How Angular Works

In this chapter, we're going to talk about the high-level concepts of Angular. We're going to take a step back so that we can see how all the pieces fit together.s

> ℹ️ If you've used AngularJS 1.x, you'll notice that Angular has a new mental-model for building applications. Don't panic! As AngularJS 1.x users ourselves we've found Angular to be both straightforward and familiar. A little later in this book we're going to talk specifically about how to convert your AngularJS 1.x apps to Angular.

In the chapters that follow, we won't be taking a deep dive into each concept, but instead we're going to give an overview and explain the foundational ideas.

The first big idea is that an Angular application is made up of *Components*. One way to think of Components is a way to teach the browser new tags. If you have an Angular 1 background, Components are analogous to *directives* in AngularJS 1.x (it turns out, Angular has directives too, but we'll talk more about this distinction later on).

However, Angular Components have some significant advantages over AngularJS 1.x directives and we'll talk about that below. First, let's start at the top: the Application.
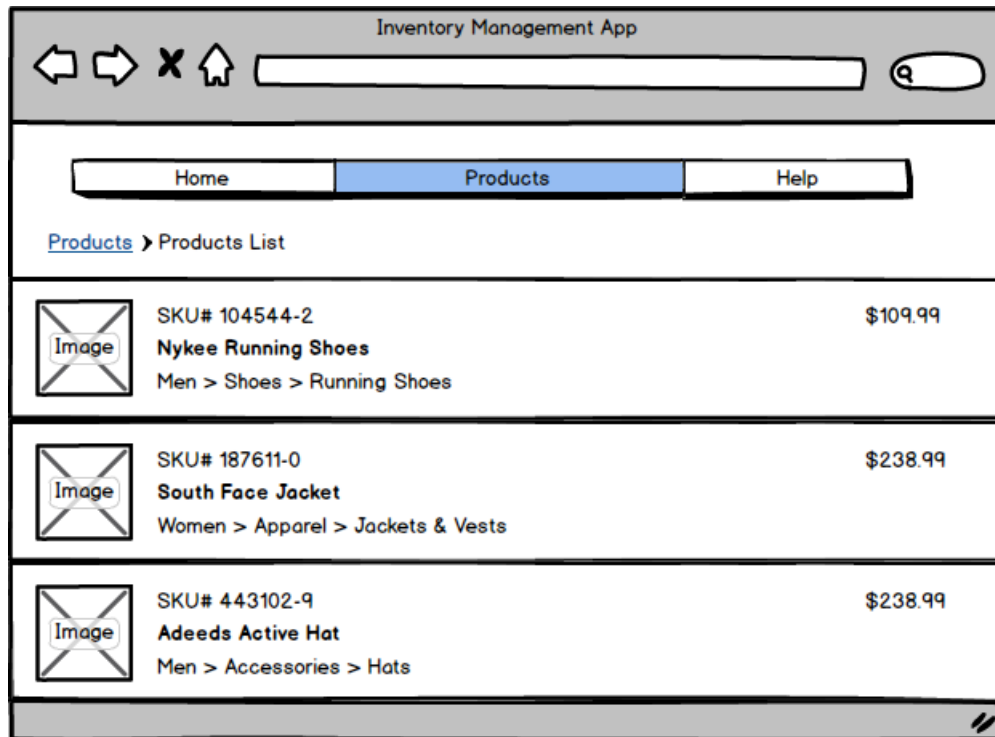
Application

An Angular Application is nothing more than a tree of Components.

At the root of that tree, the top level Component is the application itself. And that's what the browser will render when "booting" (a.k.a *bootstrapping*) the app.

One of the great things about Components is that they're **composable**. This means that we can build up larger Components from smaller ones. The Application is simply a Component that renders other Components.

Because Components are structured in a parent/child tree, when each Component renders, it recursively renders its children Components.

For example, let's create a simple inventory management application that is represented by the following page mockup:

Inventory Management App

Given this mockup, to write this application the first thing we want to do is split it into components.

In this example, we could group the page into three high level components

1. The Navigation Component
2. The Breadcrumbs Component
3. The Product List Component

The Navigation Component

This component would render the navigation section. This would allow the user to visit other areas of the application.
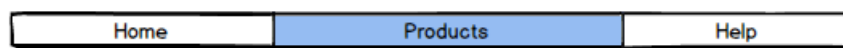
Navigation Component

The Breadcrumbs Component

This would render a hierarchical representation of where in the application the user currently is.

Products ❯ Products List

Breadcrumbs Component

The Product List Component

The Products List component would be a representation of a collection of products.

Product List Component

Breaking this component down into the next level of smaller components, we could say that the Product List is composed of multiple Product Rows.

Product Row Component

And of course, we could continue one step further, breaking each Product Row into smaller pieces:

- the **Product Image** component would be responsible for rendering a product image, given its image name
- the **Product Department** component would render the department tree, like *Men > Shoes > Running Shoes*
- the **Price Display** component would render the price. Imagine that our implementation customizes the pricing if the user is logged in to include system-wide tier discounts or include shipping for instance. We could implement all this behavior into this component.

Finally, putting it all together into a tree representation, we end up with the following diagram:
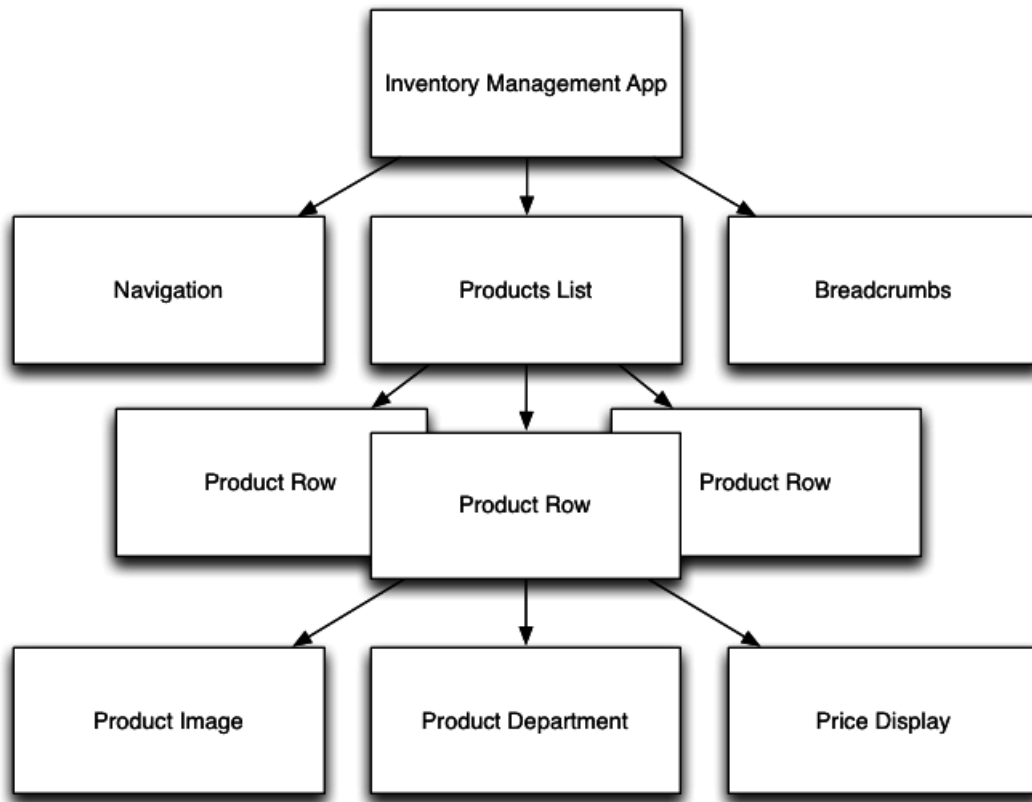
App Tree Diagram

At the top we see **Inventory Management App**: that's our application.

Under the application we have the Navigation, the Breadcrumb and the Products List components.

The Products List component has Product Rows, one for each product.

And the Product Row uses three components itself: one for the image, the department, and the price.

Let's work together to build this application.

> You can find the full code listing for this chapter in the downloads under `how-angular-works/inventory-app`.

Here's a screenshot of what our app will look like when we're done:
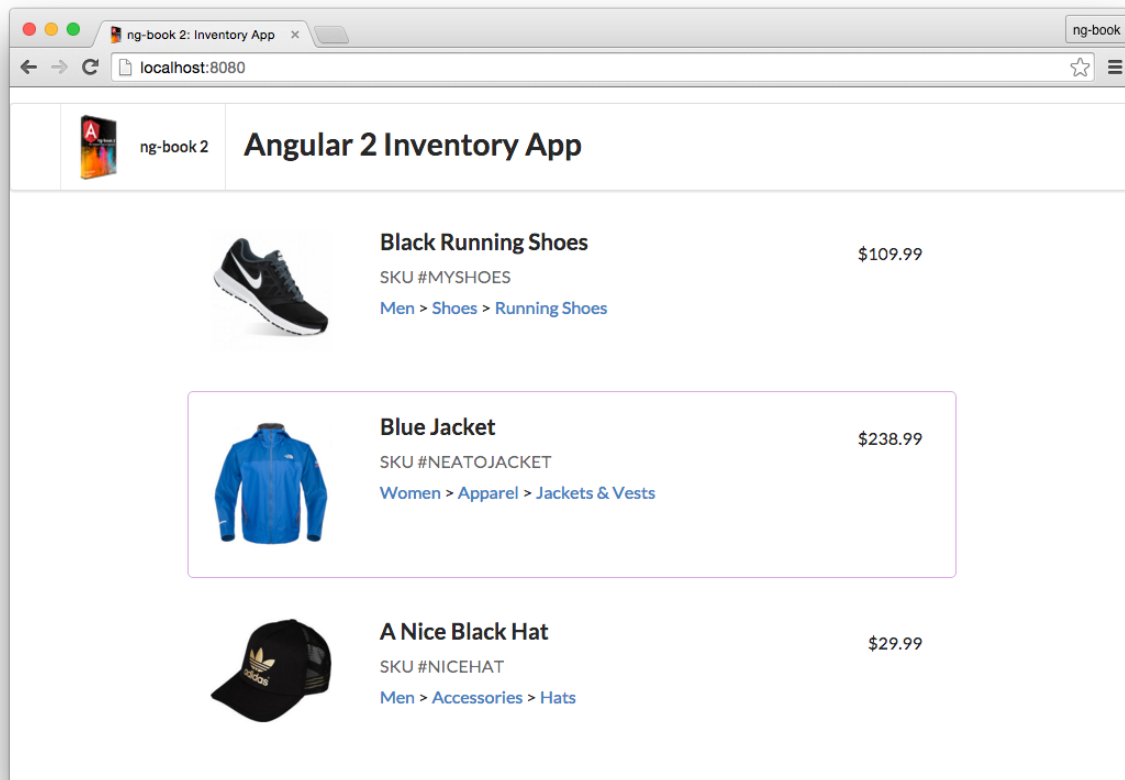
Completed Inventory App

How to Use This Chapter

In this chapter we're going to explain the fundamental concepts required when building Angular apps by walking through an app that we've built. We'll explain:

- How to break your app into components
- How to make reusable components using *inputs*
- How to handle user interactions, such as *clicking* on a component

In this chapter, we've used `angular-cli`, just as we did before. This means you can use all of the normal `ng` commands such as:

```
1  ng serve # runs the app
```

Also, in this chapter, we're not going to give step-by-step instructions on how to create each file in the app. If you'd like to follow along at home, when we introduce a new component you can run:

```
1  ng generate component NameOfNewComponentHere
```

This will generate the files you need, and you're free to type in your code there or copy and paste code from the book or from our example code.

We've provided the entire, completed application in the code download folder under `how-angular-works/inventory-app`. If you ever feel lost or need more context, take some time to look at the completed example code.

Let's get started building!

Product Model

One of the key things to realize about Angular is that it **doesn't prescribe a particular model library**.

Angular is flexible enough to support many different kinds of models (and data architectures). However, this means the choice is left to you as the user to determine how to implement these things.

We'll have **a lot** to say about data architectures in future chapters. For now, though, we're going to have our models be plain JavaScript objects.

code/how-angular-works/inventory-app/src/app/product.model.ts

```
1  /**
2   * Provides a `Product` object
3   */
4  export class Product {
5    constructor(
6      public sku: string,
7      public name: string,
8      public imageUrl: string,
9      public department: string[],
10     public price: number) {
11   }
12 }
```

If you're new to ES6/TypeScript this syntax might be a bit unfamiliar.

We're creating a new `Product` class and the `constructor` takes 5 arguments. When we write `public sku: string`, we're saying two things:

- there is a `public` variable on instances of this class called `sku`

- sku is of type string.

> 🔑 If you're already familiar with JavaScript, you can quickly catch up on some of the differences, including the public constructor shorthand, here at learnxinyminutes[39]

This Product class doesn't have any dependencies on Angular, it's just a model that we'll use in our app.

Components

As we mentioned before, Components are the fundamental building block of Angular applications. The "application" itself is just the top-level Component. Then we break our application into smaller child Components.

> 🔑 When building new Angular applications, we often follow this process: we mockup the design in wireframes (or on paper) and then we break down the parts into Components.

We'll be using Components a lot, so it's worth looking at them more closely.

Each component is composed of three parts:

- Component *Decorator*
- A View
- A Controller

To illustrate the key concepts we need to understand about components, we'll start with the top level Inventory App and then focus on the **Products List** and child components:

---

[39]https://learnxinyminutes.com/docs/typescript/
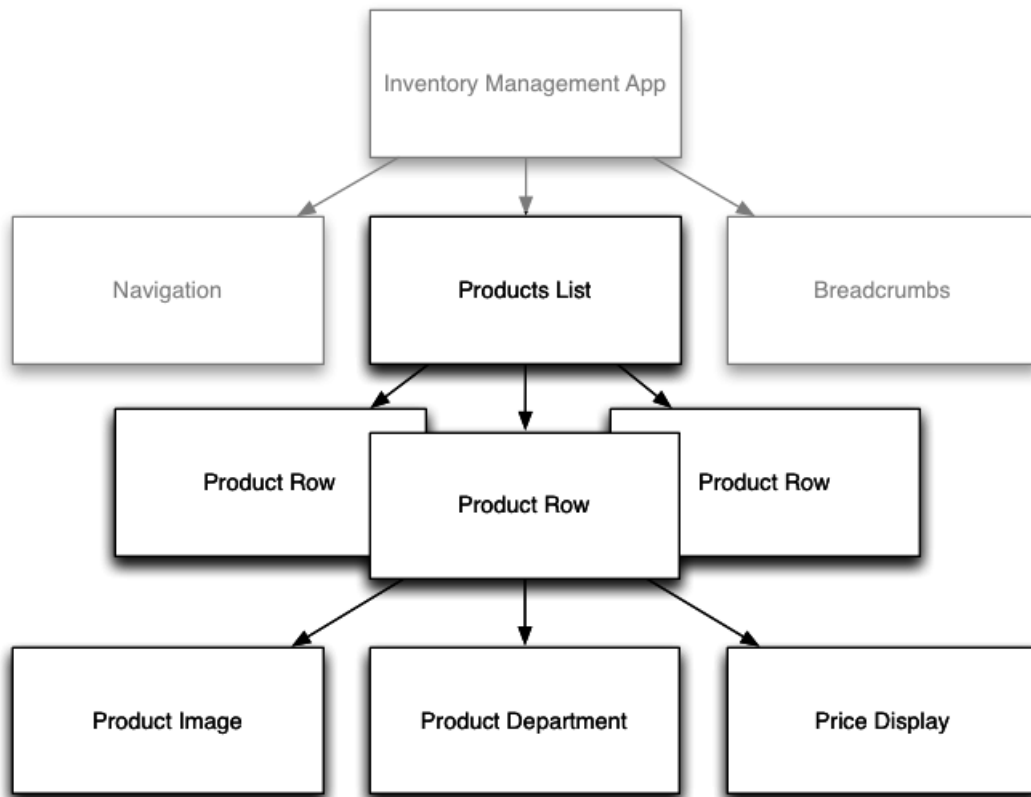
Products List Component

Here's what a basic, top-level `AppComponent` looks like:

```
1   @Component({
2     selector: 'inventory-app-root',
3     template: `
4     <div class="inventory-app">
5        (Products will go here soon)
6     </div>
7     `
8   })
9   export class AppComponent {
10    // Inventory logic here
11  }
```

> If you've been using Angular 1 the syntax might look pretty foreign! But the ideas are pretty similar, so let's take them step by step.

The @Component is called a **decorator**. It adds metadata to the class that follows it (AppComponent). The @Component decorator specifies:

- a selector, which tells Angular what element to match
- a template, which defines the view

The Component **controller** is defined by a class, the AppComponent class, in this case.

Let's take a look into each part now in more detail.

Component Decorator

The @Component decorator is where you configure your component. One of the primary roles of the @Component decorator is to configure how the outside world will interact with your component.

> There are lots of options available to configure a component (many of which we cover in the Advanced Components Chapter). In this chapter we're just going to touch on the basics.

Component selector

With the selector key, you indicate how your component will be recognized when used in a template. The idea is similar to CSS or XPath selectors. The selector is a way to define what elements in the HTML will match this component. In this case, by saying selector: 'inventory-app-root', we're saying that in our HTML we want to match the inventory-app-root tag, that is, we're defining a new tag that has new functionality whenever we use it. E.g. when we put this in our HTML:

```
1    <inventory-app-root></inventory-app-root>
```

Angular will use the AppComponent component to implement the functionality.

Alternatively, with this selector, we can also use a regular div and specify the component as an attribute:

```
1    <div inventory-app-root></div>
```

Component template

The view is the visual part of the component. By using the template option on @Component, we declare the HTML template that the component will use:

```
1    @Component({
2      selector: 'inventory-app-root',
3      template: `
4      <div class="inventory-app">
5         (Products will go here soon)
6      </div>
7      `
8    })
```

For the `template` above, notice that we're using TypeScript's backtick multi-line string syntax. Our template so far is pretty sparse: just a `div` with some placeholder text.

We can also move our template out to a separate file and use `templateUrl` instead:

```
1  @Component({
2    selector: 'inventory-app-root',
3    templateUrl: './app.component.html'
4  })
5  export class AppComponent {
6    // Inventory logic here
7  }
```

Adding A `Product`

Our app isn't very interesting without `Products` to view. Let's add some now.

We can create a new `Product` like this:

```
1    // this is just an example of how to use Product,
2    // we'll do something similar in our Angular code in a moment
3
4    // first, we have to import `Product` so that we can use it
5    import { Product } from './product.model';
6
7    // now we can create a new `Product`
8    let newProduct = new Product(
9        'NICEHAT',                               // sku
10       'A Nice Black Hat',                      // name
11       '/assets/images/products/black-hat.jpg', // imageUrl
12       ['Men', 'Accessories', 'Hats'],          // department
13       29.99);                                  // price
```

Our constructor for `Product` takes 5 arguments. We can create a new `Product` by using the `new` keyword.

> ℹ️ Normally, I probably wouldn't pass more than a few arguments to a function. Another option here is to configure the `Product` class to take an Object in the constructor, then we wouldn't have to remember the order of the arguments. That is, `Product` could be changed to do something like this:

```
1    new Product({sku: "MYHAT", name: "A green hat"})
```

> But for now, this 5 argument constructor is easy to use.

We want to be able to show this `Product` in the view. In order to make properties accessible to our template **we add them as instance variables to the Component**.

For example, if we want to access `newProduct` in our view we could write:

```
1  class AppComponent {
2    product: Product;
3
4    constructor() {
5      let newProduct = new Product(
6            'NICEHAT',
7            'A Nice Black Hat',
8            '/resources/images/products/black-hat.jpg',
9            ['Men', 'Accessories', 'Hats'],
10           29.99);
11
12     this.product = newProduct;
13   }
14 }
```

or more concisely:

```
 1  class AppComponent {
 2    product: Product;
 3
 4    constructor() {
 5      this.product = new Product(
 6            'NICEHAT',
 7            'A Nice Black Hat',
 8            '/resources/images/products/black-hat.jpg',
 9            ['Men', 'Accessories', 'Hats'],
10            29.99);
11    }
12  }
```

Notice that we did three things here:

1. **We added a `constructor`** - When Angular creates a new instance of this Component, it calls the `constructor` function. This is where we can put setup for this Component.
2. **We described an instance variable** - On `AppComponent`, when we write: `product: Product`, we're specifying that the `AppComponent` instances have a property `product` which is a `Product` object.
3. **We assigned a `Product` to `product`** - In the `constructor` we create an instance of `Product` and assigned it to the instance variable

Viewing the `Product` with Template Binding

Now that we have `product` assigned to the `AppComponent` instance, we could use that variable in our view template:

```
 1  <div class="inventory-app">
 2    <h1>{{ product.name }}</h1>
 3    <span>{{ product.sku }}</span>
 4  </div>
```

Using the {{...}} syntax is called *template binding*. It tells the view we want to use the value of the expression inside the brackets at this location in our template.

So in this case, we have two bindings:

- `{{ product.name }}`
- `{{ product.sku }}`

The `product` variable comes from the instance variable `product` on our Component instance of `AppComponent`.

What's neat about template binding is that the code inside the brackets is *an expression*. That means you can do things like this:

- `{{ count + 1 }}`
- `{{ myFunction(myArguments) }}`

In the first case, we're using an operator to change the displayed value of `count`. In the second case, we're able to replace the tags with the value of the function `myFunction(myArguments)`. Using template binding tags is the main way that you'll show data in your Angular applications.

Adding More Products

In the code above, we're only able to show a single product in our app, but we want to be able to show a list of products. Let's change our `AppComponent` to store an array of `Products` rather than a single `Product`:

```
1  class AppComponent {
2    products: Product[];
3
4    constructor() {
5      this.products = [];
6    }
7  }
```

Notice that we've renamed the variable `product` to `products`, and we've changed the type to `Product[]`. The `[]` characters at the end mean we want `products` to be an `Array` of `Products`. We also could have written this as: `Array<Product>`.

Now that our `AppComponent` holds an array of `Products`. Let's create some `Products` in the constructor:

code/how-angular-works/inventory-app/src/app/app.component.ts

```
15  export class AppComponent {
16    products: Product[];
17
18    constructor() {
19      this.products = [
20        new Product(
21          'MYSHOES',
22          'Black Running Shoes',
23          '/assets/images/products/black-shoes.jpg',
24          ['Men', 'Shoes', 'Running Shoes'],
25          109.99),
26        new Product(
27          'NEATOJACKET',
28          'Blue Jacket',
29          '/assets/images/products/blue-jacket.jpg',
30          ['Women', 'Apparel', 'Jackets & Vests'],
31          238.99),
32        new Product(
33          'NICEHAT',
34          'A Nice Black Hat',
35          '/assets/images/products/black-hat.jpg',
36          ['Men', 'Accessories', 'Hats'],
37          29.99)
38      ];
39    }
```

This code will give us some `Products` to work with in our app.

Selecting a `Product`

We (eventually) want to support user interaction in our app. For instance, the user might *select* a particular product to view more information about the product, add it to the cart, etc.

Let's add some functionality here in our `AppComponent` to handle what happens when a new `Product` is selected. To do that, let's define a new function, `productWasSelected`:

code/how-angular-works/inventory-app/src/app/app.component.ts

```
41    productWasSelected(product: Product): void {
42      console.log('Product clicked: ', product);
43    }
```

This function accepts a single argument `product` and then it will log out that the product was passed in. We'll use this function in a bit.

Listing products using `<products-list>`

Now that we have our top-level `AppComponent` component, we need to add a new component for rendering a list of products. In the next section we'll create the implementation of a `ProductsList` component that matches the selector `products-list`. Before we dive into the implementation details, here's how we will *use* this new component in our template:

code/how-angular-works/inventory-app/src/app/app.component.html

```
1  <div class="inventory-app">
2    <products-list
3      [productList]="products"
4      (onProductSelected)="productWasSelected($event)">
5    </products-list>
6  </div>
```

There is some new syntax here, so let's talk about each part:

Inputs and Outputs

When we use `products-list` we're using a key feature of Angular components: inputs and outputs:

```
1      <products-list
2        [productList]="products"                          <!-- input -->
3        (onProductSelected)="productWasSelected($event)">  <!-- output -->
4      </products-list>
```

The `[squareBrackets]` pass inputs and the `(parentheses)` handle outputs.

Data flows *in* to your component via *input bindings* and events flow *out* of your component through *output bindings*.

Think of the set of input + output bindings as defining the **public API** of your component.

`[squareBrackets]` pass inputs

In Angular, you pass data into child components via *inputs*.

In our code where we show:

```
1        <products-list
2          [productList]="products"
```

We're using an *input* of the `ProductList` component.

It can be tricky to understand where `products`/`productList` are coming from. There are two sides to this attribute:

- `[productList]` (the left-hand side) and
- `"products"` (the right-hand side)

The left-hand side `[productList]` says we want to use the `productList` *input* of the `products-list` component (we'll show how to define that in a moment).

The right-hand side `"products"` says that we want to send the *value of the expression* `products`. That is, the array `this.products` in the `AppComponent` class.

> **?** You might ask, "how would I know that `productList` is a valid input to the `products-list` component? The answer is: you'd read the docs for that component. The `inputs` (and `outputs`) are part of the "public API" of a component.
>
> You'd know the inputs for a component that you're using in the same way that you'd know what the arguments are for a function that you're using.
>
> That said, we'll define the `products-list` component in a moment, and we'll see exactly how the `productList` input is defined.

`(parens)` handle outputs

In Angular, you send data out of components via *outputs*.

In our code where we show:

```
1        <products-list
2          ...
3          (onProductSelected)="productWasSelected($event)">
```

We're saying that we want to listen to the `onProductSelected` *output* from the `ProductsList` component.

That is:

- `(onProductSelected)`, the left-hand side is the name of the `output` we want to "listen" on

- "productWasSelected", the right-hand side is the **function we want to call** when something new is sent to this output
- $event is a *special variable* here that represents **the thing emitted on** (i.e. sent to) the output.

Now, we haven't talked about **how to define inputs or outputs** on our own components yet, but we will shortly when we define the ProductsList component. For now, know that we can pass data to child components through *inputs* (like "arguments" to a function) and we can receive data out of a child component through *outputs* (sort of like "return values" from a function).

Full AppComponent Listing

We broke the AppComponent up into several chunks above. So that we can see the whole thing together, here's the full code listing of our AppComponent:

code/how-angular-works/inventory-app/src/app/app.component.ts

```
 1  import {
 2    Component,
 3    EventEmitter
 4  } from '@angular/core';
 5
 6  import { Product } from './product.model';
 7
 8  /**
 9   * @InventoryApp: the top-level component for our application
10   */
11  @Component({
12    selector: 'inventory-app-root',
13    templateUrl: './app.component.html'
14  })
15  export class AppComponent {
16    products: Product[];
17
18    constructor() {
19      this.products = [
20        new Product(
21          'MYSHOES',
22          'Black Running Shoes',
23          '/assets/images/products/black-shoes.jpg',
24          ['Men', 'Shoes', 'Running Shoes'],
25          109.99),
26        new Product(
27          'NEATOJACKET',
```

```
28          'Blue Jacket',
29          '/assets/images/products/blue-jacket.jpg',
30          ['Women', 'Apparel', 'Jackets & Vests'],
31          238.99),
32        new Product(
33          'NICEHAT',
34          'A Nice Black Hat',
35          '/assets/images/products/black-hat.jpg',
36          ['Men', 'Accessories', 'Hats'],
37          29.99)
38      ];
39    }
40
41    productWasSelected(product: Product): void {
42      console.log('Product clicked: ', product);
43    }
44  }
```

and the template:

code/how-angular-works/inventory-app/src/app/app.component.html

```
1  <div class="inventory-app">
2    <products-list
3      [productList]="products"
4      (onProductSelected)="productWasSelected($event)">
5    </products-list>
6  </div>
```

The `ProductsListComponent`

Now that we have our top-level application component, let's write the `ProductsListComponent`, which will render a list of product rows.

We want to allow the user to select **one** `Product` and we want to keep track of which `Product` is the currently selected one. The `ProductsListComponent` is a great place to do this because it "knows" all of the `Products` at the same time.

Let's write the `ProductsListComponent` in three steps:

- Configuring the `ProductsListComponent` `@Component` options
- Writing the `ProductsListComponent` controller class
- Writing the `ProductsListComponent` view template

Configuring the `ProductsListComponent` `@Component` **Options**

Let's take a look at the `@Component` configuration for `ProductsListComponent`:

code/how-angular-works/inventory-app/src/app/products-list/products-list.component.ts

```
1  import {
2    Component,
3    EventEmitter,
4    Input,
5    Output
6  } from '@angular/core';
7  import { Product } from '../product.model';
8
9  /**
10  * @ProductsList: A component for rendering all ProductRows and
11  * storing the currently selected Product
12  */
13  @Component({
14    selector: 'products-list',
15    templateUrl: './products-list.component.html'
16  })
17  export class ProductsListComponent {
18    /**
19     * @input productList - the Product[] passed to us
20     */
21    @Input() productList: Product[];
22
23    /**
24     * @output onProductSelected - outputs the current
25     *         Product whenever a new Product is selected
26     */
27    @Output() onProductSelected: EventEmitter<Product>;
```

We start our `ProductsListComponent` with a familiar option: `selector`. This selector means we can place our `ProductsListComponent` with the tag `<products-list>`. We've also defined two properties `productList` and `onProductSelected`. Notice that `productList` has a `@Input()` annotation, denoting that it is an *input* and `onProductSelected` has an `@Output()` annotation, denoting that it is an *output*.

Component inputs

Inputs **specify the parameters we expect our component to receive**. To designate an input, we use the `@Input()` decoration on a component class property.

When we specify that a Component takes an input, it is expected that the definition class **will have an instance variable** that will receive the value. For example, say we have the following code:

```
1    import { Component, Input } from '@angular/core';
2
3    @Component({
4      selector: 'my-component',
5    })
6    class MyComponent {
7      @Input() name: string;
8      @Input() age: number;
9    }
```

The `name` and `age` inputs map to the `name` and `age` properties on instances of the `MyComponent` class.

> If we need to use two different names for the attribute and the property, we could for example write `@Input('firstname') name: String;`. But the Angular Style Guide[40] suggests to avoid this.

If we want to use `MyComponent` from another template, we write something like: `<my-component [name]="myName" [age]="myAge"></my-component>`.

Notice that the attribute `name` matches the input `name`, which in turn matches the `MyComponent` property `name`. However, these don't always have to match.

For instance, say we wanted our attribute key and instance property to differ. That is, we want to use our component like this:

```
1    <my-component [shortName]="myName" [oldAge]="myAge"></my-component>
```

To do this, we would change the format of the string in the `inputs` option:

```
1    @Component({
2      selector: 'my-component'
3    })
4    class MyComponent {
5      @Input('shortName') name: string;
6      @Input('oldAge') age: number;
7    }
```

- The **property name** (`name`, `age`) represent how that incoming property will be **visible ("bound") in the controller**.
- The **@Input argument** (`shortName`, `oldAge`) configures how the property is **visible to the "outside world"**.

---

[40]https://angular.io/docs/ts/latest/guide/style-guide.html

Passing `products` through via the inputs

If you recall, in our `AppComponent`, we passed `products` to our `products-list` via the `[productList]` input:

code/how-angular-works/inventory-app/src/app/app.component.html

```
1  <div class="inventory-app">
2    <products-list
3      [productList]="products"
4      (onProductSelected)="productWasSelected($event)">
5    </products-list>
6  </div>
```

Hopefully this syntax makes more sense now: we're passing the value of `this.products` (on the `AppComponent`) in via an input on `ProductsListComponent`.

Component outputs

When you want to send data from your component to the outside world, you use *output bindings*.

Let's say a component we're writing has a button and we need to do something when that button is clicked.

The way to do this is by binding the *click* output of the button to a method declared on our component's controller. You do that using the `(output)="action"` notation.

Here's an example where we keep a counter and increment (or decrement) based on which button is pressed:

```
1  @Component({
2    selector: 'counter',
3    template: `
4      {{ value }}
5      <button (click)="increase()">Increase</button>
6      <button (click)="decrease()">Decrease</button>
7    `
8  })
9  class Counter {
10   value: number;
11
12   constructor() {
13     this.value = 1;
14   }
```

```
15
16    increase() {
17      this.value = this.value + 1;
18      return false;
19    }
20
21    decrease() {
22      this.value = this.value - 1;
23      return false;
24    }
25  }
```

In this example we're saying that every time the first button is clicked, we want the `increase()` method on our controller to be invoked. And, similarly, when the second button is clicked, we want to call the `decrease()` method.

The parentheses attribute syntax looks like this: `(output)="action"`. In this case, the output we're listening for is the `click` event on this button. There are many other built-in events we can listen to such as: `mousedown`, `mousemove`, `dbl-click`, etc.

In this example, the event is *internal* to the component. That is, calling `increase()` increments `this.value`, but there's no effect that leaves this component. When creating our own components we can also expose "public events" (component `outputs`) that allow the component to talk to the outside world.

The key thing to understand here is that in a view, we can listen to an event by using the `(output)="action"` syntax.

Emitting Custom Events

Let's say we want to create a component that emits a custom event, like `click` or `mousedown` above. To create a custom output event we do three things:

1. Specify `outputs` in the `@Component` configuration
2. Attach an `EventEmitter` to the output property
3. Emit an event from the `EventEmitter`, at the right time

ℹ️ Perhaps `EventEmitter` is unfamiliar to you. Don't panic! It's not too hard.

An `EventEmitter` is an object that helps you implement the Observer Pattern[41]. That is, it's an object that will:

1. maintain a list of subscribers and
2. publish events to them.

That's it.

Here's a short and sweet example of how you can use `EventEmitter`

```
1  let ee = new EventEmitter();
2  ee.subscribe((name: string) => console.log(`Hello ${name}`));
3  ee.emit("Nate");
4
5  // -> "Hello Nate"
```

When we assign an `EventEmitter` to an output *Angular automatically subscribes* for us. You don't need to do the subscription yourself (though in a special situation you could add your own subscriptions, if you want to).

Here's an example of how we write a component that has `outputs`:

```
1   @Component({
2     selector: 'single-component',
3     template: `
4       <button (click)="liked()">Like it?</button>
5     `
6   })
7   class SingleComponent {
8     @Output() putRingOnIt: EventEmitter<string>;
9
10    constructor() {
11      this.putRingOnIt = new EventEmitter();
12    }
13
14    liked(): void {
15      this.putRingOnIt.emit("oh oh oh");
16    }
17  }
```

---

[41]https://en.wikipedia.org/wiki/Observer_pattern

Notice that we did all three steps: 1. specified `outputs`, 2. created an `EventEmitter` that we attached to the output property `putRingOnIt` and 3. Emitted an event when `liked` is called.

If we wanted to use this output in a parent component we could do something like this:

```
 1  @Component({
 2    selector: 'club',
 3    template: `
 4      <div>
 5        <single-component
 6          (putRingOnIt)="ringWasPlaced($event)"
 7          ></single-component>
 8      </div>
 9    `
10  })
11  class ClubComponent {
12    ringWasPlaced(message: string) {
13      console.log(`Put your hands up: ${message}`);
14    }
15  }
16
17  // logged -> "Put your hands up: oh oh oh"
```

Again, notice that:

- `putRingOnIt` comes from the `outputs` of `SingleComponent`
- `ringWasPlaced` is a function on the `ClubComponent`
- `$event` contains the thing that was emitted, in this case a `string`

Writing the `ProductsListComponent` Controller Class

Back to our store example, our `ProductsListComponent` controller class needs three instance variables:

- One to hold the list of Products (that come from the `productList` input)
- One to output events (that emit from the `onProductSelected` output)
- One to hold a reference to the currently selected product

Here's how we define those in code:

code/how-angular-works/inventory-app/src/app/products-list/products-list.component.ts

```
17  export class ProductsListComponent {
18    /**
19     * @input productList - the Product[] passed to us
20     */
21    @Input() productList: Product[];
22
23    /**
24     * @output onProductSelected - outputs the current
25     *          Product whenever a new Product is selected
26     */
27    @Output() onProductSelected: EventEmitter<Product>;
28
29    /**
30     * @property currentProduct - local state containing
31     *          the currently selected `Product`
32     */
33    private currentProduct: Product;
34
35    constructor() {
36      this.onProductSelected = new EventEmitter();
37    }
```

Notice that our `productList` is an Array of `Products` - this comes in from the `inputs`.

`onProductSelected` is our output.

`currentProduct` is a property internal to `ProductsListComponent`. You might also hear this being referred to as "local component state". It's only used here within the component.

Writing the `ProductsListComponent` View Template

Here's the template for our `products-list` component:

code/how-angular-works/inventory-app/src/app/products-list/products-list.component.html

```html
1  <div class="ui items">
2    <product-row
3      *ngFor="let myProduct of productList"
4      [product]="myProduct"
5      (click)='clicked(myProduct)'
6      [class.selected]="isSelected(myProduct)">
7    </product-row>
8  </div>
```

Here we're using the `product-row` tag, which comes from the `ProductRow` component, which we'll define in a minute.

We're using `ngFor` to iterate over each `Product` in `productList`. We've talked about `ngFor` before in this book, but just as a reminder the `let thing of things` syntax says, "iterate over `things` and create a copy of this element for each item, and assign each item to the variable `thing`".

So in this case, we're iterating over the `Products` in `productList` and generating a local variable `myProduct` for each one.

> Style-wise, I probably wouldn't call this variable `myProduct` in a real app. Instead, I'd probably call it `product`, or even `p`. But here I want to be explicit about what we're passing, and `myProduct` is slightly clearer because it let's us distinguish the 'local template variable' from the input `product`.

The interesting thing to note about this `myProduct` variable is that we can now use it *even on the same tag*. As you can see, we do this on the following three lines.

The line that reads `[product]="myProduct"` says that we want to pass `myProduct` (the local variable) to the input `product` of the `product-row`. (We'll define this input when we define the `ProductRow` component below.)

The `(click)='clicked(myProduct)'` line describes what we want to do when this element is clicked. `click` is a built-in event that is triggered when the host element is clicked on. In this case, we want to call the component function `clicked` on `ProductsListComponent` whenever this element is clicked on.

The line `[class.selected]="isSelected(myProduct)"` is a fun one: Angular allows us to set `classes` conditionally on an element using this syntax. This syntax says "add the CSS class `selected` if `isSelected(myProduct)` returns `true`." This is a really handy way for us to mark the currently selected product.

You may have noticed that we didn't define `clicked` nor `isSelected` yet, so let's do that now (in `ProductsListComponent`):

clicked

code/how-angular-works/inventory-app/src/app/products-list/products-list.component.ts

```
39    clicked(product: Product): void {
40      this.currentProduct = product;
41      this.onProductSelected.emit(product);
42    }
```

This function does two things:

1. Set `this.currentProduct` to the `Product` that was passed in.
2. Emit the `Product` that was clicked on our output

isSelected

code/how-angular-works/inventory-app/src/app/products-list/products-list.component.ts

```
44    isSelected(product: Product): boolean {
45      if (!product || !this.currentProduct) {
46        return false;
47      }
48      return product.sku === this.currentProduct.sku;
49    }
```

This function accepts a `Product` and returns `true` if `product`'s `sku` matches the `currentProduct`'s `sku`. It returns `false` otherwise.

The Full ProductsListComponent Component

Here's the full code listing so we can see everything in context:

code/how-angular-works/inventory-app/src/app/products-list/products-list.component.ts

```
1   import {
2     Component,
3     EventEmitter,
4     Input,
5     Output
6   } from '@angular/core';
7   import { Product } from '../product.model';
8
9   /**
10   * @ProductsList: A component for rendering all ProductRows and
```

```
11    * storing the currently selected Product
12    */
13  @Component({
14    selector: 'products-list',
15    templateUrl: './products-list.component.html'
16  })
17  export class ProductsListComponent {
18    /**
19     * @input productList - the Product[] passed to us
20     */
21    @Input() productList: Product[];
22
23    /**
24     * @output onProductSelected - outputs the current
25     *           Product whenever a new Product is selected
26     */
27    @Output() onProductSelected: EventEmitter<Product>;
28
29    /**
30     * @property currentProduct - local state containing
31     *             the currently selected `Product`
32     */
33    private currentProduct: Product;
34
35    constructor() {
36      this.onProductSelected = new EventEmitter();
37    }
38
39    clicked(product: Product): void {
40      this.currentProduct = product;
41      this.onProductSelected.emit(product);
42    }
43
44    isSelected(product: Product): boolean {
45      if (!product || !this.currentProduct) {
46        return false;
47      }
48      return product.sku === this.currentProduct.sku;
49    }
50
51  }
```

and the template:

code/how-angular-works/inventory-app/src/app/products-list/products-list.component.html

```html
1  <div class="ui items">
2    <product-row
3      *ngFor="let myProduct of productList"
4      [product]="myProduct"
5      (click)='clicked(myProduct)'
6      [class.selected]="isSelected(myProduct)">
7    </product-row>
8  </div>
```
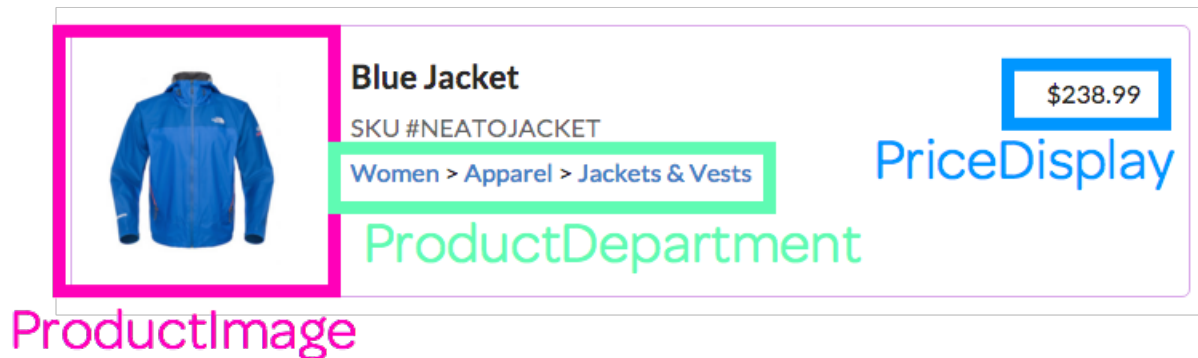
The `ProductRowComponent` Component
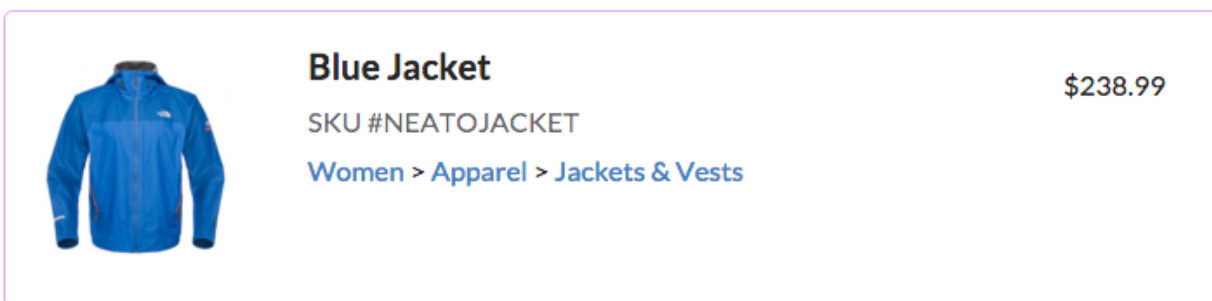
A Selected Product Row Component

Our `ProductRowComponent` displays our `Product`. `ProductRowComponent` will have its own template, but will also be split up into three smaller Components:

- `ProductImageComponent` - for the image
- `ProductDepartmentComponent` - for the department "breadcrumbs"
- `PriceDisplayComponent` - for showing the product's price

Here's a visual of the three Components that will be used within the `ProductRowComponent`:

ProductRowComponent's Sub-components

Let's take a look at the `ProductRowComponent`'s Component configuration, definition class, and template:

`ProductRowComponent` **Configuration**

The `ProductRowComponent` uses a lot of the ideas we've covered so far:

code/how-angular-works/inventory-app/src/app/product-row/product-row.component.ts

```
 1  import {
 2    Component,
 3    Input,
 4    HostBinding
 5  } from '@angular/core';
 6  import { Product } from '../product.model';
 7
 8  /**
 9   * @ProductRow: A component for the view of single Product
10   */
11  @Component({
12    selector: 'product-row',
13    templateUrl: './product-row.component.html',
14  })
15  export class ProductRowComponent {
16    @Input() product: Product;
17    @HostBinding('attr.class') cssClass = 'item';
18  }
```

We start by defining the `selector` of `product-row`. We've seen this several times now - this defines that this component will match the tag `product-row`.

Next we define that this row takes an @Input of product. This instance variable will be set to the
Product that was passed in from our parent Component.

The HostBinding decoration is new - it lets us **set attributes on the host element**. The *host* is the
element this component is attached to.

In this case, we're using the Semantic UI item class[42]. Here when we say @HostBinding('attr.class')
cssClass = 'item'; we're saying that we want to attach the CSS class item to the host element.

> **i** Using host is nice because it means we can configure our host element from *within* the
> component. This is great because otherwise we'd require the host element to specify the
> CSS tag and that is bad because we would then make assigning a CSS class part of the
> requirement to using the Component.

Instead of putting a long template string in our TypeScript file, instead we're going to move the
template to a separate HTML file and use a templateUrl to load it. We'll talk about the template in
a minute.

ProductRowComponent template

Now let's take a look at the template:

code/how-angular-works/inventory-app/src/app/product-row/product-row.component.html

```
1  <product-image [product]="product"></product-image>
2  <div class="content">
3    <div class="header">{{ product.name }}</div>
4    <div class="meta">
5      <div class="product-sku">SKU #{{ product.sku }}</div>
6    </div>
7    <div class="description">
8      <product-department [product]="product"></product-department>
9    </div>
10 </div>
11 <price-display [price]="product.price"></price-display>
```

Our template doesn't have anything conceptually new.

In the first line we use our product-image directive and we pass our product to the product input
of the ProductImageComponent. We use the product-department directive in the same way.

We use the price-display directive slightly differently in that we pass the product.price, instead
of the product directly.

---

[42]http://semantic-ui.com/views/item.html

The rest of the template is standard HTML elements with custom CSS classes and some template bindings.

Now let's talk about the three components we used in this template. They're relatively short.

The `ProductImageComponent` Component

In the `ProductImageComponent` the template is only one line, so we can put it inline:

code/how-angular-works/inventory-app/src/app/product-image/product-image.component.ts

```
 8   /**
 9    * @ProductImage: A component to show a single Product's image
10    */
11   @Component({
12     selector: 'product-image',
13     template: `
14     <img class="product-image" [src]="product.imageUrl">
15     `
16   })
17   export class ProductImageComponent {
18     @Input() product: Product;
19     @HostBinding('attr.class') cssClass = 'ui small image';
20   }
```

The one thing to note here is in the `img` tag, notice how we use the `[src]` input to `img`.

By using the `[src]` attribute, we're telling Angular that we want to use the `[src]` *input* on this `img` tag. Angular will then replace the value of the `src` attribute once the expression is resolved.

We could also have written this tag this way:

```
 1   <img src="{{ product.imageUrl }}">
```

Both styles do essentially the same thing, so feel free to pick the style that works best for your team.

The `PriceDisplayComponent` Component

Next, let's look at `PriceDisplayComponent`:

code/how-angular-works/inventory-app/src/app/price-display/price-display.component.ts

```
 1  import {
 2    Component,
 3    Input
 4  } from '@angular/core';
 5
 6  /**
 7   * @PriceDisplay: A component to show the price of a
 8   * Product
 9   */
10  @Component({
11    selector: 'price-display',
12    template: `
13    <div class="price-display">\${{ price }}</div>
14    `
15  })
16  export class PriceDisplayComponent {
17    @Input() price: number;
18  }
```

One thing to note is that we're escaping the dollar sign $ because this is a backtick string and the dollar sign is used for template variables (in ES6).

The `ProductDepartmentComponent`

Here is our ProductDepartmentComponent:

code/how-angular-works/inventory-app/src/app/product-department/product-department.component.ts

```
 1  import {
 2    Component,
 3    Input
 4  } from '@angular/core';
 5  import { Product } from '../product.model';
 6
 7  /**
 8   * @ProductDepartment: A component to show the breadcrumbs to a
 9   * Product's department
10   */
11  @Component({
12    selector: 'product-department',
```

```
13    templateUrl: './product-department.component.html'
14  })
15  export class ProductDepartmentComponent {
16    @Input() product: Product;
17  }
```

and template:

code/how-angular-works/inventory-app/src/app/product-department/product-department.component.html

```
1  <div class="product-department">
2    <span *ngFor="let name of product.department; let i=index">
3      <a href="#">{{ name }}</a>
4      <span>{{i < (product.department.length-1) ? '>' : ''}}</span>
5    </span>
6  </div>
```

The thing to note about the `ProductDepartmentComponent` Component is the `ngFor` and the `span` tag.

Our `ngFor` loops over `product.department` and assigns each department string to `name`. The new part is the second expression that says: `let i=index`. This is how you get the iteration number out of `ngFor`.

In the `span` tag, we use the `i` variable to determine if we should show the greater-than › symbol.

The idea is that given a department, we want to show the department string like:

```
1  Women > Apparel > Jackets & Vests
```

The expression `{{i < (product.department.length-1) ? '>' : ''}}` says that we only want to use the `'>'` character if we're not the last department. On the last department just show an empty string `''`.

> This format: `test ? valueIfTrue : valueIfFalse` is called the *ternary operator*.

`NgModule` **and Booting the App**

The final thing we have to do is ensure we have a `NgModule` for this app and boot it up:

code/how-angular-works/inventory-app/src/app/app.module.ts

```
1  import { BrowserModule } from '@angular/platform-browser';
2  import { NgModule } from '@angular/core';
3  import { FormsModule } from '@angular/forms';
4  import { HttpModule } from '@angular/http';
5
6  import { AppComponent } from './app.component';
7  import { ProductImageComponent } from './product-image/product-image.component';
8  import { ProductDepartmentComponent } from './product-department/product-departm\
9  ent.component';
10 import { PriceDisplayComponent } from './price-display/price-display.component';
11 import { ProductRowComponent } from './product-row/product-row.component';
12 import { ProductsListComponent } from './products-list/products-list.component';
13
14 @NgModule({
15   declarations: [
16     AppComponent,
17     ProductImageComponent,
18     ProductDepartmentComponent,
19     PriceDisplayComponent,
20     ProductRowComponent,
21     ProductsListComponent
22   ],
23   imports: [
24     BrowserModule,
25     FormsModule,
26     HttpModule
27   ],
28   providers: [],
29   bootstrap: [AppComponent]
30 })
31 export class AppModule { }
```

Angular provides a *module* system that helps organize our code. Unlike AngularJS 1.x, where all directives are essentially globals, in Angular you must specify exactly *which components* you're going to be using in your app.

While it is a bit more configuration to do it this way, it's a lifesaver for larger apps.

When you create new components in Angular, in order to use them they must be *accessible* from the current module. That is, if we want to use the ProductsListComponent component with the products-list selector in the AppComponent template, then we need to make sure that the AppComponent's module either:

1. is in the same module as the `ProductsListComponent` component or
2. The `AppComponent`'s module `imports` the module that contains `ProductsListComponent`

> Remember **every** component you write must be declared in one `NgModule` before it can be used in a template.

In this case, we're putting `AppComponent`, `ProductsListComponent`, and **all** the other components for this app in one module. This is easy and it means they can all "see" each other.

Notice that we tell `NgModule` that we want to `bootstrap` with `AppComponent`. This says that `AppComponent` will be the top-level component.

Because we are writing a browser app, we also put `BrowserModule` in the `imports` of the `NgModule`.

Booting the app

To bootstrap this app we write this in our `main.ts`:

code/how-angular-works/inventory-app/src/main.ts

```
1  import { enableProdMode } from '@angular/core';
2  import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
3
4  import { AppModule } from './app/app.module';
5  import { environment } from './environments/environment';
6
7  if (environment.production) {
8    enableProdMode();
9  }
10
11 platformBrowserDynamic().bootstrapModule(AppModule);
```

The last line in this file is what boots our `AppModule` and subsequently boots our Angular app.

Because this app was written with `angular-cli`, we can use the `ng` tool to run the app by running `ng serve`.

That said, it can be tricky to understand what's going on there. When we run our app with `ng serve` this is what happens:

- `ng serve` looks at `.angular-cli.json` which specifies `main.ts` as our entry point (and `index.html` as our index file)
- `main.ts` bootstraps `AppModule`
- `AppModule` specifies that `AppComponent` is the top level component
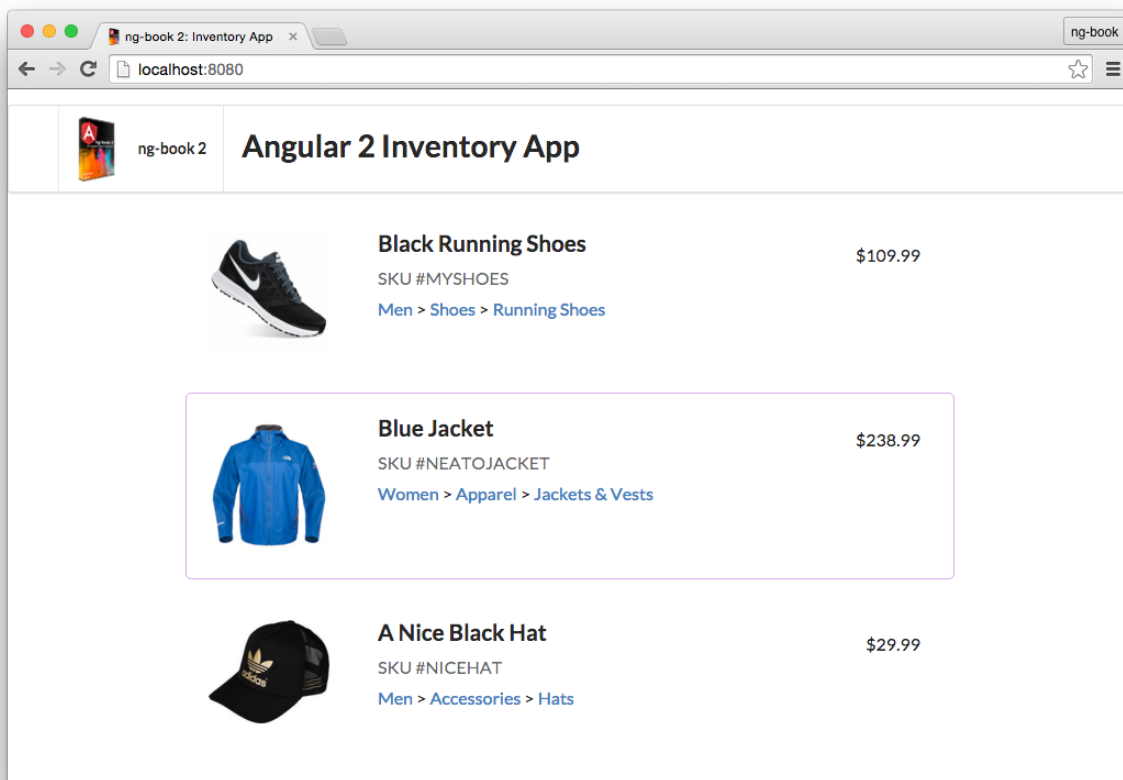- ... and then `AppComponent` renders the rest of our app!

The Completed Project

To try it out, change into the project directory and type:

```
1  npm install
2  ng serve
```

Now we have all the pieces we need for the working project!

Here's what it will look like when we're done:

Completed Inventory App

Now you can click to select a particular product and have it render a nice purple outline when selected. If you add new `Products` in your code, you'll see them rendered.

Deploying the App

We can deploy this app in the same way we deployed the app in the first chapter:

```
1  ng build --target=production --base-href '/'
```

And then push the files in `dist` to our server!

A Word on Data Architecture

You might be wondering at this point how we would manage the data flow if we started adding more functionality to this app.

For instance, say we wanted to add a shopping cart view and then we would add items to our cart. How could we implement this?

The only tools we've talked about are emitting output events. When we click add-to-cart do we simply bubble up an `addedToCart` event and handle at the root component? That feels a bit awkward.

Data architecture is a large topic with many opinions. Thankfully, Angular is flexible enough to handle a wide variety of data architectures, but that means that you have to decide for yourself which to use.

In Angular 1, the default option was two-way data binding. Two-way data binding is super easy to get started: your controllers have data, your forms manipulate that data directly, and your views show the data.

The problem with two-way data binding is that it often causes cascading effects throughout your application and makes it really difficult to trace data flow as your project grows.

Another problem with two-way data binding is that because you're passing data down through components it often forces your "data layout tree" to match your "dom view tree". In practice, these two things should really be separate.

One way you might handle this scenario would be to create a `ShoppingCartService`, which would be a singleton that would hold the list of the current items in the cart. This service could notify any interested objects when an item in the cart changes.

The idea is easy enough, but in practice there are a lot of details to be worked out.

The recommended way in Angular, and in many modern web frameworks (such as React), is to adopt a pattern of **one-way data binding**. That is, your data flows only **down** through components. If you need to make changes, you emit events that cause changes to happen "at the top" which then trickle down.

One-way data binding can seem like it adds some overhead in the beginning but it saves *a lot* of complication around change detection and it makes your systems easier to reason about.

Thankfully there are two major contenders for managing your data architecture:

1. Use an Observables-based architecture like RxJS
2. Use a Flux-based architecture

Later in this book we'll talk about how to implement a scalable data architecture for your app. For now, bask in the joy of your new Component-based application!