

# 2023.08.17.#

# Getting started with Angular

Welcome to Angular!

This tutorial introduces you to the essentials of Angular by walking you through building an e-commerce site with a catalog, shopping cart, and check-out form.

To help you get started right away, this tutorial uses a ready-made application that you can examine and modify interactively on [StackBlitz](#) —without having to set up a local work environment. StackBlitz is a browser-based development environment where you can create, save, and share projects using a variety of technologies.

/n n!

## # Prerequisites

To get the most out of this tutorial, you should already have a basic understanding of the following.

- [HTML](#) ✓
- [JavaScript](#) ✓
- [TypeScript](#) ✓

## # Take a tour of the example application

You build Angular applications with components. Components define areas of responsibility in the UI that let you reuse sets of UI functionality.

A component consists of three things:

- areas de responsabilidad en la UI  
- (funcionalidad) reusable

COMPONENT PART	DETAILS
A <u>component class</u>	Handles <u>data</u> and <u>functionality</u>
An <u>HTML template</u>	Determines the <u>UI</u>
Component-specific <u>styles</u> <i>CSS</i>	Define the <u>look and feel</u>

This guide demonstrates building an application with the following components:

## COMPONENTS

## DETAILS

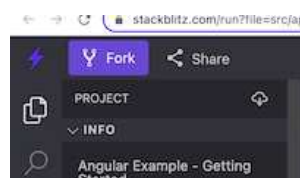
`<app-root>`The first component to load and the container for the other components`<app-top-bar>`The store name and checkout buttonLogo ↔ Checkout`<app-product-list>`The product list`<app-product-alerts>`A component that contains the application's alerts

For more information about components, see [Introduction to Components](#).

## # Create the sample project

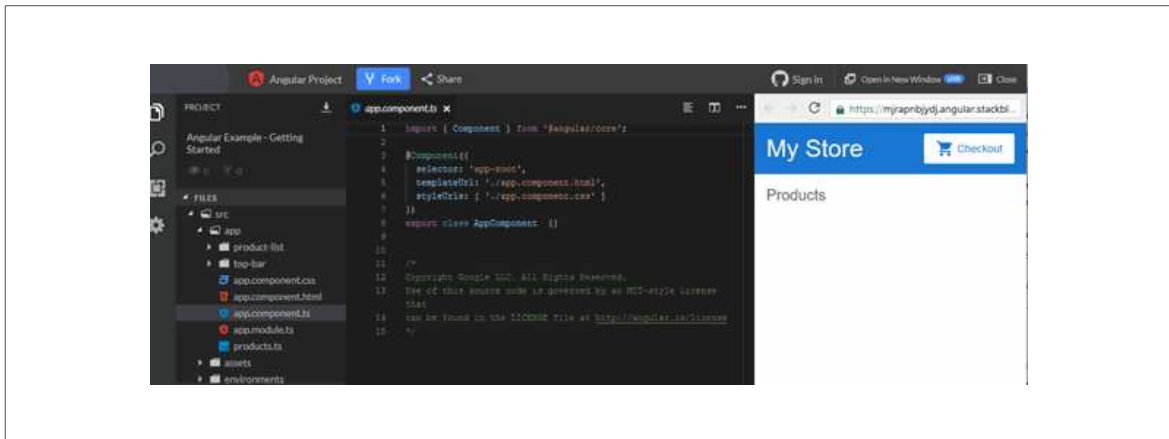
To create the sample project, generate the [ready-made sample project](#) in StackBlitz. To save your work:

1. Log into StackBlitz.
2. Fork the project you generated.
3. Save periodically.



In StackBlitz, the preview pane on the right shows the starting state of the example application. The preview features two areas:

- A top bar with the store name, `My Store`, and a checkout button
- A header for a product list, `Products`



The project section on the left shows the source files that make up the application, including the infrastructure and configuration files.

When you generate the StackBlitz example applications that accompany the tutorials, StackBlitz creates the starter files and mock data for you. The files you use throughout the tutorial are in the `src` folder.

For more information on how to use StackBlitz, see the [StackBlitz documentation](#).

## # Create the product list

In this section, you'll update the application to display a list of products. You'll use predefined product data from the `products.ts` file and methods from the `product-list.component.ts` file. This section guides you through editing the HTML, also known as the template.

1. In the `product-list` folder, open the template file `product-list.component.html`.
2. Add an `*ngFor` structural directive on a `<div>`, as follows.

src/app/product-list/product-list.component.html

```
<h2>Products</h2>
```

```
<div *ngFor="let product of products">
</div>
```

*<div \*ngFor="let product of products">  
</div>*

With `*ngFor`, the `<div>` repeats for each product in the list.

Structural directives shape or reshape the DOM's structure, by adding, removing, and manipulating elements. For more information about structural directives, see [Structural directives](#).

3. Inside the `<div>`, add an `<h3>` and `{{ product.name }}`. The `{{ product.name }}` statement is an example of Angular's [interpolation syntax](#). Interpolation `{{ }}` lets you render the property value as text.

src/app/product-list/product-list.component.html

```
<h2>Products</h2>
```

```
<div *ngFor="let product of products">
```

```
<h3>
```

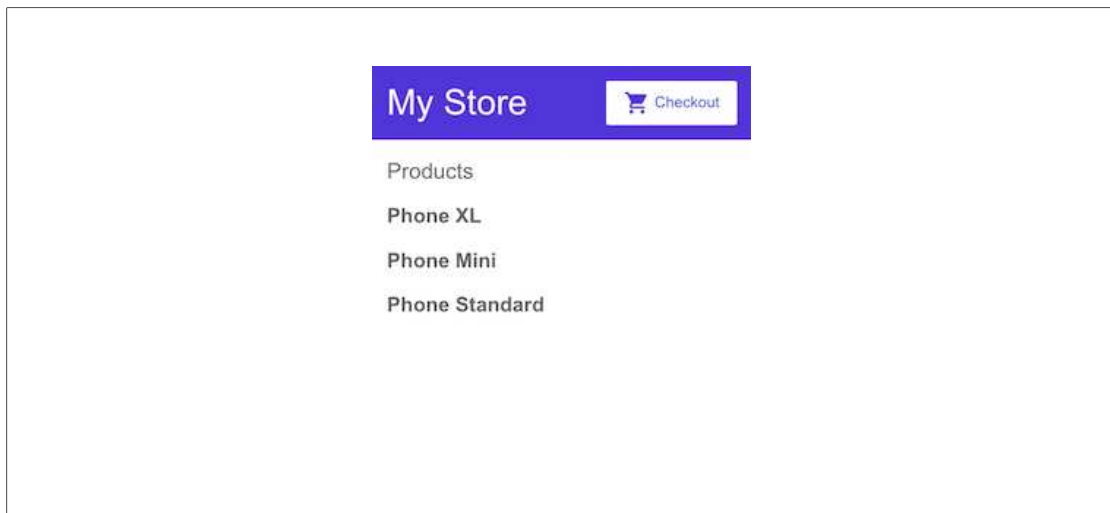
```
  {{ product.name }}
```

```
</h3>
```

```
</div>
```

*→ interpolation*

The preview pane updates to display the name of each product in the list.



4. To make each product name a link to product details, add the `<a>` element around `{{ product.name }}`.
5. Set the title to be the product's name by using the [property binding](#) `[ ]` syntax, as follows:

*(property binding "[ ]")*

src/app/product-list/product-list.component.html

```
<h2>Products</h2>

<div *ngFor="let product of products">

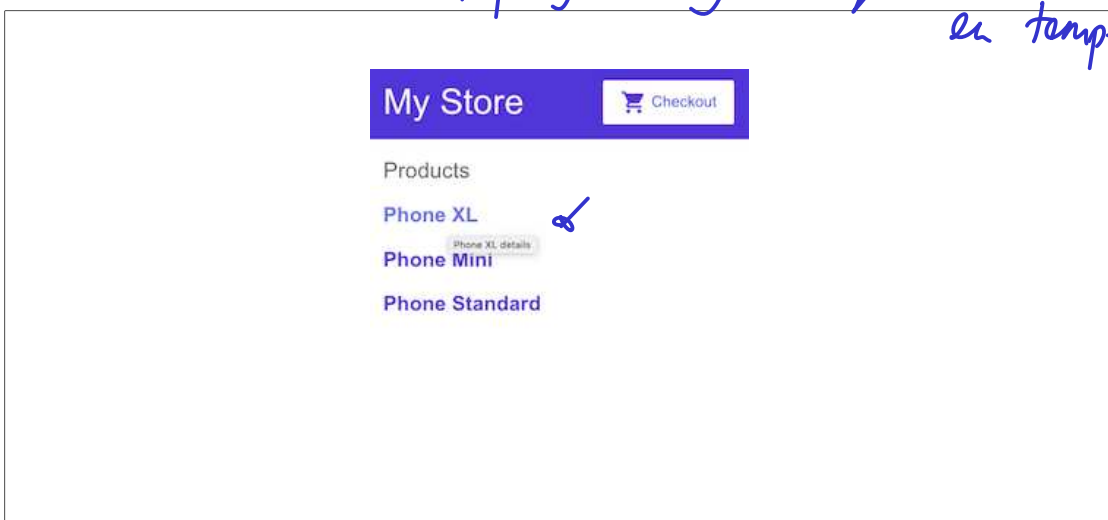
  <h3>
    <a [title]="product.name + ' details'">
      {{ product.name }}
    </a>
  </h3>

</div>
```

(`<a [title]="product.name + ' details'">`)

In the preview pane, hover over a product name to see the bound name property value, which is the product name plus the word "details". Property binding `[ ]` lets you use the property value in a template expression.

Property binding : est une valeur de propriété en template.



6. Add the product descriptions. On a `<p>` element, use an `*ngIf` directive so that Angular only creates the `<p>` element if the current product has a description.

src/app/product-list/product-list.component.html

```
<h2>Products</h2>

<div *ngFor="let product of products">

  <h3>
    <a [title]="product.name + ' details'">
      {{ product.name }}
    </a>
  </h3>

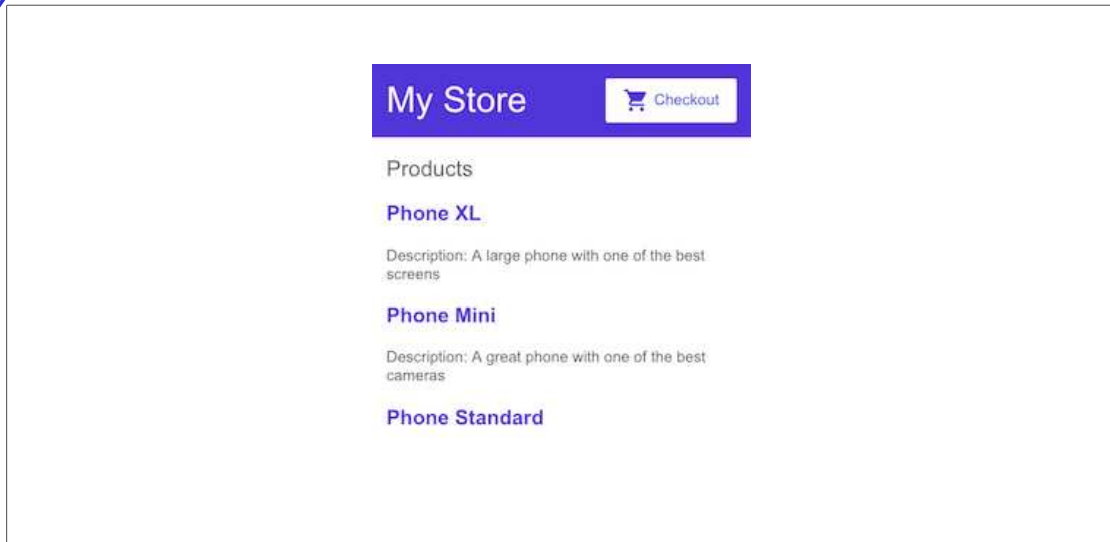
  <p *ngIf="product.description">
    Description: {{ product.description }}
  </p>

</div>
```

`<p *ngIf="product.description">`  
Desc.: {{ ... }}

The application now displays the name and description of each product in the list. Note that the final product does not have a description paragraph. Angular doesn't create the `<p>` element because the

product's description property is empty.



7. Add a button so users can share a product. Bind the button's `click` event to the `share()` method in `product-list.component.ts`. Event binding uses a set of parentheses, `()`, around the event, as in the `(click)` event on the `<button>` element.

src/app/product-list/product-list.component.html

```
<h2>Products</h2>

<div *ngFor="let product of products">

  <h3>
    <a [title]="product.name + ' details'">
      {{ product.name }}
    </a>
  </h3>

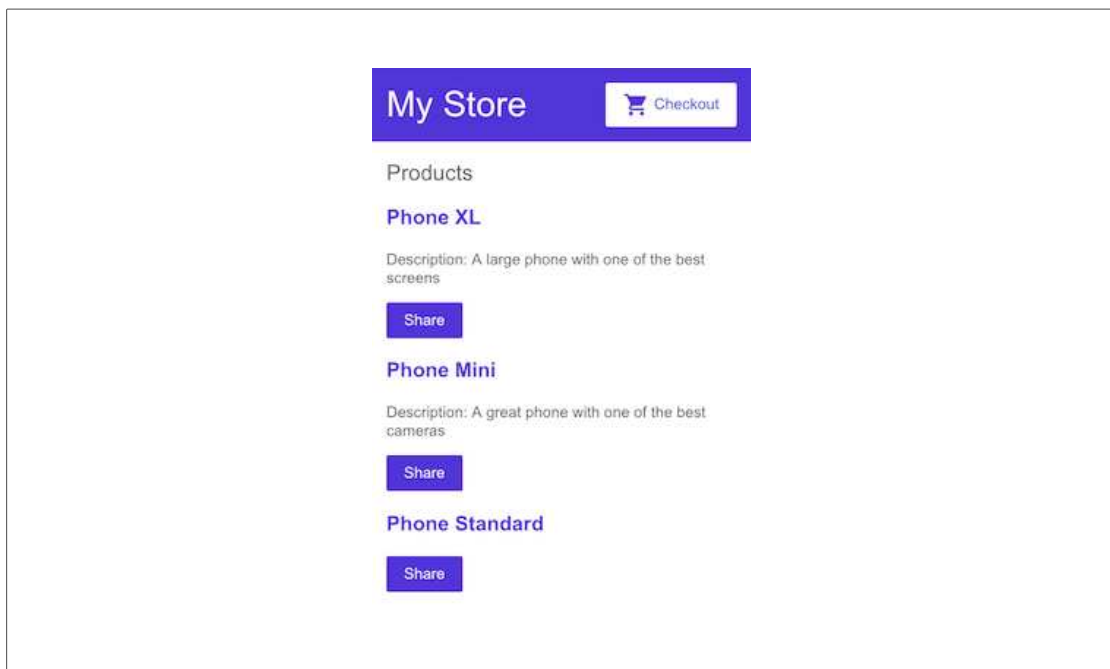
  <p *ngIf="product.description">
    Description: {{ product.description }}
  </p>

  <button type="button" (click)="share()">
    Share
  </button>

</div>
```

`<button type="button" (click)="share()">`  
Share  
`</button>`

Each product now has a **Share** button.



Clicking the **Share** button triggers an alert that states, "The product has been shared!".



In editing the template, you have explored some of the most popular features of Angular templates. For more information, see [Introduction to components and templates](#).

## # Pass data to a child component

Currently, the product list displays the name and description of each product. The `ProductListComponent` also defines a `products` property that contains imported data for each product from the `products` array in `products.ts`.

The next step is to create a new alert feature that will use product data from the `ProductListComponent`. The alert should check the product's price. If it is found to be greater than \$700, a Notify Me button should be displayed. When a user clicks on the button, they should be able to sign up for notifications that will inform them when the product goes on sale.

This section walks you through creating a child component, `ProductAlertsComponent`, that can receive data from its parent component, `ProductListComponent`.

(child: pq es ui que aparece dentro de la ui del padre)

1. Click on the plus sign above the current terminal to create a new terminal to run the command to generate the component.



2. In the new terminal, generate a new component named `product-alerts` by running the following command:

```
ng generate component product-alerts
```

*ng g component product-alerts*

The generator creates starter files for the three parts of the component:

- `product-alerts.component.ts`
- `product-alerts.component.html`
- `product-alerts.component.css`

3. Open `product-alerts.component.ts`. The `@Component()` decorator indicates that the following class is a component. `@Component()` also provides metadata about the component, including its selector, templates, and styles.

*metadata: + selector  
+ template  
+ style*

```
src/app/product-alerts/product-alerts.component.ts

@Component({
  selector: 'app-product-alerts',
  templateUrl: './product-alerts.component.html',
  styleUrls: ['./product-alerts.component.css']
})
export class ProductAlertsComponent {

}
```

Key features in the `@Component()` are as follows:

- The `selector`, `app-product-alerts`, identifies the component. By convention, Angular component selectors begin with the prefix `app-`, followed by the component name. *→ app-...*
- The `template` and `style` filenames reference the component's HTML and CSS
- The `@Component()` definition also exports the class, `ProductAlertsComponent`, which handles functionality for the component

4. To set up `ProductAlertsComponent` to receive product data, first import `Input` from `@angular/core`.

*as Input*



src/app/product-alerts/product-alerts.component.ts

```
import { Component, Input } from '@angular/core';
import { Product } from '../products';
```

5. In the `ProductAlertsComponent` class definition, define a property named `product` with an `@Input()` decorator. The `@Input()` decorator indicates that the property value passes in from the component's parent, `ProductListComponent`.

src/app/product-alerts/product-alerts.component.ts

```
export class ProductAlertsComponent {

  @Input() product: Product | undefined;

}
```

*@Input() product: Product | undef.;*

6. Open `product-alerts.component.html` and replace the placeholder paragraph with a **Notify Me** button that appears if the product price is over \$700.

src/app/product-alerts/product-alerts.component.html

```
<p *ngIf="product && product.price > 700">
  <button type="button">Notify Me</button>
</p>
```

*<p \*ngIf="product && ... > 700">  
 <button ....  
</p>*

7. The generator automatically added the `ProductAlertsComponent` to the `AppModule` to make it available to other components in the application.

src/app/app.module.ts

```
import { ProductAlertsComponent } from '../product-alerts/product-
alerts.component';

@NgModule({
  declarations: [
    AppComponent,
    TopBarComponent,
    ProductListComponent,
    ProductAlertsComponent,
  ],
```

8. Finally, to display `ProductAlertsComponent` as a child of `ProductListComponent`, add the `<app-product-alerts>` element to `product-list.component.html`. Pass the current product as input to the component using property binding.

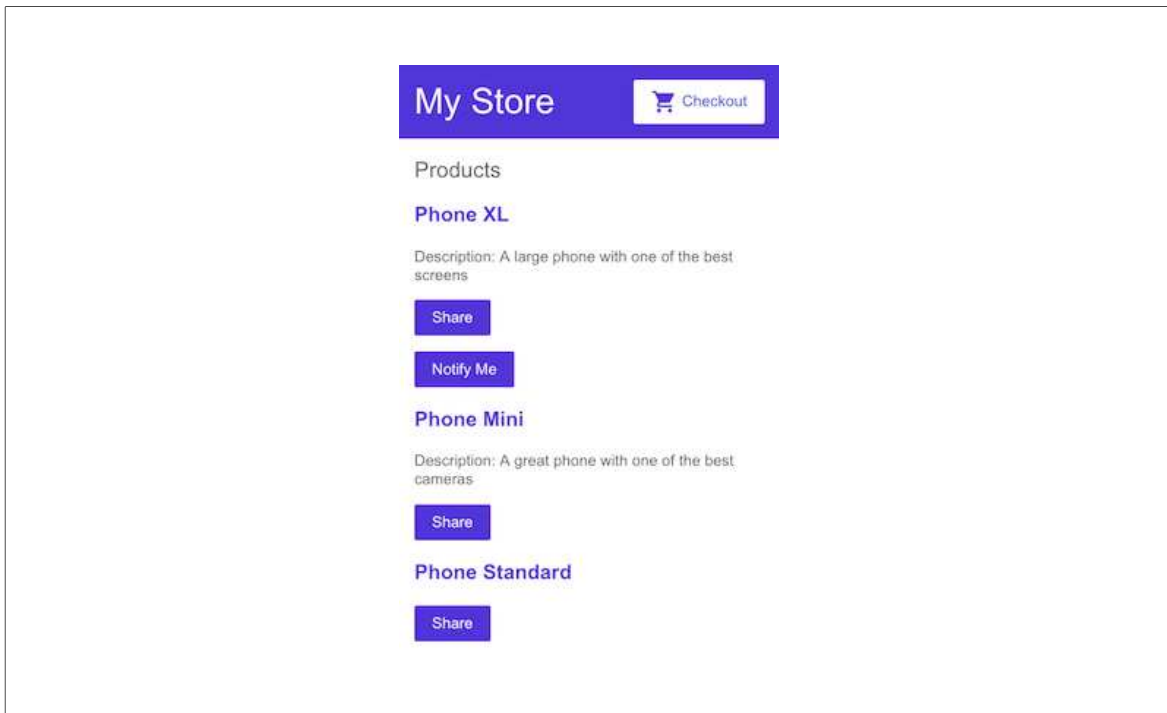
src/app/product-list/product-list.component.html

```
<button type="button" (click)="share()">
  Share
</button>

<app-product-alerts
  [product]="product">
</app-product-alerts>
```

*<app-product-alerts  
(product)="product"> </...>*

The new product alert component takes a product as input from the product list. With that input, it shows or hides the **Notify Me** button, based on the price of the product. The Phone XL price is over \$700, so the **Notify Me** button appears on that product.



## # Pass data to a parent component

To make the **Notify Me** button work, the child component needs to notify and pass the data to the parent component. The `ProductAlertsComponent` needs to emit an event when the user clicks **Notify Me** and the `ProductListComponent` needs to respond to the event.

1. In `product-alerts.component.ts`, import `Output` and `EventEmitter` from `@angular/core`.

src/app/product-alerts/product-alerts.component.ts

```
import { Component, Input, Output, EventEmitter } from '@angular/core';
import { Product } from '../products';
```

2. In the component class, define a property named `notify` with an `@Output()` decorator and an instance of `EventEmitter()`. Configuring `ProductAlertsComponent` with an `@Output()` allows the `ProductAlertsComponent` to emit an event when the value of the `notify` property changes.

src/app/product-alerts/product-alerts.component.ts

```
export class ProductAlertsComponent {
  @Input() product: Product | undefined;
  @Output() notify = new EventEmitter();
}
```

*@Output() notify = new EventEmitter();*

3. In `product-alerts.component.html`, update the **Notify Me** button with an event binding to call the `notify.emit()` method.

src/app/product-alerts/product-alerts.component.html

```
<p *ngIf="product && product.price > 700">
  <button type="button" (click)="notify.emit()">Notify Me</button>
</p>
```

4. Define the behavior that happens when the user clicks the button. The parent, `ProductListComponent` — not the `ProductAlertsComponent` — acts when the child raises the event. In `product-list.component.ts`, define an `onNotify()` method, similar to the `share()` method.

src/app/product-list/product-list.component.ts

```
export class ProductListComponent {

  products = [...products];

  share() {
    window.alert('The product has been shared!');
  }

  onNotify() {
    window.alert('You will be notified when the product goes on sale');
  }
}
```

5. Update the `ProductListComponent` to receive data from the `ProductAlertsComponent`. In `product-list.component.html`, bind `<app-product-alerts>` to the `onNotify()` method of the product list component. `<app-product-alerts>` is what displays the **Notify Me** button.

src/app/product-list/product-list.component.html

```
<button type="button" (click)="share()">
  Share
</button>
```

```
<app-product-alerts
  [product]="product"
  (notify)="onNotify()">
</app-product-alerts>
```

*(notify) = "onNotify()"*

6. Click the **Notify Me** button to trigger an alert which reads, "You will be notified when the product goes on sale".

An embedded page at angular-onlinestore.stackblitz.io  
says  
You will be notified when the product goes on sale

OK

For more information on communication between components, see [Component Interaction](#).

## What's next

In this section, you've created an application that iterates through data and features components that communicate with each other.

To continue exploring Angular and developing this application:

- Continue to [In-app navigation](#) to create a product details page.
- Skip ahead to [Deployment](#) to move to local development, or deploy your application to Firebase or your own server.

*Last reviewed on Mon Feb 28 2022*

# # Adding navigation



This guide builds on the first step of the Getting Started tutorial, [Get started with a basic Angular app](#).

At this stage of development, the online store application has [a basic product catalog](#).

In the following sections, you'll add the following features to the application:

- Type a [URL](#) in the [address bar](#) to navigate to a [corresponding product page](#)
  - Click [links on the page to navigate](#) within your [single-page application](#) *SPA*
  - Click the browser's back and forward buttons to navigate the browser history intuitively
- 

## # Associate a URL path with a component

The application already uses the Angular `Router` to navigate to the `ProductListComponent`. This section shows you how to [define a route](#) to show [individual product details](#).

1. Generate a new component for product details. In the terminal generate a new `product-details` component by running the following command:

```
ng generate component product-details
```

2. In `app.module.ts`, add a route for product details, with a `path` of `products/:productId` and `ProductDetailsComponent` for the `component`.

src/app/app.module.ts

```
@NgModule({
  imports: [
    BrowserModule,
    ReactiveFormsModule,
    RouterModule.forRoot([
      { path: '', component: ProductListComponent },
      { path: 'products/:productId', component: ProductDetailsComponent },
    ])
  ],
  declarations: [
    AppComponent,
    TopBarComponent,
    ProductListComponent,
    ProductAlertsComponent,
    ProductDetailsComponent,
  ],
})
```

*RouterModule.forRoot([C  
{path: '', component: ... 3,  
})*

3. Open `product-list.component.html`.

4. Modify the product name anchor to include a `routerLink` with the `product.id` as a parameter.

src/app/product-list/product-list.component.html

```
<div *ngFor="let product of products">

  <h3>
    <a
      [title]="product.name + ' details'"
      [routerLink]="['/products', product.id]"
      {{ product.name }}
    </a>
  </h3>

  <!-- ... -->

</div>
```

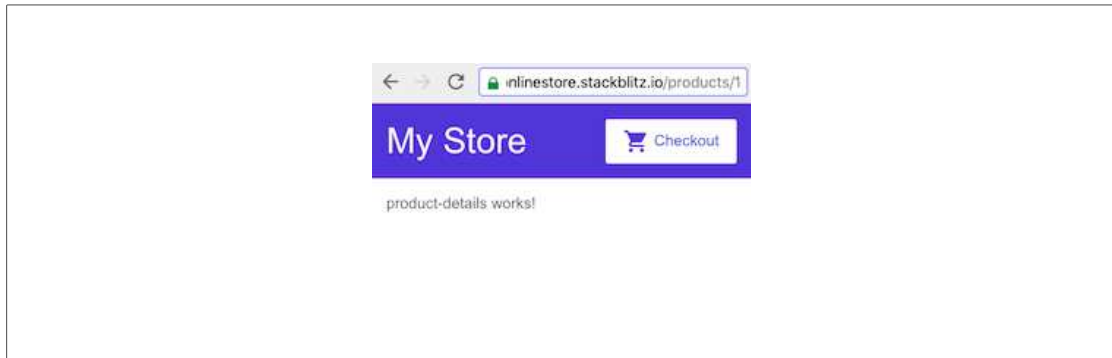
*[routerLink]="['/products', product.id]"*

The `RouterLink` directive helps you customize the anchor element. In this case, the route, or URL, contains one fixed segment, `/products`. The final segment is variable, inserting the `id` property of the current product. For example, the URL for a product with an `id` of 1 would be similar to

`https://getting-started-myfork.stackblitz.io/products/1`.

5. Verify that the router works as intended by clicking the product name. The application should display the `ProductDetailsComponent`, which currently says "product-details works!"

Notice that the URL in the preview window changes. The final segment is `products/#` where `#` is the number of the route you clicked.



---

## # View product details

- ✓ The `ProductDetailsComponent` handles the display of each product. The Angular Router displays components based on the browser's URL and your defined routes.
- ✓ In this section, you'll use the Angular Router to combine the `products` data and route information to display the specific details for each product.

1. In `product-details.component.ts`, import `ActivatedRoute` from `@angular/router`, and the `products` array from `../products`.

```
src/app/product-details/product-details.component.ts

import { Component, OnInit } from '@angular/core';
import { ActivatedRoute } from '@angular/router';

import { Product, products } from '../products';
```

2. Define the `product` property.

```
src/app/product-details/product-details.component.ts

export class ProductDetailsComponent implements OnInit {

  product: Product | undefined;
  /* ... */
}
```

3. Inject `ActivatedRoute` into the `constructor()` by adding `private route: ActivatedRoute` as an argument within the constructor's parentheses.

```
src/app/product-details/product-details.component.ts

export class ProductDetailsComponent implements OnInit {

  product: Product | undefined;

  constructor(private route: ActivatedRoute) { }

}
```

`ActivatedRoute` is specific to each component that the Angular Router loads. `ActivatedRoute` contains information about the route and the route's parameters.

By injecting `ActivatedRoute`, you are configuring the component to use a service. The Managing Data step covers services in more detail.

4. In the `ngOnInit()` method, extract the `productId` from the route parameters and find the corresponding product in the `products` array.

```
src/app/product-details/product-details.component.ts

ngOnInit() {
  // First get the product id from the current route.
  const routeParams = this.route.snapshot.paramMap;
  const productIdFromRoute = Number(routeParams.get('productId'));

  // Find the product that correspond with the id provided in route.
  this.product = products.find(product => product.id === productIdFromRoute);
}
```

*const routeParams = this.route.snapshot.paramMap;*

*busca en array productos que*

The route parameters correspond to the path variables you define in the route. To access the route parameters, we use `route.snapshot`, which is the `ActivatedRouteSnapshot` that contains information



about the active route at that particular moment in time. The URL that matches the route provides the `productId`. Angular uses the `productId` to display the details for each unique product.

5. Update the `ProductDetailsComponent` template to display product details with an `*ngIf`. If a product exists, the `<div>` renders with a name, price, and description.

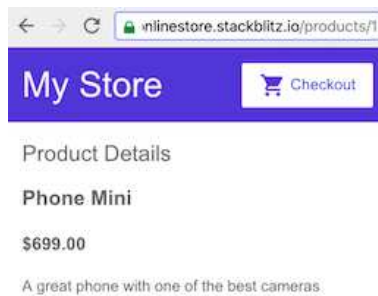
src/app/product-details/product-details.component.html

```
<h2>Product Details</h2>

<div *ngIf="product">
  <h3>{{ product.name }}</h3>
  <h4>{{ product.price | currency }}</h4>
  <p>{{ product.description }}</p>
</div>
```

The line, `<h4>{{ product.price | currency }}</h4>`, uses the `currency` pipe to transform `product.price` from a number to a currency string. A pipe is a way you can transform data in your HTML template. For more information about Angular pipes, see [Pipes](#).

When users click on a name in the product list, the router navigates them to the distinct URL for the product, shows the `ProductDetailsComponent`, and displays the product details.



For more information about the Angular Router, see [Routing & Navigation](#).

## What's next

You have configured your application so you can view product details, each with a distinct URL.

To continue exploring Angular:

- Continue to [Managing Data](#) to add a shopping cart feature, manage cart data, and retrieve external data for shipping prices
- Skip ahead to [Deployment](#) to deploy your application to Firebase or move to local development

*Last reviewed on Mon Feb 28 2022*

# # Managing data



This guide builds on the second step of the [Getting started with a basic Angular application tutorial](#), [Adding navigation](#). At this stage of development, the store application has [a product catalog](#) with two views: [a product list](#) and [product details](#). Users can click on a product name from the list to see details in a new view, with a distinct URL, or route.

This step of the tutorial guides you through [creating a shopping cart](#) in the following phases:

- Update the [product details view](#) to include a [Buy button](#), which [adds the current product to a list of products](#) that a [cart service](#) manages
- Add a [cart component](#), which [displays the items in the cart](#)
- Add a [shipping component](#), which retrieves shipping prices for the items in the cart by using Angular's `HttpClient` to retrieve shipping data from a `.json` file

---

## / Create the shopping cart service

In Angular, [a service is an instance of a class](#) that you can [make available to any](#) part of your application using Angular's [dependency injection system](#).

Currently, users can view product information, and the application can simulate sharing and notifications about product changes.

The next step is to build a [way for users to add products to a cart](#). This section walks you through adding a [Buy button](#) and setting up a [cart service to store information about products in the cart](#).

## / Define a cart service

This section walks you through creating the `CartService` that tracks products added to shopping cart.

1. In the terminal generate a new `cart` service by running the following command:

```
ng generate service cart
```

*ng g service cart*

2. Import the `Product` interface from `./products.ts` into the `cart.service.ts` file, and in the `CartService` class, define an `items` property to store the array of the current products in the cart.

src/app/cart.service.ts

```
import { Product } from './products';
import { Injectable } from '@angular/core';
/* . . . */
@Injectable({
  providedIn: 'root'
})
export class CartService {
  items: Product[] = [];
  /* . . . */
}
```

3. Define methods to add items to the cart, return cart items, and clear the cart items.

src/app/cart.service.ts

```
@Injectable({
  providedIn: 'root'
})
export class CartService {
  items: Product[] = [];
  /* . . . */

  addToCart(product: Product) {
    this.items.push(product);
  }

  getItems() {
    return this.items;
  }

  clearCart() {
    this.items = [];
    return this.items;
  }
  /* . . . */
}
```

*items: Product[] = [];*

*this.items.push(...)*

*get --*

*clear --*

- The `addToCart()` method appends a product to an array of `items`
- The `getItems()` method collects the items users add to the cart and returns each item with its associated quantity
- The `clearCart()` method returns an empty array of items, which empties the cart

## Use the cart service

This section walks you through using the `CartService` to add a product to the cart.

1. In `product-details.component.ts`, import the `cart service`.

```
src/app/product-details/product-details.component.ts

import { Component, OnInit } from '@angular/core';
import { ActivatedRoute } from '@angular/router';

import { Product, products } from '../products';
import { CartService } from '../cart.service';
```

2. Inject the cart service by adding it to the `constructor()`.

```
src/app/product-details/product-details.component.ts

export class ProductDetailsComponent implements OnInit {

  constructor(
    private route: ActivatedRoute,
    private cartService: CartService
  ) {}
}
```

3. Define the `addToCart()` method, which adds the current product to the cart.

```
src/app/product-details/product-details.component.ts

export class ProductDetailsComponent implements OnInit {

  addToCart(product: Product) {
    this.cartService.addToCart(product);
    window.alert('Your product has been added to the cart!');
  }
}
```

*this.cartService.addToCart(product)*  
*window.alert('...')*

The `addToCart()` method does the following:

- Takes the current `product` as an argument
- Uses the `CartService` `addToCart()` method to add the product to the cart
- Displays a message that you've added a product to the cart

4. In `product-details.component.html`, add a button with the label Buy, and bind the `click()` event to the `addToCart()` method. This code updates the product details template with a Buy button that adds the current product to the cart.

```
src/app/product-details/product-details.component.html

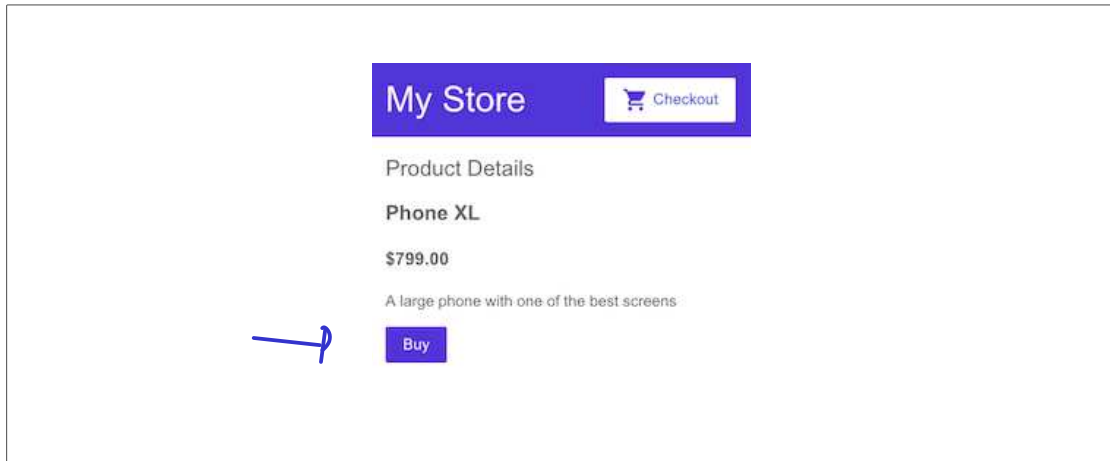
<h2>Product Details</h2>

<div *ngIf="product">
  <h3>{{ product.name }}</h3>
  <h4>{{ product.price | currency }}</h4>
  <p>{{ product.description }}</p>
  <button type="button" (click)="addToCart(product)">Buy</button>
</div>
```

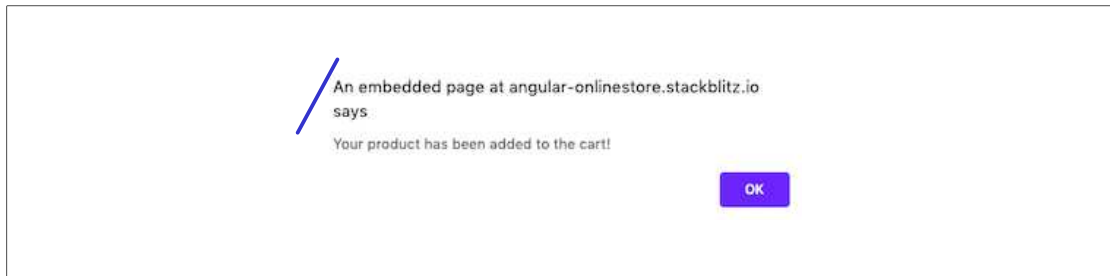
*<button type="button" (click)="addToCart(product)">*

5. Verify that the new **Buy** button appears as expected by refreshing the application and clicking on a product's name to display its details.

Buy </button>)



6. Click the **Buy** button to add the product to the stored list of items in the cart and display a confirmation message.



## Create the cart view

For customers to see their cart, you can create the cart view in two steps:

1. Create a cart component and configure routing to the new component.
2. Display the cart items.

## Set up the cart component

To create the cart view, follow the same steps you did to create the `ProductDetailsComponent` and configure routing for the new component.

1. Generate a new component named `cart` in the terminal by running the following command:

```
ng generate component cart
```

This command will generate the `cart.component.ts` file and its associated template and styles files.

```
src/app/cart/cart.component.ts

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

@Component({
  selector: 'app-cart',
  templateUrl: './cart.component.html',
  styleUrls: ['./cart.component.css']
})
export class CartComponent {

}
```

2. Notice that the newly created `CartComponent` is added to the module's `declarations` in `app.module.ts`.

```
src/app/app.module.ts

import { CartComponent } from './cart/cart.component';

@NgModule({
  declarations: [
    AppComponent,
    TopBarComponent,
    ProductListComponent,
    ProductAlertsComponent,
    ProductDetailsComponent,
    CartComponent,
  ],
```

3. Still in `app.module.ts`, add a `route` for the component `CartComponent`, with a `path` of `cart`.

```
src/app/app.module.ts

@NgModule({
  imports: [
    BrowserModule,
    ReactiveFormsModule,
    RouterModule.forRoot([
      { path: '', component: ProductListComponent },
      { path: 'products/:productId', component: ProductDetailsComponent },
      { path: 'cart', component: CartComponent },
    ])
  ],
```

4. Update the `Checkout` button so that it routes to the `/cart` URL. In `top-bar.component.html`, add a `routerLink` directive pointing to `/cart`.

src/app/top-bar/top-bar.component.html

```
<a routerLink="/cart" class="button fancy-button">
  <em class="material-icons">shopping_cart</em>Checkout
</a>
```

*<a routerLink="/cart" class='button fancy-button'>*

5. Verify the new `CartComponent` works as expected by clicking the **Checkout** button. You can see the "cart works!" default text, and the URL has the pattern `https://getting-started.stackblitz.io/cart`, where `getting-started.stackblitz.io` may be different for your StackBlitz project.



## Display the cart items

This section shows you how to use the cart service to display the products in the cart.

1. In `cart.component.ts`, import the `CartService` from the `cart.service.ts` file.

src/app/cart/cart.component.ts

```
import { Component } from '@angular/core';
import { CartService } from '../cart.service';
```

2. Inject the `CartService` so that the `CartComponent` can use it by adding it to the `constructor()`.

src/app/cart/cart.component.ts

```
export class CartComponent {

  constructor(
    private cartService: CartService
  ) { }
}
```

3. Define the `items` property to store the products in the cart.

src/app/cart/cart.component.ts

```
export class CartComponent {

  items = this.cartService.getItems();

  constructor(
    private cartService: CartService
  ) { }
}
```

This code sets the items using the `CartService` `getItems()` method. You defined this method when you created `cart.service.ts`.

4. Update the cart template with a header, and use a `<div>` with an `*ngFor` to display each of the cart items with its name and price. The resulting `CartComponent` template is as follows.

src/app/cart/cart.component.html

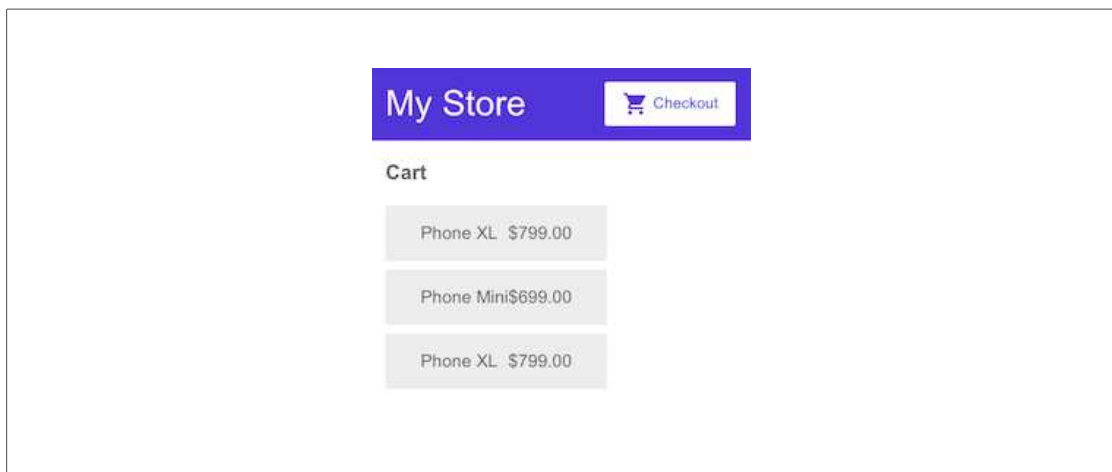
```
<h3>Cart</h3>

<div class="cart-item" *ngFor="let item of items">
  <span>{{ item.name }}</span>
  <span>{{ item.price | currency }}</span>
</div>
```

5. Verify that your cart works as expected:

- Click **My Store**.
- Click on a product name to display its details.
- Click **Buy** to add the product to the cart.
- Click **Checkout** to see the cart.





For more information about services, see [Introduction to Services and Dependency Injection](#).

## Retrieve shipping prices

Servers often return data in the form of a stream. Streams are useful because they make it easy to transform the returned data and make modifications to the way you request that data. Angular `HttpClient` is a built-in way to fetch data from external APIs and provide them to your application as a stream.

This section shows you how to use `HttpClient` to retrieve shipping prices from an external file.

The application that StackBlitz generates for this guide comes with predefined shipping data in `assets/shipping.json`. Use this data to add shipping prices for items in the cart.

`src/assets/shipping.json`

```
[
  {
    "type": "Overnight",
    "price": 25.99
  },
  {
    "type": "2-Day",
    "price": 9.99
  },
  {
    "type": "Postal",
    "price": 2.99
  }
]
```

```
[
  {
    "type": "Overnight",
    "price": 25.99
  },
]
```

## Configure AppModule to use HttpClient

To use Angular's `HttpClient`, you must configure your application to use `HttpClientModule`.

Angular's `HttpClientModule` registers the providers your application needs to use the `HttpClient` service throughout your application.

1. In `app.module.ts`, import `HttpClientModule` from the `@angular/common/http` package at the top of the file with the other imports. As there are a number of other imports, this code snippet omits them for brevity. Be sure to leave the existing imports in place.

src/app/app.module.ts

```
import { HttpClientModule } from '@angular/common/http';
```

2. To register Angular's `HttpClient` providers globally, add `HttpClientModule` to the `AppModule` `@NgModule()` `imports` array.

src/app/app.module.ts

```
@NgModule({
  imports: [
    BrowserModule,
    HttpClientModule,
    ReactiveFormsModule,
    RouterModule.forRoot([
      { path: '', component: ProductListComponent },
      { path: 'products/:productId', component: ProductDetailsComponent },
      { path: 'cart', component: CartComponent },
    ])
  ],
  declarations: [
    AppComponent,
    TopBarComponent,
    ProductListComponent,
    ProductAlertsComponent,
    ProductDetailsComponent,
    CartComponent,
  ],
  bootstrap: [
    AppComponent
  ]
})
export class AppModule { }
```

## Configure CartService to use HttpClient

The next step is to inject the `HttpClient` service into your service so your application can fetch data and interact with external APIs and resources.

1. In `cart.service.ts`, import `HttpClient` from the `@angular/common/http` package.

src/app/cart.service.ts

```
import { HttpClient } from '@angular/common/http';
import { Product } from './products';
import { Injectable } from '@angular/core';
```

2. Inject `HttpClient` into the `CartService` `constructor()`.

src/app/cart.service.ts

```
@Injectable({
  providedIn: 'root'
})
export class CartService {
  items: Product[] = [];

  constructor(
    private http: HttpClient
  ) {}
  /* . . . */
}
```

## Configure `CartService` to get shipping prices

To get shipping data, from `shipping.json`, You can use the `HttpClient` `get()` method.

1. In `cart.service.ts`, below the `clearCart()` method, define a new `getShippingPrices()` method that uses the `HttpClient` `get()` method.

src/app/cart.service.ts

```
@Injectable({
  providedIn: 'root'
})
export class CartService {
  /* . . . */
  getShippingPrices() {
    return this.http.get<{type: string, price: number}[]>('/assets
/shipping.json');
  }
}
```

*this.http.get<{type:string, price: number}[]>('...')*

For more information about Angular's `HttpClient`, see the [Client-Server Interaction](#) guide.

## Create a shipping component

Now that you've configured your application to retrieve shipping data, you can create a place to render that data.

1. Generate a cart component named `shipping` in the terminal by running the following command:

```
ng generate component shipping
```

This command will generate the `shipping.component.ts` file and its associated template and styles files.

```
src/app/shipping/shipping.component.ts

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

@Component({
  selector: 'app-shipping',
  templateUrl: './shipping.component.html',
  styleUrls: ['./shipping.component.css']
})
export class ShippingComponent {

}
```

2. In `app.module.ts`, add a route for shipping. Specify a `path` of `shipping` and a component of `ShippingComponent`.

```
src/app/app.module.ts

@NgModule({
  imports: [
    BrowserModule,
    HttpClientModule,
    ReactiveFormsModule,
    RouterModule.forRoot([
      { path: '', component: ProductListComponent },
      { path: 'products/:productId', component: ProductDetailsComponent },
      { path: 'cart', component: CartComponent },
      { path: 'shipping', component: ShippingComponent },
    ])
  ],
  declarations: [
    AppComponent,
    TopBarComponent,
    ProductListComponent,
    ProductAlertsComponent,
    ProductDetailsComponent,
    CartComponent,
    ShippingComponent
  ],
  bootstrap: [
    AppComponent
  ]
})
export class AppModule { }
```

There's no link to the new shipping component yet, but you can see its template in the preview pane by entering the URL its route specifies. The URL has the pattern: `https://angular-ynqttp--4200.local.webcontainer.io/shipping`, where the `angular-ynqttp--4200.local.webcontainer.io` part may be different for your StackBlitz project.

## Configuring the ShippingComponent to use CartService

This section guides you through modifying the `ShippingComponent` to retrieve shipping data via HTTP from the `shipping.json` file.

1. In `shipping.component.ts`, import `CartService`.

src/app/shipping/shipping.component.ts

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

import { Observable } from 'rxjs';
import { CartService } from '../cart.service';
```

2. Inject the cart service in the `ShippingComponent` `constructor()`.

src/app/shipping/shipping.component.ts

```
constructor(private cartService: CartService) { }
```

3. Define a `shippingCosts` property that sets the `shippingCosts` property using the `getShippingPrices()` method from the `CartService`. Initialize the `shippingCosts` property inside `ngOnInit()` method.

src/app/shipping/shipping.component.ts

```
export class ShippingComponent implements OnInit {

  shippingCosts!: Observable<{ type: string, price: number }[]>;

  ngOnInit(): void {
    this.shippingCosts = this.cartService.getShippingPrices();
  }

}
```

↖ Type price

4. Update the `ShippingComponent` template to display the shipping types and prices using the `async` pipe.

src/app/shipping/shipping.component.html

```
<h3>Shipping Prices</h3>

<div class="shipping-item" *ngFor="let shipping of shippingCosts | async">
  <span>{{ shipping.type }}</span>
  <span>{{ shipping.price | currency }}</span>
</div>
```

The `async` pipe returns the latest value from a stream of data and continues to do so for the life of a given component. When Angular destroys that component, the `async` pipe automatically stops. For detailed information about the `async` pipe, see the [AsyncPipe API documentation](#).

5. Add a link from the `CartComponent` view to the `ShippingComponent` view.

Idea:  
| valor de acción ...  
| username online

src/app/cart/cart.component.html

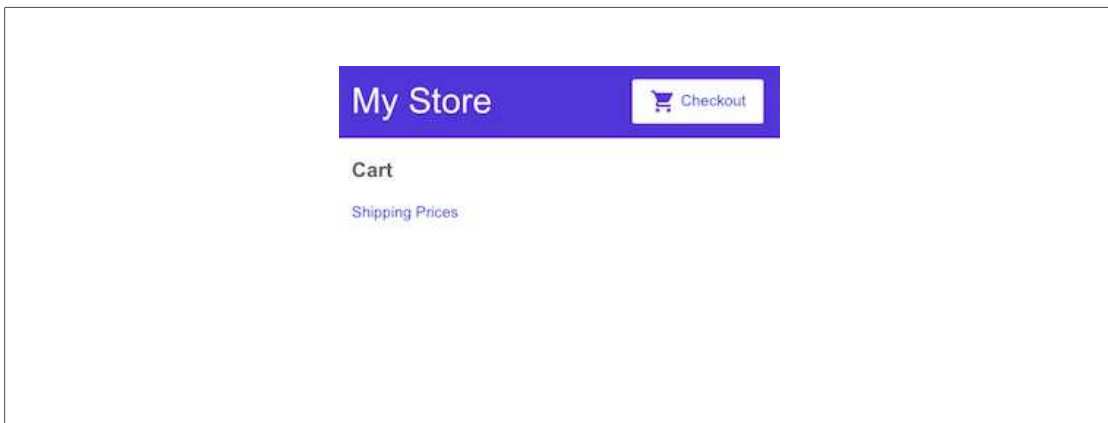
```
<h3>Cart</h3>

<p>
  <a routerLink="/shipping">Shipping Prices</a>
</p>

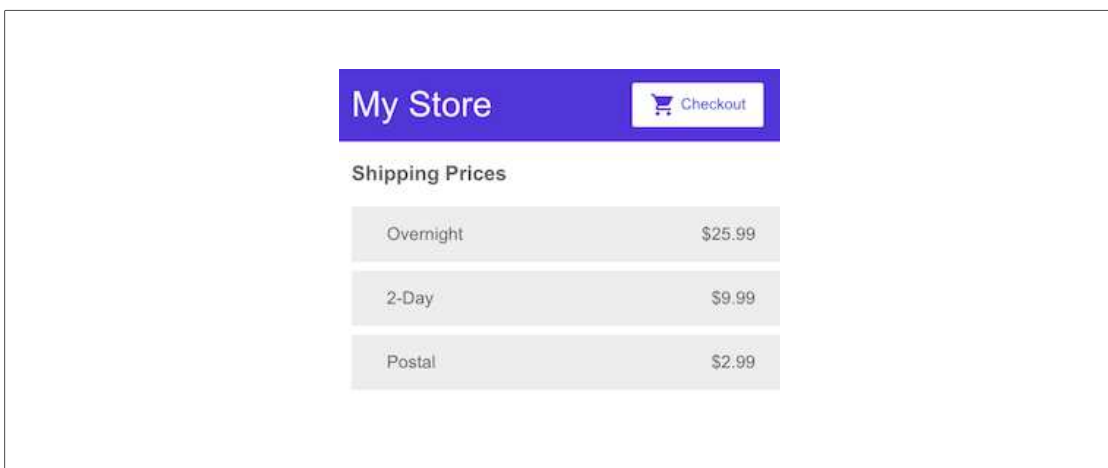
<div class="cart-item" *ngFor="let item of items">
  <span>{{ item.name }}</span>
  <span>{{ item.price | currency }}</span>
</div>
```

*enlarge*

6. Click the **Checkout** button to see the updated cart. Remember that changing the application causes the preview to refresh, which empties the cart.



Click on the link to navigate to the shipping prices.



## What's next

You now have a store application with a product catalog, a shopping cart, and you can look up shipping prices.

To continue exploring Angular:

- Continue to [Forms for User Input](#) to finish the application by adding the shopping cart view and a checkout form
- Skip ahead to [Deployment](#) to move to local development, or deploy your application to Firebase or your own server

*Last reviewed on Mon Feb 28 2022*





# Using forms for user input



This guide builds on the [Managing Data](#) step of the Getting Started tutorial, [Get started with a basic Angular app](#).

This section walks you through adding a form-based checkout feature to collect user information as part of checkout.

---

## Define the checkout form model

This step shows you how to set up the checkout form model in the component class. The form model determines the status of the form.

1. Open `cart.component.ts`.

2. Import the `FormBuilder` service from the `@angular/forms` package. This service provides convenient methods for generating controls.

```
src/app/cart/cart.component.ts

import { Component } from '@angular/core';
import { FormBuilder } from '@angular/forms';

import { CartService } from '../cart.service';
```

3. Inject the `FormBuilder` service in the `CartComponent` `constructor()`. This service is part of the `ReactiveFormsModule` module, which you've already imported.

```
src/app/cart/cart.component.ts

export class CartComponent {

  constructor(
    private cartService: CartService,
    private formBuilder: FormBuilder,
  ) {}
}
```

4. To gather the user's name and address, use the `FormBuilder` `group()` method to set the `checkoutForm` property to a form model containing `name` and `address` fields.

```
src/app/cart/cart.component.ts

export class CartComponent {

  items = this.cartService.getItems();

  checkoutForm = this.formBuilder.group({
    name: '',
    address: ''
  });

  constructor(
    private cartService: CartService,
    private formBuilder: FormBuilder,
  ) {}
}
```

5. Define an `onSubmit()` method to process the form. This method allows users to submit their name and address. In addition, this method uses the `clearCart()` method of the `CartService` to reset the form and clear the cart.

The entire cart component class is as follows:

src/app/cart/cart.component.ts

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

import { CartService } from '../cart.service';

@Component({
  selector: 'app-cart',
  templateUrl: './cart.component.html',
  styleUrls: ['./cart.component.css']
})
export class CartComponent {

  items = this.cartService.getItems();

  checkoutForm = this.formBuilder.group({
    name: '',
    address: ''
  });

  constructor(
    private cartService: CartService,
    private formBuilder: FormBuilder,
  ) {}

  onSubmit(): void {
    // Process checkout data here
    this.items = this.cartService.clearCart();
    console.warn('Your order has been submitted', this.checkoutForm.value);
    this.checkoutForm.reset();
  }
}
```

---

## / Create the checkout form

Use the following steps to add a checkout form at the bottom of the Cart view.

1. At the bottom of `cart.component.html`, add an HTML `<form>` element and a **Purchase** button.
2. Use a `formGroup` property binding to bind `checkoutForm` to the HTML `<form>`.

src/app/cart/cart.component.html

```
<form [formGroup]="checkoutForm">

  <button class="button" type="submit">Purchase</button>

</form>
```

3. On the `form` tag, use an `ngSubmit` event binding to listen for the form submission and call the `onSubmit()` method with the `checkoutForm` value.

src/app/cart/cart.component.html (cart component template detail)

```
<form [formGroup]="checkoutForm" (ngSubmit)="onSubmit()">
  <button ...>
</form>
```

4. Add `<input>` fields for `name` and `address`, each with a `formControlName` attribute that binds to the `checkoutForm` form controls for `name` and `address` to their `<input>` fields. The complete component is as follows:

src/app/cart/cart.component.html

```
<h3>Cart</h3>

<p>
  <a routerLink="/shipping">Shipping Prices</a>
</p>

<div class="cart-item" *ngFor="let item of items">
  <span>{{ item.name }} </span>
  <span>{{ item.price | currency }}</span>
</div>

<form [formGroup]="checkoutForm" (ngSubmit)="onSubmit()">

  <div>
    <label for="name">
      Name
    </label>
    <input id="name" type="text" formControlName="name">
  </div>


  <div>
    <label for="address">
      Address
    </label>
    <input id="address" type="text" formControlName="address">
  </div>

  <button class="button" type="submit">Purchase</button>

</form>
```

After putting a few items in the cart, users can review their items, enter their name and address, and submit their purchase.

My Store

 Checkout

Cart

Shipping Prices

Phone XL \$799.00

Phone Mini\$699.00

Phone XL \$799.00

NAME

ADDRESS

Purchase

To confirm submission, open the console to see an object containing the name and address you submitted.

## What's next

You have a complete online store application with a product catalog, a shopping cart, and a checkout function.

Continue to the ["Deployment"](#) section to move to local development, or deploy your app to Firebase or your own server.

*Last reviewed on Wed Sep 15 2021*



# Deploying an application



Deploying your application is the process of compiling, or building, your code and hosting the JavaScript, CSS, and HTML on a web server.

This section builds on the previous steps in the [Getting Started](#) tutorial and shows you how to deploy your application.

---

## Prerequisites

A best practice is to run your project locally before you deploy it. To run your project locally, you need the following installed on your computer:

- [Node.js](#)
- The [Angular CLI](#) . From the terminal, install the Angular CLI globally with:

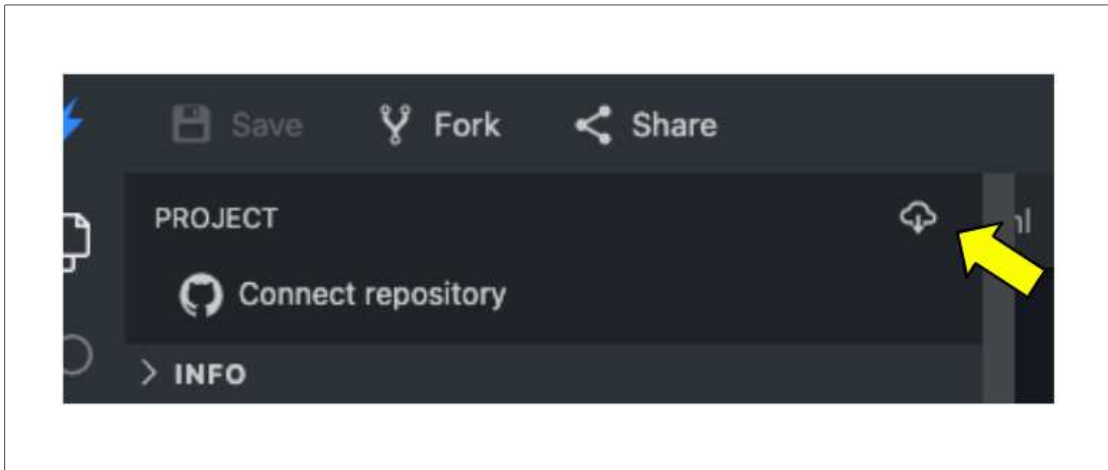
```
npm install -g @angular/cli
```

With the Angular CLI, you can use the command `ng` to create new workspaces, new projects, serve your application during development, or produce builds to share or distribute.

---

## Running your application locally

1. Download the source code from your StackBlitz project by clicking the `Download Project` icon in the left menu, across from `Project`, to download your project as a zip archive.



2. Unzip the archive and change directory to the newly created project. For example:

```
cd angular-ynqttp
```

3. To download and install npm packages, use the following npm CLI command:

```
npm install (en package.json)
```

4. Use the following CLI command to run your application locally:

```
ng serve
```

5. To see your application in the browser, go to `http://localhost:4200/`. If the default port 4200 is not available, you can specify another port with the port flag as in the following example:

```
ng serve --port 4201
```

While serving your application, you can edit your code and see the changes update automatically in the browser. To stop the `ng serve` command, press `Ctrl + c`.

---

## Building and hosting your application



1. To build your application for production, use the `build` command. By default, this command uses the `production` build configuration.

```
ng build
```

This command creates a `dist` folder in the application root directory with all the files that a hosting service needs for serving your application.

If the above `ng build` command throws an error about missing packages, append the missing dependencies in your local project's `package.json` file to match the one in the downloaded StackBlitz project.

2. Copy the contents of the `dist/my-project-name` folder to your web server. Because these files are static, you can host them on any web server capable of serving files; such as `Node.js`, Java, .NET, or any backend such as [Firebase](#), [Google Cloud](#), or [App Engine](#). For more information, see [Building & Serving](#) and [Deployment](#).

## What's next

In this tutorial, you've laid the foundation to explore the Angular world in areas such as mobile development, UX/UI development, and server-side rendering. You can go deeper by studying more of Angular's features, engaging with the vibrant community, and exploring the robust ecosystem.

### Learning more Angular

For a more in-depth tutorial that leads you through building an application locally and exploring many of Angular's most popular features, see [Tour of Heroes](#).

To explore Angular's foundational concepts, see the guides in the Understanding Angular section such as [Angular Components Overview](#) or [Template syntax](#).

### Joining the community

[Tweet that you've completed this tutorial](#), tell us what you think, or submit [suggestions for future editions](#).

Keep current by following the [Angular blog](#).

### Exploring the Angular ecosystem

To support your UX/UI development, see [Angular Material](#).

The Angular community also has an extensive [network of third-party tools and libraries](#).