right side? Let's use `blockquote-reverse` class:

```
<blockquote class="blockquote blockquote-reverse">
  <p>Love all, trust a few, do wrong to none.</p>
  <footer class="blockquote-footer">William Shakespeare in
    <cite>All's Well That Ends Well</cite>
  </footer>
</blockquote>
```



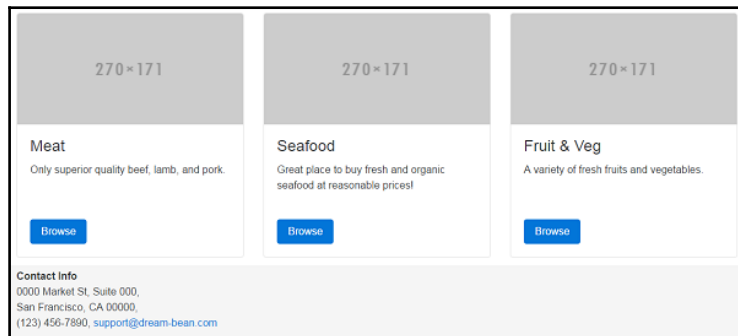
Love all, trust a few, do wrong to none.

William Shakespeare in *All's Well That Ends Well* —

Address

We use address element to display the customer contact information at the bottom of the page:

```
<footer class="footer">
  <div class="container">
    <address>
      <strong>Contact Info</strong><br>
      0000 Market St, Suite 000,<br>
      San Francisco, CA 00000,<br>
      (123) 456-7890,
      <a href="mailto:#">support@dream-bean.com</a>
    </address>
  </div>
</footer>
```



Display content with Carousel

Another component we can use to take customers extra attention is Carousel. It helps us create elegant and interactive image or text slideshow. The Carousel is a combination of different components, each of them playing very specific role.

Carousel Container

The container wraps all other content so the plugin JavaScript code can find it by `carousel` and `slide` classes. It must have an `id` for carousel controls and inner components to function proper. If you want the carousel to start animation when page loads use `data-ride="carousel"` property.

```
<div id="welcome-products"
      class="carousel slide" data-ride="carousel">
```

Carousel Inner

This container holds carousel items as scrollable content and marks with `carousel-inner` class.

```
<div class="carousel-inner" role="listbox">
```

Carousel Item

This one marked with `carousel-item` class keeps the content of slide like image, text or combination of them. You need to wrap the text-based content with `carousel-caption` container. The `active` class marks the item as initial and without it the carousel won't be visible.

```
<div class="carousel-item active">
  
  <div class="carousel-caption">
    <h3>Bread & Pastry</h3>
  </div>
</div>
```

Carousel Indicators

Carousel may have indicators to display and control the slideshow via click or tap to select a particular slide. Usually, it is an ordered list marked with `carousel-indicators` class.

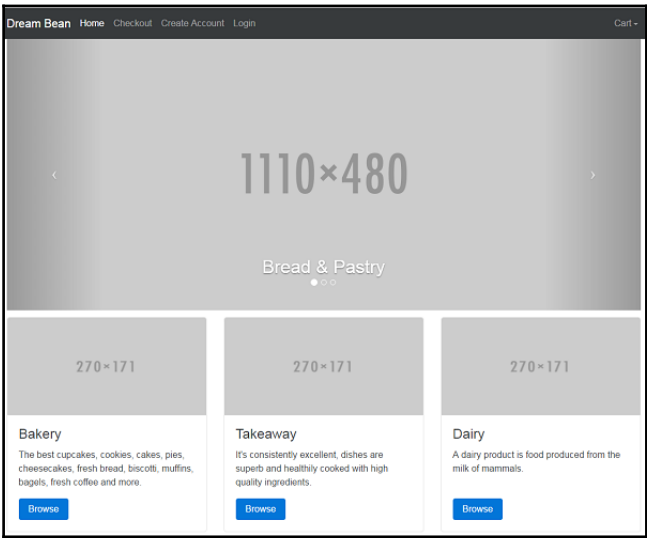
Every item of the list must have the `data-target` property keeps the carousel container id. Because of ordered list, you don't need to sort it. If you need to alter the slide position about the current location use `data-slide` property accepts the keywords `prev` and `next`. Another option is that use the `data-slide-to` property to pass the index of the slide. Use `active` class to mark the initial indicator.

```
<ol class="carousel-indicators">
  <li data-target="#welcome-products" data-slide-to="0"
      class="active"></li>
  <li data-target="#welcome-products" data-slide-to="1"></li>
  <li data-target="#welcome-products" data-slide-to="2"></li>
</ol>
```

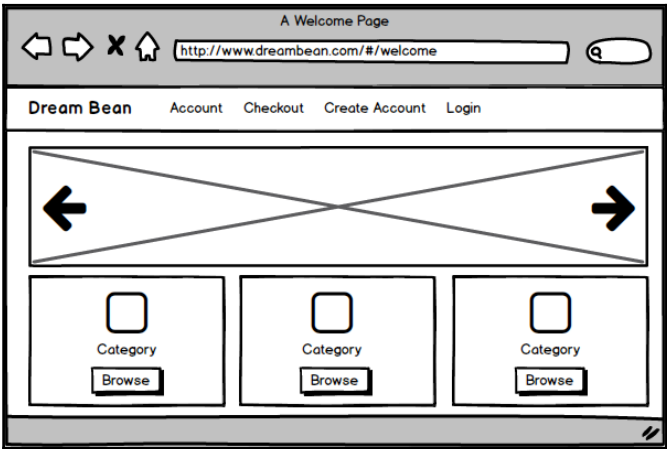
Carousel Controls

You can use an alternative way to display slides via carousel control buttons. In this case, the two anchor elements playing the role of the buttons. Add the `left` or the `right` classes to the particular button with the conjunction of `carousel-control`. Use carousel container id as a link in the `href` property. Set the `prev` or `next` to the `data-slide` property.

```
<a class="left carousel-control" href="#welcome-products"
    role="button" data-slide="prev">
  <span class="icon-prev" aria-hidden="true"></span>
  <span class="sr-only">Previous</span>
</a>
<a class="right carousel-control" href="#welcome-products"
    role="button" data-slide="next">
  <span class="icon-next" aria-hidden="true"></span>
  <span class="sr-only">Next</span>
</a>
```



Let's compare the final result and wireframe of the Welcome page.

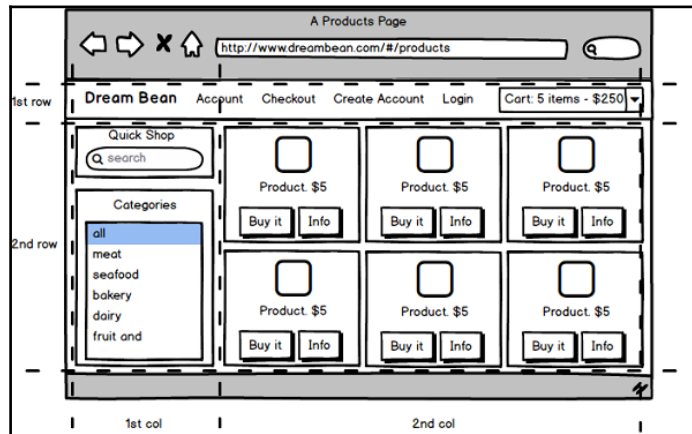


The wireframe of the Welcome Page

As you see they look absolutely the same. Actually, we finish with a Welcome page, and now it's time to move on the Products page development.

Products Page Layout

Let's have a look on wireframe of a Products page and imagine splitting it into rows and columns as we did for the Welcome page:



The first row still contains our navigation header, but other content I put into another single row. There are two columns with Quick Shop and Categories in one, and the grid includes set of products in another one. Why I split the Products page like that? The answer to my question is pretty straight forward. The Bootstrap always render content by rows and then by columns inside them. On devices with the small layout, the header in the first row usually collapsed into hamburger menu. At the bottom, it displays the second row with the Quick Shop, Categories and below the set of products aligned vertically.

I clone the last project and clean the code, but save the navigation header and footer, because I don't want to mix the development of the Products page with the original one. Let's talk about components in the first column.

Quick Shop Component

This component is just a search input with the button. I don't implement business logic, just design the page. This one based on the Card element we know from the Chapter 2. I would like to use **Input Group component**, so let's see what it can do?

Input Group


This one is a group of form controls and text combined in one line. It was designed to extend controls of form by adding text, buttons, or group of buttons on either side of the input field and align them against each other. Create an Input Group component is very easy, just wrap the input with element marked with `input-group` class and append or prepend another one with an `input-group-addon` class. You can use Input Group out of any form, but we need to mark input element with the `form-control` class to have it a with equals 100%.

- Use Input Group for the textual input elements only.

Text Addons

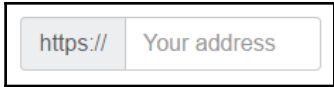
Here is an example of text field with appended addon:

```
<div class="input-group">
  <input type="text" class="form-control"
    placeholder="Price in USD">
  <span class="input-group-addon">.00</span>
</div>
```



Another example with prepended addon:

```
<div class="input-group">
  <span class="input-group-addon">https://</span>
  <input type="text" class="form-control"
    placeholder="Your address">
</div>
```



And finally we can combine all of them together:

```
<div class="input-group">
  <span class="input-group-addon">$</span>
  <input type="text" class="form-control"
```

```
        placeholder="Price per unit">
    <span class="input-group-addon">.00</span>
</div>
```

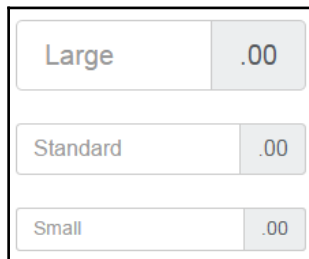


A Bootstrap input group. It consists of a light gray button with a dollar sign (\$), a text input field with the placeholder text "Price per unit", and a light gray button with the text ".00".

Sizing

There are two form sizing classes `input-group-lg` and `input-group-sm` to make Input Group bigger or smaller than the standard one. You need to apply one to element marked with `input-group` class and content within will automatically resize.

```
<div class="input-group input-group-lg">
  <input type="text" class="form-control"
    placeholder="Large">
  <span class="input-group-addon">.00</span>
</div>
<div class="input-group">
  <input type="text" class="form-control"
    placeholder="Standard">
  <span class="input-group-addon">.00</span>
</div>
<div class="input-group input-group-sm">
  <input type="text" class="form-control"
    placeholder="Small">
  <span class="input-group-addon">.00</span>
</div>
```



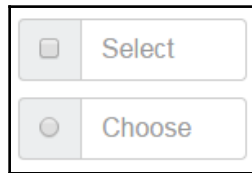
Three Bootstrap input groups stacked vertically, each enclosed in a black border. The first group is labeled "Large" and has a ".00" addon. The second group is labeled "Standard" and has a ".00" addon. The third group is labeled "Small" and has a ".00" addon. Each group consists of a text input field and a light gray button with the ".00" value.

Checkboxes and Radio Options Addons

We can use checkbox or radio option instead of text addons.

```
<div class="input-group">
  <span class="input-group-addon">
    <input type="checkbox">
  </span>
  <input type="text" class="form-control"
    placeholder="Select">
</div>

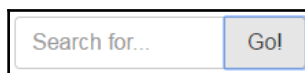
<div class="input-group">
  <span class="input-group-addon">
    <input type="radio">
  </span>
  <input type="text" class="form-control"
    placeholder="Choose">
</div>
```



Button Addons

The most familiar elements are buttons, and you can use them within the Input Group. Just add one extra level of complexity, and you get it.

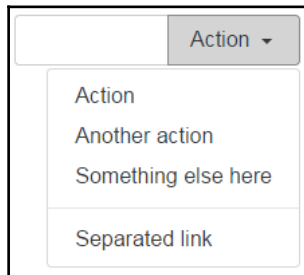
```
<div class="input-group">
  <input type="text" class="form-control"
    placeholder="Search for...">
  <span class="input-group-btn">
    <button class="btn btn-secondary" type="button">Go!</button>
  </span>
</div>
```



Dropdown Menu Addons

We can use button to show the dropdown menu. We will speak about Dropdown Menus a bit later in this Chapter.

```
<div class="input-group">
  <input type="text" class="form-control">
  <div class="input-group-btn">
    <button type="button"
      class="btn btn-secondary dropdown-toggle"
      data-toggle="dropdown">
      Action
    </button>
    <div class="dropdown-menu dropdown-menu-right">
      <a class="dropdown-item" href="#">Action</a>
      <a class="dropdown-item" href="#">Another action</a>
      <a class="dropdown-item" href="#">Something else here</a>
      <div role="separator" class="dropdown-divider"></div>
      <a class="dropdown-item" href="#">Separated link</a>
    </div>
  </div>
</div>
```

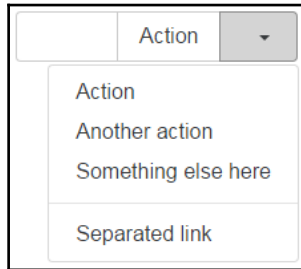


Segmented Buttons

Sometimes it could be useful to split button and dropdown menu, so that layout available as well.

```
<div class="input-group">
  <input type="text" class="form-control">
  <div class="input-group-btn">
    <button type="button" class="btn btn-secondary">Action</button>
    <button type="button" class="btn btn-secondary dropdown-toggle"
      data-toggle="dropdown">
    <span class="sr-only">Toggle Dropdown</span>&nbsp;
```

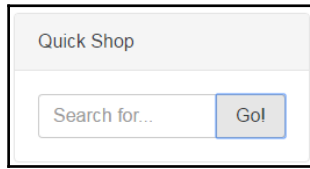
```
</button>
<div class="dropdown-menu dropdown-menu-right">
<a class="dropdown-item" href="#">Action</a>
<a class="dropdown-item" href="#">Another action</a>
<a class="dropdown-item" href="#">Something else here</a>
<div role="separator" class="dropdown-divider"></div>
<a class="dropdown-item" href="#">Separated link</a>
</div>
</div>
</div>
```



Now, when we know how to use Input Group let's create Quick Shop component.

```
<div class="container">
  <div class="row">
    <div class="col-md-3">
      <div class="card">
        <div class="card-header">
          Quick Shop
        </div>
        <div class="card-block">
          <div class="input-group">
            <input type="text" class="form-control"
              placeholder="Search for...">
            <span class="input-group-btn">
              <button class="btn btn-secondary"
                type="button">Go!</button>
            </span>
          </div>
        </div>
      </div>
    <!-- /.card -->
  </div>
  <!-- /.col -->
</div>
<!-- /.row -->
</div>
```

```
<!-- /.container -->
```



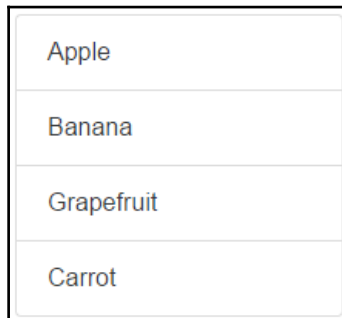
Categories Component

The Categories Component laying under the Quick Shop. I would like to use the **List Group** component to keep categories which customer can choose.

List Group

This one is a flexible component for displaying an unordered list of elements like simple items or custom content with easy. Just mark any unordered list element with `list-group` class and every item with `list-group-item` to quickly create the list group component.

```
<ul class="list-group">
  <li class="list-group-item">Apple</li>
  <li class="list-group-item">Banana</li>
  <li class="list-group-item">Grapefruit</li>
  <li class="list-group-item">Carrot</li>
</ul>
```



List with Labels

Sometimes we need to display a bit more information about every item like counts, activities etc. For that purpose we can add **Label** to each item and list group automatically position it to the right.

```
<ul class="list-group">
  <li class="list-group-item">
    <span class="label label-default label-pill
      pull-xs-right">15</span>
    Apple
  </li>
  <li class="list-group-item">
    <span class="label label-default label-pill
      pull-xs-right">5</span>
    Banana
  </li>
  <li class="list-group-item">
    <span class="label label-default label-pill
      pull-xs-right">0</span>
    Grapefruit
  </li>
  <li class="list-group-item">
    <span class="label label-default label-pill
      pull-xs-right">3</span>
    Carrot
  </li>
</ul>
```

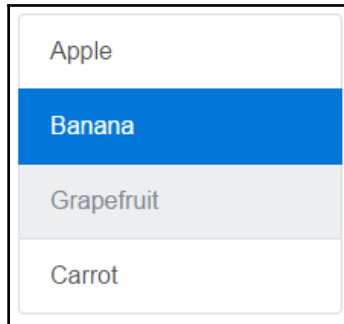
Apple	15
Banana	5
Grapefruit	0
Carrot	3

Linked List Group

We can quickly create a vertical menu with Linked List Group component. This kind of list based on `div` tag instead of `ul`. The whole item of this list is an anchor element, and it can be:

- Clickable
- Hoverable
- Highlighted with the help of an active class
- Disabled with the aid of class the same name

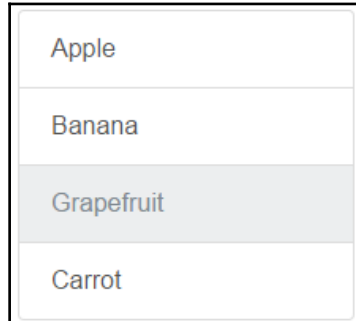
```
<div class="list-group">
  <a href="#" class="list-group-item">Apple</a>
  <a href="#" class="list-group-item active">Banana</a>
  <a href="#" class="list-group-item disabled">Grapefruit</a>
  <a href="#" class="list-group-item">Carrot</a>
</div>
```



Button List Group

If you prefer use buttons instead of anchor elements, just change the tag name.

```
<div class="list-group">
  <button type="button" class="list-group-item">Apple</button>
  <button type="button" class="list-group-item">Banana</button>
  <button type="button" class="list-group-item disabled">
    Grapefruit
  </button>
  <button type="button" class="list-group-item">Carrot</button>
</div>
```



- Usage of standard `btn` class in List Group is prohibited.

Contextual Classes

You can, as well, style individual list items with contextual classes. Just add a contextual class suffix to `list-group-item` class. The item with active class displays in a darkened version.

```
<div class="list-group">
  <a href="#" class="list-group-item
    list-group-item-success">Apple</a>
  <a href="#" class="list-group-item
    list-group-item-success active">Watermelon</a>
  <a href="#" class="list-group-item
    list-group-item-info">Banana</a>
  <a href="#" class="list-group-item
    list-group-item-warning">Grapefruit</a>
  <a href="#" class="list-group-item
    list-group-item-danger">Carrot</a>
</div>
```



Custom Content

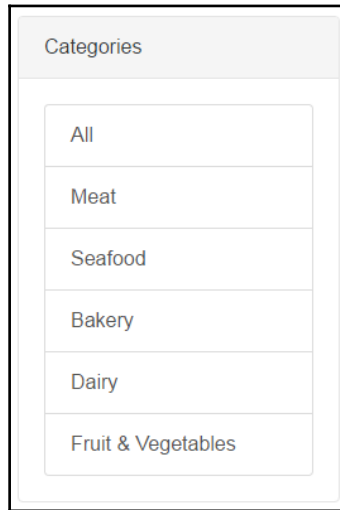
Finally, you can add any HTML within every item of List Group component, and make it clickable with anchor element. Bootstrap 4 provides `list-group-item-heading` and `list-group-item-text` classes for heading and the text content. The item with `active` class displays custom content in a darkened version.

```
<div class="list-group">
  <a href="#" class="list-group-item list-group-item-success">
    <h4 class="list-group-item-heading">Apple</h4>
    <p class="list-group-item-text">It is sweet.</p>
  </a>
  <a href="#" class="list-group-item list-group-item-success
    active">
    <h4 class="list-group-item-heading">Watermelon</h4>
    <p class="list-group-item-text">
      It is a fruit and a vegetable.
    </p>
  </a>
</div>
```




Now, it's time to create our Categories component.

```
<div class="card">
  <div class="card-header">
    Categories
  </div>
  <div class="card-block">
    <div class="list-group">
      <a href="#" class="list-group-item">All</a>
      <a href="#" class="list-group-item">Meat</a>
      <a href="#" class="list-group-item">Seafood</a>
      <a href="#" class="list-group-item">Bakery</a>
      <a href="#" class="list-group-item">Dairy</a>
      <a href="#" class="list-group-item">Fruit & Vegetables</a>
    </div>
  </div>
</div>
```



We are finished with first column, so let's go to develop the second one contains the grid with set of products.

Grid of Products

We need to display a set of products in a grid of rows and columns inside the second column.

Nested Rows

We can nest additional rows inside any column to create a more complex layout similar the one we have.

```
<div class="col-md-9">
  <div class="row">
    <div class="col-xs-12 col-sm-6 col-lg-4">
      <!-- The Product 1 -->
    </div>
    <!-- /.col -->
    <div class="col-xs-12 col-sm-6 col-lg-4">
      <!-- The Product 2 -->
    </div>
    <!-- /.col -->
    <div class="col-xs-12 col-sm-6 col-lg-4">
```

```
        <!-- The Product N -->
    </div>
    <!-- /.col -->
</div>
</div>
```

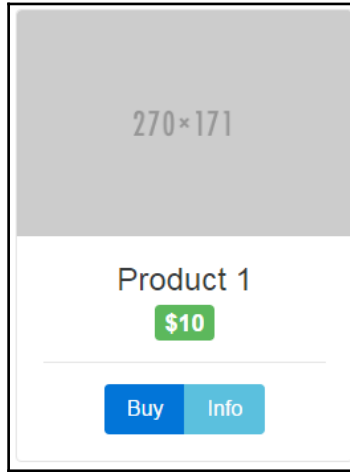
We create as many columns as we need within one row and Bootstrap will display them properly depends on viewport size:

- One column takes the whole size on extra small viewport size
- Two columns on small viewport
- Three columns on large and bigger viewports

Product Component

In a similar way, we use the Card to display information and controls in the Product Component.

```
<div class="card">
  
  <div class="card-block text-xs-center">
    <h4 class="card-title">Product 1</h4>
    <h4 class="card-subtitle">
      <span class="label label-success">$10</span>
    </h4>
    <hr>
    <div class="btn-group" role="group">
      <button class="btn btn-primary">Buy</button>
      <button class="btn btn-info">Info</button>
    </div>
  </div>
</div>
<!-- /.card -->
```



Let's talk a bit about elements we used here.

Image

As far as we use images in Card element, I think it's a good idea to talk about images with responsive behavior and image shapes.

Responsive Images

You can make any image responsive with `img-fluid` class. It applies the following to the picture and scales it with the parent element:

- Set `max-width` property equals `100%`
- Set `height` property to `auto`

```
<div class="container">
  <div class="row">
    <div class="col-md-3">
      
    </div>
  </div>
</div>
```



Image Shapes

In cases when you need render images

- With rounded corners use `img-rounded` class
- Within the circle use `img-circle` class
- As a thumbnail to use `img-thumbnail` class



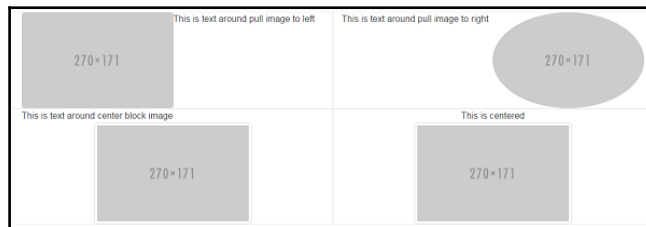
Image Alignment

To align images horizontally, we can use either text alignment or helper float classes:

- Use `text-*-center` classes on the parent of the picture to center it
- Use `center-block` class on the image to center it
- Use `pull-*-left` or `pull-*-right` classes to float the image to left or right accordingly

```
<div class="container">
  <div class="row">
    <div class="col-md-6 table-bordered">
      This is text around pull image to left
      
    </div>
```

```
<div class="col-md-6 table-bordered">
  This is text around pull image to right
  
</div>
<div class="col-md-6 table-bordered">
  This is text around center block image
  
</div>
<div class="col-md-6 text-xs-center table-bordered">
  This is centered<br>
  
</div>
</div>
</div>
```



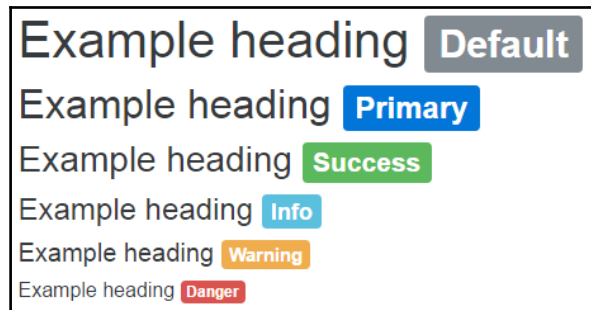
I use the `table-border` class in the code above only to display borders.

Label

If I need highlight some information in the string of text I will use **Label**. To create a Label I need apply `label` class together with contextual `label-*` to span element:

```
<div class="container">
  <div class="row">
    <div class="col-md-12">
      <h1>Example heading
        <span class="label label-default">Default</span>
      </h1>
      <h2>Example heading
        <span class="label label-primary">Primary</span>
      </h2>
      <h3>Example heading
        <span class="label label-success">Success</span>
      </h3>
    </div>
  </div>
</div>
```

```
</h3>
<h4>Example heading
  <span class="label label-info">Info</span>
</h4>
<h5>Example heading
  <span class="label label-warning">Warning</span>
</h5>
<h6>Example heading
  <span class="label label-danger">Danger</span>
</h6>
</div>
</div>
</div>
```



The Label uses relative font size of parent element to always scales to match the size of it. If you need Label looks like Badge use label-pill class to achieve the same effect.

```
<div class="container">
  <div class="row">
    <div class="col-md-12">
      <span class="label label-pill label-default">Default</span>
      <span class="label label-pill label-primary">Primary</span>
      <span class="label label-pill label-success">Success</span>
      <span class="label label-pill label-info">Info</span>
      <span class="label label-pill label-warning">Warning</span>
      <span class="label label-pill label-danger">Danger</span>
    </div>
  </div>
</div>
```



Button Group

We can group buttons together either horizontally or vertically with **Button Group** component. Buttons oriented horizontally by default. To create the Button Group use buttons with `btn` class in container with `btn-group` class.

```
<div class="container">
  <div class="row">
    <div class="col-md-12">
      <div class="btn-group" role="group">
        <button type="button"
          class="btn btn-default">Left</button>
        <button type="button"
          class="btn btn-secondary">Middle</button>
        <button type="button"
          class="btn btn-danger">Right</button>
      </div>
    </div>
  </div>
</div>
```

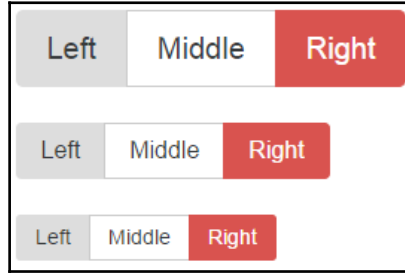


Sizing

There are two sizes to make Button group bigger or smaller than the standard size. Add either `btn-group-lg` or `btn-group-sm` classes to the button group to change size of all buttons in the group at once.

```
<div class="btn-group btn-group-lg" role="group">
  <button type="button" class="btn btn-default">Left</button>
  <button type="button" class="btn btn-secondary">Middle</button>
  <button type="button" class="btn btn-danger">Right</button>
</div><br><br>
<div class="btn-group" role="group">
  <button type="button" class="btn btn-default">Left</button>
  <button type="button" class="btn btn-secondary">Middle</button>
  <button type="button" class="btn btn-danger">Right</button>
</div><br><br>
<div class="btn-group btn-group-sm" role="group">
  <button type="button" class="btn btn-default">Left</button>
  <button type="button" class="btn btn-secondary">Middle</button>
  <button type="button" class="btn btn-danger">Right</button>
```

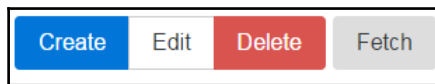

</div>



Button Toolbar

We can combine button groups into **Button Toolbar** for more complex components.

```
<div class="btn-toolbar" role="toolbar">
  <div class="btn-group" role="group">
    <button type="button" class="btn btn-primary">Create</button>
    <button type="button" class="btn btn-secondary">Edit</button>
    <button type="button" class="btn btn-danger">Delete</button>
  </div>
  <div class="btn-group" role="group">
    <button type="button" class="btn btn-default">Fetch</button>
  </div><br><br>
</div>
```



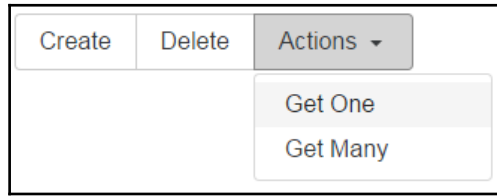
Nesting Dropdown

We can make Dropdown as a part of Button Group via nesting it into another button group.

```
<div class="btn-group" role="group">
  <button type="button" class="btn btn-secondary">Create</button>
  <button type="button" class="btn btn-secondary">Delete</button>

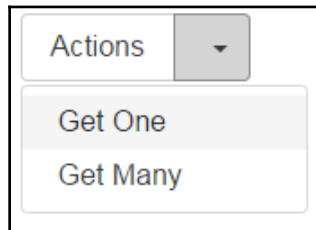
  <div class="btn-group" role="group">
    <button id="btnGroupDrop1" type="button"
      class="btn btn-secondary dropdown-toggle"
      data-toggle="dropdown" aria-haspopup="true">
```

```
        aria-expanded="false">
        Actions
    </button>
    <div class="dropdown-menu" aria-labelledby="btnGroupDrop1">
        <a class="dropdown-item" href="#">Get One</a>
        <a class="dropdown-item" href="#">Get Many</a>
    </div>
</div>
</div>
```



As well, you can create Split Dropdown Menu component with Button Group:

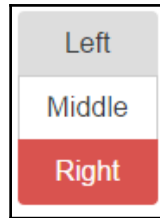
```
<div class="btn-group" role="group">
    <button type="button" class="btn btn-secondary">Actions</button>
    <button id="btnGroupDrop1" type="button"
        class="btn btn-secondary dropdown-toggle"
        data-toggle="dropdown" aria-haspopup="true"
        aria-expanded="false">
        <span class="sr-only">Toggle Dropdown</span>&nbsp;
    </button>
    <div class="dropdown-menu" aria-labelledby="btnGroupDrop1">
        <a class="dropdown-item" href="#">Get One</a>
        <a class="dropdown-item" href="#">Get Many</a>
    </div>
</div>
```



Vertical Button Group

If you need orient Button Group vertically replace `btn-group` with `btn-group-vertical` class.

```
<div class="btn-group-vertical" role="group">
  <button type="button"
    class="btn btn-default">Left</button>
  <button type="button"
    class="btn btn-secondary">Middle</button>
  <button type="button"
    class="btn btn-danger">Right</button>
</div>
```



The vertical Button Group doesn't support Split Dropdown Menus.

Dropdown Menu

We talk a lot about **Dropdown Menu**, so let's have a look at it closer. It is a toggleable overlay for displaying a list of links. The Dropdown Menu is a combination of several components.

Dropdown Container

This one wraps all other elements. Usually, it is a `div` element with `dropdown` class, or any another one uses `position: relative`.

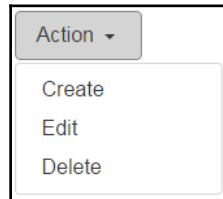
Dropdown Trigger

It is any item that the user can click or tap to expand the dropdown. We need to mark it with `dropdown-toggle` class and set `data-toggle="dropdown"` property.

Dropdown Menu with Items

The Dropdown Menu itself is a combination of elements with `dropdown-item` classes, and wrapper contains all of them marked with `dropdown-menu` class. It is a listless component. For menu items, you can use anchor or button elements.

```
<div class="dropdown">
  <button class="btn btn-secondary dropdown-toggle"
    type="button" id="dropdownMenu1"
    data-toggle="dropdown" aria-haspopup="true"
    aria-expanded="false">
    Action
  </button>
  <div class="dropdown-menu" aria-labelledby="dropdownMenu1">
    <a class="dropdown-item" href="#">Create</a>
    <a class="dropdown-item" href="#">Edit</a>
    <a class="dropdown-item" href="#">Delete</a>
  </div>
</div>
```

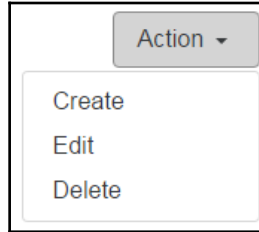


Menu Alignment

The dropdown menu aligned to left by default. If you need align it to the right side than you need to apply `dropdown-menu-right` class to dropdown menu. I add `text-xs-right` class to the parent element to align the whole component to the right side.

```
<div class="col-md-3 text-xs-right">
  <div class="dropdown">
    <button class="btn btn-secondary dropdown-toggle"
      type="button" id="dropdownMenu1"
      data-toggle="dropdown" aria-haspopup="true"
      aria-expanded="false">
      Action
    </button>
    <div class="dropdown-menu dropdown-menu-right"
      aria-labelledby="dropdownMenu1">
      <a class="dropdown-item" href="#">Create</a>
```

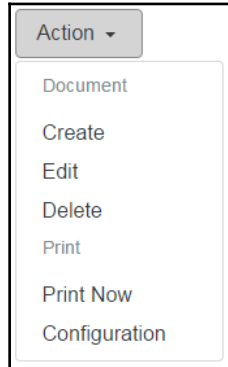
```
<a class="dropdown-item" href="#">Edit</a>
<a class="dropdown-item" href="#">Delete</a>
</div>
</div>
</div>
```



Menu Headers and Dividers

The Dropdown Menu may have several header elements. You can add them with help of heading elements and dropdown-header classes.

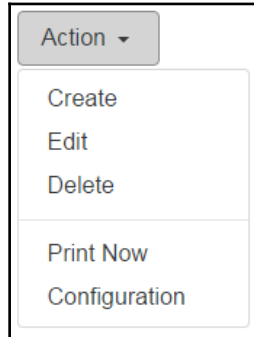
```
<div class="dropdown">
  <button class="btn btn-secondary dropdown-toggle" type="button"
    id="dropdownMenu1" data-toggle="dropdown"
    aria-haspopup="true" aria-expanded="false">
    Action
  </button>
  <div class="dropdown-menu" aria-labelledby="dropdownMenu1">
    <h6 class="dropdown-header">Document</h6>
    <a class="dropdown-item" href="#">Create</a>
    <a class="dropdown-item" href="#">Edit</a>
    <a class="dropdown-item" href="#">Delete</a>
    <h6 class="dropdown-header">Print</h6>
    <a class="dropdown-item" href="#">Print Now</a>
    <a class="dropdown-item" href="#">Configuration</a>
  </div>
</div>
```



Menu Divider

We can segregate group of menu items not only with headers but with dividers as well. Use `dropdown-divider` class to mark menu items as dividers.

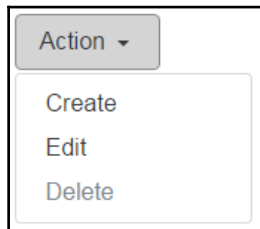
```
<div class="dropdown">
  <button class="btn btn-secondary dropdown-toggle" type="button"
    id="dropdownMenu1" data-toggle="dropdown"
    aria-haspopup="true" aria-expanded="false">
    Action
  </button>
  <div class="dropdown-menu" aria-labelledby="dropdownMenu1">
    <a class="dropdown-item" href="#">Create</a>
    <a class="dropdown-item" href="#">Edit</a>
    <a class="dropdown-item" href="#">Delete</a>
    <div class="dropdown-divider"></div>
    <a class="dropdown-item" href="#">Print Now</a>
    <a class="dropdown-item" href="#">Configuration</a>
  </div>
</div>
```



Disabled Menu Items

If that is necessary we can disable menu items to style them as disabled via `disabled` class.

```
<div class="dropdown">
  <button class="btn btn-secondary dropdown-toggle" type="button"
    id="dropdownMenu1" data-toggle="dropdown"
    aria-haspopup="true" aria-expanded="false">
    Action
  </button>
  <div class="dropdown-menu" aria-labelledby="dropdownMenu1">
    <a class="dropdown-item" href="#">Create</a>
    <a class="dropdown-item" href="#">Edit</a>
    <a class="dropdown-item disabled" href="#">Delete</a>
  </div>
</div>
```



Table

There are new classes to build consistently styled and responsive tables. Because we need

Table to design Shopping Cart component I would like to discover it now. It is an opt-in, so it's very easy to transform any table to Bootstrap Table via adding `table` class. As result we have a basic table with horizontal dividers.

```
<table class="table">
  <thead>
    <tr>
      <th>#</th>
      <th>First Name</th>
      <th>Last Name</th>
      <th>Username</th>
    </tr>
  </thead>
  <tfoot>
    <tr>
      <th colspan="4">Number <strong>2</strong></th>
    </tr>
  </tfoot>
  <tbody>
    <tr>
      <th scope="row">1</th>
      <td>Mark</td>
      <td>Otto</td>
      <td>@mdo</td>
    </tr>
    <tr>
      <th scope="row">2</th>
      <td>Jacob</td>
      <td>Thornton</td>
      <td>@fat</td>
    </tr>
  </tbody>
</table>
```

#	First Name	Last Name	Username
1	Mark	Otto	@mdo
2	Jacob	Thornton	@fat
Number 2			

Inverse Table

The `table-inverse` class inverts the colors of the table.

```
<table class="table table-inverse">
```

#	First Name	Last Name	Username
1	Mark	Otto	@mdo
2	Jacob	Thornton	@fat
Number 2			

Striped Rows

We can alter the background colors of rows with `table-striped` class.

```
<table class="table table-striped">
```

#	First Name	Last Name	Username
1	Mark	Otto	@mdo
2	Jacob	Thornton	@fat
Number 2			

Bordered Table

If you need table with all borders around use `table-bordered` class.

```
<table class="table table-bordered">
```

#	First Name	Last Name	Username
1	Mark	Otto	@mdo
2	Jacob	Thornton	@fat
Number 2			

Hoverable Rows

To achieve the “hover” effect while mouse over the rows of the table use `table-hover` class.

```
<table class="table table-hover">
```

#	First Name	Last Name	Username
1	Mark	Otto	@mdo
2	Jacob	Thornton	@fat
Number 2			

Table Head Options

There are two classes to change the of `thead` element of the `table`. Add the `thead-default` class to apply slightly gray background color:

```
<table class="table">  
  <thead class="thead-default">
```

#	First Name	Last Name	Username
1	Mark	Otto	@mdo
2	Jacob	Thornton	@fat
Number 2			

The `thead-inverse` class inverts the text and background colors of `thead`.

```
<table class="table">
  <thead class="thead-inverse">
```

#	First Name	Last Name	Username
1	Mark	Otto	@mdo
2	Jacob	Thornton	@fat
Number 2			

Small Table

We can half the padding of the table to make it smaller with `table-sm` class.

```
<table class="table table-sm">
```

#	First Name	Last Name	Username
1	Mark	Otto	@mdo
2	Jacob	Thornton	@fat
Number 2			

Contextual Classes

There are five contextual classes to apply to individual rows or cells: `table-active`,

`table-success`, `table-info`, `table-warning`, and `table-danger`.

Responsive Tables

The responsive tables support horizontal scrolling on small and extra small devices (under 768px). On devices bigger than small you don't see any differences. Wrap ant table with div element with the `table-responsive` class to achieve that effect.

```
<div class="table-responsive">
  <table class="table">
    ...
  </table>
</div>
```

Reflow Tables

There is class `table-reflow` to help mke contents of atable reflow.

```
<table class="table table-reflow">
  <thead>
    <tr>
      <th>#</th>
      <th>First Name</th>
      <th>Last Name</th>
      <th>Username</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th scope="row">1</th>
      <td>Mark</td>
      <td>Otto</td>
      <td>@mdo</td>
    </tr>
    <tr>
      <th scope="row">2</th>
      <td>Jacob</td>
      <td>Thornton</td>
      <td>@fat</td>
    </tr>
  </tbody>
</table>
```

#	1	2
First Name	Mark	Jacob
Last Name	Otto	Thornton
Username	@mdo	@fat

Shopping Cart Component

We didn't touch the last component on wireframe of Products page that is a Shopping Cart. This one is the union of the cart information and dropdown contains the table of items the customer added into the cart.

We display the cart information as a text of the button:

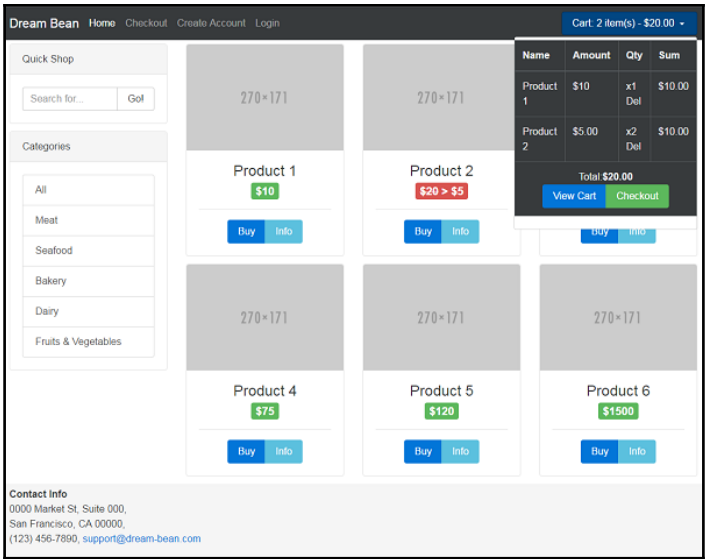
```
<button class="btn btn-primary dropdown-toggle" type="button"
        id="cartDropdownMenu" data-toggle="dropdown"
        aria-haspopup="true" aria-expanded="false">
    Cart: 2 item(s) - $20.00
</button>
```

I use inversed, bordered table to print out set of productes cusomer added into the cart:

```
<div class="dropdown-menu dropdown-menu-right"
    aria-labelledby="cartDropdownMenu">
    <table class="table table-bordered table-inverse">
        <thead>
            <tr>
                <th>Name</th><th>Amount</th><th>Qty</th><th>Sum</th>
            </tr>
        </thead>
        <tfoot>
            <tr>
                <td colspan="4" style="text-align:center">
                    Total:<strong>$20.00</strong><br>
                    <div class="btn-group">
                        <button class="btn btn-primary">View Cart</button>
                        <button class="btn btn-success">Checkout</button>
                    </div>
                </td>
            </tr>
        </tfoot>
    </table>
```

```
</tfoot>
<tbody>
  <tr>
    <td>Product 1</td><td>$10</td><td>x1<br>
      <span class="delete-cart">Del</span>
    </td>
    <td>$10.00</td>
  </tr>
  <tr>
    <td>Product 2</td><td>$5.00</td><td>x2<br>
      <span class="delete-cart">Del</span>
    </td>
    <td>$10.00</td>
  </tr>
</tbody>
</table>
</div>
```

I combined everything we learned and here is how the Products page looks like for now:



Summary

We've covered a lot in this Chapter, and it's time to interrupt our journey, take a break, and recap it all.

Bootstrap allowed us to capture customer attention with Jumbotron and Carousel slideshow quite easy.

We also looked at the powerful responsive grid system included with Bootstrap and created a simple two-column layout. While we were doing this, we learned about the five different column class prefixes as well as nesting our grid. To adapt our design, we discovered some of the helper classes included with the framework to allow us to float, center, and hide elements.

In this chapter, we saw in detail how to use Input, Button, and List Groups in our project. A simple but powerful component like Dropdown and Table helped us to create our components quickly and more efficiently.

In *Chapter 4, Creating the template*, we're going to discover more Bootstrap fundamentals and continue to build the project we started to develop in this chapter.

In this chapter, the readers will learn how to create a UI template using some built-in Angular 2 directives. The readers will become familiar with the template syntax. We will show how to bind properties and events in an HTML page and transform display using pipes.

4

Creating the Template

In this chapter, we'll learn how to build a UI template using built-in Angular 2 directives. You'll become familiar with the template syntax, and how to bind properties and events in an HTML page and transform display using pipes. Of course, we need to discuss design principals standing behind the Angular 2.

At the end of the chapter, you will have a solid understanding of :

- Template expression
- Various binding types
- Input and Output properties
- Using built-in directives
- Local template variables
- Pipe and Elvis operator
- Custom pipes
- Design components of our application

Dive deeper in Angular 2

We read mostly three chapters and didn't touch Angular 2 yet. I think it's time to invite Angular 2 on stage to demonstrate how this framework can help us in creating components for our project. As I mentioned in Chapter 1, the architecture of Angular 2 builds on top of web components standard so we can define custom HTML selectors and program them. That means we can create a set of Angular 2 elements to use them in the project. In previous chapters, we designed and developed two pages, and you can find many repetitive markups so that we can reuse our Angular 2 components there as well.

Open terminal and move to `ecommerce-carousel` folder. Install NPM modules if its

necessary via command:

```
npm init
```

You need to run the following command to launch the `lite-server` and open a web browser with a Welcome Page:

```
npm start
```

Now open Microsoft Visual Studio Code and open `app.component.html` from `app` folder. We are ready to analyze the Welcome Page.

Welcome Page Analysis

The Welcome Page has quite simple structure, so I would like to create the following Angular 2 components to encapsulate inside the current markup and future business logic:

- Navbar with menus
- Slideshow based on Carousel Bootstrap component
- Grid of Products based on Card Bootstrap component

I will follow the Angular 2 Style Guide (<https://angular.io/docs/ts/latest/guide/style-guide.html>) while developing our project to keep application code cleaner, easy to read and maintain. I recommend following my example on your plans otherwise the development results could be unpredictable and extremely costly.

Single Responsibility Principle

We will apply the **Single Responsibility Principle** to all aspects of the project, so whenever we need to create a Component or Service, we will create the new file for it and try to keep inside maximum 400 lines of code. Benefits of keeping one component per file is evident:

- Make code more reusable and less error prone
- Easy to read, test and maintain
- Prevents collisions with team in source control
- Avoid unwanted code coupling

- The Component Router can lazily load it at runtime

Naming conventions

There is not a secret that the **Naming Conventions** are crucial to readability and maintainability. The ability to find files and understand what they contain may have a significant impact on future development, so we should be consistent and descriptive in naming and organizing files to find content at a glance. The conventions comprise the following rules:

- The recommended pattern for all features describes the name then its type:
`feature.type.ts`
- There are words in descriptive name should be separated by dashes: `feature-list.type.ts`
- There are well-known type names include `service`, `component`, `directive`, and `pipe`: `feature-list.service.ts`

Barrels

There are barrel modules – TypeScript files that imports, aggregates, and re-exports other modules. The purpose of them is only one – reduce the number of import statements in code. They provide a consistent pattern to introduce everything exported in the barrel from a folder. Considered name of this file is `index.ts`.

Application Structure

We keep application code in the `app` folder. For easy and quick access to files, it is recommended to maintain the folder structure flat as long as possible until there is a clear value in creating a new folder.

Folders-by-Feature Structure

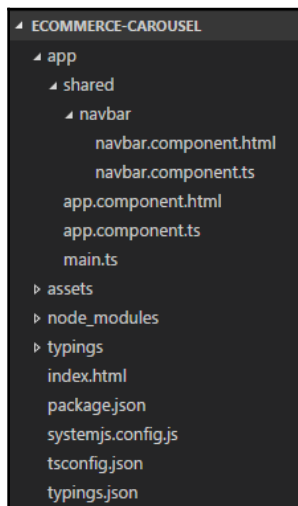
For small projects, you can save all files in the `app` folder. Our project has many features, so we put each of them in their folders, include TypeScript, HTML, Style Sheet, and Spec files. The name of the each folder represents the feature it has.

Shared Folder

There are some features we can use in multiple places. Better move them into the shared folder and separate them into folders if necessary. If in your project features exists, define the overall layout to save them here as well.

Navigation Component

There is a navigation component needed for the entire application, so I presume we need to create a shared folder, the navbar folder, files `navbar.component.ts` and `navbar.component.html`, so here the folder structure of our project as it stands now:



Open the `navbar.component.ts` file and paste the following content:

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

@Component({
  selector: 'db-navbar',
  templateUrl: 'app/shared/navbar/navbar.component.html'
})
export class NavbarComponent {}
```

In the code, we just defined `NavbarComponent` class with a `@Component` decorator to tell Angular that the class, which it attached to, is a component. We use `import` statement here

to specify the module, where TypeScript compiler can find the `@Component` decorator.

Decorators

The decorators are a proposed standard for ECMAScript 2016 and available as crucial part of TypeScript defining reusable structural pattern. Each decorator follows the form of `@expression`. The `expression` is a function evaluates at runtime with information about the decorated statement to change the behave and state of it. We can use a **decorator function**, which returns as a result of evaluation of the `expression` to customize how decorator applies to a declaration. It is possible to attach one or multiple decorators to any class, method, accessor, property, or parameter declarations.

The `@Component` is a class decorator applies at runtime to the constructor of `NavbarComponent` class for following purposes:

- To modify the class definition with a set of parameters passing through
- To add proposed methods organizing the component lifecycle

We must define the `selector` parameter for every `@Component` decorator and use `kebab-case` for naming it. The **style guide** recommends identifying component as elements via the `selector` because it provides consistency for components that represent content with a template. I use `db-navbar` selector name for `NavigationComponent` as a combination of:

- The `db` prefix displays the Dream Bean company name abbreviations.
- The `navbar` as name of the feature



Always use the prefix for selector name to prevent name collision with components from other libraries

The template is required part of the `@Component` decorator because we associate it with putting content on the page. You can supply the `template` as an inline string in the code or `templateUrl` as an external resource. It is better to keep the content of template as an external resource:

- When it has more than three lines
- Because some editors do not support the syntax hints for inline templates

- It is easier to read the logic of component when not mixed with inline templates

Now, open the `app.component.html` and find the `nav` element on the top. Cut it with content and paste into `navbar.component.html` and replace it with:

```
<db-navbar></db-navbar>
```

Now we need to add the `NavbarComponent` into `AppComponent`. Open `app.component.ts` to add reference on `NavbarComponent` to the `AppComponent`:

```
import { Component } from '@angular/core';
import {bootstrap} from '@angular/platform-browser-dynamic';

import {NavbarComponent} from './navbar/navbar.component';

@Component({
  selector: 'my-app',
  templateUrl: 'app/app.component.html',
  directives: [NavbarComponent]
})
export class AppComponent {}
```

Tree of Components

Every Angular application has a top level element to display the content. In our application, it is an `AppComponent`. In Chapter 3 we split the Welcome Page into Bootstrap components, now we move them into separate modules, and compose them back with the help of an Angular framework. The Angular framework renders application as a tree of Components, from a top level element, its children and further down. When we need to add a child to any component, we do register it via `directives` property.

Let's come back to the `navbar.component.html` to find other repetitive elements. In the place where we display navigation bar we have navigation items:

```
<div class="nav navbar-nav">
  <a class="nav-item nav-link active" href="#">
    Home <span class="sr-only">(current)</span>
  </a>
  <a class="nav-item nav-link" href="#">Checkout</a>
  <a class="nav-item nav-link" href="#">Create Account</a>
  <a class="nav-item nav-link" href="#">Login</a>
</div>
```

Because we have duplicates in the markup I propose to create an array of links and keep

them inside the `NavbarComponent` as a property, so Angular can display them here.

NavItem Object

I suggest create a separate class `NavItem` to keep information about navigation, because each item should have `href`, `label`, and `active` properties:

```
export class NavItem {
  // Navigation link
  href: string;
  // Navigation Label
  label: string;
  // Status of Navigation Item
  active: boolean;
}
```

Copy and paste the code in between the top of `NavbarComponent` class and last import statement. Now we can add property `navItems` into `NavbarComponent` that exposes the navigation items:

```
@Component({
  selector: 'db-navbar',
  templateUrl: 'app/shared/navbar/navbar.component.html'
})
export class NavbarComponent {
  // App name
  appName: string = 'Dream Bean';
  // Navigation items
  navItems: NavItem[] = [
    {href: '#', label: 'Home', active: true},
    {href: '#', label: 'Checkout', active: false},
    {href: '#', label: 'Create Account', active: false},
    {href: '#', label: 'Login', active: false}
  ];
}
```

I add the `appName` property to keep the application name out of the template as well. We are ready to use the data binding, but before let's take a closer look at Template Expressions and Statements.

Template Expressions

The **Template Expression** is the central part of data binding. Its primary purpose is to

execute expression to produce a value so that Angular can assign it to a binding property of HTML element, directive or component. We can put expression in template in two forms:

- Within the interpolation curly braces. The Angular first evaluates the content inside the braces and then converts to a string: `{{a + 1 - getVal()}}`
- Within the quotes when setting a property of view element to the value of template expression: `<button [disabled]="isUnchanged">Disabled</button>`

The Template Expression bases on JavaScript-like language. We can use any JavaScript expressions with following restrictions:

- It is prohibited to use assignments like `=`, `+=`, `-=`
- Do not use `new` keyword
- Do not create chaining expressions with `;` or `,`
- Avoid usage of increment `++` and decrement `--` operators
- Bitwise operators `|` and `&` and new template expression operators `|` and `?` are not supported

Expression Context

The content of Template Expression only belongs to the component instance and cannot refer to variables or functions in the global context. The component instance provides everything the template expression can use. It is usually the context of the expression, but can include objects other than component like a template reference variable.

Template Reference Variable

The **Template Reference Variable** is a reference to a DOM element or directive within a template. You can use it as a variable with any native DOM element and Angular 2 components. We can reference it on the some one, on a sibling or any child elements. There is two forms of how can we define it:

- Within prefix hash (`#`) and variable name:

```
<input #product placeholder="Product ID">
<button (click)="findProduct(product.value)">Find</button>
```

- The canonical alternative with `ref-` prefix and variable name:

```
<input ref-product placeholder="Product ID">
<button (click)="findProduct(product.value)">Find</button>
```

In both places the variable `product` pass its value to an event handler.

Expression Guidelines

Authors of Angular Framework recommends following this guideline in your Template Expressions:

- It should change only the value of the target property. Changes to other application states are prohibited.
- It must be as quick as possible because it executes more often than other code. Consider to cache values of computation for better performance when the computation is expensive.
- Please avoid creating the complex template expressions. Usually, you can get value from the property or call the method. Move complex logic into the method of the component.
- Please create idempotent expressions that always return the same thing until one of its dependent values changes. It is not allowed to change dependent values in the period of the event loop.

Expression Operators

Template Expression Language includes a few operators for specific scenarios.

The Elvis Operator

There are very common situations where the data we want to bind to the view is undefined temporarily. Say, we render a template and simultaneously fetch data from the server. There is a period where the data is unclear since the fetch call is asynchronous. As the Angular doesn't know this by default, it throws an error. In the following markup we see that the `product` can be equals `null`:

```
<div>Product: {{product.name | uppercase}}</div>
```

The render view may fail with a null reference error or worse yet entirely disappear:

```
TypeError: Cannot read property 'name' of null in [null].
```


Every time, when you write markup, you need to analyze it. If you decide that the `product` variable must never be `null`, but it is `null` – you find the programming error that should be caught and fixed, so it is the reason to throw an exception. Contrariwise, a `null` value can be on the property from time to time, since the fetch call is asynchronous. In the last case, the view must render without exceptions, and `null` property path must display as blank. We can solve this problem with a few ways:

- Include undefined checks
- Make sure that data always have initial value

Both of them are useful and have merit but looks cumbersome. As an example, we can wrap code in `NgIf` and check existence of `product` variable and its properties:

```
<div *ngIf="product && product.name">
  Product: {{product.name | uppercase}}
</div>
```

This code noteworthy, but it becomes cumbersome and looks ugly especially if the path is long. No one solution is elegant like **Elvis operator**. The Elvis or **safe navigation operator** is a convenient way to protect template expression evaluation out of `null` or undefined exceptions in property path.

```
<div>Product: {{product?.name | uppercase}}</div>
```

The Angular stops expression evaluation when it hits the first null value, display blank, and application doesn't crash.

The Pipe Operator

One of the primary purposes of the template is displaying data. We can show the raw data with to string values directly to the view. But most of the time we need transform the raw data into a simple format, add currency symbol to raw floats, etc., so we understand that some values need a bit of message before display. I feel like; we desire lot of the same transformations in many applications. The Angular Framework gives us pipes – a way to write display-value transformations that we can declare in templates.

Pipes are simple functions that accept an input value and return a transformed value. We can use them within template expressions, using the pipe operator (`|`):

```
<div>Product: {{product.name | uppercase}}</div>
```

The `uppercase` is a pipe function we placed after the pipe operator. It is possible to chain

expressions through multiple pipes:

```
<div>Product: {{product.name | uppercase | lowercase}}</div>
```

Pipe chain always starts the transformation from the first pipe converts the product name into uppercase, then to lowercase. It is possible to pass the parameters to a pipe:

```
<div>Expiry Date: {{product.expDate | date:'longDate'}}</div>
```

Here we have a pipe with configuration argument dictates to transforms the expiry date into the long date format: August 2, 1969. There is list of common pipes available in Angular 2:

- The `async` subscribes to an Observable or Promise and returns the latest value it has emitted.
- The `date` formats a value to a string based on the requested format.
- The `i18nSelect` is a generic selector that displays the string that matches the current value.
- The `percent` formats a number as a local percent.
- The `uppercase` implements uppercase transforms to text.
- The `number` formats a number as local text. i.e. group sizing and the separator and other locale-specific configurations base on the active locale.
- The `json` transforms any input value using `JSON.stringify`. Useful for debugging.
- The `replace` creates a new String with some or all of the matches of a pattern replaced by a replacement.
- The `currency` formats a number as local currency.
- The `i18nPlural` maps a value to a string that pluralizes the value correctly.
- The `lowercase` transforms text to lowercase.
- The `slice` creates a new List or String containing only a subset (slice) of the elements.

The Custom Pipes

We can create the custom pipe similar to `json` for our needs as follow:

- Import `Pipe` and `PipeTransform` from Angular core module
- Create a class `JsonPipe` implements `PipeTransform`
- Apply the `@Pipe` decorator to `JsonPipe` class and give it a name `db-json`
- Write the `transform` function with input value of `string` type

Here is the final version of our pipe:

```
import {Pipe, PipeTransform} from '@angular/core';

@Pipe({name: 'db-json'})
export class JsonPipe implements PipeTransform {
  transform(value: any): string {
    return JSON.stringify(value);
  }
}
```

Now we need a component to demonstrate our pipe:

```
import {Component} from '@angular/core';
import {JsonPipe} from '../shared/json.pipe';

@Component({
  selector: 'receiver',
  template: `
    <h2>Receiver</h2>
    <p>Received: {{data | db-json}}</p>
  `,
  pipes: [JsonPipe]
})
export class PowerBoosterComponent {
  data: any = {x: 5, y: 6};
}
```

Template Statements

The **Template Statement** is another important part of data binding. We use template statements to respond to an event raised by binding targets like element, directive or component. It bases on JavaScript-like language such as the template expression, but

Angular parses it different because:

- It supports the basic assignment =
- It supports the chaining expressions with ; or ,

Statement Context

The Statement Expression, like the template expression, can refer only to the component instance to which it is binding event or to a template reference variable. You may use reserved `$event` in event binding statement that represents the payload of the raised event.

Statement Guidelines

Authors of Angular Framework recommends to avoid creating the complex statement expressions. Usually, you can assign a value to the property or call the method. Move complex logic into the method of the component.

Data Binding

I mentioned the **Data Binding** in Chapter 1 in passing, but now we dive deeper in this crucial tool of Angular Framework. The Data Binding is the mechanism to update parts of the template with binding markup via properties or methods of a Component.

The Data Binding flow supports three directions between data sources and targets HTML elements:

- The One-way binding from the data source to target HTML. This group includes Interpolation, Property, Attribute, Class, and Style binding types.

```
{{expression}}  
[target] = "expression"  
bind-target = "expression"
```

- The One-way binding from the target HTML to the data source. This one is event data binding.

```
(target) = "statement"  
on-target = "statement"
```

- The two-way data binding.

```
[(target)] = "expression"  
bindon-target = "expression"
```

The `target` is the directive or component input property receives data from outside. We must explicitly declare any input property before start uses it. There are two ways to do that:

Mark the property with `@Input` Decorator:

```
@Input() product: Product;
```

Identify property as element of `inputs` array of directive or component metadata:

```
@Component({  
  inputs: ['product']  
})
```

The hosting parent element can use `product` property name:

```
<div>  
  <db-product [product]="product"></db-product>  
</div>
```

It is possible to use **alias** for the property to get it the different public name from the internal one to meets conventional expectations:

```
@Input('bestProduct') product: Product;
```

Now any hosting parent element can use `bestProduct` property name instead of `product`:

```
<div>  
  <db-product [bestProduct]="product"></db-product>  
</div>
```

HTML Attributes vs. DOM Properties

The HTML attributes and DOM Properties are not the same things. We are using HTML attributes only to initialize DOM properties, and we cannot change their values later.



Template binding works with DOM properties and events, not HTML attributes.

Interpolation

When we need to show the property value of component on the page we use double curly brackets markup to tell Angular how to display it. Let's update our code inside the `navbar.component.html` to such way:

```
<a class="navbar-brand" href="#">{{appName}}</a>
```

Angular automatically pulls the value of the property `appName` from the `NavbarComponent` class and inserts it into the page. When the property changes the framework updates the page. Interpolation is just a syntactic sugar to make our live easy. In reality, it is one of the forms of Property Binding.

Property Binding

The **Property Binding** is a technique to set the property of an element, component or directive. We can change the markup above such this way:

```
<a class="navbar-brand" href="#" [innerHTML]="appName"></a>
```

We can change the classes via `ngClass` property:

```
<div [ngClass]="classes">Binding to the classes property</div>
```

Here is how can we change the property of the component or directive:

```
<product-detail [product]="currentProduct"></product-detail>
```

For the reason, that template expression may contain the malicious content the Angular sanitizes the values before displaying them. Neither interpolation nor property binding does not allow the HTML with script tags to leak into the web browser.

Attribute Binding

There are several attributes of HTML element do not have corresponding DOM properties like ARIA, SVG, and table span. If you try to write code like this:

```
<tr><td colspan="{{1 + 1}}">Table</td></tr>
```

You will immediately get the following exception because table data tag has `colspan` attribute, but does not have `colspan` property:

```
browser_adapter.js:77 EXCEPTION: Error: Uncaught (in promise): Template
```

```
parse errors:
Can't bind to 'colspan' since it isn't a known native property ("
<tr><td [ERROR ->]colspan="{{1 + 1}}">Three-Four</td></tr>
")
```

In that particular case, we can use **Attribute Binding** as part of Property Binding. It uses the prefix `attr` followed by the dot (`.`) and the name of the attribute. Everything else – the same:

```
<tr><td [attr.colspan]="1 + 1">Three-Four</td></tr>
```

Class Binding

The Angular provides support for the **Class Binding**. By analogy to Attribute Binding, we use the prefix `class`, optionally followed by a dot (`.`) and the name of a CSS class. The following statement:

```
<div class="meat special">Meat special</div>
```

We can replace it with binding to a string of the desired class name `meatSpecial`:

```
<div [class]="meatSpecial">Meat special</div>
```

Or add the template expression `isSpecial` evaluates true or false to tell Angular to add or remove class `special` from the target element:

```
<div [class.special]="isSpecial">Show special</div>
```



Use the **NgClass** directive for managing multiple class names at the same time.

Style Binding

It is possible to manage the styles of the target element via **Style Binding**. We use the prefix `style`, optionally followed by a dot (`.`) and the name of a CSS style property:

```
<button [style.color]="isSpecial?'red':'green'">Special</button>
```



Use the `NgStyle` directive when setting several inline styles at the same time.

The data always flows in one direction in the Property Binding, from data property of the component to the target element. We cannot use the Property Binding to get the value from the target element or call a method on the target element. If the element raises the events, we can listen to them via an Event Binding.

Event Binding

Any user action on the page is generating events, so the authors of Angular Framework introduced the **Event Binding**. The syntax of this binding quite simple and it consists a **Target Event** within the parentheses, equal sign, and a quoted template statement. The Target Event is the name of the event:

```
<button (click)="onSave()">Save</button>
```

There is exist the canonical format of Event Binding you can use if prefer. It supports the prefix `on-` in front of name without parentheses in such way:

```
<button on-click="onSave()">Save</button>
```

In cases when the name of event does not exist on the element or output property is unknown, Angular reports an “unknown directive” error.

We can use the information Event Binding transfers about the event via an **event object** `$event`. The Angular uses the Target Event to determines the shape of the `$event` thereby if DOM element generates an event, the `$event` is a DOM event object, and it contains `target` and `target.value` properties. Check this code:

```
<div #product>
  <input [value]="product.name"
        (input)="product.name=$event.target.value"><br>
  {{product.name}}
</div>
```

We define the local variable `product` and bind the value of input element to its name, and we attach the input event to listen to changes. When the user starts to type, the component generates the DOM input event, and the binding executes the statement.

Custom Events

The JavaScript provides a dozen number of events for the whole bunch of scenarios out of the box, but sometimes we want to fire our own, custom events for particular needs. It would be good to use them because custom events provide an excellent level of decoupling in the application. JavaScript provides **CustomEvent** that does all sort of awesome things, but Angular exposes **EventEmitter** class we can use in **directives** and **Components** to emit custom events. What we need is that to create a property of type `EventEmitter` and call `emit` method to fire the event. It is possible to passing in a message payload that can be anything. This property regarding Angular works as output because it fires events from the directive or component to outside. We must explicitly declare any output property before start uses it. There are two ways to do that:

Mark the property with `@Output` Decorator:

```
@Output() select:EventEmitter<Product>
```

Identify property as element of `outputs` array of directive or component metadata:

```
@Component({
  outputs: ['select']
})
```

If that necessary, we can use **alias** for the property to get it the different public name from the internal one to meets conventional expectations:

```
@Output('selected') select:EventEmitter<Product>
```

Presume the customer selects the product in the grid of products. We can listen to mouse click event in markup and handle it in `browse` method of the component:

```
<a class="btn btn-primary" (click)="browse(product)">Browse</a>
```

When the method handles the mouse event we can fire the custom event with selected product:

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

export class Product {
  name: string;
  price: number;
}

@Component({
  selector: 'db-product',
```

```
    templateUrl: 'app/product/product.component.html'
  })
  export class ProductComponent {
    @Input product: Product;
    @Output() select: EventEmitter<Product> =
      new EventEmitter<Product>();
    browse($event) {
      this.select.emit(<Product>$event);
    }
  }
}
```

From now any hosting parent component can bind to the `select` event firing by the `ProductComponent`:

```
<db-product [product]="product"
  (select)="productSelected($event)"></db-product>
```

When the `select` event fires, the Angular calls the `productSelected` method in parent component and pass the `Product` in the `$event` variable.

Two-way Data Binding

Most of the time we need only the one-way binding where data follows from component to view or vice versa. Usually, we do not capture input that needs to be applied back to the DOM, but in some scenarios, it might be very useful, thus why Angular supports **Two-way Data Binding**. As shown above, we can use the Property Binding to input the data into directive or component properties with help of square brackets:

```
<input [value]="product.selected"></input>
```

In the opposite direction is denoted by surrounding an event name with parentheses:

```
<input (input)="product.selected=$event.target.value">Browse</a>
```

We can combine those techniques to have the best of both worlds with the help of `ngModel` directive. There are two forms of two-way data binding:

- Where the parentheses go inside the brackets. It is easy to remember as it shapes like “banana in a box”:

```
<input [(ngModel)]="product.selected"></input>
```

- Use the canonical prefix `bindon-`:

```
<input bindon-ngModel="product.selected"></input>
```

When Angular parses the markup and meets one of this forms, it uses the `ngModel` input and `ngModelChange` output to create the two-way data binding and hide the details behind the scene.



The `ngModel` directive only works for HTML elements supported by a `ControlValueAccessor`.

We cannot use `ngModel` in custom component until we implement a suitable value accessor.

Built-in Directives

Angular has a small amount of powerful built-in directives cover many routine operations we need to do in templates.

NgClass

We use a Class Binding to add and remove a single class:

```
<div [class.special]="isSpecial">Show special</div>
```

In scenarios when we need to manage many classes at once better use `NgClass` directive. Before usage, we need to create a key: value control object, where the key is a CSS class name and the value is a boolean. If the value is true, the Angular add the class from the key to the element and if it is false then remove. Here is the method returns the key: value control object:

```
getClasses() {  
  let classes = {  
    modified: false,  
    special: true  
  };  
  return classes;  
}
```

So, it's time to add `NgClass` property and bind it to the `getClasses` method:

```
<div [ngClass]="getClasses()">This is special</div>
```

NgStyle

The Style Binding helps set inline styles, based on the state of the component.

```
<button [style.color]="isSpecial?'red':'green'">Special</button>
```

If we need set many inline styles better use `NgStyle` directive, but before use it we need to create the key: value control object. The key of the object is the style name, the value is anything appropriated for the particular style. Here is the key: value control object:

```
getStyles() {  
  let styles = {  
    'font-style': 'normal',  
    'font-size': '24px'  
  };  
  return styles;  
}
```

Let's add the `NgStyle` property and bind it to the `getStyles` method:

```
<div [ngStyle]="getStyles()">  
  This div has a normal font with 8 px size.  
</div>
```

NgIf

We can use different techniques to manage the appearance of elements in DOM. One of them use hidden property to hide unwanted part of the page away:

```
<h3 [hidden]="!specialName">  
  Your special is: {{specialName}}  
</h3>
```

In the code above, we bind `specialName` variable to the HTML `hidden` property. Another one uses built-in directive like `NgIf` to add or remove the element from the page entirely:

```
<h3 *ngIf="!specialName">  
  Your special is: {{specialName}}  
</h3>
```

The difference between hiding and delete is matter. Benefits of invisible elements is evidence:

- It is showing very quick

- It preserves the previous state and ready to display
- It is not necessary to reinitialize

The side effect of hidden element is that:

- It still exists on the page and its behavior continues
- It ties up to resources, utilizes connections to the backend, etc.
- Angular keeps listening events and checking for changes that could affect data bindings and so on.

The `NgIf` directive works differently:

- It removes component and all children entirely
- Removed elements do not utilize resources
- Angular stops change detection, detaches the element from DOM and destroys it.



I recommend, to use `ngIf` to remove unwanted components rather than hide them.

NgSwitch

If we want to display only one element tree from many element trees based on some condition, we can use `NgSwitch` directive. To make it works we need:

- To define a container element contains `NgSwitch` directive with switch expression
- Define inner elements with `NgSwitchCase` directive per element
- Establish no more than an item with `NgSwitchDefault` directive

The `NgSwitch` is inserts nested elements based on which match expression in `NgSwitchCase` matches the value evaluated from switch expression:

```
<div [ngSwitch]="condition">
```

```
<p *ngSwitchCase="true">The true value</p>
<p *ngSwitchCase="false">The false value</p>
<p *ngSwitchDefault>Unknown value</p>
</div>
```

If a matching expression is not found than an element with `NgSwitchDefault` directive is displayed.

NgFor

The `NgFor` directive, in contrast to `NgSwitch` rendering each item in the collection. We can apply it to simple HTML elements or components with the following syntax:

```
<div *ngFor="let product of products">{{product.name}}</div>
```

The text assigned to `NgFor` directive is not a template expression. It is **microsyntax** – the language that Angular interprets how to iterate over the collection. Say more; the Angular translates instruction into a new set of elements and bindings. The `NgFor` directive iterates over `products` array to return the instance of the `Product` and stamps out instances of the `DIV` element to which it is applied. The `let` keyword in expression creates a **template input variable** called the `product`, available in the scope of host and it children elements, so we can use its properties like we are doing in interpolation `{{product.name}}`.



A template input variable is neither the template nor the state reference variables.

Sometimes it can be useful to know a bit more about the currently iterated element. The `NgFor` directive provides several exported index-like values:

- The `index` value sets to the current loop iteration from 0 to the length of collection
- The `first` is the boolean value indicating whether the item is the first in the iteration
- The `last` is the boolean value indicating whether the item is the last one in the collection
- The `even` is the boolean value indicating whether the item has an even index

- The `odd` is the boolean value indicating whether the item has an odd index

So we can use any of those values to capture one in local variable and use it inside iteration context:

```
<div *ngFor="let product of products; let i=index">
  {{i + 1}} - {{product.name}}
</div>
```

Now, let's imagine the array of products we query from the backend. Each refresh operation returns the list containing a small, if not all, number of changed items. Because Angular doesn't now about changes, it discards the old DOM elements and rebuilds new list with new DOM elements. With a huge number of items in the list, the `NgFor` directive can perform poorly, freeze the UI and make web application entirely unresponsive. We can fix the problem if we give to the Angular a function to track items inside collection to avoid this DOM rebuild nightmare. The tracking relying on object identity so that we can use any one or many properties to compare new and old items inside the collection. The term object identity refers to object equality based on “===” identity operator. Here is an example of tracking function:

```
trackByProductId(index: number, product: Product): any {
  return product.id;
}
```

It's time to add the tracking function to the `NgFor` directive expression:

```
<div *ngFor="let product of products; trackBy:trackByProductId;
  let i=index">
  {{i + 1}} - {{product.name}}
</div>
```

The tracking function cannot remove the DOM changes but can reduce the number of them and make the UI smoother and better responsive.

Structural Directives

We quite often see the asterisk prefix in built-in directives, but I didn't explain the purpose. It's time to unveil the “secret” Angular developers keep away from us.

We are developing single page application and first or last, we end up with the necessity to manipulate with DOM efficiently. The Angular framework helps in appearing and disappearing portions of the page according to application state with several built-in directives. In general, Angular has three kinds of directives:

- **Component:** It is a directive with a template, and we create a lot of them in our project.
- **Attribute Directive:** This kind of directive changes the appearance or behavior of an element.
- **Structural Directive:** It changes the DOM layout by adding and removing DOM elements.

Structural directives use the HTML 5 `template` tag to manage the appearance of components on the page. Templates allow declaring fragments of HTML markup as prototypes. We can insert them into the page anywhere – the head, body or frameset, but without display:

```
<template id="special_template">
  <h3>Your are special</h3>
</template>
```

To use template we must clone and insert it into the DOM:

```
// Get the template
var template: HTMLTemplateElement =
  <HTMLTemplateElement>document.
    querySelector("#special_template");
// Find place where
var placeholder: HTMLElement =
  <HTMLElement>document.
    querySelector("place");
// Clone and insert template into the DOM
placeholder.appendChild(template.content.cloneNode(true));
```

Angular keeps the content of structural directive in the `template` tag, replace it with a `script` tag and use it when it is necessary. Because the template form is verbose, the Angular developers introduced the “syntactic sugar” – asterisk (*) prefix for directives to hide verbosity:

```
<h3 *ngIf="condition">Your are special</h3>
```

When Angular read and parse the above HTML markup, it replaces the asterisk back to template form:

```
<template [ngIf]="condition">
  <h3>Your are special</h3>
</template>
```


Custom Structural Directive

Let's create the structural directive similar to `NgIf` we can use to display the content on the page depends on condition. Open the project in Microsoft Studio Code and create `if.directive.ts` file with following content:

```
import {Directive, Input} from '@angular/core';

@Directive({ selector: '[dbIf]' })
export class IfDirective {
}
```

We import `Directive` to apply it to the `IfDirective` class. We can use our directive in any HTML element or component as a property. Because we manipulate with the content of the template we need `TemplateRef`. Moreover, Angular uses `especial renderer` `ViewContainerRef` to render the content of template, so we need to import both of them and inject into constructor as private variables:

```
import {Directive, Input} from '@angular/core';
import {TemplateRef, ViewContainerRef} from '@angular/core';

@Directive({ selector: '[dbIf]' })
export class IfDirective {
  constructor(
    private templateRef: TemplateRef<any>,
    private viewContainer: ViewContainerRef
  ) { }
}
```

And last thing, the property to keep the boolean condition, so directive adds or removes the template based on that value. It must have the same name as directive plus we can make it read-only:

```
@Input() set dbIf(condition: boolean) {
  if (condition) {
    this.viewContainer.createEmbeddedView(this.templateRef);
  } else {
    this.viewContainer.clear();
  }
}
```

If the condition is true, the code above calls the view container to create embedded view that references on the template content, otherwise – remove it. Here is the final version of our directive:

```
import {Directive, Input} from '@angular/core';
```

```
import {TemplateRef, ViewContainerRef} from '@angular/core';

@Directive({ selector: '[dbIf]' })
export class IfDirective {
  constructor(
    private templateRef: TemplateRef<any>,
    private viewContainer: ViewContainerRef
  ) { }

  @Input() set dbIf(condition: boolean) {
    if (condition) {
      this.viewContainer.
        createEmbeddedView(this.templateRef);
    } else {
      this.viewContainer.clear();
    }
  }
}
```

Now we can add our directive into `directives` array of the host component to use it instead of `NgIf`.

Category Product Component

We continue in the creation of Angular components of our application. Now, when we know everything about templates, it's time to create the Category Product. Navigates into a shared folder and create `category` directory there. Create the file `category.ts`. Copy and paste the code below:

```
export class Category {
  // Unique Id
  id: string;
  // The title
  title: string;
  // Description
  desc: string;
  // Path to image
  image: string;
}
```

So, each category of product has unique identifier, title, description and image. Then create the file `category-card.component.ts`, copy and paste the following code:

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

```
import {Category} from './category';

@Component({
  selector: 'db-category-card',
  templateUrl:
    'app/shared/category/category-card.component.html'
})
export class CategoryCardComponent {
  @Input() category: Category;
  @Output() select: EventEmitter<Category> =
    new EventEmitter<Category>();

  browse() {
    this.select.emit(this.category);
  }
}
```

This a Category component we use in a grid of categories. It has the input property category and output event select. Let's have a look how markup is looks like:

```
<div class="col-xs-12 col-sm-6 col-md-4">
  <div class="card">
    
    <div class="card-block">
      <h4 class="card-title">{{category.title}}</h4>
      <p class="card-text">{{category.desc}}</p>
      <a class="btn btn-primary" (click)="browse()">Browse</a>
    </div>
  </div>
</div>
<!-- /.col -->
```

It is exact copy of markup from app.component.html. We use interpolation Data Binding everywhere. Now create category-slide.component.ts, copy and past next code:

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

import {Category} from './category';

@Component({
  selector: 'db-category-slide',
  templateUrl:
    'app/shared/category/category-slide.component.html'
})
export class CategorySlideComponent {
  @Input() category: Category;
  @Output() select: EventEmitter<Category> =
```

```
        new EventEmitter<Category>();

        browse() {
            this.select.emit(this.category);
        }
    }
}
```

The source code of this file looks absolutely similar to the card category, but markups not:

```

<div class="carousel-caption">
    <h2>{{category.title}}</h2>
</div>
```

This one is copy of HTML from carousel component. It's time to update `app.component.ts` file:

```
import {Component} from '@angular/core';
import {bootstrap} from '@angular/platform-browser-dynamic';

import {NavbarComponent} from '../shared/navbar/navbar.component';
import {Category, CategoryCardComponent, CategorySlideComponent} from
'../shared/category/index';

@Component({
    selector: 'my-app',
    templateUrl: 'app/app.component.html',
    directives: [NavbarComponent, CategoryCardComponent,
CategorySlideComponent]
})
export class AppComponent {
    // Slide Categories
    slideCategories: Category[] = [
        { id: '1', title: 'Bakery', image: 'http://placeholder.it/1110x480',
desc: 'The best cupcakes, cookies, cakes, pies, cheesecakes, fresh bread,
biscotti, muffins, bagels, fresh coffee and more.' },
        { id: '2', title: 'Takeaway', image: 'http://placeholder.it/1110x480',
desc: 'It's consistently excellent, dishes are superb and healthily cooked
with high quality ingredients.' },
        { id: '3', title: 'Dairy', image: 'http://placeholder.it/1110x480', desc:
'A dairy product is food produced from the milk of mammals, primarily cows,
water buffaloes, goats, sheep, yaks, horses.' },
    ];

    // Card categories
    cardCategories: Category[] = [
        { id: '1', title: 'Bakery', image: 'http://placeholder.it/1110x480',
desc: 'The best cupcakes, cookies, cakes, pies, cheesecakes, fresh bread,
```

```
biscotti, muffins, bagels, fresh coffee and more.' },
  { id: '2', title: 'Takeaway', image: 'http://placeholder.it/1110x480',
    desc: 'It's consistently excellent, dishes are superb and healthily cooked
    with high quality ingredients.' },
  { id: '3', title: 'Dairy', image: 'http://placeholder.it/1110x480', desc:
    'A dairy product is food produced from the milk of mammals, primarily cows,
    water buffaloes, goats, sheep, yaks, horses.' },
  { id: '4', title: 'Meat', image: 'http://placeholder.it/1110x480', desc:
    'Only superior quality beef, lamb, and pork.' },
  { id: '5', title: 'Seafood', image: 'http://placeholder.it/1110x480',
    desc: 'Great place to buy fresh seafood.' },
  { id: '6', title: 'Fruit & Veg', image: 'http://placeholder.it/1110x480',
    desc: 'A variety of fresh fruits and vegetables.' },
  ];

  selectCategory(category: Category) {
    console.log('Selected category', category.title);
  }
}
```

I import `CategoryCardComponent` and `CategorySlideComponent` so we can add them into directives property of `Component`. I defined two variables to keep data for cards in grid and slides.

Here is the changes in `app.component.html`:

```
<!-- Indicators -->
<ol class="carousel-indicators">
  <li data-target="#welcome-products"
    *ngFor="let category of slideCategories;let first=first; let i=index"
      attr.data-slide-to="{{i}}" [ngClass]="{active: first}"></li>
</ol>
```

We use `NgFor` directive here with `first` and `index` values to initialize `data-slide-to` attribute and `active` class of first component.

```
<!-- Content -->
<div class="carousel-inner" role="listbox">
  <div *ngFor="let category of slideCategories;let first=first"
    class="carousel-item" [ngClass]="{active: first}">
    <db-category-slide
      [category]="category" (select)="selectCategory($event)">
    </db-category-slide>
  </div>
</div>
```

In this markup we form the content of carousel images, so we use `NgFor` directive in

carousel-inner component. We use the first value to manage active class of first component.

```
<div class="row">
  <db-category-card *ngFor="let category of cardCategories"
    [category]="category" (select)="selectCategory($event)">
  </db-category-card>
</div>
```

Hire the last section of our changes where we create the cards grid with help of `NgFor` directive.

Summary

We speak about the structure of the Angular application and how important to maintain the folder structure flat. Because we following the Single Responsibility principles, so we create one component per file and keep it size as small as possible. Every Angular application has shared folder where we keep features we can use in multiple places.

We spoke a lot about Decorators, Tree of Components and Templates. We know that Template Expression and Template Statements are the crucial part of Data Binding. Both of them based on restricted version of the JavaScript-like language.

Template Expression includes the Elvis and Pipe operators for specific scenarios. The Data Binding supports three flow directions and includes Interpolation, Property Binding, Attribute Binding, Class Binding, Style Binding, Event Binding, and Two-way Binding.

Angular has several very powerful Directives help us to manipulate with DOM elements like `NgFor`, `NgIf`, `NgClass`, `NgStyle`, and `NgSwitch`. We learned why we use the asterisk prefix and what Structural Directives are?

In *Chapter 5, Routing*, we will set up the top navigation with Bootstrap. You will become familiar with Angular's component router and how to configure it. Plus we will continue to build the project we started to develop in previous chapters.

5

Routing

Many web applications require more than one page or view, and Angular is well-equipped to handle this with its router. The router is a JavaScript code manages the navigation between views as user perform application tasks. In this chapter, we will take a look at how we can create static routes as well as routes containing parameters and how to configure them. Also, we will discover some of the pitfalls we might face. In this chapter, we will set up the top navigation with Angular.

At the end of the chapter, you will have a solid understanding:

- Component Router
- Router configuration
- Router Link and Router Outlet
- Creating components and Navigation for our Application

The Modern Web Applications

You heard about **Single-Page Applications (SPA)** many times, but what the reason to develop web applications like that? What the benefits?

The main idea to use SPA is quite simple – users would like to use web applications which look like and behave the native. The SPA is web applications that load a single HTML page and dynamically update that page as the user interacts with multiple components on it. Some of the components support many states like open, collapsed, etc. Implementing all of these features with server-side rendering is hard to do, so much of the work happens on the client side, in JavaScript. This is achieved by separating the data from the presentation of data by having a model layer that handles data and a view layer that reads from the models.

This idea brings some level of complexity on the code and results in changing minds about the development process. Now we start thinking about the conceptual parts of application, file and module structures, performance issues over bootstrapping, so on.

Routing

Since we are making a Single Page Application and we don't want any page refreshes, we'll use routing capabilities of Angular. The Routing module is a crucial part of Angular. It helps to update the URL of the browser as the user navigates through the application, from one side. From another side, it allows changes to the URL of the browser to drive navigation through the web application. Thus allowing the user to create bookmarks to locations deep within the Single Page Application. As a bonus, we can split the application into multiple bundles and load them on demand.

With the introduction of HTML 5, browsers acquired the ability to create programmatically the new browser history entries that change the displayed URL without the need for a new request. It is achieved using the `pushState` method of `history` that exposes the browser's navigational history to JavaScript. So now, instead of relying on the anchor hack to navigate routes, modern frameworks can count on `pushState` to perform history manipulation without reloads.

The Angular Router uses this model to interpret a browser URL as an instruction to navigate to a client-generated view. We can pass optional parameters along to the view component to help it decide what specific content to present.

Routing Path

Before we begin, let's plan out exactly what routes we're going to need for the Dream Bean grocery store website:

- The Welcome view uses `#!/welcome` path. It is going to be our entry point for the application, which will list all categories in a grid and slideshow
- The Products utilizes the `#!/products` path. We'll be able to see the goodies within the chosen category there
- We show the Product view on `#!/product/:id`. Here we will display the information about the product
- The `#!/cart` path is where we will see the Cart view lists all items in the user's

shopping cart

- On Checkout view with the `/#/checkout/:id` path, we will include a form that will allow user to add a contact information, and provides the order information and purchase conditions

These are all of our essential routes; so let's take a look at how we can create them.

Installing the Router

The Router packaged as a module inside of the Angular, but it is not a part of the Angular core, so we need manually to include it inside bootstrapping configuration in `systemjs.config.js` file:

```
var packageNames = [  
  '@angular/common',  
  '@angular/compiler',  
  '@angular/core',  
  '@angular/http',  
  '@angular/platform-browser',  
  '@angular/platform-browser-dynamic',  
  '@angular/router',  
  '@angular/testing'  
];
```

The Base URL

If we decided to use routing, then we should add the `base` element as the first child in the `head` tag. Reference in this tag resolves relative URLs and hyperlinks and tells to router how to compose navigation URLs. For our project, I assigned the `"/"` to the `href` of the `base` element, because `app` folder is the application root:

```
<base href="/">
```

If we deploy the application to the server within particular context like `portal`, then we must change this value accordingly:

```
<base href="/portal">
```

The Component Router

The actual routing from one view to another happens with the help of the **Component Router**. It is an optional service, represents the component view for specific URL. It has its library package, and we must import from it before usage:

```
import { ROUTER_DIRECTIVES } from '@angular/router';
```

The Router Configuration

The application must have only one Router. We should configure it, so it knows how to map the browser's URL to the corresponding `Route` and determine the component to display. The primary way to do that uses the `provideRouter` function with an array of routes and bootstrap the application with it.

Creating Basic Routes

Let's import necessary elements from router package:

```
import {provideRouter, RouterConfig} from '@angular/router';
```

Now create the constant to keep the application routes:

```
const routes: RouterConfig = [
  { path: 'welcome', component: WelcomeComponent },
  { path: 'products', component: ProductListComponent },
  { path: 'product/:id', component: ProductComponent }
];
```

We define the array of route objects that describe how to navigate. Each `Route` maps a URL path to a component to display. The router parses and constructs the URL helping us to use:

- Path references to the base element and eliminates the necessity to use leading slashes.
- Absolute path

Query Parameters

The second item in the router configuration points only to `products`, but as I mentioned early, we'll be able to see the goodies within the chosen category there. It sounds like the

information we would like to include into URL is optional:

- We can leave the request without extra information to get all products
- We can use the particular category to fetch the products belongs to

These kinds of parameters do not fit easily into a URL path, so usually, it is complicated or impossible to create the pattern matching required to translate an incoming URL to a named route. Fortunately, the Component Router supports the **URL Query String** for conveying any arbitrary information during navigation.

Router Parameters

The third element in the `routes` array has an `id` in its path. It is a token for a **route parameter** – the value corresponding view component will use to find and present the product information. In our example, the URL `'product/20'` keep the value 20 of the `id` parameter. The `ProductComponent` can use this value to find and display the product with `id` equals 20.

Route vs. Query Parameters

Here are the general rules to help you to choose what the parameters to use. Use the route parameters when:

- The value is required
- The value is necessary for navigation to another route

Use the query parameters when:

- The value is optional
- The value is complex or can contains multi-variance

Register Routing in bootstrap

In the end, we should use the `provideRouter` function to registry the router dependencies in the `bootstrap` method:

```
bootstrap(AppComponent, [
```

```
        provideRouter(routes)
    });
```

It returns the configured Router service provider. We will use this service in many places of our project later.

Redirecting Routes

Usually, when user typing the address of the Dream Bean website it does provide the website domain name: `http://www.greambean.com`

This URL does not match any configured routes, and Angular cannot show any component at that moment. The user must click on some link to navigate to the view, or we can teach the configuration to display the particular route with help of `redirectTo` property:

```
const routes: RouterConfig = [
  { path: '', redirectTo: 'welcome', pathMatch: 'full' },
  { path: 'welcome', component: WelcomeComponent },
  { path: 'products', component: ProductListComponent },
  { path: 'product/:id', component: ProductComponent }
];
```

After those changes if the user navigates to the original URL, the Router translates from initial URL('') to the default one URL('welcome') and displays the Welcome View.

The redirect Route has a required property `pathMatch` to tell the Router how to match the URL to the path. We have two options for this value:

- The `full` shows that selected route must match the entire URL
- The `prefix` dictates the Router to match the redirect route to any URL begins with the prefixed value in the `path`.

Router Outlet

Now, when we settled the router configuration, it's time to present some components on the screen. But wait, we need the place for them, and this is why the **Router Outlet** is coming to the stage.

The `RouterOutlet` is a placeholder that Angular dynamically fills based on the application's route. To use `RouterOutlet` we need to import the Router Directives because

they keep the definition about it:

```
import {ROUTER_DIRECTIVES} from '@angular/router';
```

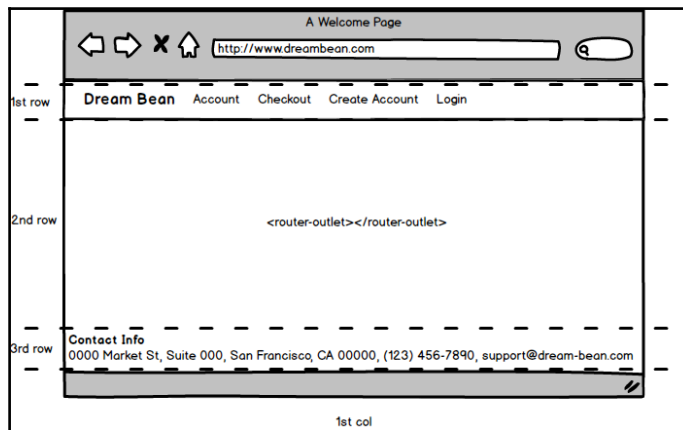
Then, include them in AppComponent:

```
@Component({
  selector: 'my-app',
  templateUrl: 'app/app.component.html',
  directives: [ROUTER_DIRECTIVES, NavbarComponent,
               FooterComponent]
})
export class AppComponent {}
```

In the end, update the HTML with application component markup represent the RouterOutlet as a component:

```
<db-navbar></db-navbar>
<router-outlet></router-outlet>
<db-footer></db-footer>
```

Here is a wireframe splits the SPA into three rows:



The wireframe of the SPA

In first row, we keep the NavigationComponent. In last row – AddressComponent. All space in between is the place where RouterOutlet will display the corresponding view.

Welcome View

We configured application routes, updated the AppComponent, so now we need to create the Welcome View because it is a crucial part of the routing. I moved almost all code from AppComponent into WelcomeComponent as well as markup between HTML files. I use the link to navigate from Welcome View to the Products View with selected category instead of making a call to the selectCategory method, so I delete the last one also.

Category Card View

Let's open the category-card.component.html file and change the markup as follow:

```
<div class="col-xs-12 col-sm-6 col-md-4">
  <div class="card">
    
    <div class="card-block">
      <h4 class="card-title">{{category.title}}</h4>
      <p class="card-text">{{category.desc}}</p>
      <a class="btn btn-primary"
        (click)="filterProducts(category)">Browse</a>
    </div>
  </div>
</div>
```

When user click on **Browse** button the Angular calls the filterProducts method with category specified as a parameter.

Open category-card.component.ts file, import the Router from the library and add the reference in the constructor of the component:

```
import {Component, Input} from '@angular/core';
import {Router} from '@angular/router';

import {Category} from './category';

@Component({
  selector: 'db-category-card',
  templateUrl:
    'app/shared/category/category-card.component.html'
})
export class CategoryCardComponent {
  @Input() category: Category;

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

```
filterProducts(category: Category) {  
  this.router.navigate(['/products'],  
    {queryParams: { category: category.id} });  
}
```

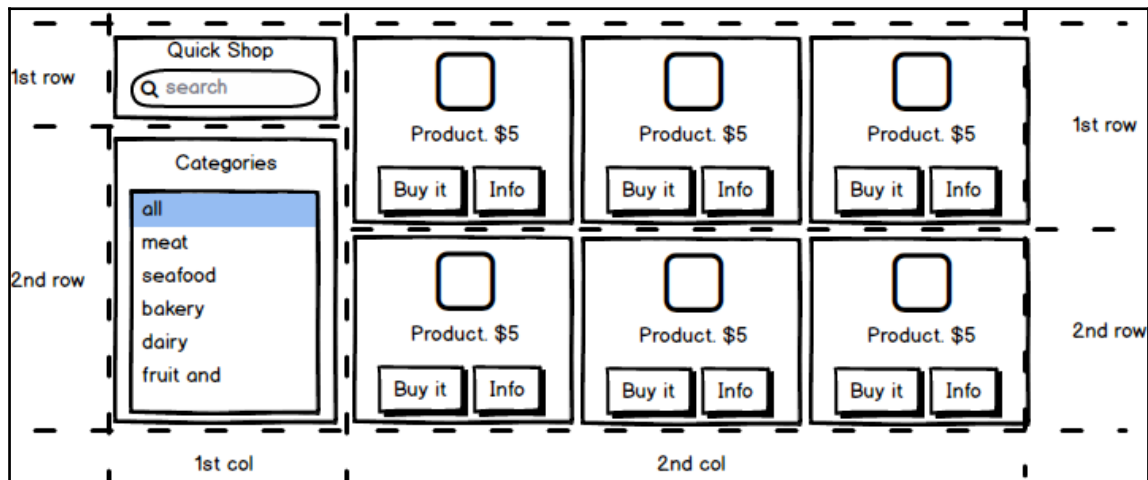
Take attention on `filterProducts` method. We use router configured in bootstrapping of application and available in this component. Because we decided to use query parameters, I invoke navigation method and pass as the second parameter object the same name. We can convey any information and Angular will convert it into the query string of URL like that:

```
/products?category=1
```

We are done with Welcome View and now moving to Products View.

Products View

The products view provides a listing of all goodies within the chosen category. From it, a customer can see all product information, and add any of the listed products to his or her shopping cart. A user can also navigate to any of the provided categories or use 'Quick Shop' feature to search products by name.



The wireframe of the Products View

The layout of this component is a composition of two columns:

- The first column contains the Quick Shop and List of Categories

- The second column is a nested column combined into the rows

Quick Shop Component

This one is an Input Group field for search and quick shop the products available in the grocery. We use the **URL Query String** for conveying the search information as we did for the category because we don't know what the user type in the search field.

Let's create `product-search.component.html` with following markup:

```
<div class="card">
  <div class="card-header">Quick Shop</div>
  <div class="input-group">
    <input #search type="text" class="form-control"
      placeholder="Search for...">
    <span class="input-group-btn">
      <button class="btn btn-secondary" type="button"
        (click)="searchProduct (search.value) ">Go!
      </button>
    </span>
  </div>
</div>
```

I use the Bootstrap 4 Input Groups with button inside the Card component. The template reference variable `search` grants us direct access to an input element so that we can use the text value in `searchProduct` method, when user type the product name and click on the **Go!** button.

I create the `ProductSearch` component similar to `CategoryCard` one:

```
import {Component} from '@angular/core';
import {Router} from '@angular/router';

import {Product} from '../product';

@Component({
  selector: 'db-product-search',
  templateUrl: 'app/product/product-search.component.html'
})
export class ProductSearchComponent {

  constructor(private router: Router) {}

  searchProduct(value: string) {
    this.router.navigate(['/products'],
```



```
        { queryParams: { search: value } });  
    }  
}
```

I use navigation method of Router to search product by name with the following URL:

```
/products?search=Apple
```



Now, we are ready to create the `CategoryList` component user can use to select the category.

List of Categories Component

In Chapter 3 we introduced the flexible Bootstrap 4 `List Group` component. The `Categories` is a list of unordered items so that we can use this particular one to render categories quickly. I use the same mechanism to update the URL with the specific category as we use in `CategoryCard` component. Let's have a look at our markup:

```
<div class="card">  
  <div class="card-header">Categories</div>  
  <div class="card-block">  
    <div class="list-group list-group-flush">  
      <a class="list-group-item"  
        *ngFor="let category of categories"  
        (click)="filterProducts(category)">  
        {{category.title}}</a>  
    </div>  
  </div>  
</div>
```

We have the `Card` component wraps the `List Group`. The built-in `NgFor` directive helps to organize iteration through categories to display the titles.

Router Links

Most of the time, the user navigates between views as a result of some actions on links, like click happens on the anchor tag. We can bind the router to links on a page, so when the user clicks on the link, it will navigate to appropriate application view.



The router logs activity in the history journal of the browser, so the back and forward buttons work as expected.

The Angular team introduced a `RouterLink` directive to the anchor tag, to bind it to the template expression contains the array of route link parameters. Let's create the `ProductCard` component with the help of `RouterLink`.

Product Card

I suppose it is a good idea to present the Product as a card. I create the `product-card.component.html` with the following markup:

```
<div class="col-xs-12 col-sm-6 col-md-4">
  <div class="card">
    
    <div class="card-block">
      <h4 class="card-title">{{product.title}}</h4>
      <p class="card-text">{{product.desc}}</p>
      <a class="btn btn-primary"
        [routerLink]="['/product', product.id]">Browse</a>
    </div>
  </div>
</div>
```

In our example, the `RouterLink` binds in the anchor tag. Take attention on the template expression we bind to the `routerLink`. Obviously, it is an array so that we can add more than one item, and Angular will combine them to build the URL. We can specify all pieces of route exclusively like `"product/1"`, but I intentionally leave them as separated items of an array easy to maintain. Let's parse it:

- The first item identifies the parent root `" /product "` path
- There are no parameters for this parent element like `"product/groups/1"`, so we are done with it

- The second item identifies the child route for product requires the id

The navigation with `RouterLink` so flexible, so we can write an application with multiple levels of routing with link parameters array.

Create a `product-card.component.ts`. We need import the Router Directives and declare them in the component, so now the `RouterLink` is available on markup with next code:

```
import {Component, Input} from '@angular/core';
import {ROUTER_DIRECTIVES} from '@angular/router';

import {Product} from '../product';

@Component({
  selector: 'db-product-card',
  templateUrl: 'app/product/product-card.component.html',
  directives: [ROUTER_DIRECTIVES]
})
export class ProductCardComponent {
  @Input() product: Product;
}
```

Products Grid Component

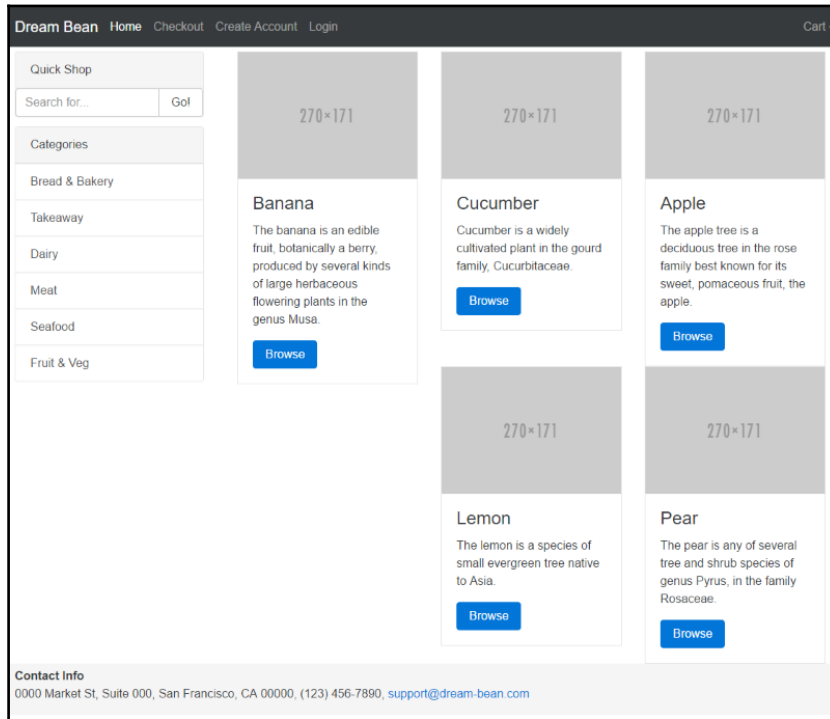
We need to show the products as a grid with three columns and multiple rows. The Card component is the most suitable one to display the Product information and navigate to the Product View. All of the cards in the row must have the same width and height. How can we display them in the particular place inside the parent grid layout?

Card Groups

We can use the Bootstrap 4 **Card Groups** to present multiple cards as a single attached element with equal width and height. We need only include all cards within a parent element marked with `card-group` class:

```
<div class="card-group">
  <db-product-card *ngFor="let product of products"
    [product]="product"></db-product-card>
</div>
```

The result is not what I want because some cards attached to each other:

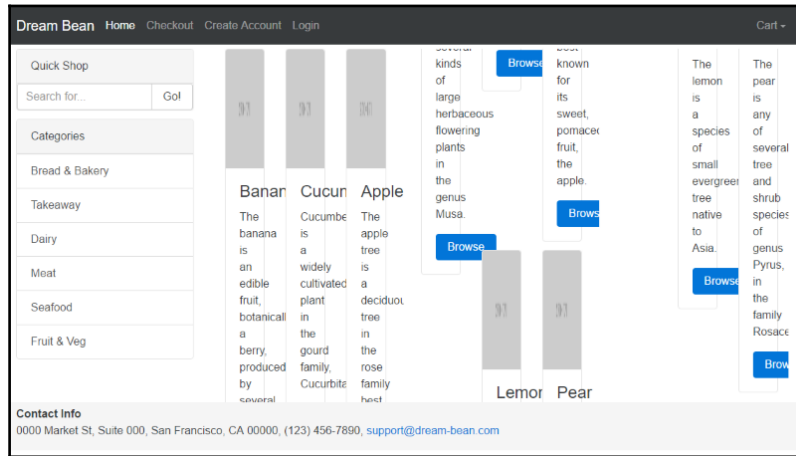


Card Columns

Another layout is **Card Columns** from Bootstrap 4. It allows you to display multiple cards in each column. Each card inside a column is stacked on top of other. Include all cards within a `card-columns` class:

```
<div class="card-columns">
  <db-product-card *ngFor="let product of products"
    [product]="product"></db-product-card>
</div>
```

The result looks quite funny:



Card Desks

The last layout is a **Card Desk** from Bootstrap 4. It is similar to Card Group, except the cards aren't attached to each other. This one requires two wrapping elements: `card-deck-wrapper` and a `card-deck`. It uses table styles for the sizing and the gutters on `card-deck`. The `card-deck-wrapper` is used to negative margin out the border-spacing on the `card-deck`.

Let's create `product-card.component.html` file with following content:

```
<div class="card-deck-wrapper">
  <div class="card-deck">
    <div class="card" *ngFor="let product of products">
      <div class="card-header text-xs-center">
        {{product.title}}
      </div>
      
      <div class="card-block text-xs-center"
        [ngClass]="setClasses(product)">
        <h4 class="card-text">
          Price: ${{product.price}}
        </h4>
      </div>
      <div class="card-footer text-xs-center">
        <a class="btn btn-primary"
          (click)="buy(product)">Buy Now</a>
      </div>
    </div>
  </div>
</div>
```

```
        <a class="btn btn-secondary"
          [routerLink]="['/product', product.id]">
            More Info
        </a>
    </div>
    <div class="card-block">
        <p class="card-text">{{product.desc}}</p>
    </div>
</div>
</div>
</div>
```

The Card Desk works perfectly enough with one row, so we expose the products input in ProductCard component :

```
import {Component, Input} from '@angular/core';
import {ROUTER_DIRECTIVES} from '@angular/router';

import {Product} from './product';

@Component({
  selector: 'db-product-card',
  templateUrl: 'app/product/product-card.component.html',
  directives: [ROUTER_DIRECTIVES]
})
export class ProductCardComponent {
  @Input() products: Product[];

  setClasses(product: Product) {
    return {
      'card-danger': product.isSpecial,
      'card-inverse': product.isSpecial
    };
  }

  buy(product: Product) {
    console.log('We bought', product.title);
  }
}
```

Method `setClasses` helps change the card's background if the product has a special price. We call the `buy` method when the user clicks on **Buy Now** button.

Here is a markup of the ProductGrid compont:

```
<db-product-card *ngFor="let row of products"
  [products]="row"></db-product-card>
```

Quite neat, isn't it?

We need to transform an array of products into an array of rows with three product per line. Please pay attention to the code in constructor:

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

import {Product, getProducts} from './product';
import {ProductCardComponent} from './product-card.component';

@Component({
  selector: 'db-product-grid',
  templateUrl: 'app/product/product-grid.component.html',
  directives: [ProductCardComponent]
})
export class ProductGridComponent {
  products: any = [];

  constructor() {
    let index = 0;
    let products: Product[] = getProducts();
    let length = products.length;

    this.products = [];

    while (length) {
      let row: Product[] = [];
      if (length >= 3) {
        for (let i = 0; i < 3; i++) {
          row.push(products[index++]);
        }
        this.products.push(row);
        length -= 3;
      } else {
        for (; length > 0; length--) {
          row.push(products[index++]);
        }
        this.products.push(row);
      }
    }
  }
}
```

We split the products into multiple rows contain the three columns maximum.

Router Change Events

As we mentioned in router configuration the `ProductList` component represents the Product View when user navigates in the URL like:

```
/products?category=1
```

Or

```
/products?search=apple
```

We can subscribe to route change events to inform the `ProductGrid` about the changes happens in URL. Import `Router` from library, add `router` property and subscribe on route changes:

```
constructor(private router: Router) {  
    this.router  
        .routerState  
        .queryParams  
        .subscribe(params => {  
            let category: string = params['category'];  
            let search: string = params['search'];  
            // Return filtered data from getProducts function  
            let products: Product[] =  
                getProducts(category, search);  
            // Transform products to appropriate data  
            // to display  
            this.products = this.transform(products);  
        });  
}
```

In the code above, we are listening the changes happens only in `queryParams` and use them to filter data in `getProducts` function. Later, with help of `transform` method we translate the filtered products in data appropriate to display.

```
transform(source: Product[]) {  
    let index = 0;  
    let length = source.length;  
  
    let products = [];  
  
    while (length) {  
        let row: Product[] = [];  
        if (length >= 3) {  
            for (let i = 0; i < 3; i++) {  
                row.push(source[index++]);  
            }  
        }  
    }  
}
```



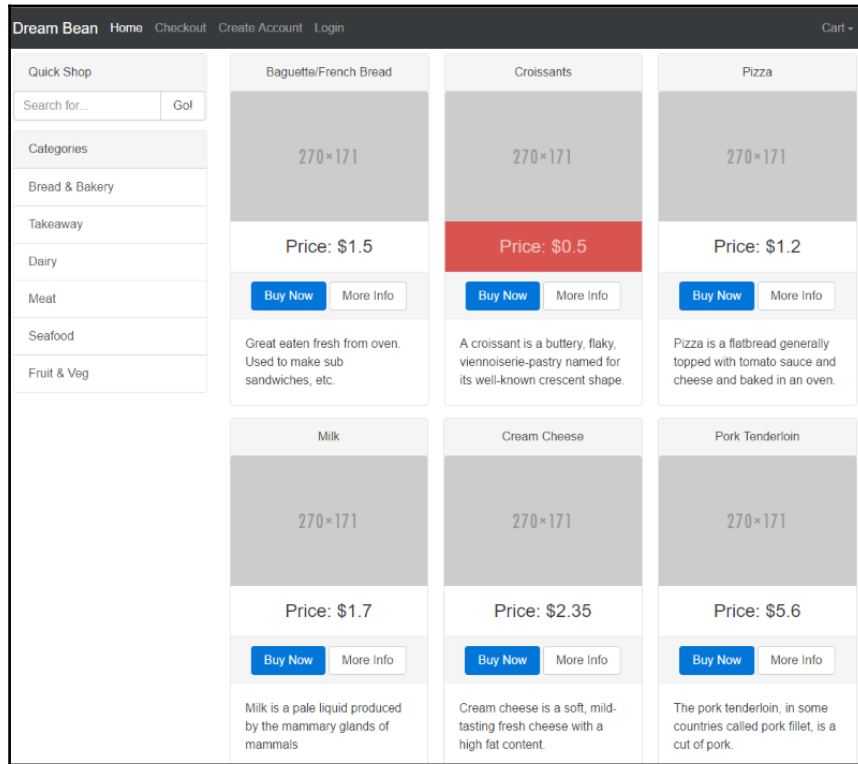
```
        products.push(row);
        length -= 3;
    } else {
        for (; length > 0; length--) {
            row.push(source[index++]);
        }
        products.push(row);
    }
}

return products;
}
```

The last one, we must change the `getProducts` function:

```
export function getProducts(category?: string, search?: string) {
    if (category) {
        return products.filter(
            (product: Product, index: number, array: Product[]) => {
                return product.categoryId === category;
            });
    } else if (search) {
        let lowSearch = search.toLowerCase();
        return products.filter(
            (product: Product, index: number, array: Product[]) => {
                return product.title.toLowerCase().
                    indexOf(lowSearch) !== -1;
            });
    } else {
        return products;
    }
}
```

It filters data by category, searches text or leaves them as is, depends on what the parameters we send to the function. Save, the code and try to play with filter data:



Routing Strategies

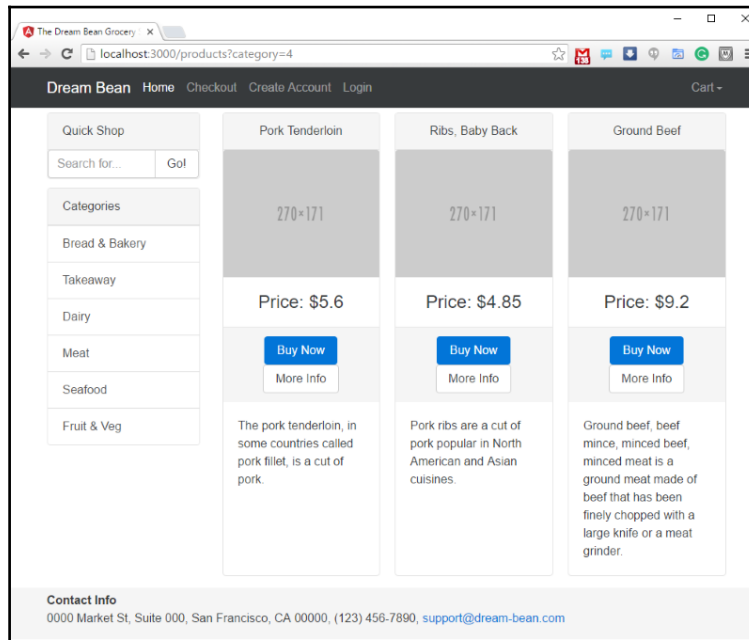
All of our essential routes configured, and we now have access to separate view for all of them. That's great, but maybe you are not happy with the path following the `#` symbol in the URL. As I mentioned, the modern web browsers support `pushState` technique helps change a location and history in the browser without a request to the server. The `Router` using this method to build the URL. The Angular Component Router uses different `LocationStrategy` providers support old and new ways:

- The `PathLocationStrategy` provides the default, HTML 5 style based on `pushState`
- The `HashLocationStrategy` utilizing the hash in URL style

Choosing the strategy is crucial for future development because it won't be easy to change it

later, so better do it at the right time. I used the second method for demonstration purposes only. Now it's time to change it to the original mode and `lite-server` can help us with that. The fact is that it support redirect to a fallback page when a route is not found.

Open `main.ts` file and comment out or delete `LocationStrategy`, `HashLocationStrategy` from import and bootstrap configuration. Save is and check the how application works without hash in the browser's URL:



Summary

In this chapter, we transformed our application from a single page into a multi-page view and multi-route app that we can build upon Dream Bean grocery store. We started by planning out the essential routes in our application before writing any line of code.

We then build static and dynamic routes containing parameters.

Finally, we looked at how we can remove the # symbol from the URL using HTML5's `pushState` and how we can link both types of routes.

In Chapter 6, Dependency Injection, we will talk about dependency injection that teaches the readers how to decouple the requirements of an application and how to create a consistent source of data as a service. Plus we will continue to build the project we started to develop in previous chapters.

6

Dependency Injection

This chapter is about dependency injection that teaches the readers how to decouple the requirements of an application and how to create a consistent source of data as a service. readers will learn about Injector and Provider classes. We will also discuss Injectable decorator that is required for the creation of an object.

At the end of the chapter, you will have a solid understanding:

- What is Dependency Injection?
- Separation of concerns
- Creating a service
- The Injector and Provider classes
- Injectable and Inject decorators
- Creating data services for our Application

What is Dependency Injection

Today, I would talk about the concept of **Dependency Injection** with some concrete examples that will hopefully demonstrate the problems it tries to solve and the benefits it gives to the developer. Angular is mostly based on Dependency Injection, which you may or may not be familiar. If you already know the concept of Dependency Injection, you can safely skip this chapter and instead read the next one.

Dependency Injection is probably one of the most famous design patterns I know, and you have probably already used it. I think it is one of the most difficult ones to explain well, partly due to the nonsense examples used in most introductions to Dependency Injection. I have tried to come up with examples that fit the Angular world better.

The Real Life Example

Imagine, you start your own business, and tend to travel a lot by air, so you need to arrange flights. You know the phone number of the airline agency, and you are aware of the flight bookings yourself.

Thus your typical travel planning routine might look like the following:

- Decide the destination, and desired arrival date and time
- Call up the airline agency and convey the necessary information to obtain a flight booking
- Pick up the tickets and be on your way

Now if you suddenly change the preferred agency, and its contacts mechanisms, you would be subject to the following relearning scenarios:

- The new agency, and its new contact mechanisms (say the new agencies offer internet based services and the way to make the bookings is over the internet instead of over the phone)
- The typical conversational sequence through which the necessary bookings get done (Data instead of voice).

You need to adjust yourselves to the new scenario. It could lead to a substantial amount of time getting spent in the readjustment process.

Assume your business is growing and you get a secretary in a company, so whenever you needed to travel, you send an email to him or her to just state the destination, desired arrival date and time. The flight reservations are made for you and the tickets get delivered to you.

Now if the preferred agency gets changed, the secretary would become aware of change, would perhaps readjust its workflow to be able to communicate with the agency. However, you would have no relearning required. You still continue to follow the same protocol as earlier, since the secretary makes all the necessary adaptation in a manner that you do not need to do anything differently.

In both the scenarios, you are the client and are dependent upon the services provided by the agency. However, the second scenario has a few differences.

- You don't need to know the contact point of the agency – the secretary does it for

you.

- You don't need to know the exact conversational sequence by which agency conduct its activities via voice, email, website, etc. – you are aware of a particular standardized conversational series with the secretary
- The services you are dependent upon are provided to you in a manner that you do not need to readjust should the service provider change.

That is dependency injection in “real life”.

The Dependency Injection

Both, the Angular and custom components we used in our project is a part of a set of collaborating components. They depend upon each other to complete their intended purpose, and they need to know:

- Which components to communicate?
- Where to locate them?
- How to communicate to them?

When the way to access is changed, such changes can potentially require the source of a lot of components to be modified. Here are the plausible solutions we can use to prevent dramatic changes of components:

- We can embed the logic of location and instantiation as part of usual logic of components
- We can create the “external” piece of code to assume the responsibility of location and instantiation and supply the references when necessary

We can look at last solution as on “Secretary” from our “real life” example. We don't need to change the code of components when the way to locate and an external dependency changes. This solution is the implementation of Dependency Injection, where “external” piece of code is the part of Angular Framework.

Usage of Dependency Injection requires to declare the components and lets the framework work out of complexities of instantiation, initialization, sequencing, and supplying the references as needed.

An injection is the passing of dependency to a dependent object that would use it. There are at least three common ways for a component to accept dependency:

- **Constructor injection:** Where the dependencies are provided through a class constructor
- **Setter injection:** Where Injector utilizes the component exposed setter methods to inject the dependency
- **Interface injection:** where the dependency itself provides a method that will inject the dependency into any component passed to it.

Constructor Injection

This method requires the component to provide a parameter in a constructor for the dependency. We injected the Router instance in the code of the `ProductGridService` component:

```
constructor(private router: Router) {  
  this.router  
    .routerState  
    .queryParams  
    .subscribe(params => {  
      let category: string = params['category'];  
      let search: string = params['search'];  
      // Return filtered data  
      let products: Product[] =  
        getProducts(category, search);  
      // Transform products to appropriate data  
      // to display  
      this.products = this.transform(products);  
    });  
}
```

Constructor injection is most preferable method can be used to ensure the component is always in a valid state, but its lacks the flexibility to have its dependencies changed later.

Another Injection methods

Setter and Interface methods are not implemented in the Angular Framework.

Components vs. Services

The Angular 2 is distinguishing the code of web application on:

- The components that represent the visual part
- The reusable data services

The **data service** is a simple class that provides methods for returning or updating some data.

The Injector

An **Injector** is a replacement for a new operator we are using to resolve the constructor dependencies automatically. When code in application asks about dependencies in the constructor, the `Injector` resolves them.

```
import {Injectable} from '@angular/core';

@Injectable()
export ProductGridService {
  constructor(private router: Route) {...}
}

const injector = new Injector([Router, ProductGridService]);
const service = injector.get(ProductGridService);
```

With a new statement, we create an instance of `Injector`. We are passing the array of service providers into the injector, or it won't know how to create them.

With an `Injector`, creating a `ProductGridService` is very easy, because it takes full responsibilities to provide and inject the `Router` into the `ProductGridService`.

Let's talk why we imported and applied the `Injectable` decorator to the class?

Injectable Decorator

We create multiple types in the application for the particular needs. Some of them may have dependencies to other. We must mark any type available for an injector with an `Injectable` decorator. `Injector` uses a class constructor metadata to get the parameter types and determine dependent types for instantiation and injection. Any dependent type must be marked with `Injectable` decorator or injector will report an error when trying to

instantiate it.



Add `@Injectable()` to every service class to prevent Dependency Injection errors.

We must import and apply the `Injectable` decorator to all class of our services explicitly to makes them an available to an injector for instantiation. Without this decorator, the Angular doesn't know about the existence of those types.

Inject Decorator

As I mentioned, the Injector uses a class constructor metadata to determine dependent types:

```
constructor(private router: Route) {...}
```

Injector uses the TypeScript generated metadata to inject the instance of `Router` type into the constructor. For injecting the TypeScript primitives such string, boolean or array we should manually let it know that a parameter must be injected with `@Inject` decorator:

```
import {Inject} from '@angular/core';

constructor(@Inject('config') private config) {...}
```

We can use `@Inject` for non-primitive types as well, but it is not necessary because most of the time Angular 2 knows what to do with them.

Optional Decorator

In the cases when class has optional dependencies we can use the `@Optional` decorator to mark the constructor parameters:

```
import {Optional} from '@angular/core';

constructor(@Optional('config') private config) {
  if (config) {
    // Use the config
    ...
  }
}
```

I expected that `config` property can be `null` because of the `@Optional` decorator and added a conditional statement in the code above.

Configuring the Injector

In the example above, I used the `new` operator to create `Injector`, but in real life, it's not necessary:

```
const injector = new Injector([Router, ProductGridService]);
```

The Angular framework creates an application injector for us during the bootstrap of application:

```
bootstrap(AppComponent, [Router, ProductService]);
```

The second parameter in this function is an array of providers. We use it for configuration injector via registering the providers that create the services our application require. This way is the most obvious one we can use to configure the injector with preregistered Angular service, such as `Router`. By the way, we can configure injector with alternative providers under the right circumstances:

- Provides an object behaves or looks like the original one
- Provides a substitute class
- Provides a factory function

The preferred way for custom services is to register them in `providers` property of `@Component` decorator, for example for `AppComponent` class:

```
@Component({
  selector: 'my-app',
  templateUrl: 'app/app.component.html',
  directives: [ROUTER_DIRECTIVES, NavbarComponent,
               FooterComponent],
  precompile: [WelcomeComponent, ProductListComponent],
  providers: [CategoryService]
})
export class AppComponent {
}
```

In the both ways, we used a short-hand expression when registered the provider in the injector. The Angular transforms it into the following verbose format:

```
[{provide: Router, useClass: Router}]
```

The property `provide` in the first place is the **token** that serves as the key for:

- Locating a dependency value
- Registering the provider

The second property `useClass` is a definition object similar to many other “use” things and tells to framework how to create the dependency. With the help of the “use” definitions, we can quickly switch implementations, define constants and factory functions. Let's look to all of them.

Class Providers

Most of the time we will utilize the `useClass` definition to ask the different class to provide the service. We can create own `BetterRouter` class as an extension of the original one and register it such that:

```
[{ provide: Router, useClass: BetterRouter }]
```

The injector knows how to build `BetterRouter` and will sort it out.

Aliased class providers

In scenarios when we need to use many providers of the same class, we can use the **useExisting** definition:

```
class BetterRouter extends Router {}

var injectorClass = Injector.resolveAndCreate([
  BetterRouter, {provide: Router, useClass: BetterRouter}
]);
var injectorAlias = Injector.resolveAndCreate([
  BetterRouter, {provide: Router, useExisting: BetterRouter}
]);
```

Look at example below where `useExisting` helps organize mock requests:

```
var injector = Injector.resolveAndCreate([
  HTTP_PROVIDERS,
  MockBackend,
  { provide: XHRBackend, useExisting: MockBackend }
]);
```

```
]);  
var http = injector.get(Http);  
var backend = injector.get(MockBackend);
```

The code below demonstrates how to use the `MockBackend` instead of the real one, makes AJAX requests:

```
var people = [{name: 'Jeff'}, {name: 'Tobias'}];  
  
// Listen for any new requests  
backend.connections.observer({  
  next: connection => {  
    var response = new Response({body: people});  
    setTimeout(() => {  
      // Send a response to the request  
      connection.mockRespond(response);  
    });  
  }  
});  
  
http.get('people.json').observer({  
  next: res => {  
    // Response came from mock backend  
    console.log('first person', res.json()[0].name);  
  }  
});
```

Another useful place for `useExisting` is to providing multiple values of custom pipes, custom directives or custom validators:

```
@Directive({  
  selector: '[custom-validator]',  
  providers: [{ provide: NG_VALIDATORS,  
                useExisting: CustomValidatorDirective, multi: true }]  
})  
class CustomValidatorDirective implements Validator {  
  validate(c: Control): { [key: string]: any } {  
    return { "custom": true };  
  }  
}
```

With the help of option `multi`, it is possible to add the `CustomValidatorDirective` to the default collections to have it available globally in the application.

Value Providers

Sometimes we need to use configuration object, string or function in our application is not always an instance of a class. Here is the interface defines the structure of configuration:

```
export interface Config {
  url: string;
  title: string;
}

export const CUSTOM_CONFIG: Config = {
  url: 'www.dreambean.com',
  title: 'Dream Bean Co.'
};
```

We can register such the ready-made objects with the `useValue` definition. There is no `Config` class, so we cannot use it for the token. Instead, we can use string literal to register and resolve dependencies:

```
providers: [{ provide: 'app.config', useValue: CUSTOM_CONFIG }]
```

And now we can inject it into any constructor with help of `@Inject` decorator:

```
constructor(@Inject('app.config') config: Config) {
  this.title = config.title + ':' + config.url;
}
```

Unfortunately, using string tokens opens up potential for naming collisions. The Angular is rescue and provides elegant solution with `Opaque Token` for non-class dependencies:

```
import { OpaqueToken } from '@angular/core';

export let CONFIG = new OpaqueToken('app.config');
```

We are registering the `CUSTOM_CONFIG` in the injector with the value provider:

```
providers: [{ provide: CONFIG, useValue: CUSTOM_CONFIG }]
```

Inject it into any constructor:

```
constructor(@Inject(CONFIG) config: Config) {
  this.title = config.title + ':' + config.url;
}
```

Multiple Values

With help of option `milti` it is possible to add other values to the same binding later:

```
bootstrap(AppComponent, [
  provide('languages', {useValue: 'en', multi:true }),
  provide('languages', {useValue: 'fr', multi:true })
]);
```

Somewhere in the code we can get multiple values of the languages:

```
constructor(@Inject('languages') languages) {
  console.log(languages);
  // Logs: "['en','fr']"
}
```

Factory Providers

In the cases when we need to create the dependent value dynamically based on information changed at any moment after the bootstrap happened, we can use `useFactory` definition.

Let's imagine we use the `SecurityService` to authorize the user. The `CategoryService` must know the fact about the user. The authorization can change during the user session because he or she can log in and log out at any moment many times. The direct injection the `SecurityService` into `CategoryService` creates a precedent to inject it into all services of the application.

The solution is quite neat – use the primitive boolean authorization property instead of `SecurityService` to control `CategoryService`:

```
categories: Category[] = [...];

constructor(private authorized: boolean) { }

getCategories() {
  return this.authorized ? this.categories : [];
}
```

The `authorized` property will update dynamically, so we cannot use value provider, but we have to take over the creation of new instance of the `CategoryService` with a factory function:

```
let categoryServiceFactory = (securityService: SecurityService) => {
  return new CategoryService(securityService.authorized);
}
```

In the factory provider we inject the `SecurityService` along to the factory function:

```
export let categoryServiceProvider = {
  provide: CategoryService,
  useFactory: categoryServiceFactory,
  deps: [SecurityService]
};
```

The Hierarchy of Injectors

The Angular 1 has only one injector across the application, and it manages the creation and resolving all dependencies quite nice. Every registered dependency becomes to be a singleton, so only one instance of it available across the application. That solution has a side effect where you need to have more than one instance of the same dependency injecting into different parts of the application. Because the Angular 2 application is a tree of components, the framework has a **Hierarchical Dependency Injection** system – the tree of injectors exists in parallel to the component tree of application. Every component has an injector, of its own or shared with another components at the same level in the tree. When the component at the bottom of the tree requests a dependency, the Angular tries to find it with a provider registered in that components injector. If the provider doesn't exist on this level, the injector passes the request to its parent injector and so on until finding the injector that can handle the request. The Angular throws an exception if it runs out of ancestors. This solution helps us to create different instances of the same dependency on various levels and components. The particular service instance is still a singleton, but only in the scope of the host component instance and its children.

Category Service

I mentioned in Chapter 5 about the necessity to decouple the data from the presentation logic when to implement SPA. I partially realized it in `Category` and `Product`. The `CategoryListComponent` and `WelcomeComponent` use `categories` returns from `getCategories` function. Right now it is not suffering, but when we start getting and updating data from the server, we will need more functions. Better hide the implementation detail inside the single reusable data service class to use it in multiple components.

Let's refactor the category data acquisition business to a single service that provides categories, and share that service with all components that need them.

Rename the `category.ts` into `category.service.ts` to follow a name convention in which we spell the name of a service in lowercase followed by `.service`. If the service

name is multi-word, we will spell the base filename in lower dash-case. Add import statement to the top of the file:

```
import {Injectable} from '@angular/core';
```

Now create the `CategoryService` class and move the `categories` variable, `getCategories` and `getCategory` functions inside:

```
@Injectable()
export class CategoryService {
  categories: Category[] = [
    { id: '1', title: 'Bread & Bakery', imageL:
'http://placeholder.it/1110x480', imageS: 'http://placeholder.it/270x171',
desc: 'The best cupcakes, cookies, cakes, pies, cheesecakes, fresh bread,
biscotti, muffins, bagels, fresh coffee and more.' },
    { id: '2', title: 'Takeaway', imageL:
'http://placeholder.it/1110x480', imageS: 'http://placeholder.it/270x171',
desc: 'It's consistently excellent, dishes are superb and healthily cooked
with high quality ingredients.' },
    { id: '3', title: 'Dairy', imageL: 'http://placeholder.it/1110x480',
imageS: 'http://placeholder.it/270x171', desc: 'A dairy product is food
produced from the milk of mammals, primarily cows, water buffaloes, goats,
sheep, yaks, horses.' },
    { id: '4', title: 'Meat', imageL: 'http://placeholder.it/1110x480',
imageS: 'http://placeholder.it/270x171', desc: 'Only superior quality beef,
lamb, and pork.' },
    { id: '5', title: 'Seafood', imageL:
'http://placeholder.it/1110x480', imageS: 'http://placeholder.it/270x171',
desc: 'Great place to buy fresh seafood.' },
    { id: '6', title: 'Fruit & Veg', imageL:
'http://placeholder.it/1110x480', imageS: 'http://placeholder.it/270x171',
desc: 'A variety of fresh fruits and vegetables.' }
  ];

  getCategories() {
    return this.categories;
  }

  getCategory(id: string): Category {
    for (let i = 0; i < this.categories.length; i++) {
      if (this.categories[i].id === id) {
        return this.categories[i];
      }
    }
    throw new CategoryNotFoundException(
      `Category ${id} not found`);
  }
}
```

Don't forget to add `this` to all references to `categories` property.

Injector Provider for Category Service

We must register a service provider with the injector to tell Angular how to create the service. The best place to do that is in the `providers` property of a component. Because we need only one instance of `categories` per application, we can register it in the `AppComponent`. Open the `app.component.ts` file, import the `CategoryService` and change `@Component` decorator with the following code:

```
@Component({
  selector: 'my-app',
  templateUrl: 'app/app.component.html',
  directives: [ROUTER_DIRECTIVES, NavbarComponent,
    FooterComponent],
  precompile: [WelcomeComponent, ProductListComponent],
  providers: [CategoryService]
})
export class AppComponent {
}
```

Open the terminal and run the TypeScript compiler and server with command:

```
npm start
```

We get full bunch of issues, mostly about the wrong name of file `category.ts` we renamed into `category.service.ts`. That issue we can easily fix. Another problem – usage of functions `getCategory` and `getCategories`. To fix that issue we need import the `CategoryService`:

```
import {Category, CategoryService} from './category.service';
```

And inject it in constructors in all necessary places such that:

```
export class CategoryListComponent {

  categories: Category[];

  constructor(private router: Router,
    private categoryService: CategoryService) {
    this.categories = this.categoryService.getCategories();
  }

  filterProducts(category: Category) {
    this.router.navigate(['/products'],
```

```
        { queryParams: { category: category.id} });  
    }  
}
```

Move initialization of all variables inside the constructor for now similarly to categories in example above.

Product Service

Rename the `product.ts` into `product.service.ts`. Create class `ProductService` and move the `products` variable, `getProducts` and `getProduct` functions into it:

```
export class ProductService {  
  
    private products: Product[] = [  
    // ...  
    ];  
    getProducts(category?: string, search?: string) {  
        if (category) {  
            return this.products.filter((product: Product, index: number,  
array: Product[]) => {  
                return product.categoryId === category;  
            });  
        } else if (search) {  
            let lowSearch = search.toLowerCase();  
            return this.products.filter((product: Product, index: number,  
array: Product[]) => {  
                return product.title.toLowerCase().indexOf(lowSearch) !=  
-1;  
            });  
        } else {  
            return this.products;  
        }  
    }  
  
    getProduct(id: string): Product {  
        for (let i = 0; i < this.products.length; i++) {  
            if (this.products[i].id === id) {  
                return this.products[i];  
            }  
        }  
        throw new ProductNotFoundException(`Product ${id} not found`);  
    }  
}
```

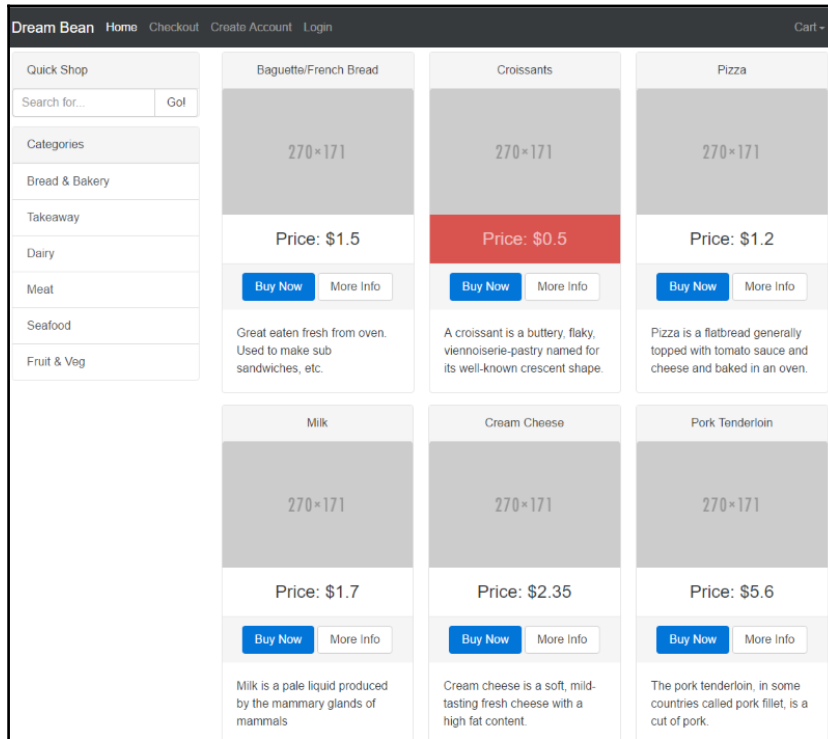
Fix the import in all classes have references on old methods.

Injector Provider for Product Service

We follow the same procedure for `ProductService` to register a service provider. Because we need only one instance of service per application, we can register it in the `AppComponent`. Open the `app.component.ts` file, import the `ProductService` and change `@Component` decorator with the following code:

```
@Component({
  selector: 'my-app',
  templateUrl: 'app/app.component.html',
  directives: [ROUTER_DIRECTIVES, NavbarComponent,
    FooterComponent],
  precompile: [WelcomeComponent, ProductListComponent],
  providers: [CategoryService, ProductService]
})
export class AppComponent {
}
```

Now restart application to see all of your products:



The Shopping Cart

A shopping cart is a piece of software that acts as an online store's catalog and allows users to select items for eventual purchase. It's known as a shopping basket. A shopping cart (or basket) allows a user to collect items while browsing an online catalog of products. The user should click on the **Buy Now** button to add the selected item into the cart. The total amount and a number of items in the cart presents in Navigation Bar component. The user is able to move to a checkout or view the cart to manage the number of purchased items.

The cart must store the items the user puts in the cart. The items should be:

- Fetchable to display the cart content
- Updatable to change the quantity of an item in the cart
- Removable

With this in mind, let's first create the basic cart functionality: adding, updating, and deleting items and define a barebones item class and walk through the code usage.

The Cart Model and Cart Item

At the beginning, the `Cart` class needs an internal array for storing all the items in the cart:

```
export class Cart {
  count: number = 0;
  amount: number = 0;
  items: CartItem[] = [];
}
```

Next, it must count the number and keep the amount of all items. The `CartItem` is an interface defining the structure of data the cart can use:

```
export interface CartItem {
  product: Product;
  count: number;
  amount: number;
}
```

The Cart Service

The `CartService` keep the cart instance to make it available across whole application:

```
cart: Cart = new Cart();
```

The `addProduct` method should add items to the cart:

```
addProduct(product: Product) {
  // Find CartItem in items
  let item: CartItem = this.findItem(product.id);
  // Check was it found?
  if (item) {
    // Item was found.
    // Increase the count of the same products
    item.count++;
    // Increase amount of the same products
    item.amount += product.price;
  } else {
    // Item was not found.
    // Create the cart item
    item = {
      product: product,
```

```
        count: 1,
        amount: product.price
    };
    // Add item to items
    this.cart.items.push(item);
}
// Increase count in the cart
this.cart.count++;
// Increase amount in the cart
this.cart.amount += product.price;
}
```

The method takes one argument of type `Product` and tries to find the item contains the same one. The method needs to increment the number of products and increase the amount of found cart item. Otherwise, it creates the new `CartItem` instance and assigns the product to it. After all, it is growing the total number of items and amount in the shopping cart.

Next, the `removeProduct` method of the class can be used to remove the product quickly from the cart:

```
removeProduct(product: Product) {
    // Find CartItem in items
    let item: CartItem = this.findItem(product.id);
    // Check is item found?
    if (item) {
        // Decrease the count
        item.count--;
        // Check was that the last product?
        if (!item.count) {
            // It was last product
            // Delete item from items
            this.remove(item);
        }
        // Decrease count in the cart
        this.cart.count--;
        // Decrease amount in the cart
        this.cart.amount -= product.price;
    }
}
```

The method takes one argument of type `product` and tries to find the item contains the same one. The method needs to decrement the number of the goods associated with this item cart. It removes the cart item includes no one product. In the end, it reduces the total number of items and amount in the shopping cart.

Method `removeItem` removes the particular item and reduces the total number of items and amount in the shopping cart:

```
removeItem(item: CartItem) {  
    // Delete item from items  
    this.remove(item);  
    // Decrease count in the cart  
    this.cart.count -= item.count;  
    // Decrease amount in the cart  
    this.cart.amount -= item.amount;  
}
```

The following private method `findItem` helps to find `CartItem` by `Product id`:

```
private findItem(id: string): CartItem {  
    for (let i = 0; i < this.cart.items.length; i++) {  
        if (this.cart.items[i].product.id === id) {  
            return this.cart.items[i];  
        }  
    }  
    return null;  
}
```

The last private `remove` decreases the number of items in the cart:

```
private remove(item: CartItem) {  
    // Find the index of cart item  
    let indx: number = this.cart.items.indexOf(item);  
    // Check was item found  
    if (indx !== -1) {  
        // Remove element from array  
        this.cart.items.splice(indx, 1);  
    }  
}
```

The Cart Menu Component

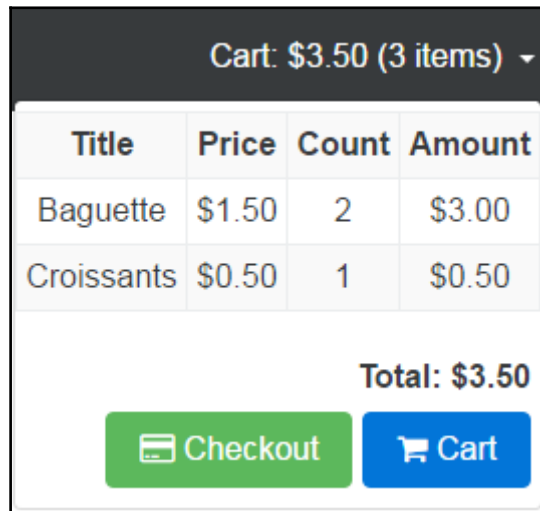
The key aspect that I find must present on the shopping cart design is that at first glance the user should be able to find out how many items are there in the shopping cart. You need to keep your user informed about how items are in the shopping cart so that users are aware of what they have added to the cart without having to use the dropdown.



Make sure shoppers can easily see the items in their cart and that they appear above the fold rather than another page.

That is quite significant UX design pattern. If you keep shopping cart content somewhere in

the sidebar or near the top right of your pages, you are taking away extra steps in the checkout process and make it easier for shoppers to move throughout the site and keep track of items and order totals the whole time.



With this in mind, let's create CartMenu component:

```
import {Component, Input} from '@angular/core';
import {ROUTER_DIRECTIVES} from '@angular/router';

import {Cart, CartService} from '../cart.service';

@Component({
  selector: 'db-cart-menu',
  templateUrl: 'app/cart/cart-menu.component.html',
  directives: [ROUTER_DIRECTIVES]
})
export class CartMenuComponent {

  private cart: Cart;

  constructor(private cartService: CartService) {
    this.cart = this.cartService.cart;
  }
}
```

The purpose of the local `cart` variable is to represent on view the content and update it with changes happens after user adds or removes the product to the cart.

We display the total number of items and amount in the label of dropdown menu:

```
<a class="nav-link dropdown-toggle" data-toggle="dropdown"
  href="#" role="button" aria-haspopup="true"
  aria-expanded="false">
  Cart: {{cart.amount | currency:'USD':true:'1.2-2'}}
    ({{cart.count}} items)
</a>
```

Take attention on the currency pipe with following parameters:

- The first parameter is the ISO 4217 currency code, such as “USD” for the US dollar and “EUR” for the euro.
- * On the second place is a boolean indicating whether to use the currency symbol (e.g. \$) or the currency code (e.g. USD) in the output
- On the last place we add the digit info in the next format:
minIntegerDigits.minFractionDigits-maxFractionDigits

I recommend using this pipe here and in all other places where you need to display the amount of currency.

We display the content of cart inside Bootstrap 4 table:

```
<div class="table-responsive">
  <table class="table table-sm table-striped table-bordered
    table-cart">
    <tbody>
      <tr>
        <td class="font-weight-bold">Title</td>
        <td class="font-weight-bold">Price</td>
        <td class="font-weight-bold">Count</td>
        <td class="font-weight-bold">Amount</td>
      </tr>
      <tr *ngFor="let item of cart.items">
        <td>{{item.product.title}}</td>
        <td>{{item.product.price |
          currency:'USD':true:'1.2-2'}}</td>
        <td>{{item.count}}</td>
        <td>{{item.amount |
          currency:'USD':true:'1.2-2'}}</td>
      </tr>
    </tbody>
  </table>
</div>
```

At the bottom of the menu we display the total amount and two buttons to navigate to Cart and Checkout:

```
<div class="row">
  <div class="col-md-12">
    <div class="total-cart pull-xs-right">
      <b>Total:
        {{cart.amount | currency:'USD':true:'1.2-2'}}
      </b>
    </div>
  </div>
</div>
<div class="row">
  <div class="col-md-12">
    <a [routerLink]="['/cart']"
      class="btn btn-primary pull-xs-right btn-cart">
      <i class="fa fa-shopping-cart" aria-hidden="true"></i>
      Cart
    </a>
    <a [routerLink]="['/checkout']"
      class="btn btn-success pull-xs-right btn-cart">
      <i class="fa fa-credit-card" aria-hidden="true"></i>
      Checkout
    </a>
  </div>
</div>
</div>
```

Update the Cart via Service

And the last thing we must do is inject the `CartService` into the `ProductGrid` component and start to listen to 'addToCart' events. In the method with the same name we call the `addProduct` of `CartService` to add the selected goodie into the shopping cart:

```
addToCart(product:Product) {
  this.cartService.addProduct(product);
}
```

Now, try to click on **Buy Now** on different products and see changes happened in the Navigation Bar. Click dropdown to display the shopping cart content:

The screenshot displays the Dream Bean web application interface. At the top, a navigation bar includes links for Home, Checkout, Create Account, and Login, along with a shopping cart icon showing a total of \$9.10 for 4 items. The main content area is divided into three columns. The left column features a 'Quick Shop' section with a search bar and a 'Go!' button, followed by a 'Categories' list including Bread & Bakery, Takeaway, Dairy, Meat, Seafood, and Fruit & Veg. The middle column displays two product cards: 'Pork Tenderloin' priced at \$5.6 and 'Ribs' priced at \$4.85. Each card includes a placeholder image (270x171), a 'Buy Now' button, a 'More Info' button, and a brief description. The right column shows a shopping cart table with the following items: Baguette (\$1.50, 2 units, \$3.00), Croissants (\$0.50, 1 unit, \$0.50), and Pork Tenderloin (\$5.60, 1 unit, \$5.60). The total cart amount is \$9.10. Below the table are 'Checkout' and 'Cart' buttons, and a 'Price: \$9.2' label. At the bottom, a 'Contact Info' section provides the address: 0000 Market St, Suite 000, San Francisco, CA 00000, and contact details: (123) 456-7890 and support@dream-bean.com.

Title	Price	Count	Amount
Baguette	\$1.50	2	\$3.00
Croissants	\$0.50	1	\$0.50
Pork Tenderloin	\$5.60	1	\$5.60

Total: \$9.10

Checkout Cart

Price: \$9.2

Buy Now More Info

Buy Now More Info

Buy Now More Info

Contact Info
0000 Market St, Suite 000, San Francisco, CA 00000, (123) 456-7890, support@dream-bean.com

Summary

You are familiar with dependency injection that Angular relies heavily on. As we've seen, we split our Angular code into visual components and services. Each of them depends on upon one another, and dependency injection provides referential transparency. Dependency Injection allows us to tell Angular what services our visual components depends on upon, and the framework will resolve these for us.

We created the classes for products and categories to hide the functionality into reusable services. Plus we created the shopping cart component and service and wired the last to products, so the user can add the products to the cart.

In Chapter 7, Working with Forms, we will talk about how to use Angular 2 directives related to form creation and how to link a code based form component to the HTML form. Plus we will continue to build the project we started to develop in previous chapters.