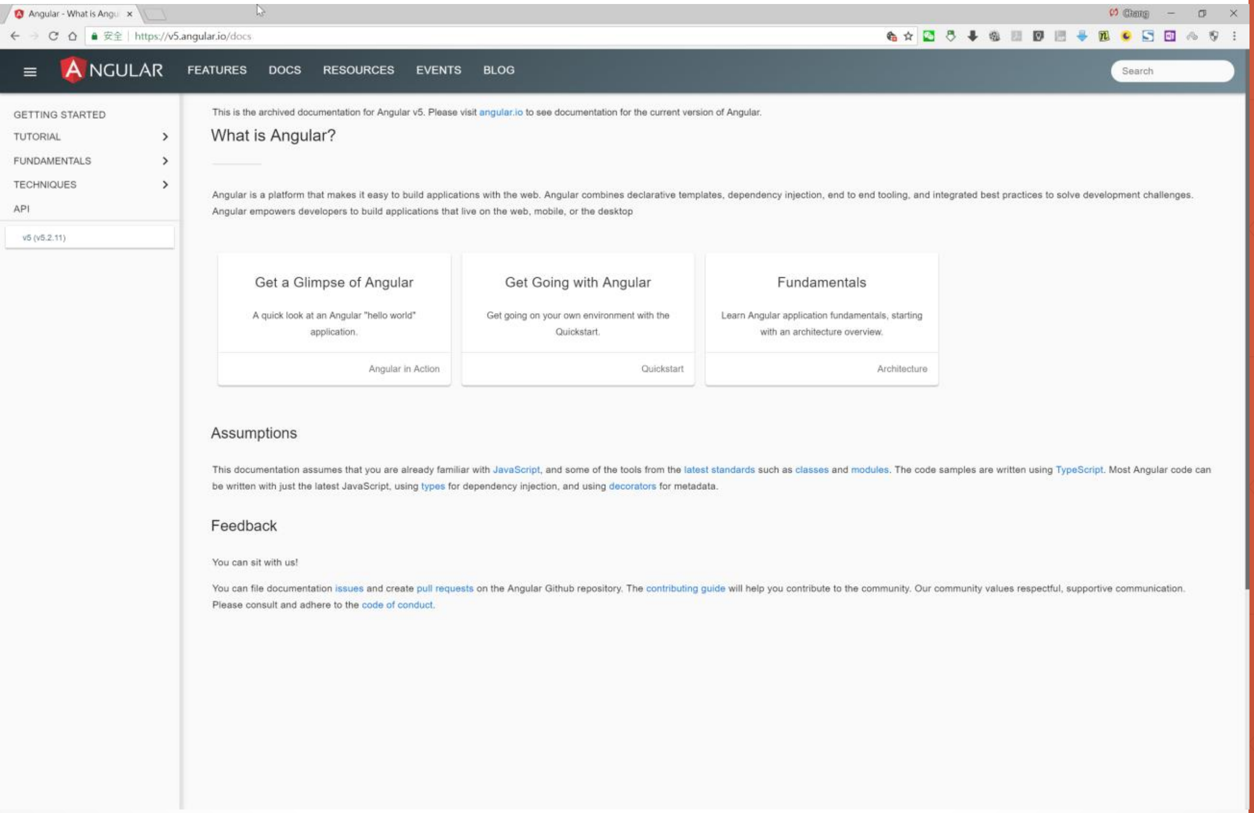
Angular 開發說明

技術文件指引

Angular

官網與開發說明文件

- Angular 5 系統說明文件
- https://v5.angular.io/docs



This is the archived documentation for Angular v5. Please visit angular.io to see documentation for the current version of Angular.

What is Angular?

Angular is a platform that makes it easy to build applications with the web. Angular combines declarative templates, dependency injection, end to end tooling, and integrated best practices to solve development challenges. Angular empowers developers to build applications that live on the web, mobile, or the desktop

Get a Glimpse of Angular

A quick look at an Angular "hello world" application.

Angular in Action

Get Going with Angular

Get going on your own environment with the Quickstart.

Quickstart

Fundamentals

Learn Angular application fundamentals, starting with an architecture overview.

Architecture

Assumptions

This documentation assumes that you are already familiar with JavaScript, and some of the tools from the latest standards such as classes and modules. The code samples are written using TypeScript. Most Angular code can be written with just the latest JavaScript, using types for dependency injection, and using decorators for metadata.

Feedback

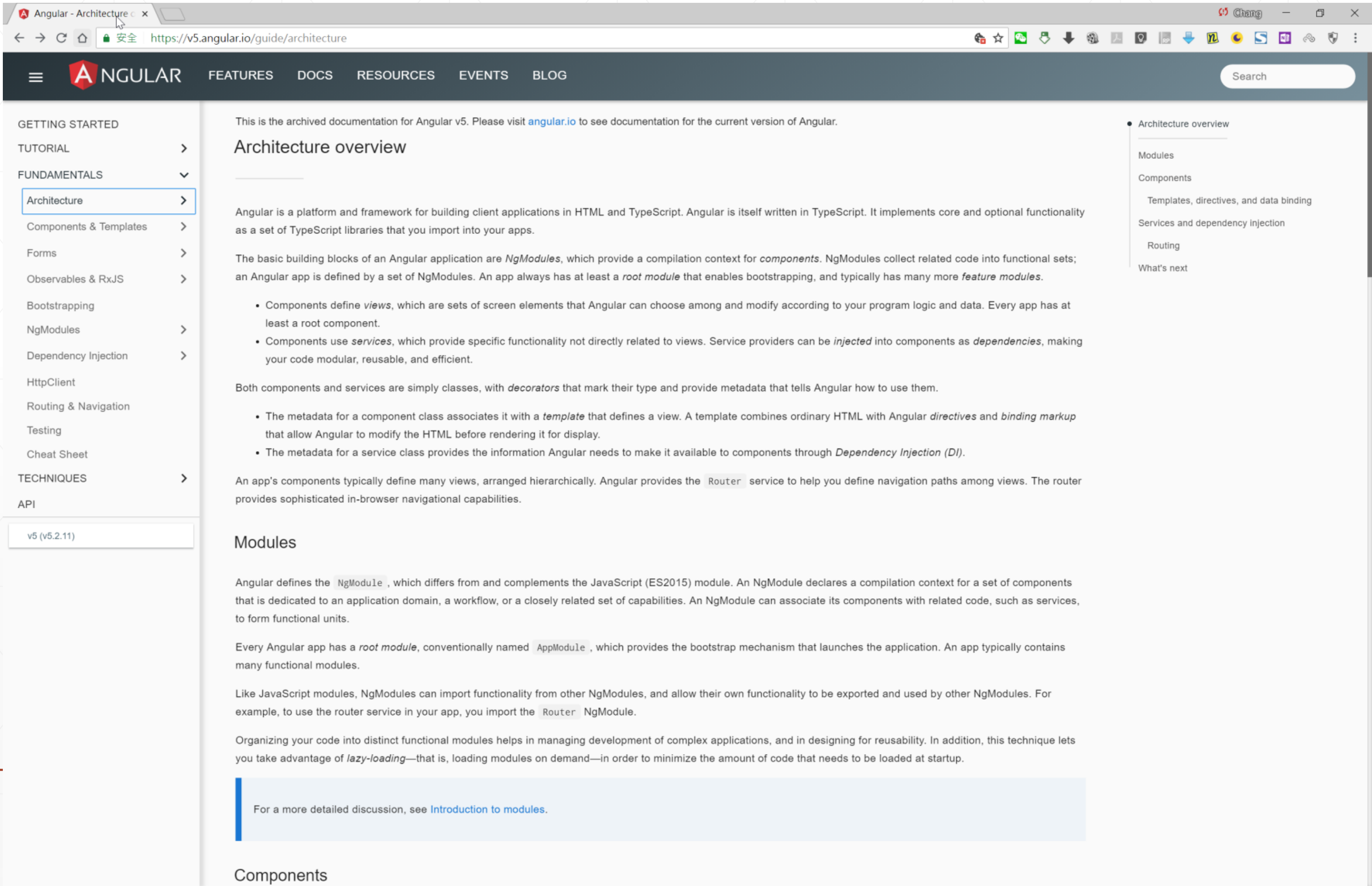
You can sit with us!

You can file documentation issues and create pull requests on the Angular Github repository. The contributing guide will help you contribute to the community. Our community values respectful, supportive communication. Please consult and adhere to the code of conduct.

Angular 系統說明與技術說明

- Angular 基礎架構
  https://v5.angular.io/guide/architecture
- Angular 開發案例
  https://v5.angular.io/tutorial

Angular 基礎架構
https://v5.angular.io/guide/architecture



This is the archived documentation for Angular v5. Please visit angular.io to see documentation for the current version of Angular.

Architecture overview

Angular is a platform and framework for building client applications in HTML and TypeScript. Angular is itself written in TypeScript. It implements core and optional functionality as a set of TypeScript libraries that you import into your apps.

The basic building blocks of an Angular application are *NgModules*, which provide a compilation context for *components*. NgModules collect related code into functional sets; an Angular app is defined by a set of NgModules. An app always has at least a *root module* that enables bootstrapping, and typically has many more *feature modules*.

- Components define *views*, which are sets of screen elements that Angular can choose among and modify according to your program logic and data. Every app has at least a root component.
- Components use *services*, which provide specific functionality not directly related to views. Service providers can be *injected* into components as *dependencies*, making your code modular, reusable, and efficient.

Both components and services are simply classes, with *decorators* that mark their type and provide metadata that tells Angular how to use them.

- The metadata for a component class associates it with a *template* that defines a view. A template combines ordinary HTML with Angular *directives* and *binding markup* that allow Angular to modify the HTML before rendering it for display.
- The metadata for a service class provides the information Angular needs to make it available to components through *Dependency Injection (DI)*.

An app's components typically define many views, arranged hierarchically. Angular provides the `Router` service to help you define navigation paths among views. The router provides sophisticated in-browser navigational capabilities.

Modules

Angular defines the `NgModule`, which differs from and complements the JavaScript (ES2015) module. An NgModule declares a compilation context for a set of components that is dedicated to an application domain, a workflow, or a closely related set of capabilities. An NgModule can associate its components with related code, such as services, to form functional units.

Every Angular app has a *root module*, conventionally named `AppModule`, which provides the bootstrap mechanism that launches the application. An app typically contains many functional modules.

Like JavaScript modules, NgModules can import functionality from other NgModules, and allow their own functionality to be exported and used by other NgModules. For example, to use the router service in your app, you import the `Router` NgModule.

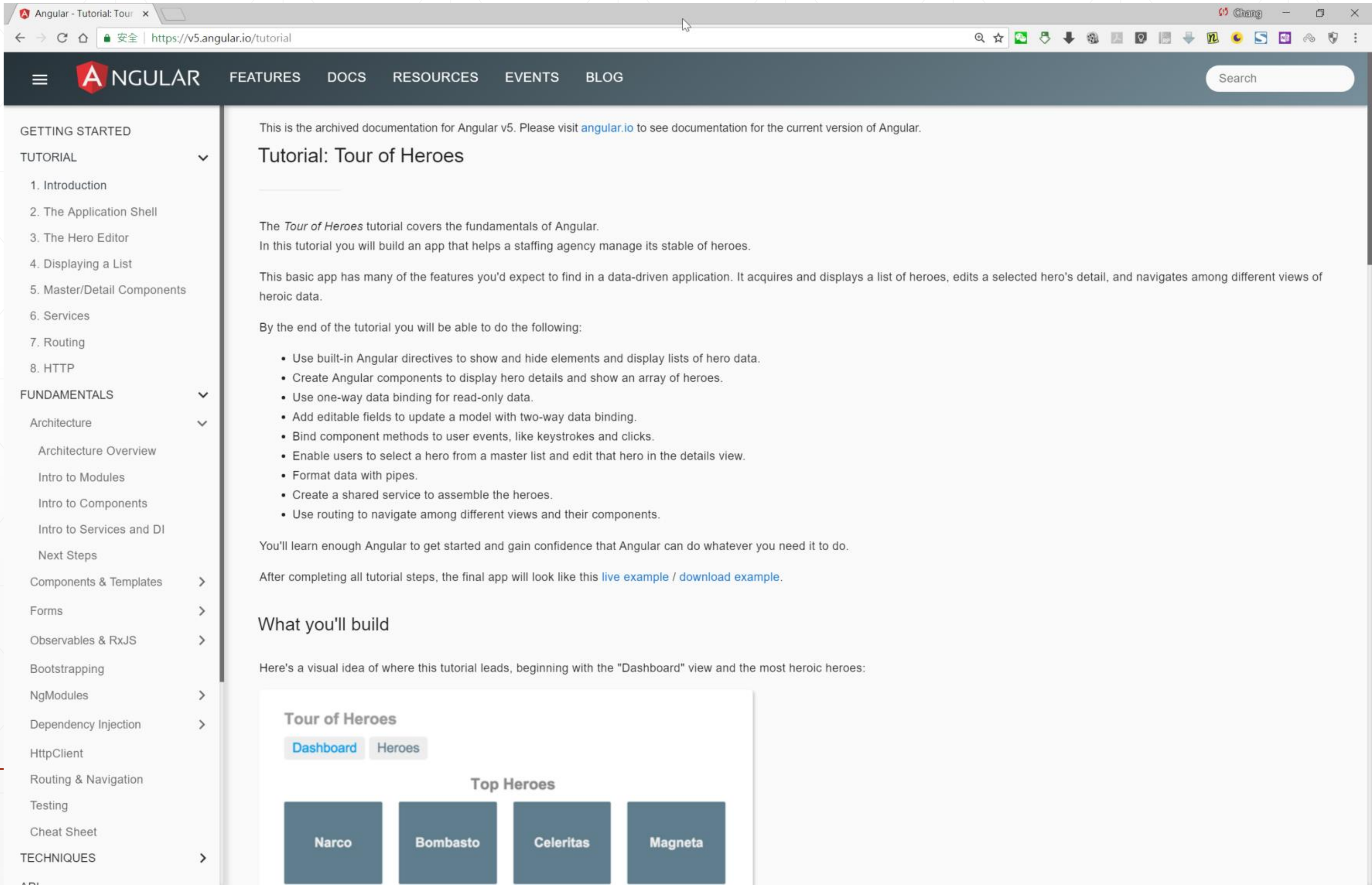
Organizing your code into distinct functional modules helps in managing development of complex applications, and in designing for reusability. In addition, this technique lets you take advantage of *lazy-loading*—that is, loading modules on demand—in order to minimize the amount of code that needs to be loaded at startup.

> For a more detailed discussion, see Introduction to modules.

Components

Angular 開發案例

https://v5.angular.io/tutorial



GETTING STARTED
TUTORIAL
1. Introduction
2. The Application Shell
3. The Hero Editor
4. Displaying a List
5. Master/Detail Components
6. Services
7. Routing
8. HTTP
FUNDAMENTALS
Architecture
Architecture Overview
Intro to Modules
Intro to Components
Intro to Services and DI
Next Steps
Components & Templates
Forms
Observables & RxJS
Bootstrapping
NgModules
Dependency Injection
HttpClient
Routing & Navigation
Testing
Cheat Sheet
TECHNIQUES

This is the archived documentation for Angular v5. Please visit angular.io to see documentation for the current version of Angular.

Tutorial: Tour of Heroes

The *Tour of Heroes* tutorial covers the fundamentals of Angular.
In this tutorial you will build an app that helps a staffing agency manage its stable of heroes.

This basic app has many of the features you'd expect to find in a data-driven application. It acquires and displays a list of heroes, edits a selected hero's detail, and navigates among different views of heroic data.

By the end of the tutorial you will be able to do the following:

- Use built-in Angular directives to show and hide elements and display lists of hero data.
- Create Angular components to display hero details and show an array of heroes.
- Use one-way data binding for read-only data.
- Add editable fields to update a model with two-way data binding.
- Bind component methods to user events, like keystrokes and clicks.
- Enable users to select a hero from a master list and edit that hero in the details view.
- Format data with pipes.
- Create a shared service to assemble the heroes.
- Use routing to navigate among different views and their components.

You'll learn enough Angular to get started and gain confidence that Angular can do whatever you need it to do.

After completing all tutorial steps, the final app will look like this live example / download example.

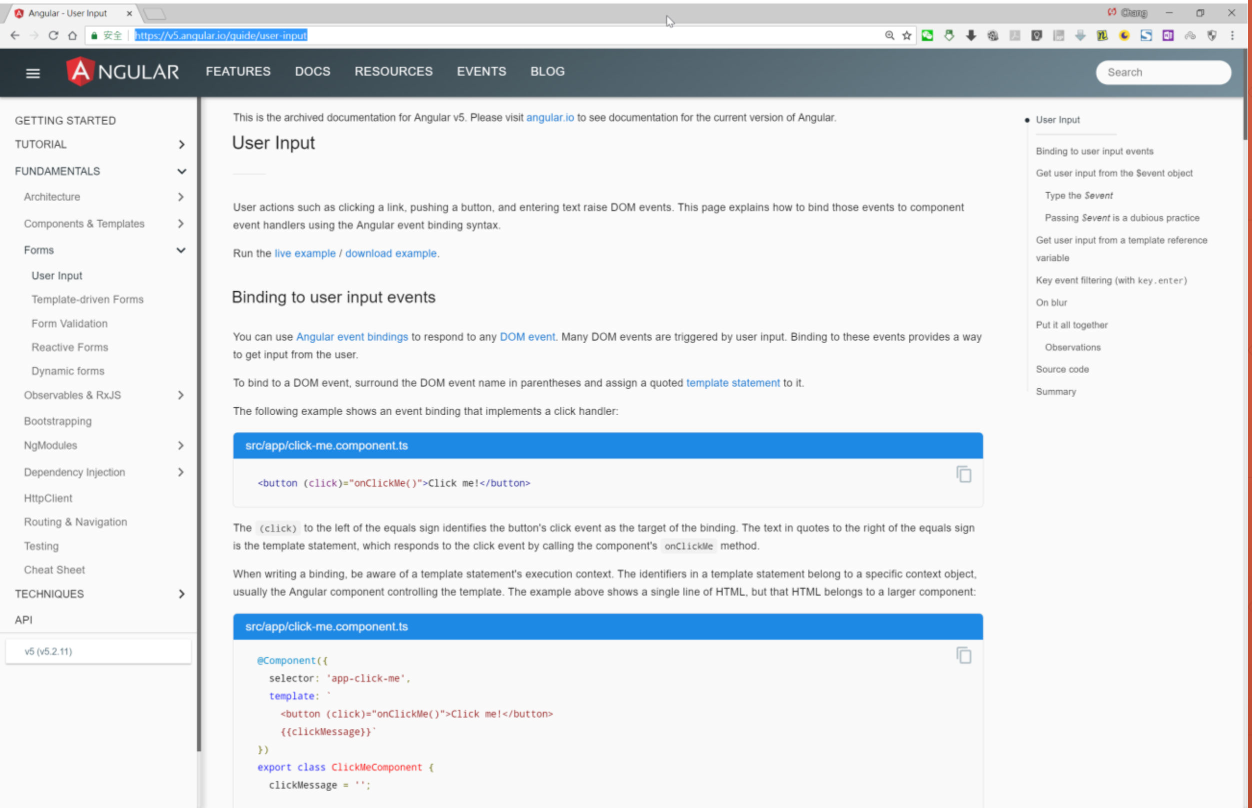
What you'll build

Here's a visual idea of where this tutorial leads, beginning with the "Dashboard" view and the most heroic heroes:

- Forms
- Template-driven forms

Angular

Forms



This is the archived documentation for Angular v5. Please visit angular.io to see documentation for the current version of Angular.

User Input

User actions such as clicking a link, pushing a button, and entering text raise DOM events. This page explains how to bind those events to component event handlers using the Angular event binding syntax.

Run the live example / download example.

Binding to user input events

You can use Angular event bindings to respond to any DOM event. Many DOM events are triggered by user input. Binding to these events provides a way to get input from the user.

To bind to a DOM event, surround the DOM event name in parentheses and assign a quoted template statement to it.

The following example shows an event binding that implements a click handler:

src/app/click-me.component.ts

```
<button (click)="onClickMe()">Click me!</button>
```

The `(click)` to the left of the equals sign identifies the button's click event as the target of the binding. The text in quotes to the right of the equals sign is the template statement, which responds to the click event by calling the component's `onClickMe` method.

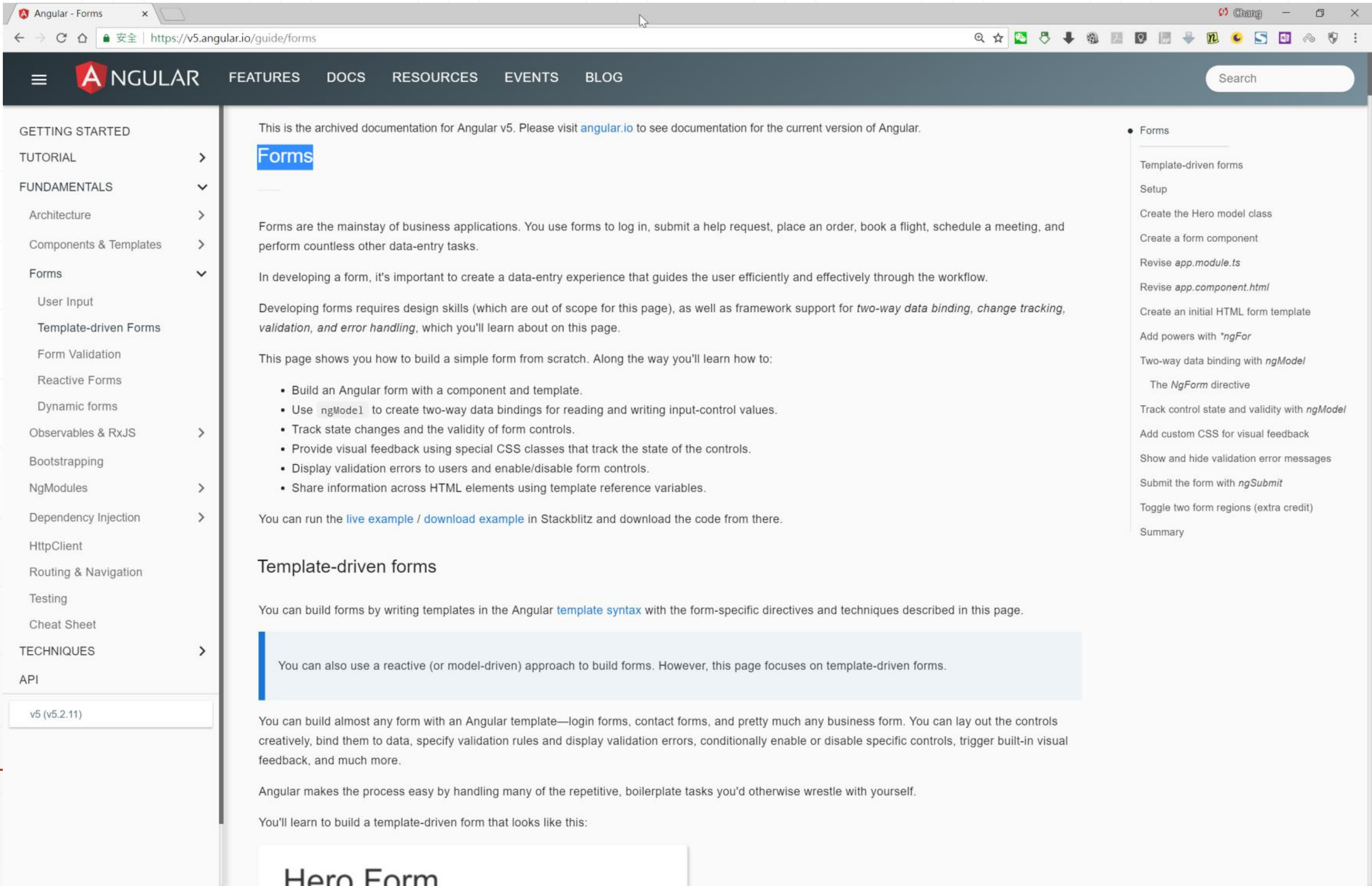
When writing a binding, be aware of a template statement's execution context. The identifiers in a template statement belong to a specific context object, usually the Angular component controlling the template. The example above shows a single line of HTML, but that HTML belongs to a larger component:

src/app/click-me.component.ts

```
@Component({
  selector: 'app-click-me',
  template: `
    <button (click)="onClickMe()">Click me!</button>
    {{clickMessage}}`
})
export class ClickMeComponent {
  clickMessage = '';
```

Forms

Forms
https://v5.angular.io/guide/forms



This is the archived documentation for Angular v5. Please visit angular.io to see documentation for the current version of Angular.

Forms

Forms are the mainstay of business applications. You use forms to log in, submit a help request, place an order, book a flight, schedule a meeting, and perform countless other data-entry tasks.

In developing a form, it's important to create a data-entry experience that guides the user efficiently and effectively through the workflow.

Developing forms requires design skills (which are out of scope for this page), as well as framework support for *two-way data binding, change tracking, validation, and error handling*, which you'll learn about on this page.

This page shows you how to build a simple form from scratch. Along the way you'll learn how to:

- Build an Angular form with a component and template.
- Use `ngModel` to create two-way data bindings for reading and writing input-control values.
- Track state changes and the validity of form controls.
- Provide visual feedback using special CSS classes that track the state of the controls.
- Display validation errors to users and enable/disable form controls.
- Share information across HTML elements using template reference variables.

You can run the live example / download example in Stackblitz and download the code from there.

Template-driven forms

You can build forms by writing templates in the Angular template syntax with the form-specific directives and techniques described in this page.

> You can also use a reactive (or model-driven) approach to build forms. However, this page focuses on template-driven forms.

You can build almost any form with an Angular template—login forms, contact forms, and pretty much any business form. You can lay out the controls creatively, bind them to data, specify validation rules and display validation errors, conditionally enable or disable specific controls, trigger built-in visual feedback, and much more.

Angular makes the process easy by handling many of the repetitive, boilerplate tasks you'd otherwise wrestle with yourself.

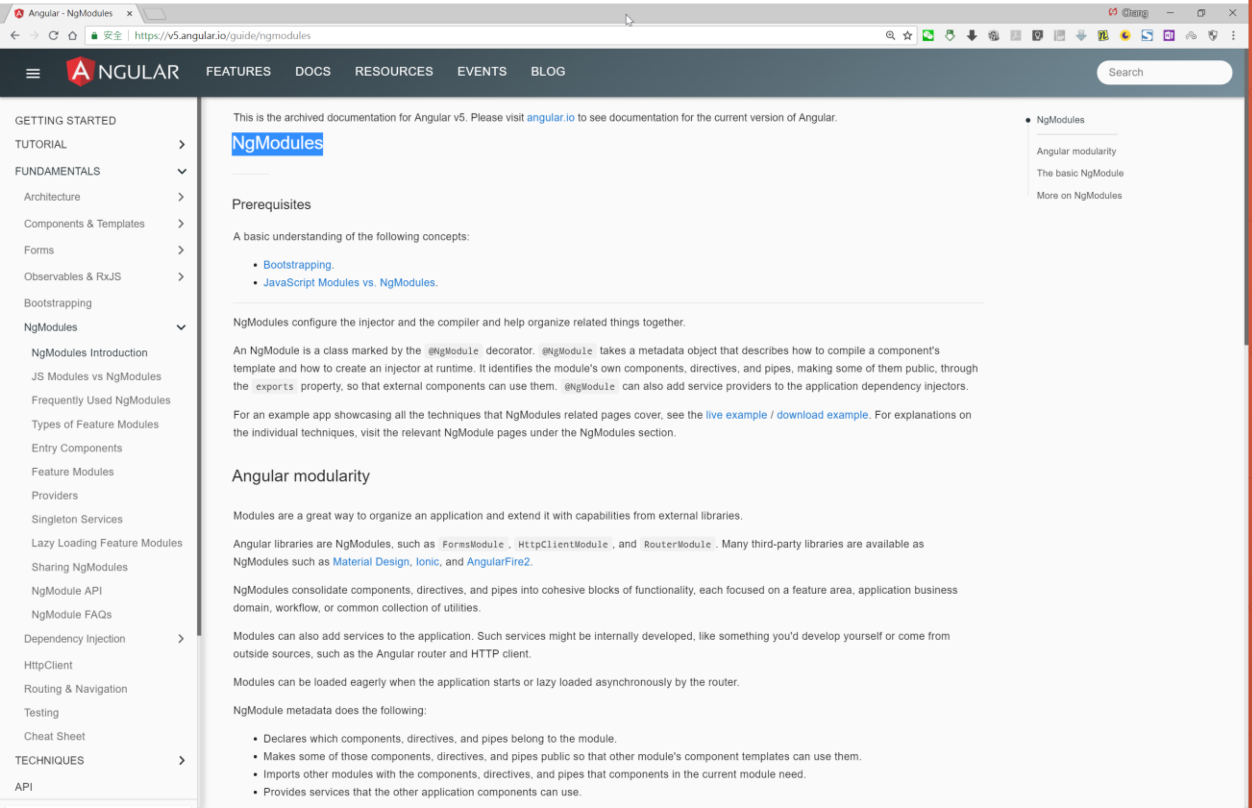
You'll learn to build a template-driven form that looks like this:

Hero Form

Angular

NgModule

- NgModules
  https://v5.angular.io/guide/ngmodules



This is the archived documentation for Angular v5. Please visit angular.io to see documentation for the current version of Angular.

NgModules

Prerequisites

A basic understanding of the following concepts:

- Bootstrapping.
- JavaScript Modules vs. NgModules.

NgModules configure the injector and the compiler and help organize related things together.

An NgModule is a class marked by the `@NgModule` decorator. `@NgModule` takes a metadata object that describes how to compile a component's template and how to create an injector at runtime. It identifies the module's own components, directives, and pipes, making some of them public, through the `exports` property, so that external components can use them. `@NgModule` can also add service providers to the application dependency injectors.

For an example app showcasing all the techniques that NgModules related pages cover, see the live example / download example. For explanations on the individual techniques, visit the relevant NgModule pages under the NgModules section.

Angular modularity

Modules are a great way to organize an application and extend it with capabilities from external libraries.

Angular libraries are NgModules, such as `FormsModule`, `HttpClientModule`, and `RouterModule`. Many third-party libraries are available as NgModules such as Material Design, Ionic, and AngularFire2.

NgModules consolidate components, directives, and pipes into cohesive blocks of functionality, each focused on a feature area, application business domain, workflow, or common collection of utilities.

Modules can also add services to the application. Such services might be internally developed, like something you'd develop yourself or come from outside sources, such as the Angular router and HTTP client.

Modules can be loaded eagerly when the application starts or lazy loaded asynchronously by the router.

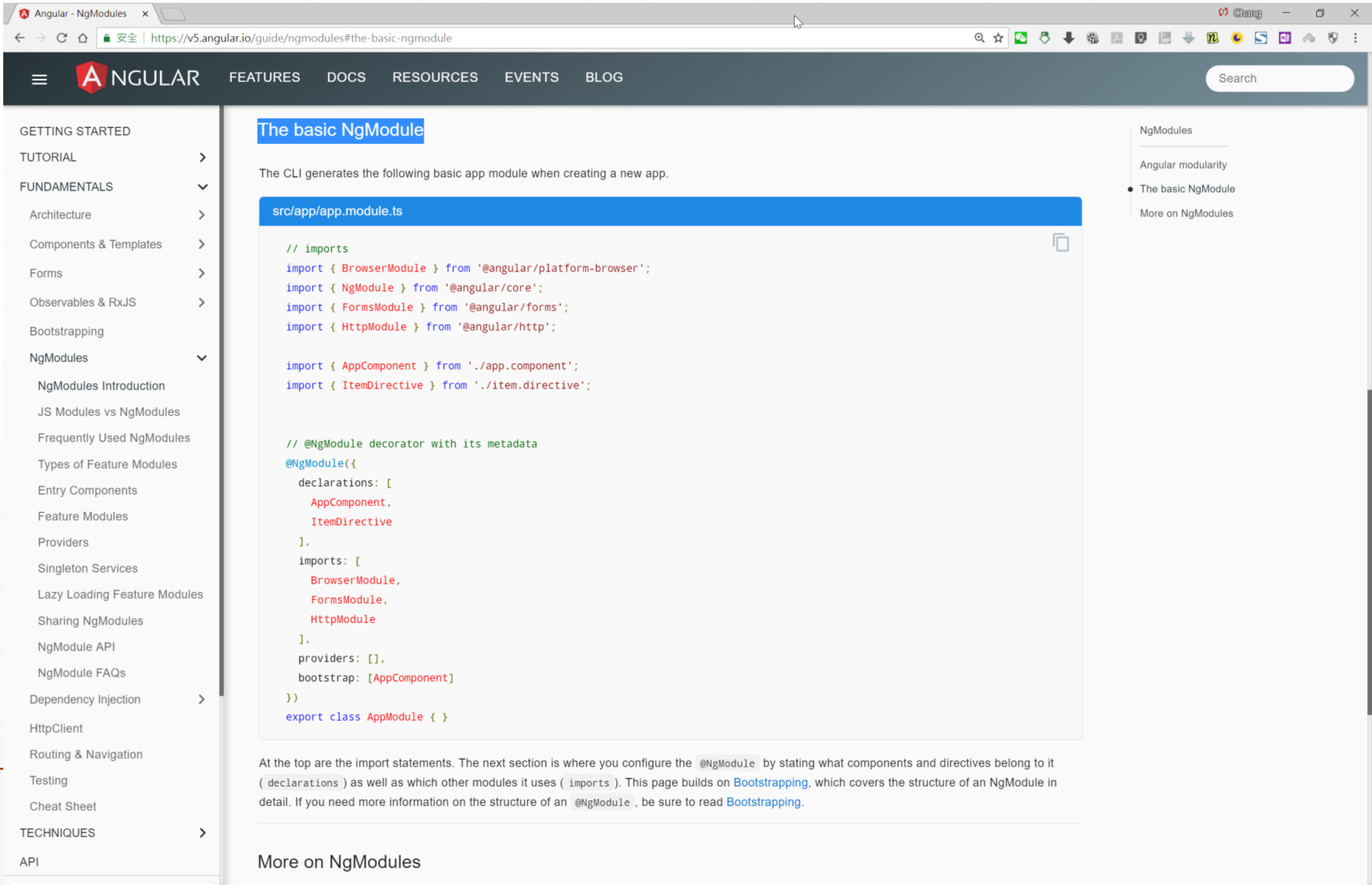
NgModule metadata does the following:

- Declares which components, directives, and pipes belong to the module.
- Makes some of those components, directives, and pipes public so that other module's component templates can use them.
- Imports other modules with the components, directives, and pipes that components in the current module need.
- Provides services that the other application components can use.

NgModules

The basic NgModule

https://v5.angular.io/guide/ngmodules#the-basic-ngmodule



The basic NgModule

The CLI generates the following basic app module when creating a new app.

src/app/app.module.ts

```
// imports
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpModule } from '@angular/http';

import { AppComponent } from './app.component';
import { ItemDirective } from './item.directive';


// @NgModule decorator with its metadata
@NgModule({
  declarations: [
    AppComponent,
    ItemDirective
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

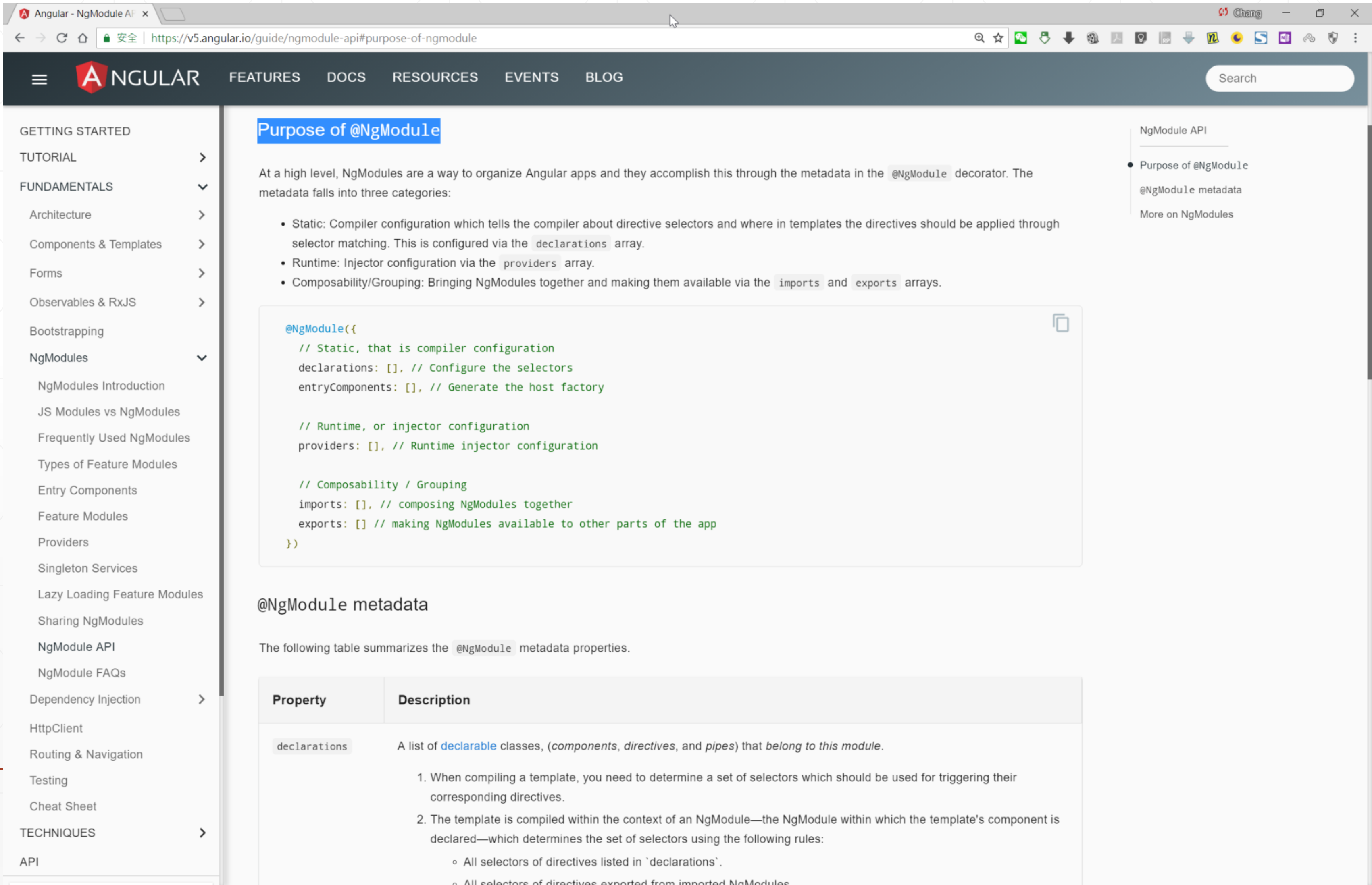
At the top are the import statements. The next section is where you configure the `@NgModule` by stating what components and directives belong to it (`declarations`) as well as which other modules it uses (`imports`). This page builds on Bootstrapping, which covers the structure of an NgModule in detail. If you need more information on the structure of an `@NgModule` , be sure to read Bootstrapping.

More on NgModules

NgModules

Purpose of @NgModule

https://v5.angular.io/guide/ngmodule-api#purpose-of-ngmodule



Purpose of @NgModule

At a high level, NgModules are a way to organize Angular apps and they accomplish this through the metadata in the `@NgModule` decorator. The metadata falls into three categories:

- Static: Compiler configuration which tells the compiler about directive selectors and where in templates the directives should be applied through selector matching. This is configured via the `declarations` array.
- Runtime: Injector configuration via the `providers` array.
- Composability/Grouping: Bringing NgModules together and making them available via the `imports` and `exports` arrays.

```
@NgModule({
  // Static, that is compiler configuration
  declarations: [], // Configure the selectors
  entryComponents: [], // Generate the host factory

  // Runtime, or injector configuration
  providers: [], // Runtime injector configuration

  // Composability / Grouping
  imports: [], // composing NgModules together
  exports: [] // making NgModules available to other parts of the app
})
```

@NgModule metadata

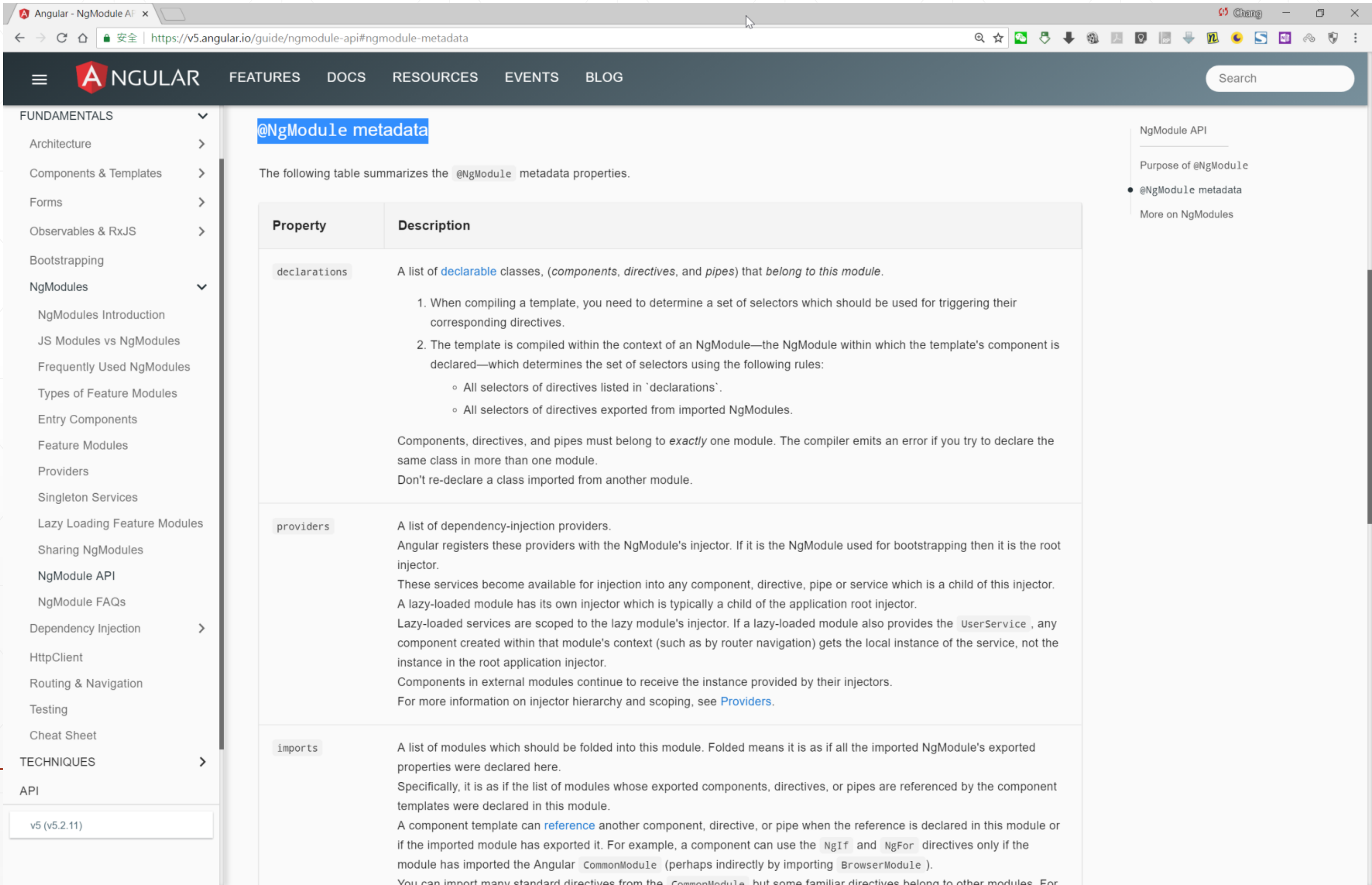
The following table summarizes the `@NgModule` metadata properties.

Property	Description
`declarations`	A list of declarable classes, (*components*, *directives*, and *pipes*) that *belong to this module*. 1. When compiling a template, you need to determine a set of selectors which should be used for triggering their corresponding directives. 2. The template is compiled within the context of an NgModule—the NgModule within which the template's component is declared—which determines the set of selectors using the following rules: ◦ All selectors of directives listed in `declarations`.

NgModules

@NgModule Metadata

https://v5.angular.io/guide/module-types


Angular - NgModule API

https://v5.angular.io/guide/ngmodule-api#ngmodule-metadata

@NgModule metadata

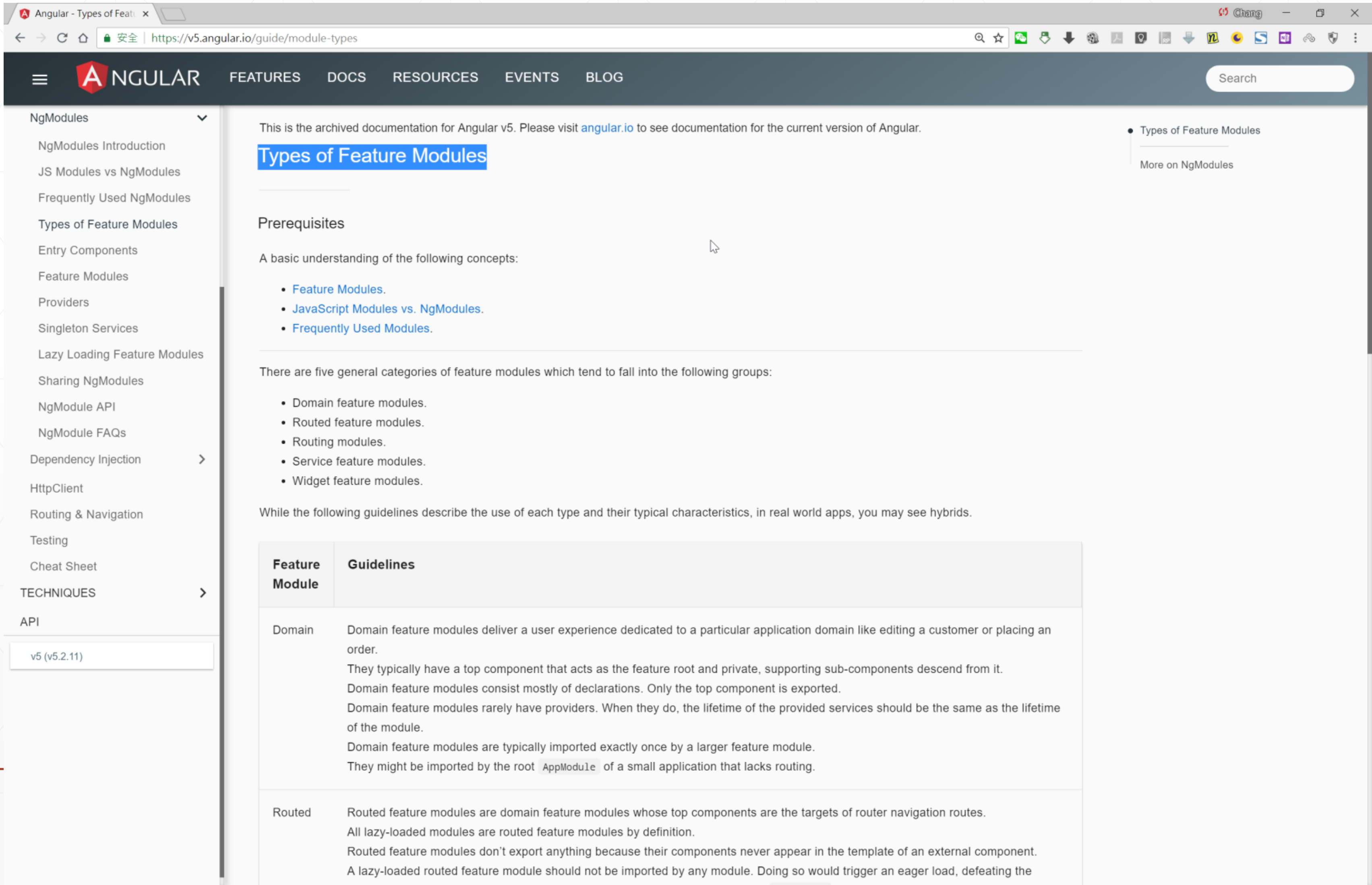
The following table summarizes the `@NgModule` metadata properties.

Property	Description
`declarations`	A list of declarable classes, (*components*, *directives*, and *pipes*) that *belong to this module*. 1. When compiling a template, you need to determine a set of selectors which should be used for triggering their corresponding directives. 2. The template is compiled within the context of an NgModule—the NgModule within which the template's component is declared—which determines the set of selectors using the following rules: ◦ All selectors of directives listed in `declarations`. ◦ All selectors of directives exported from imported NgModules. Components, directives, and pipes must belong to *exactly* one module. The compiler emits an error if you try to declare the same class in more than one module. Don't re-declare a class imported from another module.
`providers`	A list of dependency-injection providers. Angular registers these providers with the NgModule's injector. If it is the NgModule used for bootstrapping then it is the root injector. These services become available for injection into any component, directive, pipe or service which is a child of this injector. A lazy-loaded module has its own injector which is typically a child of the application root injector. Lazy-loaded services are scoped to the lazy module's injector. If a lazy-loaded module also provides the `UserService`, any component created within that module's context (such as by router navigation) gets the local instance of the service, not the instance in the root application injector. Components in external modules continue to receive the instance provided by their injectors. For more information on injector hierarchy and scoping, see Providers.
`imports`	A list of modules which should be folded into this module. Folded means it is as if all the imported NgModule's exported properties were declared here. Specifically, it is as if the list of modules whose exported components, directives, or pipes are referenced by the component templates were declared in this module. A component template can reference another component, directive, or pipe when the reference is declared in this module or if the imported module has exported it. For example, a component can use the `NgIf` and `NgFor` directives only if the module has imported the Angular `CommonModule` (perhaps indirectly by importing `BrowserModule`). You can import many standard directives from the `CommonModule` but some familiar directives belong to other modules. For

NgModules

Types of Feature Modules

https://v5.angular.io/guide/module-types



This is the archived documentation for Angular v5. Please visit angular.io to see documentation for the current version of Angular.

Types of Feature Modules

Prerequisites

A basic understanding of the following concepts:

- Feature Modules.
- JavaScript Modules vs. NgModules.
- Frequently Used Modules.

There are five general categories of feature modules which tend to fall into the following groups:

- Domain feature modules.
- Routed feature modules.
- Routing modules.
- Service feature modules.
- Widget feature modules.

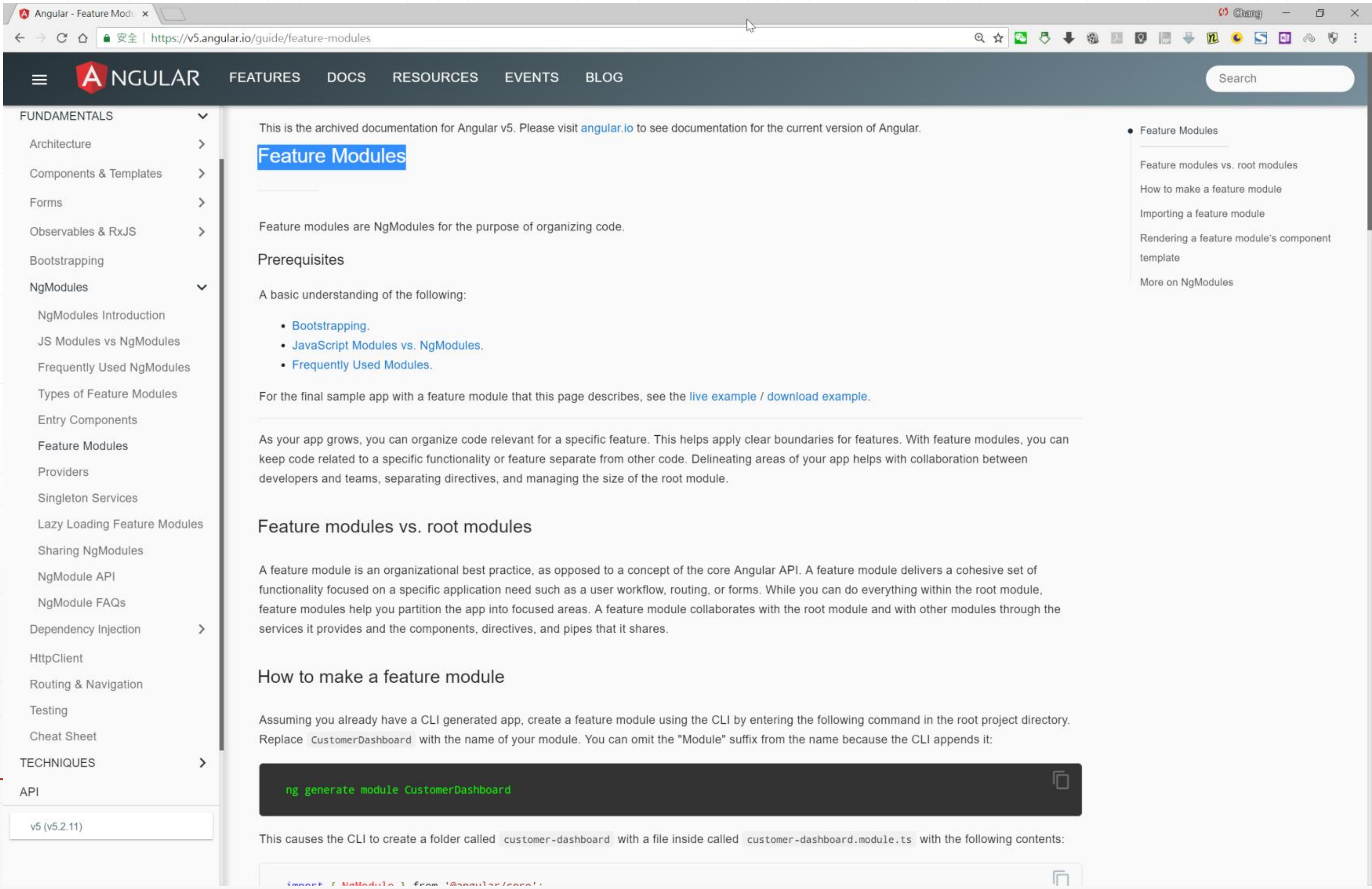
While the following guidelines describe the use of each type and their typical characteristics, in real world apps, you may see hybrids.

Feature Module	Guidelines
Domain	Domain feature modules deliver a user experience dedicated to a particular application domain like editing a customer or placing an order. They typically have a top component that acts as the feature root and private, supporting sub-components descend from it. Domain feature modules consist mostly of declarations. Only the top component is exported. Domain feature modules rarely have providers. When they do, the lifetime of the provided services should be the same as the lifetime of the module. Domain feature modules are typically imported exactly once by a larger feature module. They might be imported by the root `AppModule` of a small application that lacks routing.
Routed	Routed feature modules are domain feature modules whose top components are the targets of router navigation routes. All lazy-loaded modules are routed feature modules by definition. Routed feature modules don't export anything because their components never appear in the template of an external component. A lazy-loaded routed feature module should not be imported by any module. Doing so would trigger an eager load, defeating the

NgModules

Feature Modules

https://v5.angular.io/guide/feature-modules



This is the archived documentation for Angular v5. Please visit angular.io to see documentation for the current version of Angular.

Feature Modules

Feature modules are NgModules for the purpose of organizing code.

Prerequisites

A basic understanding of the following:

- Bootstrapping.
- JavaScript Modules vs. NgModules.
- Frequently Used Modules.

For the final sample app with a feature module that this page describes, see the live example / download example.

As your app grows, you can organize code relevant for a specific feature. This helps apply clear boundaries for features. With feature modules, you can keep code related to a specific functionality or feature separate from other code. Delineating areas of your app helps with collaboration between developers and teams, separating directives, and managing the size of the root module.

Feature modules vs. root modules

A feature module is an organizational best practice, as opposed to a concept of the core Angular API. A feature module delivers a cohesive set of functionality focused on a specific application need such as a user workflow, routing, or forms. While you can do everything within the root module, feature modules help you partition the app into focused areas. A feature module collaborates with the root module and with other modules through the services it provides and the components, directives, and pipes that it shares.

How to make a feature module

Assuming you already have a CLI generated app, create a feature module using the CLI by entering the following command in the root project directory. Replace `CustomerDashboard` with the name of your module. You can omit the "Module" suffix from the name because the CLI appends it:

```
ng generate module CustomerDashboard
```

This causes the CLI to create a folder called `customer-dashboard` with a file inside called `customer-dashboard.module.ts` with the following contents: