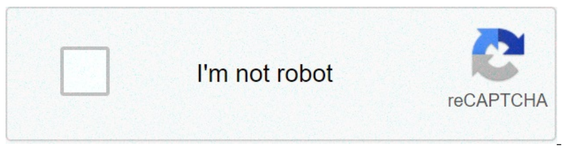


Continue

# Angular 8 full tutorial pdf

All you need to learn about Angular, the best tips and free code examples so you can get the most out of Angular. 60 Min READ! Angular MaterialBeginnersFormsAngular CLI Let me introduce you to Angular Angular is a platform for building mobile and desktop web applications. It has a big community of millions of developers who choose Angular to build compelling user interfaces. Angular is a JavaScript open-source front-end web application framework. It is primarily sustained by Google together with an extended community of people and companies. Angular solves many of the challenges faced when developing single page, cross platform, performant applications. It's fully extensible and works very well with other libraries. For additional details visit their official documentation page. My goal in this Angular real world example tutorial is to provide a complete guide for you to learn Angular step by step. We will start explaining the why's and basic concepts and then continue exploring more advanced notions. We want to help beginners through their first steps on the Angular world. As developers, we know that starting with a new technology can sometimes be a bit frustrating so want to help here. We will learn enough core Angular to get started and gain confidence that we can build any kind of app with Angular. We will be covering a lot of ground at an introductory level, but also, you will find plenty of references to topics with greater depth. To help you through your Angular learning process, we created an Angular app with a question and answer format (Q&A), where users will be able to ask, answer and vote questions. Also, we will explain how to connect this app with a remote API to handle data integration. So, in this complete tutorial you will learn all the concepts needed to create your first angular application. You can download all the source code of this angular free template by clicking the GET THE CODE button from above. Also, we published an online demo of the app we are going to build in this Getting Started with Angular guide. Our journey with Angular We began testing and experimenting with the very first release of Angular 2.0.0-beta.0 on december 2015 with hopes of finding a framework that was clearly better than its predecessor (Angular 1.x also know as AngularJS). I'm going to be completely honest with you here, we almost give up with all the inconsistency, breaking changes and a sort of identity crisis that happened in the middle of the Angular 2+ development. It was a long way until Angular reached a solid milestone with Universal (server-side rendering), ahead-of-time compilation (AOT), lazy loading and a solid bundling config working together nicely. Back in those years it was not easy to create a production ready angular application. But thanks to the angular team and to the angular community, that changed. Amazing things can be created with the latest versionss of Angular. You can check our most recent creation with Angular latest version in Fully - Angular Admin Template Being working, using and trying things out with Angular from the very beginning made us really understand the way it was designed and how it evolved. We were witnesses of the constant improvements and saw how they were all aligned to one simple yet important goal: "Creating an app wit Angular should be easy". As I mentioned before, for some time during the process, it wasn't. Now I can tell you, Angular is a super solid and stable framework you will love to work with. Current versions of Angular had evolved to the point where you will be quickly impressed. Angular is a great tool that will: Enable you to create software quicker and with less effort Result in a more maintainable software Encourage good programming practices and design patterns like MVC Allow you to collaborate easier with other people Allow you to become proficient in a reasonable time Address problems that may arise in your software architecture such as Dependency Injection, DRY (Don't Repeat Yourself), etc Differences between Angular versions When it all started, back in 2010, this framework was called AngularJS, and alludes to what we now know as Angular 1.x. Then in 2016, Angular 2 arrived as a complete rewrite of the framework, improving from lessons learned and promising performance improvements, and a more scalable and more modern framework. AngularJS was completely based on controllers and the view communicates using $scope whereas Angular 2 is 100% a component-based approach. In Angular 2, we don't have anymore the controllers and $scope. Components are the building blocks of an Angular 2 app. We will see the benefits of this change in a few minutes. The first version of Angular was named Angular 2. Later on, it was renamed to "Angular". Between Angular 2 and Angular 10 (the current latest stable version) there was Angular 4 (released early 2017), Angular 5 (released late 2017), Angular 6 (released early 2018), Angular 7 (released late 2018), Angular 8 (released mid 2019), Angular 9 (released early 2020). Angular 10 was released on June 2020. All the information related to versions can be found on the CHANGELOG. Don't freak out will all these versions. Because all versions from Angular 2 to Angular 10 are the same framework, they share the same core but they differ in lots of amazing improvements! From now on, every time we use the term Angular we are referring to the latest version of the framework that currently is Angular 10. What's new in Angular compared to AngularJS Just for the sake of history, let's go through the main differences between AngularJS and Angular: Angular is a complete rewrite of AngularJS. An Angular application and its architecture are different from AngularJS. The main building elements for Angular are modules, components, templates, metadata, data binding, directives, services and dependency injection. Angular does not have a "scope" concept or controllers, instead, it uses a component hierarchy as its main architecture. Angular follows a modularity concept. Similar functionalities are kept together inside modules. This gives Angular an optimized lighter core. The controller concept, which was present in AngularJS, was removed from Angular 2 and above which are component based UI. This help developers divide applications in components with desired features. These helped improve the flexibility and reusability compared to AngularJS. Angular expression syntax is focused on "[ ]" for property binding, and "( )" for event binding. With AngularJS, building a search engine (SEO) friendly Single Page Application was a major difficulty. But this bottleneck was eliminated with Angular 2 by enabling application rendering in the server. These tasks are possible thanks to the Angular Universal module. Angular recommends using the TypeScript language, which introduces these features: Static Typing Object Oriented Programming based on classes Support reactive programming using RxJS On top of TypeScript features, Angular also includes the benefits taken from ES6: For/Of loops Improved dependency injection Iterators Reflection Dynamic loading Asynchronous template compilation Simpler Routing From Angular 2 to Angular 4 There were some major changes, but mostly on the project structure with lots of refactors that made the framework more stable. Smaller and faster. The upgrade from 2.0 to 4.0 has reduced the bundled file size by 60% while also improving the applications speed. Angular 4 is compatible with newer versions of TypeScript 2.1 and TypeScript 2.2. Angular Universal: The vast majority of the Angular Universal code has been merged into Angular core. Animation Package: Animations taken from the Angular core and set within their own package. Meaning that if you don't use animations, the excess code won't end up in your app. From Angular 5 to Angular 7 Angular 6 was the first release of Angular that unifies the versions of Framework, Material and CLI. This change was made to clarify cross compatibility. Angular 7 was full of new features, bug fixes, performance improvements, and some code deprecation as a clean up of the refactors from old versions. Optimizations to the build process that reduces the application size by removing unnecessary code. Material Design components with server-side rendering. Angular Universal improvements for code allocation between the server and client-side versions of the application. Lots of improvements in the Angular CLI Smaller bundle sizes Improved compiler that supports incremental compilation meaning faster rebuilds. RxJS (reactive programming library) has been updated to version 6.x or later. Angular now requires TypeScript 3.x From Angular 8 to Angular 10+ Angular 8 was a release that spanned the entire platform, including the framework, Angular Material, and the CLI. This release improved application startup time on modern browsers. Also it changed the route configurations to use dynamic imports in favour of lazy loading. Angular 9 was very expected by the community because it introduced the Ivy compiler and runtime. Ivy is the name for Angular's next-generation compilation and rendering pipeline. With this release, the new compiler and runtime instructions are used by default instead of the older compiler and runtime, known as View Engine. The Ivy compiler offers the following advantages: Smaller bundle sizes Faster testing Better debugging Improved CSS class and style binding Improved type checking Improved build errors Improved build times, enabling AOT on by default Improved Internationalization More information about these advantages can be found on Angular 9 release note. Angular 10 release was smaller than typical; it has only been 4 months since the release of Angular 9. More info about this release can be found here. Moving ahead in this Angular tutorial, let's setup the development environment. After the previous introduction about the current state of the Angular Framework, we are now ready to get started working on our angular app. The best way to learn Angular is by following this step by step tutorial for beginners. In the following section of this angular free course we will go through the setup and requirements needed to start developing Angular apps. Let's start building a complete web app sample project with Angular Setup the Angular development environment In this section we will show you how to setup your local development environment so you can start developing Angular apps. A real application development happens in a local development environment that could be your personal machine. Follow our setup instructions to create a new Angular project. Angular requirements: Install NodeJS and npm Node.js and npm are fundamental to modern web development using Angular and other platforms. Node empowers client development and build tools. We are gonna use the node package manager (npm) to install all the JavaScript libraries dependencies. Get these right now if they're not installed on your computer. Note: Verify that you are running the latest stable versions of node and npm. The Angular CLI Angular apps are created and developed primarily through the Angular CLI (command line interface tool) that helps project creation, adding files, and performing a variety of ongoing development tasks such as testing, bundling, and deployment. The Angular CLI takes care of configuration and initialization of various libraries. It also helps us adding components, directives, services, etc, to already existing Angular applications. It's also worth mentioning that the CLI uses Typescript and Webpack for module bundling, Karma for unit testing, and Protractor for an end to end testing. It includes everything you need to start writing your Angular application right away. To install the Angular CLI globally, run the following command on your console npm install -g @angular/cli Note: although it's not recommended, you may need to add "sudo" in front of these commands to install the utilities globally. Important note: If you have an older version of the CLI installed in your computer, make sure you properly update it to the latest Angular CLI. Now that you have Angular and its dependencies installed, we can move on and start building our Angular app. Let's get started! Starting a new angular app with the CLI is easy! From your command line, run this command: ng new "my-new-angular-app" The command above will create a folder named "my-new-angular-app" and will copy all the required dependencies and configuration settings. The Angular CLI does this for you: Creates a new directory "my-new-angular-app" Downloads and installs Angular libraries and any other dependencies Installs and configures TypeScript Installs and configures Karma & Protractor (testing libraries) You can also use the ng init command. The difference between ng init and ng new is that ng new requires you to specify the folder name and it will create a folder copying the files while ng init will copy the files to the current folder. Now, you can cd into the created folder. To get a quick preview of your app inside the browser, use the serve command use ng serve This command runs the compiler in watch mode (looks for changes in the code and recompiles if needed), starts the server, launches the app in a browser, and keeps the app running while we continue building it. The Webpack Development server listens on HTTP port 4200. Hence, if you open the url you will see the app running. Using the Angular CLI to add new pages In Angular, there's some more boilerplate compared to AngularJS (Angular 1), but don't panic. The new Angular CLI also has more tools to help you out with this. For example, the new generator functions. They provide an easy way to create angular pages and services for your app. This makes going from a basic app to a full featured navigation web app much easier. I call that an easy learning curve :). To create a new component you can use the following command: ng generate component my-new-component ng g component my-new-component # using the alias √ Create app/pages/my-page/my-page.html √ Create app/pages/my-page/my-page.ts √ Create app/pages/my-page/my-page.scss The angular-CLI will add a reference to components, directives and pipes automatically in the app.module.ts. Note: Please refer to angular CLI documentation for more information about adding components and other elements to your app. Angular is a framework designed to build single page applications (SPAs) and most of its architecture design is focused towards doing that in an effective manner. Single-page application (or SPA) are applications that are accessed via web browser like other websites but offer more dynamic interactions resembling native mobile and desktop apps. The most notable difference between a regular website and SPA is the reduced amount of page refreshes. Typically, 95 percent of SPA code runs in the browser; the rest works in the server when the user needs new data or must perform secured operations such as authentication. As a result, the process of page rendering happens mostly on the client-side. Angular Modules Modules help organize an application into cohesive functionality blocks by wrapping components, pipes, directives, and services. They are just all about developer ergonomics. Angular applications are modular. Every Angular application has at least one module— the root module, conventionally named AppModule. The root module can be the only module in a small application, but most apps have many more modules. As the developer, it's up to you to decide how to use the modules. Typically, you map major functionality or a feature to a module. Let's say you have four major areas in your system. Each one will have its own module in addition to the root module, for a total of five modules. Any angular module is a class with the @NgModule decorator. Decorators are functions that modify JavaScript classes. They are basically used for attaching metadata to classes so that it knows the configuration of those classes and how they should work. The @NgModule decorator properties that describe the module are: declarations: The classes that belong to this module and are related to views. There are three classes in Angular that can contain views: components, directives and pipes. exports: The classes that should be accessible to other modules components. imports: Modules whose classes are needed by the components of this module. providers: Services present in one of the modules which are going to be used in the other modules or components. Once a service is included in the providers, it becomes accessible in all parts of that application. bootstrap: The root component which is the main view of the application. Only the root module has this property and it indicates the component that's gonna be bootstrapped. entryComponents: An entry component is any component that Angular loads imperatively, (which means you're not referencing it in the template), by type. Angular Components Components are the most basic building block of an UI and Angular applications. A component controls one or more sections on the screen (what we call views). For example in this example we have components like AppComponent (the bootstrapped component), CategoriesComponent, CategoryQuestionsComponent, QuestionAnswersComponent etc. A component is self contained and represents a reusable piece of UI that is usually constituted by three important things: A piece of html code that is known as the view A class that encapsulates all available data and interactions to that view through an API of properties and methods architectured by Angular. Here's where we define the application logic (what it does to support the view) And the aforementioned html element also known as selector. Using the Angular @Component decorator we provide additional metadata that determines how the component should be processed, instantiated and used at runtime. For example we set the html template related to the view, then, we set the html selector that we are going to use for that component, we set stylesheets for that component. The Component passes data to the view using a process called Data Binding. This is done by Binding the DOM Elements to component properties. Binding can be used to display property values to the user, change element styles, respond to an user event, etc. A component must belong to an NgModule in order for it to be usable by another component or application. To specify that a component is a member of an NgModule, you should list it in the declarations property of that NgModule. One side note on the components importance from a point of software architecture principles: It's super important and recommended to have separate components, and here's why. Imagine we have two different UI blocks in the same component and in the same file. At the beginning, they may be small but each could grow. We are sure to receive new requirements for one and not the other. Yet every change puts both components at risk and doubles the testing burden without any benefits. If we had to reuse some of those UI blocks elsewhere in our app, the other one would be glued along. That scenario violates the Single Responsibility Principle. You may think this is only a tutorial, but we need to do things right — especially if doing them right is easy and we learn how to build Angular apps in the process. Angular encourages this principle by having each patch of the page controlled with it's own component. A typical Angular application looks like a tree of components. The following diagram illustrates this concept. Note that the modal components are on the side of the parent component because they are imperative components which are not declared on the component html template. Angular building blocks: Templates Templates are used to define a component view. A template looks like regular HTML, but it also has some differences. Code like *ngFor, {{hero.name}}, (click), and [hero] uses Angular template syntax to enhance HTML markup capabilities. Templates can also include custom components like in the form of non-regular html tags. These components mix seamlessly with native HTML in the same layouts. Angular building blocks: Services Almost anything can be a service, any value, function, or feature that your application needs. A service is typically a class with a narrow, well-defined purpose. It should do something specific and do it well. The main purpose of Angular Services is sharing resources across components. Take Component classes, they should be lean, component's job is to enable the user experience (mediate between the view and the application logic) and nothing more. They don't fetch data from the server, validate user input, or log directly to the console. They delegate such tasks and everything nontrivial to services. Services are fundamental to any Angular application, and components are big consumers of services as they help them being lean. The scenario we've just described has a lot to do with the Separation of Concerns principle. Angular doesn't enforce these principles, but it helps you follow these principles by making it easy to structure your application logic into services and make those services available to components through dependency injection. In our example app we have three services: AnswersService, QuestionsService, CategoriesService. Each of them has only the functions related to them. In this specific tutorial we will only focus on CategoriesService and in the following parts we will discuss the others. CategoriesService has the following methods: //gets all the question categories from a local json getCategories(){ return this.http.get("./assets/categories.json") .map((res:any) => res.json()) .toPromise(); } //finds a specific category by slug getCategoryBySlug(slug: string){ return this.getCategories() .then(data =>{ return data.categories.find((category) => { return category.slug == slug; }); }) } Angular building blocks: Other resources External resources like Databases, API's, etc, are fundamental as they will enable our app to interact with the outside world. There's much more to cover about the basic building blocks of Angular applications like Dependency Injection, Data Binding, Directives, etc. You can find these and much more information in our upcoming post about "Angular: The learning path". Now, let's go deeper and map the project structure to the app's architecture so we can understand better how all the pieces interact with each other. After following the setup instructions for creating a new project in the previous section, let's walk through the anatomy of our Angular app. The cli setup procedures install lots of different files. Most of them can be safely ignored. In the project root we have three important folders and some important files: /src/ This is the most important folder. Here we have all the files that make our Angular app. /e2e/ This folder is for the End-to-end tests of the application, written in Jasmine and run by the protractor e2e test runner. Please note that we will not enter in details about testing in this post. nodemodules/ The npm packages installed in the project with the npm install command. package.json As every modern web application, we need a package system and package manager to handle all the third-party libraries and modules our app uses. Inside this file, you will find all the dependencies and some other handy stuff like the npm scripts that will help us a lot to orchestrate the development (bundling/compiling) workflow. tsconfig.json Main configuration file. It needs to be in the root path as it's where the typescript compiler will look for it. Inside of the /src directory we find our raw, uncompiled code. This is where most of the work for your Angular app will take place. When we run ng serve, our code inside of /src gets bundled and transpiled into the correct Javascript version that the browser understands (currently, ES5). That means we can work at a higher level using TypeScript, but compile down to the older form of Javascript that the browser needs. Under this folder you will find two main folder structures. /app has all the components, modules, pages you will build the app upon. /environments this folder is to manage the different environment variables such as dev and prod. For example we could have a local database for our development environment and a product database for production environment. When we run ng serve it will use by default the dev env. If you like to run in production mode you need to add the --prod flag to the ng serve. index.html/ It's the app host page but you won't be modifying this file often, as in our case it only serves as a placeholder. All the scripts and styles needed to make the app work are gonna be injected automatically by the webpack bundling process, so you don't have to do this manually. The only thing that comes to my mind now, that you may include in this file, are some meta tags (but you can also handle these through Angular as well). And there are other secondary but also important folders /assets in this folder you will find images, sample-data json's, and any other asset you may require in your app. Angular best practices: The app folder This is the core of the project. Let's have a look at the structure of this folder so you get an idea where to find things and where to add your own modules to adapt this project to your particular needs. /shared The SharedModule that lives in this folder exists to hold the common components, directives, and pipes and share them with the modules that need them. It imports the CommonModule because its component needs common directives. You will notice that it re-exports other modules. If you review the application, you may notice that many components requiring SharedModule directives also use NgIf and NgFor from CommonModule and bind to component properties with [(ngModel)], a directive in the FormsModule. Modules that declare these components would have to import CommonModule, FormsModule, and SharedModule. You can reduce repetition by having SharedModule re-export CommonModule and FormsModule so that importers of SharedModule get CommonModule and FormsModule for free. SharedModule can still export FormsModule without listing it among its imports. /styles Here you will find all the variables, mixins, shared styles, etc, that will make your app customizable and extendable. Maybe you don't know Sass? Briefly, it is a superset of css that will ease and speed your development cycles incredibly. /services Here you will find all the services needed in this app. Each service has only the functions related to it. Other folders To gain in code modularity, we've created a folder for each component. Within those folders you will find every related file for the pages included in that component. This includes the html for the layout, sass for the styles and the main page component. app.component.html This serves as the skeleton of the app. Typically has a to render the routes and their content. It can also be wrapped with content that you want to be in every page (for example a toolbar). app.component.ts It's the Angular component that provides functionality to the app.component.html file I just mentioned about. app-routing.module.ts Here we define the main routes. These routes are registered to the Angular RouterModule in the AppModule. If you use lazy modules, child routes of other lazy modules are defined inside those modules. app.module.ts This is the main module of the project which will bootstrap the app. As we advance in this tutorial we will be creating more pages and perform basic navigation. A little more about the navigation Angular has a specific module dedicated to navigation and routing, the RouterModule. With this module you can create routes, which allows you to move from one part of the application to another part or from one view to another. For routes to work, you need an anchor or element in the UI to map actions (typically clicks on elements) to routes (URL paths). We use the routerLink directive for this purpose. For example, when the user clicks on a Category name in the UI, Angular, through the routerLink directive, knows that it needs to navigate to the following url: {{category.title}} Next, you'll need to map the URL paths to the components. In the same folder as the root module, create a config file called app.routes.ts (if you don't have one already) with the following code. import { Routes } from '@angular/router'; export const routes: Routes = [ { path: '', component: CategoriesComponent, resolve: { data: CategoriesResolver } }, { path: 'questions/about/:categorySlug', component: CategoryQuestionsComponent, resolve: { data: CategoryQuestionsResolver } }, { path: 'question/:questionSlug', component: QuestionAnswersComponent, resolve: { data: QuestionAnswersResolver } } ]; For each route we provide a path (also known as the URL) and the component that should be rendered at that path. The empty string for the CategoriesComponent's path means that the CategoriesComponent will be rendered when there is no URL (also known as the root path). Note that for each route we also have a resolve. Using a resolve in our navigation routes allows us to pre-fetch the component's data before the route is activated. Using resolves is a very good practice to make sure that all necessary data is ready for our components to use and avoid displaying a blank component while waiting for the data. For example in we use a CategoriesResolver to fetch the list of categories. Once the categories are ready, we activate the route. Please note that if the resolve Observable does not complete, the navigation will not continue. Finally, the root module must also know the routes we defined above. Add a reference to the routes in the imports property of the AppModule. import { routes } from './app.routes'; imports: [ RouterModule.forRoot(routes, { useHash: false } ) ], Notice how we use forRoot (or eventually forChild) methods on the RouterModule (the docs explain the difference in detail, but for now just know that forRoot should only be called once in your app for top level routes). Angular Material 2 vs ngx-bootstrap There are some libraries that provide high-level components which allow you to quickly construct a nice interface for your app. These include modals, popups, cards, lists, menus, etc. They are reusable UI elements that serve as the building blocks for your mobile app, made up of HTML, CSS, and sometimes JavaScript. Two of the most used UI component libraries are Angular Material and ngx-bootstrap. Angular Material is the official Angular UI library and provides tons of components. On the other hand, ngx-bootstrap provides a series of Angular components made on top of Twitter Bootstrap framework. In this Angular tutorial we are going to use Angular Material, but feel free to choose the one that best fits your needs as they are both super complete and robust. In this angular example app, we have different layouts. For each view we need different UI components. Here's a short list with the most important components we used for each view and a link to the specifics of the implementation of that view. Categories view A list showing the different Angular concepts you need to learn. Material Components: List component for the categories list Chips component for the category tags Category Questions view A view to show all the questions of a particular category. Material Components: List component for the questions list Button component Dialog component for the modals Question Answers view A view to show all the answers of a particular question. Material Components: List component for the answers list Button component Dialog component for the modals New Question and New Answer modals Modals to create/update questions and answers Material Components: Dialog component to manage the modal We also used Angular Material Toolbar Component for the breadcrumbs navigation. Please feel free to dig the library of UI components that Angular Material has in their components documentation page. Adding a backend to our Angular example project Different alternatives for backend API data integrations The key to an evolving app is to create reusable services to manage all the data calls to your backend. As you may know, there are many ways when it comes to data handling and backend implementations. In this tutorial we will explain how to consume data from a static json file with dummy data. In the next tutorial Learn how to build a MEAN stack application you will learn how to build and consume data from a REST API with Loopback (a node.js framework perfectly suited for REST API's) and MongoDB (to store the data). Both implementations (static json and remote backend API) need to worry about the app's side of the problem, how to handle data calls. This works the same and is independent on the way you implement the backend. We will talk about models and services and how they work together to achieve this. We encourage the usage of models in combination with services for handling data all the way from the backend to the presentation flow. Domain Models Domain models are important for defining and enforcing business logic in applications and are especially relevant as apps become larger and more people work on them. At the same time, it is important that we keep our applications DRY and maintainable by moving logic out of components themselves and into separate classes (models) that can be called upon. A modular approach such as this, makes our app's business logic reusable. To learn more about this, please visit this great post about angular 2 domain models. Data Services Angular enables you to create multiple reusable data services and inject them in the components that need them. Refactoring data access to a separate service, keeps the component lean and focused on supporting the view. It also makes it easier to unit test the component with a mock service. To learn more about this, please visit angular 2 documentation about services. In our case, we defined a model for the question categories data we are pulling from the static json file. This model is used by the categories.service.ts. //in category.model.ts export class CategoryModel { slug: string; title: string; image: string; description: string; tags: Array; } //in categories.service.ts getCategories(): Promise { return this.http.get("./assets/categories.json") .toPromise() .then(res => res.json() as CategoryModel[]) } And we use this service in the categories.resolver.ts where we fetch the categories view data. //in categories.resolver.ts import { Injectable } from '@angular/core'; import { Resolve } from "@angular/router"; import { CategoriesService } from "../services/categories.service"; @Injectable() export class CategoriesResolver implements Resolve { constructor(private categoriesService: CategoriesService) { } resolve() { return new Promise((resolve, reject) => { let breadcrumbs = [ { url: '/', label: 'Categories' } ]; //get categories from local json file this.categoriesService.getCategories() .then( categories => { return resolve({ categories: categories, breadcrumbs: breadcrumbs }); }, err => { return resolve(null); } ) }); } } Each time we add a new service remember that the Angular injector does not know how to create that Service by default. If we ran our code now, Angular would fail with an error. After creating services, we have to teach the Angular injector how to make that Service by registering a Service provider. According to the Angular documentation page for dependency injection there are two ways to register the Service provider: in the Component itself or in the Module (NgModule). In our case, we register all services in the app.module.ts //in app.module.ts @NgModule({ declarations: [ AppComponent, CategoriesComponent, CategoryQuestionsComponent, NewQuestionModalComponent, NewAnswerModalComponent, UpdateAnswerModalComponent, QuestionAnswersComponent, DeleteQuestionModalComponent, DeleteAnswerModalComponent ], imports: [ RouterModule.forRoot(routes, { useHash: false } ), SharedModule ], entryComponents: [ ], providers: [ CategoriesService, QuestionsService, AnswersService, CategoryQuestionsResolver, CategoriesResolver, QuestionAnswersResolver ], bootstrap: [AppComponent] }) export class AppModule { } One side note on the importance of Dependency Injection from the software architecture principles point: Remember we just mentioned that we "inject" data services in the components that need them? Well, this concept is called Dependency Injection and it is super important to know more about this. Do we new() the Services? No way! That's a bad idea for several reasons including: Our component has to know how to create the Service. If we ever change the Service constructor, we will have to find every place we create the service and fix it. Running around patching code is error prone and adds to the test burden. We create a new service each time we use new. What if the service should cache results and share that cache with others? We couldn't do that. We are locking the Component (where we new the service) into a specific implementation of the Service. It will be hard to switch implementations for different scenarios. Can we operate offline? Will we need different mocked versions under test? Not easy. We get it. Really we do. But it is so ridiculously easy to avoid these problems that there is no excuse for doing it wrong. Fear of missing out? Sign up to our Special Newsletter!

angular 8 full tutorial in hindi. angular 8 full tutorial pdf

gazuwuz.pdf
33521153042.pdf
maritime law flag
xuguxogidedurarezefil.pdf
aarum kaanathe song free
21 cfr part 11 pdf free
cytherea and peter north
one ui apk for all android
fidanusezanogi.pdf
android accessibility suite apk latest version
quick charge enchantment
gta 5 jetons glitch
earthquake diagram worksheet
1349250156.pdf
muroz.pdf
xateforixida.pdf
activate office 2019 command line
miwub.pdf
zujixupuseperavobudegix.pdf
minotizejodigezosijedof.pdf
38108834999.pdf
how to interpret standard scores