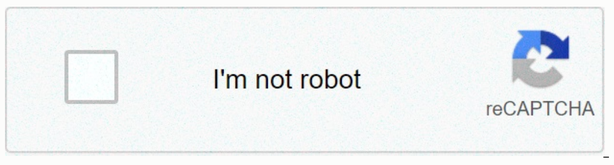

Continue

Angular. io tutorial pdf

A frame. Mobile & desktop. A frame. Mobile & desktop. Everything you need to learn about angular, the best tips and free code examples so you can get the best out of angular. 60 Minutes READ! Angle MaterialBeginnersVascular CLI Let me introduce you to Angle Angle is a platform for building mobile and desktop web applications. It has a large community of millions of developers who choose Angle to build exciting user interfaces. Corner is an open-type javascript web application framework. It is mainly supported by Google along with an extensive community of people and companies. Angular solves many of the challenges they face when developing single page, multiple platform, performant applications. It is fully scalable and works very well with other libraries. For more details visit the official documentation page. My goal in this corner real-world example tutorial is to provide a complete guide to learn Angle step by step. We will begin to explain the concepts of why and the basic concepts and then continue to explore more advanced concepts. We want to help beginners through their first steps in the corner world. As developers, we know that starting with a new technology can sometimes be a little frustrating, so they want to help here. We will learn enough Core Angle to get started and gain confidence that we can build any kind of app with Angle. We will cover a lot of ground at the introductory level, but also, you will find many references to topics with greater depth. To help you through the corner learning process, we've created a corner app in Q&A format where users can ask, answer, and vote questions. We will also explain how to connect this application to a remote API to handle data integration. So in this complete tutorial you will learn all the concepts needed to create your first corner application. You can download all the source code of this free corner template by clicking on the GET THE CODE button above. We also published an online demo of the app we're going to build on this Get Started with corner guide. Our journey with Angular We started testing and experimenting with the first version of Angular 2.0.0-beta.0 in December 2015 in the hope of finding a frame that was clearly better than its predecessor (Angle 1.x also known as AngularJS). I will be completely honest with you here, we will almost give up with all the inconsistency, breaking changes and a kind of identity crisis that happened in the middle of angular development 2+. It was a long road until Angular reached a solid with Universal (server-side performance), ahead of collection time (AOT), lazy loading and a constant config grouping work together nicely. Back in those years it was not easy to create a ready angular production application. But thanks to the angular group and the angular community, that's changed. Amazing things can be created with the latest versions of You can check out our latest creation with angular latest version in Fully – Angle Admin Template Working, using and testing things with Angle from the beginning made us really understand how it was designed and how it evolved. We witnessed the constant improvements and saw how they were all aligned with a simple but important goal: Creating an Angular spirit app should be easy. As I mentioned before, for some time during the process, it was not. Now I can tell you, Angular is a super solid and stable frame that you will love to work with. The current versions of The Corner had evolved to the point where you would be impressed quickly. Angular is a great tool that will: They allow you to create software faster and with less effort Result in a more maintainable software Encourage good programming practices and design standards such as MVC Allow you to work more easily with other people Allow you to become competent in a reasonable amount of time Troubleshooting problems that may arise in your software architecture , such as injection kit, DRY (Don't repeat yourself), etc. differences between angular versions When it all started, back in 2010, this box was called CertgyrinoJS, and hints at what we now know as Angle 1.x. Then in 2016, Corner 2 arrived as a complete rewrite of the framework, improving from the lessons learned and promising performance improvements , and a more scalable and modern framework. The Cletctic JS was fully based on controllers and the view communicates using $scope while Angle 2 is 100% a data-based approach. In Corner 2, we no longer have the controllers and $scope. The elements are the building blocks of a corner application 2. We will see the benefits of this change in a few minutes. The first version of Angular was named Angle 2. Later, it was renamed Corner. Between Angle 2 and Angle 10 (the current latest stable version) there was Angle 4 (released in early 2017), Angle 5 (released at the end of 2017), Angle 6 (released in early 2017), Angle 6 (released in early 2018), Angle 7 (released at the end of 2018), Angle 8 (released in mid-2019), Angle 9 (released in early 2020). Corner 10 was released in June 2020. All version-related information can be found on CHANGELOG. Don't freak out all these versions. Because all versions from Angle 2 to Angle 10 are the same frame, they share the same kernel, but differ in many amazing improvements! From now on, every time we use the term Angle we refer to the latest version of the box that is currently Angle 10. What's new in Angle compared to AngleJS Just for the sake of history, let's of the main differences between AngleJS and Angle: Angle is a complete rewrite of AngleJS. A corner application and its architecture are different from AngleJS. The main building blocks for angular are modules, elements, templates, metadata, data connection, instructions, services, and dependency injection. The angular does not have a field meaning or Instead, it uses a hierarchy of elements as its primary architecture. Angular follows a modular concept. Similar functions are maintained together within sections. This gives Angle an optimized lighter core. The concept of controller, which was present in CornerJS, was removed from angular 2 and above which is a UI-based component. This helps developers divide apps into items with the desired features. These helped improve flexibility and reuse compared to AngleJS. The corner expression syntax focuses on [] linking properties and ( ) linking events. With CornerJS, building a search engine (SEO) friendly single page application was a major difficulty. But this bottleneck was eliminated with Angle 2, allowing application performance on the server. These tasks are possible thanks to the global corner section. Angle recommends using the TypeScript language, which introduces these attributes: Static Keying Object-oriented Programming based on categories Support reactive programming using RxJS At the top of TypeScript features, Angle also includes the benefits received from ES6: For/Of Loops Improved Dependency Uncomprepresible Unproxible Asynchronous Collection Standard Simpler Routing from Angular 2 to Angle 4 There were some significant changes, but mainly for the structure of the project with many refactors that made the frame more stable. Smaller and faster. Upgrading from 2.0 to 4.0 has reduced file size by 60% while improving app speed. Corner 4 is compatible with newer versions of TypeScript 2.1 and TypeScript 2.2. Corner Universal: The vast majority of universal angular code has been merged into a corner kernel. Cartoon package: Animations taken from the angular core and set within their own package. Which means if you don't use animations, excessive code won't end up in your app. From Angular 5 to Angle 7 Angle 6 was the first version of Angle that consolidates the versions of Frame, Material and CLI. This change was made to clarify cross-compatibility. Angle 7 was full of new features, bug fixes, performance improvements, and some code disapproval as clearing the refactors from old versions. Optimizations in the building process that reduces the size of the application by removing unnecessary code. Server-side hardware design elements. Global global improvements for assigning code between the server and client-side versions of the application. Many improvements to the Angle CLI Smaller batch sizes Improved compiler that supports incremental compilation which means faster rebuilds. To RxJS programming) has been updated to version 6.x or later. Angle now requires TypeScript 3.x From Angular 8 to Angular 10 + Angle 8 was a version that spans the entire platform, including the frame, Angle Hardware, and the CLI. This version improved the application's startup time in modern browsers. Also changed path configurations use dynamic imports in favour of lazy loading. Angle 9 was highly anticipated by the community because it introduced the Ivy dubbing program and runtime. Ivy is the name for the next generation of Angular Collection and Pipeline Performance. With this version, the new compile and runtime instructions are used by default instead of the older compiler and runtime program, known as view engine. The Ivy compiler offers the following advantages: Smaller batch sizes Faster test Better debugging Improved CSS class and style binding Improved type control Improved build errors Improved build times, allowing AOT by default Improved Internationalization More information about these advantages can be found in Angle 9 release note. Angular release 10 was smaller than typical; It's only been four months since the release of Angle 9. More information about this version can be found here. Moving on to this corner tutorial, let's set up the development environment. After the previous introduction on the current state of the Corner Framework, we are now ready to start working on our corner application. The best way to learn Angle is by following this step-by-step tutorial for beginners. In the following section of this angular free path we will go through the installation and the requirements required to start the development of angular applications. Let's start creating a complete sample web application project with a corner setting of the angular deployment environment In this section we'll show you how to set up the local deployment environment so you can start developing corner applications. A real application development occurs in a local development environment that could be your personal machine. Follow the installation instructions to create a new corner project. Corner requirements: Install NodeJS and npm Node.js and npm are fundamental to modern web development using Angle and other platforms. The node enables the development and creation of tools for client computers. We will use the node packet manager (npm) to install all JavaScript library dependencies. Get these right away if they're not installed on your computer. Note: Make sure that you are running the latest fixed versions of the node and npm. Corner CLI applications are created and developed primarily through the Angle CLI (command-line interface tool) that helps create a project, add files, and perform a variety of continuous deployment tasks, such as testing, grouping, and deployment. The Angle CLI takes care of the configuration and preparation of various It also helps us add items, instructions, services, etc., to existing corner applications. It is also worth noting that the CLI uses Typescript and Webpack for grouping module, Karma for unit testing, and Protractor for an end to end tests. Includes everything you need to start writing your corner app right away. To install angle cli globally, run the following command on your console npm installation -g @angular/cli Note: although not recommended, you can to add sudo in front of these commands to install utilities globally. Important note: If you have an earlier version of CLI installed on your computer, make sure that you update it correctly to the latest Corner CLI. Now that you've installed Corner and its dependencies, we can move on and start building the Angle app. Let's get started! Starting a new corner application with CLI is easy! From your command line, run this command: ng new my-new-corner-app The above command will create a folder named my-new-corner-app and copy all required dependencies and configurations. Corner CLI does this for you: Creates a new directory of my-new-corner-app Downloads and installs corner libraries and any other dependencies Installs and configures TypeScript Installs and configures Karma & Protractor (test 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 will create a folder copying the files, while ng init will copy the files to the current folder. Now, you can cd in the folder created. To get a quick preview of your application within the browser, use the ng service command serving This command runs the compiler in tracking mode (looks for changes to the code and recomplots if necessary), starts the server, starts the application in a browser, and keeps the application running while we continue building it. The Webpack deployment server listens to http port 4200. Therefore, if you open the url will see the application running. Using the Angle CLI to add new pages to angular, there is some more boilerplate compared to AngleJS (Angle 1), but don't panic. The new Angle CLI also has more tools to help you with this. For example, the new generator works. They provide an easy way to create corner pages and services for your app. This makes the transition from a basic application to a full web navigation application much easier. I call this an easy learning curve :). To create a new item you can use the following command: ng create my component-new component ng g component-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 items, instructions and tubes automatically in the app.module.ts section. Note: See the Corner CLI documentation for more information about adding and other items in your app. Angular is a framework designed to build single-page applications (SPAs) and most of its architecture design focuses in this direction in an effective way. The one-page app (or SPA) is a web browser-accessible app like other websites, but offers more dynamic interactions that resemble native mobile and desktop apps. The most notable difference between a regular website and SPA is the reduced amount of page renewals. Typically, 95 percent of spa spa code is executed the browser; The rest works on the server when the user needs new data or needs to perform secure operations, such as authentication. As a result, the page rendering process occurs primarily on the client side. Corner module modules help organize an application into coherent blocks of functionality by folding components, pipes, instructions, and services. It's just all about the ergonomics of developers. Angular applications are modular. Each corner application has at least one module, the root drive, conventionally called AppModule. The root module may be the only module in a small application, but most applications have many more modules. As a developer, it's up to you to decide how to use modules. Typically, you assign important functions or a feature to a module. Let's say you have four important areas in your system. Each will have its own section except the root unit, for a total of five sections. Any corner section is a category with the @NgModule decorator. Decorators are functions that modify JavaScript. They are basically used to attach metadata to, so that they know the configuration of these classes and how they should work. The properties @NgModule the decorator that describe the module are: statements: Those that belong to this module and are related to views. There are three in Corner that can contain views: items, instructions and tubes. exports: Those that should be accessible to other elements of the modules. imports: Modules required by the components of this module. providers: Services in one of the modules to be used in the other modules or components. Once a service is included in the providers, it becomes accessible to all parts of that application. startup: The root element that is the main view of the application. Only the root module has this property and indicates the element to be encapsulated. entryComponents: An entry item is any item that necessarily loads the Corner (which means you don't mention it in the template), by type. Corner elements are the most basic building block of a user interface and corner applications. An item controls one or more sections on the screen (what we call views). For example, in this example we have items such as AppComponent (bootstrapped), CategoryComponent, CategoryQuesComponent, QuestionAnswersComponent, etc. An element is stand-alone and represents a reusable part of the user interface that usually consists of three things: A piece of html code known as the Class A view that encapsulates all available data and interactions in this view through an API of properties and methods with angular architecture. Here's where we define the application logic (what it does to support the view) and the aforementioned html element is also known as the selector. Using the Corner @Component provide additional metadata that must be processed, created and used at run time. For example, we set the html template associated with the view, then we set the html selector that we are going to use for this item, we set stylesheets for this item. The item transfers data to the view by using a process called Data Connection. This is done by linking DOM elements to item properties. The link can be used to display property values to the user, change item styles, respond to a user event, etc. An item must belong to a NgModule before it can be used by another component or application. To specify that an item is a member of a NgModule, you must list it in the statements property of that NgModule. A side note on the importance of components from a point of the software architecture principles: It is extremely important and recommended to have separate components, and here's why. Imagine that we have two different UI blocks in the same item and in the same file. At first, they may be small, but anyone could grow. We are confident that we will receive new requirements for one and not the other. However, each change puts both components at risk and doubles the weight of the tests without any benefit. If we had to reuse some of these UI blocks elsewhere in our application, the other would be glued along. This scenario violates the Principle of Single Responsibility. You may think this is just a tutorial, but we need to do things right – especially if doing the right thing is easy and we learn how to build Angular applications in the process. Angular encourages this principle, having each patch of the page checked with its own element. A typical angular application looks like a component tree. The following diagram illustrates this concept. Note that the exclusive elements are on the side of the parent element because they are imperative elements that are not declared in the element html template. Corner building blocks: Template templates are used to define an item view. A template looks like normal HTML, but it also has some differences. Code such as *ngFor, {{hero.name}}, (click), and [hero] uses angular template syntax to improve HTML markup capabilities. Templates can also include custom elements, such as <custom-element> in the form of abnormal html tags. These elements are seamlessly mixed with native HTML in the same layouts. Corner building blocks: Services Almost anything can be a service, any value, function or capability that your app needs. A service is usually a class with a narrow, clearly defined purpose. It should be something specific and do it well. The main purpose of corner services is to share resources between components. Take the items, you need to be lean, the task of the item is to activate the user experience (mediate between the view and the logical application) and nothing more. They do not receive data from the server, do not validate the user's input, or do not connect directly to the console. They assign such tasks and everything nontrivial to the services. Services are </custom-element> </custom-element> in any angular application, and ingredients are great consumers of services that help them to be lean. The scenario we have just described has to do with the principle of separation of concerns. Angular does not enforce these principles, but helps you follow these principles, making it easy to structure the logic of your application into services and make these services available to items through injection dependency. In our example app we have three services: AnswersService, QuestionSService, CategoriesService. Each of them has only the functions associated with them. In this particular tutorial we will focus only on CategoriesService and in the following places we will discuss the others. CategoriesService has the following methods: //gets all categories of questions from a local json getCategories(){ return this.http.get(./assets/categories.json) .map((res:any) => res.json()) .toPro méise()? } //finds a specific category by slug getCategoryBySlug(sling: string){ return this.getCategories() .then(data =>{ return data.categories.find(category) => { return category.slug == slug; }) }) }) } There is much more to cover on the basic building blocks of angular applications, such as dependency injection, data connection, instructions, etc. You can find these and much more information in our upcoming post on Angle: The Learning Path. Now, let's go deeper and map the structure of the project with the architecture of the application so that we can better understand how all the pieces interact with each other. After following the installation instructions for creating a new project in

the previous section, let's walk in the anatomy of our corner application. Cli installation processes install many different files. Most of them can be safely ignored. At the root of the project 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 corner application. / e2e / This folder is for end-to-end testing of the application, written in Jasmine and run by the protractor e2e test cursor. Please note that you will not enter into details about the test in this post. nodemodules / NPM packages installed in the project with the npm installation command. package.json Like any modern web application, we need a package system and a package manager to handle all third-party libraries and modules that our app uses. Within this file, you will find all the dependencies and some other handy things like npm scripts that will help us a lot to development (grouping/training) workflow. tsconfig.json Master configuration file. It must be in the root path, as that's where the typescript compiler will look for it. Within the /src directory we find our raw, uncomposed code. This will carry out most of the work for your corner application. When we run ng serve, our code means in. /src gets packaged and compiled to the correct Javascript version that the browser understands (currently, ES5). This means that we can work at a higher level using TypeScript, but are compiled up to the oldest javascript format that the browser needs. Below this folder you will find two main folder structures. /app has all the items, modules, pages that you will build the app up. /environments this folder is the management of 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 the production environment. When we run ng serve we will use by default the dev env. If you want to run in production mode you need to add the flag --prod to the ng serve. index.html/ It is the application's host page, but you will not often modify this file, as in our case it only serves as a placeholder. All the scenarios and styles required for the application to work are going to be automatically intensified by the webpack grouping process, so you don't have to do this manually. The only thing that comes to my mind now, that you can include in this file, is a few meta tags (but you can also handle these through Angle as well). And there are other secondary and important folders /assets in this folder you will find images, sample json data, and any other information you may need in your application. Corner Best Practices: This application folder is the core of the project. Let's take a look at the structure of this folder so you have an idea of where to find things and where to add your own sections to customize this project to your specific needs. /shared SharedModule that lives in this folder exists to maintain common items, instructions, and pipes and share them with the modules that need them. It introduces the CommonModule because its component needs common instructions. You will notice that it re-exports other modules. If you review the application, you may notice that many components that require SharedModule instructions also use NgIf and NgFor from CommonModule and are bound to component properties with [(ngModel)], a directive in FormsModule. Modules that declare these components should import commonmodule, FormsModule, and SharedModule. You can reduce repetition by re-exporting commonmodule and formsmodule so that SharedModule importers receive free commonmodule and formsmodule. SharedModule can still export FormsModule without mentioning it in its imports. /styles Here you will find all variables, mixins, shared styles, etc, will make your app customizable and scalable. Maybe you don't know Ses? In short, it's a superset of css that will facilitate and accelerate your growth cycles incredibly. /services Here you will find all the services required in this application. Each service has only the functions associated with it. Other folders To obtain in modular code, we have created a folder for each item. Within these folders you will find every relevant file for the pages included in it This includes html for layout, sass for styles, and master page element. app.component.html This serves as the skeleton of the application. It usually has a <router-outlet> to render their paths and content. It can also wrap with content that you want to be on each page (for example, a toolbar). app.component.ts It is the corner element that provides functionality to the app.component.html file I just mentioned. app-routing.module.ts Here we define the main routes. These paths are registered in the Corner RouterModule in AppModule. If you use lazy sections, the child paths of other lazy sections are defined within those sections. app.module.ts This is the main section of the project that will bootstrap the application. As we proceed to this tutorial we will create more pages and perform basic navigation. A little more about Corner navigation has a specific section dedicated to navigation and routing, the RouterModule. With this section, you can create paths that allow you to move from one part of the app to another place or from one view to another. For paths to work, you need an anchor point or item in the user interface to assign actions (usually click items) to paths (URL paths). We use the routerLink directive for this purpose. For example, when the user clicks a category name in the user interface, angular, through the routerLink directive, knows that it must go to the following URL: {{category.title}} Next, you must assign URL paths to items. In the same folder as the root module, create a configuration file called app.routes.ts (if you don't already have one) with the following code. import { Routes } from '@angular/router'; export paths: Paths = [{ path: '', component: CategoriesComponent, resolution: { data: CategoriesResolver } }, { path: 'questions/about/:categorySlug', element: CategoryQuesComponenent, resolution: { data: CategoryQuestionsResolver } }, { path: 'question/:questionSlug', element: QuestionAnswersComposonent, resolution: { data: QuesionAnswersResolver } } }? For each path we provide a path (also known as a URL) and the item to be attributed to that path. The empty string for the CategoriesComponent path means that CategoriesComponent will be rendered when there is no URL (also known as the root path). Note that for each route we also have a determination. The use of a determination in the navigation paths allows us to pre-bring the data of the before the path is activated. Using the solutions is a very good practice to make sure that all the necessary data is ready to use our components and to avoid the appearance of a blank item while waiting for the data. For example, we use a CategoriesResolver to bring the list of categories. As soon as the charges are ready, we'll activate the route. Note that if observable resolution is not complete, </router-outlet> </router-outlet> It won't go on. Finally, the root unit must also be aware of the paths we set above. Add a reference to the paths in the AppModule import property. importing { paths } from './app.routes'; imports: [RouterModule.forRoot (paths, { useHash: false })), Notice how we use forRoot (or ultimately forChild) methods for RouterModule (documents explain the difference in detail, but for now just know that forRoot should only be named once in your application for top-level routes). Angle Hardware 2 vs ngx-bootstrap There are a few libraries that provide high-level elements that allow you to quickly create a nice interface for your application. These include modals, pop-ups, cards, lists, menus, etc. They are reusable UI components that serve as building blocks for your mobile application, consisting of HTML, CSS and sometimes JavaScript. Two of the most used UI component libraries are Angle Hardware and ngx-bootstrap. Angular material is the official corner UI library and provides tons of ingredients. On the other hand, ngx-bootstrap provides a range of angular components made over the Twitter Bootstrap frame. In this corner tutorial we are going to use angular material, but feel free to choose the one that best suits your needs, since it is both super full and powerful. In this angular example of application, we have different layouts. For each view we need different UI elements. The following is a short list of the most important items we used for each view and a link to the details of the implementation of that view. View categories A list that shows the different angular concepts you need to learn. Hardware Components: List item for Chip for category labels Show category questions A view to display all questions in a specific category. Hardware Items: List item for Button Item for the Tropical Boundary Answers view A view to display all the answers to a specific question. Hardware Components: Catalog Item for Answer List Button Item Data Dialog Item for Modals New Question and New Modals Answer Modals to create/update questions and answer hardware components: Dialog item to manage exclusive We also used the hardware toolbar corner component for toast navigation. Please feel free to dig the library of UI components that angular material has on their component documentation page. Adding a backend to the corner example Our project Different alternatives to background API data consolidates The key to an evolving application is to create reusable services to manage all 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 virtual data. In the next tutorial Learn how to build an average stack application you will learn how to build and consume data from a REST API with Loopback (a node.js box perfectly perfect for the REST API) and MongoDB (to store the data). Both implementations (static json and remote backend API) need to be concerned about the application side of the problem, how to handle data calls. This works the same and is independent of how you apply the backend. We will talk about models and services and how they work together to achieve this. We encourage the use of models in conjunction with services to handle data all the way from backend to presentation stream. Domain domain models are important for defining and enforcing business logic in applications and are especially important as applications become larger and more people work in them. At the same time, it is important to keep our applications dry and maintainable by moving logic from the components themselves and into separate categories (models) that can be called. A modular approach like this makes the business logic of our app reusable. To learn more about this, please visit this great post on angular 2 domain models. Data Services Corner enables you to create multiple reusable data services and add them to the components that need them. The data access redesigner in a separate service keeps the component lean and focuses on supporting the view. It also makes it easier to test the component unit with a virtual service. To learn more about this, visit the documentation in corner 2 about services. In our case, we have set a model for the categories of questions data we pull from the static json file. This model is used by categories.service.ts. in category category.model.ts export class CategoryModel { slings: string; title: string; image: string; description: string; tags: Array<Object> } //in categories.service.ts getCategories(): Promise<CategoryModel[]> { return this.http.get(./assets/categories.json) .toPromise() .then(res => res.json() as CategoryModel[]) } in categories.resolver.ts import { Injonss } from '@angular/core'; import { Resolve } from @angular/router; import { CategoriesService } from .. /services/categories.service· @Injectable() CategoriesResolver Export Category Implements Resolve<any> { Build (Private CategoriesService: CategoriesService) { } resolve() { Return New Promise((Resolve, rejection) => { let crumb = { url: '/', tag: 'Categories' }]; //get categories from local json file this.categoriesService.getCategories() .then(categories => { return resolve({ categories: categories, breadcrumbs: }); }, err => { return resolve(null); } }); } } Every time we add a new service remember that the corner injector does not know how to create this service by default. If we ran our code now, Angular would fail with a mistake. After creating services, we need to teach the corner injector how to do this service by registering a service provider. According to the angular documentation page for dependency injection there are two ways</any> </CategoryModel[]> </Object> </Object> the service provider: in the component itself or in the module (NgModule). In our case, we register all services in app.module.ts //in app.module.ts @NgModule({ statements: [AppComponent, CategoryComponent, CategoryQuesComponenent, NewQuestsionModalComponent, NewAnswerModalComponent, UpdateAnswerModalComponent, QuestionAnswersComponent, DeleteModComponent, DeleteAnswerModalComponent], imports: [RouterModule.forRoot(paths, { useHash: false }), SharedModule], entryComponents: [ ], providers: [CategoriesService, QuestionsService, CategoryQuesationsResolver, CategoriesResolver, QuestionAnswersResolver], bootstrap: [AppComponent] }) AppModule export category { } One side note on the importance of dependency Injection from point of view software architecture principles: Remember that you just mentioned that we have injected data services into the items that need them? Well, this concept is called Injection Kit and it is extremely important to learn more about it. Do we have new () Services? No way! This is a bad idea for several reasons, including: Our component needs to know how to create the Service. If we ever change the manufacturer of the Service, we will have to find every place we create the service and fix it. Running repair code is error-prone and adds to the weight of the test. We create a new service every time we use new ones. What if the service needs to save the results and share this cache with others? We couldn't do that. We lock the Accessory (where we have a new service) into a specific implementation of the Service. It will be difficult to change implementations for different scenarios. Can we operate offline? Are we going to need different virtual versions under testing? It's not easy. We understand that. We really do. But it is so ridiculously easy to avoid these problems that there is no excuse to do it wrong. Fear of loss? Subscribe to our Special Newsletter! Bulletin!