

**blog** (<https://ionicframework.com/blog/>)

# Full-Stack TypeScript with Ionic, Angular, and NestJS Part 1 (<https://ionicframework.com/blog/full-stack-typescript-with-ionic-angular-and-nestjs-part-1/>)



By Ely Lucas (<https://twitter.com/elylucas>) on April 11, 2019 in **ENGINEERING** (/BLOG/CATEGORY,



1/3

## Introduction

Welcome to the first post in a new series we're kicking off all about building a full stack TypeScript app using Ionic, Angular and NestJS.

TypeScript is a powerful language that is a superset of JavaScript, with some additional features added to help build out large scale applications. One of my favorite features of TypeScript is its optional static type checking. I was a longtime static language dev (C#) before coming to the front-end, and even though I enjoyed the dynamic nature of JavaScript, I would often run into runtime errors and bugs in my client-side code because there was no type checking to help ensure the system I wrote was correct. TypeScript helps fix this by providing a way to add types to your variables and objects that are evaluated at dev time, but removed and turned into plain JavaScript when running in the browser.

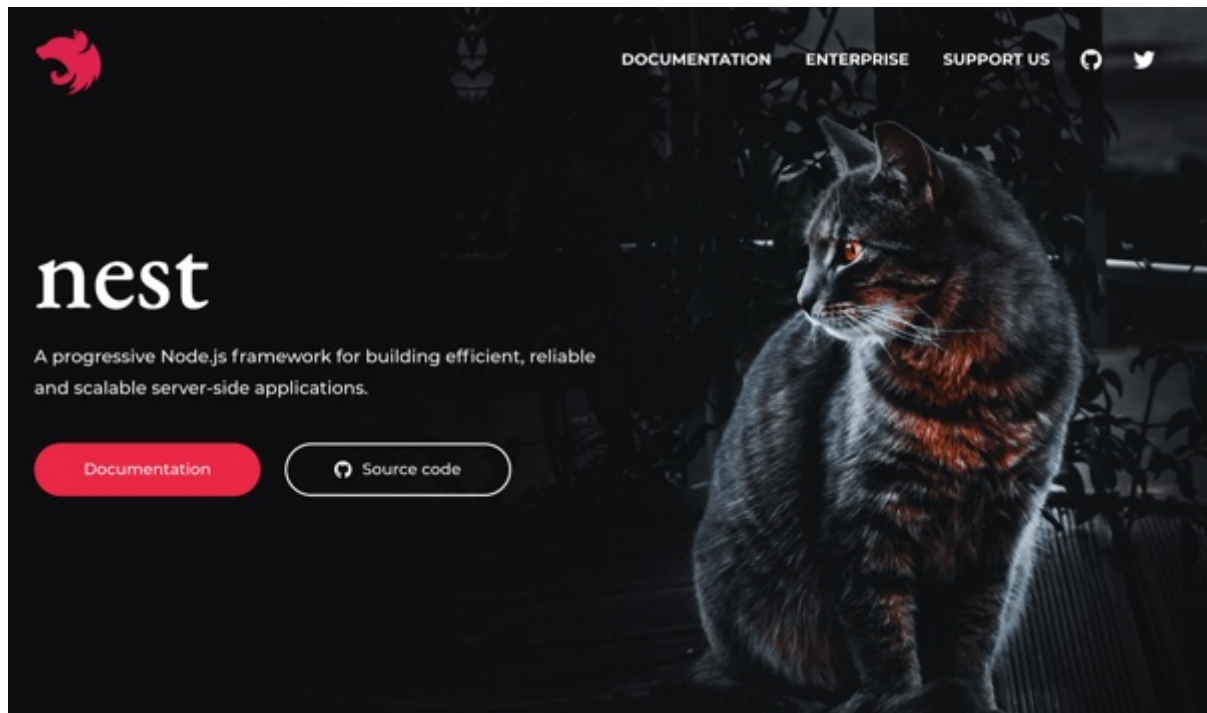
Beyond type checking, TypeScript also helps you speed up your development by providing code completion and refactoring, as well as letting you use modern JavaScript language features that might not be available in all browsers yet.

Ionic and Angular developers have adopted TypeScript as their primary language for their client-side development for a few years now. However, TypeScript can run anywhere JavaScript can, which includes the server!

Front-end devs often turn to Node for their backend needs. The reasons are numerous, as Node provides a great runtime that works with the language with which they are most familiar (JavaScript). A couple of years ago, I was starting a new Node project and wanted to use TypeScript in it because I was curious if I could use TypeScript's decorators to help define routes and place them directly on the controllers. Additionally, I liked this pattern from ASP.Net MVC and figured it would be possible with TypeScript.

It didn't take much searching until I found a little, somewhat unknown, project called NestJS. Nest billed itself as a Node framework built with the power of TypeScript. I was immediately intrigued!

# Meet NestJS



NestJS (<https://nestjs.com/>) (just Nest from here on out), is a Node framework meant to build server-side applications. Not only is it a framework, but it is also a platform to meet many backend application needs, like writing APIs, building microservices, or doing real-time communications through web sockets. For us Ionic devs, it is a great solution to write our backends with because it fits in so well with the rest of our ecosystem.

Nest is also heavily influenced by Angular, and, as an Angular dev, you will immediately find its concepts familiar. The creators of Nest strived to make the learning curve as small as possible, while still taking advantage of many higher level concepts such as modules, controllers, and dependency injection.

However, this doesn't mean that Nest is only for Angular devs. On its own, Nest is a powerful framework that anyone looking to build server-side applications on Node should consider. Moreover, Nest is also similar to other

MVC (model, view, controller) frameworks out there, like ASP.Net MVC and Spring. Developers coming from other enterprise frameworks will find Nest familiar.

Since you use TypeScript (or JavaScript) to write a Nest application, one of the most intriguing features of Nest is the ability to use the same language on the client that you use on the server. This helps speed up development as your developers don't need to context switch between the idiosyncrasies different languages can have. The unified language also opens doors for some potential code reuse between your front-end and back-end.

Are you interested in taking Nest for a spin and building out a backend for you Ionic apps? Great, let's get started!

## Our App

A typical full-stack tutorial has you build out a todo app. Todo apps are way too lame to waste on great tech such as Ionic, Angular, and Nest. So, in this tutorial series, we are going to build an app for space rangers 🚀. Our app will provide a list of missions a space ranger can go on, and have them enter their own missions. Once finished, they can mark their mission as complete.

You might be thinking this sounds like another todo app, but it's not. It's way cooler. It has space kitties. We will call it GoSpaceRanger.

## Getting Started

I'm going to assume you are all set to get started with Ionic and Angular, but if not, head over to our getting started guide

(<https://ionicframework.com/docs/installation/cli>) to get those setup.

First, create a new directory called `go-space-ranger` wherever you like to do your developer things and `cd` into it. We will create two applications (an Ionic and a Nest app) via CLIs in this directory.

Nest also comes with a CLI that is relatively similar to that of Ionic and Angular. Install it via NPM:

```
npm i -g @nestjs/cli
```

Use the Nest CLI to kick off a new app:

```
nest new gsr-server
```

When asked, select NPM as your package manager.

Go into the `gar-server` directory Nest created, and start up the development server by running:

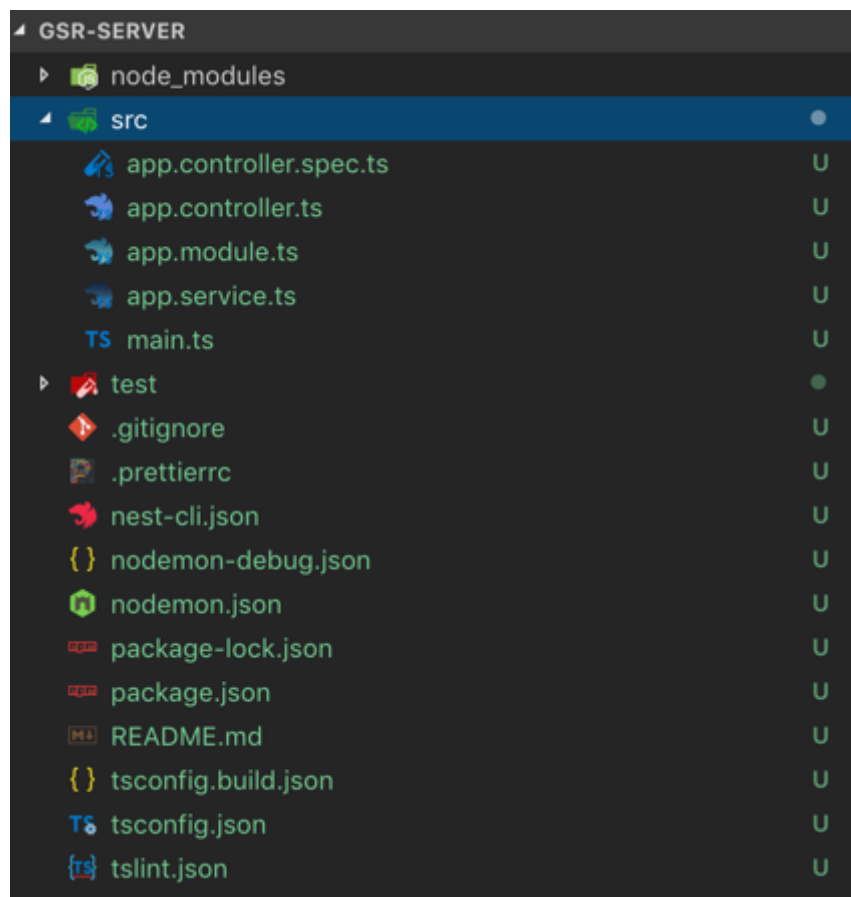
```
npm run start:dev
```

This command kicks off a development server and loads your Nest app on a default port of 3000. Open up your browser to `http://localhost:3000`, and you should see:



Open up the `gsr-server` directory in your editor of choice. I'll be showing VS Code in this tutorial and will talk about some tips on how to use it, but feel free to use what you are comfortable with. Let's take a brief tour of the default Nest project.

## The Nest Project Structure



There is nothing too crazy going on here. By default, Nest gives you a few goodies out of the box, that you would typically have to set up on your own. For code formatting and code style, Nest includes a Prettier and TSLint setup. Nodemon is included to recompile your app on each code change. Typescript is all set up as well. The config files for all these tools have sensible defaults, but feel free to modify them to fit your style.

The `src` folder contains all your app code.

The `main.ts` is the file that bootstraps your application. By default, Nest uses Express as the web framework to serve the HTTP requests. If you needed to do any additional configuration or add any Express middleware, you would do it here.

The `app.module.ts` file is familiar to Angular devs. The module serves the same concept here in Nest that it does in Angular. Modules provide logical separation in your app and give the configuration needed for each section.

When it comes down to it, Nest implements an MVC (model, view, controller) design pattern. In MVC, controllers listen in for incoming requests, call into services for data access and business logic, then serve out the models from the services back to the client.

The `app.controller.ts` file is a simple implementation of a controller:

```
import { Controller, Get } from '@nestjs/common';
import { AppService } from './app.service';

@Controller()
export class AppController {
  constructor(private readonly appService: AppService) {}

  @Get()
  getHello(): string {
    return this.appService.getHello();
  }
}
```

Controllers are just pure ES6 classes that have a `@Controller` decorator on them. The decorator takes a string param that specifies the route the controller should listen for requests at. Since it is omitted here, this `AppController` listens to the root path `'/'`.

Nest also uses decorators on methods in a controller to designate which HTTP verbs (Get, Post, Put, Delete, etc...) a method should respond to. In our `AppController`, when a GET request comes in, Nest responds by invoking the `getHello` method.

Our controller has a service named `appService` injected into it. A Nest service is another ES6 class that is registered with Nest's dependency injection framework as a provider.

```
import { Injectable } from '@nestjs/common';

@Injectable()
export class AppService {
  getHello(): string {
    return 'Hello World!';
  }
}
```



This simple class has a single method that returns a string. If your server is still running (via NPM), go ahead and modify the string, then go back to your browser and refresh. Since the server is running in dev mode, Nodemon automatically recompiles and restarts the app every time a file changes.

Now that we know the basic structure of a Nest app, let's create some of our own services and controllers.

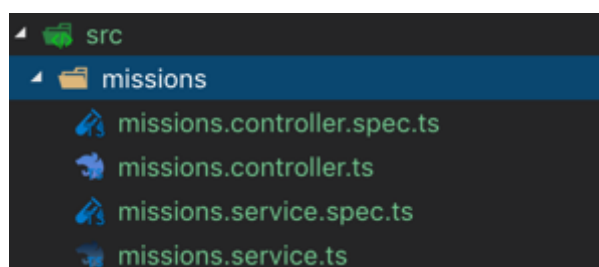
## Your First Nest Service and Controller

Our GoSpaceRanger app returns back a list of “missions” that our space rangers can respond to. So, to kick off our new app, let's create a model to represent a mission. Create a new folder in `src` named `models`, then create a new `mission.model.ts` and paste in the following code:

```
export class Mission {  
  id?: number;  
  title: string;  
  reward: number;  
  active: boolean;  
}
```

Next, we can use the Nest CLI to generate a service and controller for us:

```
nest g service missions  
nest g controller missions
```



These commands will create a controller and a service file, along with spec files for testing.

For now, the Missions Service returns a hard-coded list of missions. Update the class in the `mission.service.ts` file to the following:

```
@Injectable()
export class MissionsService {
  missions: Mission[] = [
    {
      id: 1,
      title: 'Rescue cat stuck in asteroid',
      reward: 500,
      active: true,
    },
    {
      id: 2,
      title: 'Escort Royal Fleet',
      reward: 5000,
      active: true,
    },
    {
      id: 3,
      title: 'Pirates attacking the station',
      reward: 2500,
      active: false,
    },
  ];

  async getMissions(): Promise<Mission[]> {
    return this.missions;
  }
}
```

Here, we have a few missions defined in an array and a `getMissions` method, which simply returns the missions array.

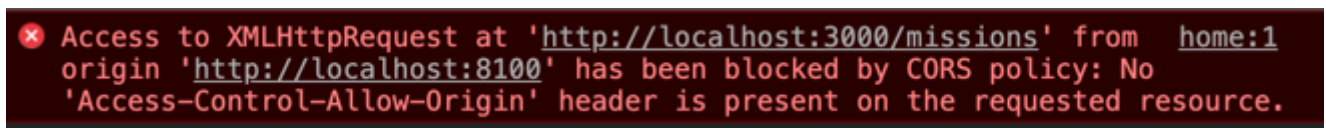
Next, update the `missions.controller.ts` controller to include the method that handles the GET request:

```
@Controller('missions')
export class MissionsController {
  constructor(private missionsService: MissionsService) {}

  @Get()
  getMissions() {
    return this.missionsService.getMissions();
  }
}
```

Now, if you back to your browser and visit <http://localhost:3000/missions>, you should see the list of missions returned in JSON format.

However, trying to access this data from a web app (like what we will build in a moment), you would get the following error in the browser:



This error is because the browsers same origin policy is kicking in and denying you to make an XHR request to a domain that the web page did not originate from. To get around this, we enable Cross-Origin Resource Sharing (CORS). Fortunately, Nest makes this easy, and all we have to do is open up the `main.ts` file and add `app.enableCors()` right after `app` is defined like so:

```
async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.enableCors();
  await app.listen(3000);
}
bootstrap();
```

With that in place, we are ready to move forward making requests from our upcoming Ionic app. We will begin to build that out next.

# Create GoSpaceRanger Ionic App

In your main project folder (the parent folder where we created the `gsr-server` Nest project), run the following command to create a new Ionic project:

```
ionic start gsr-client sidemenu
```

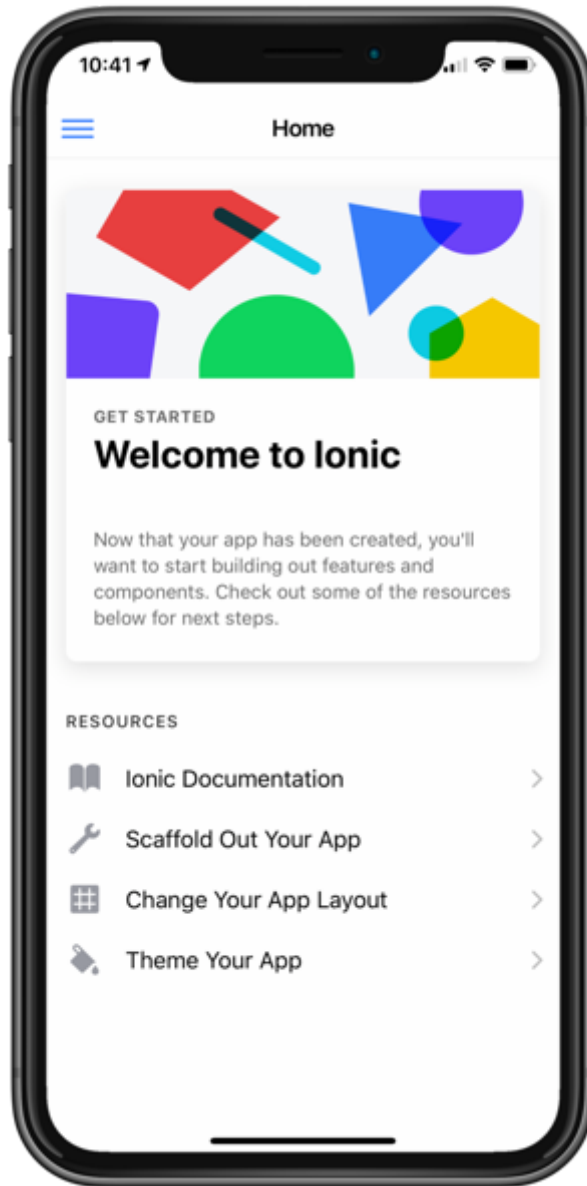
When asked if you want to install the AppFlow SDK, select no.

This command creates an Ionic project using the side menu template. Our app consists of a simple list page and a detail page, so starting with the side menu template gives us what we need initially, and we will use the side menu itself in the future.

Start up the development server:

```
ionic serve
```

After building the app, your browser should automatically open to `http://localhost:8100/home` and show you the blank starter:



Let us create a new Angular service via the Ionic CLI:

```
ionic g service services/missions
```

```
@Injectable({
  providedIn: 'root'
})
export class MissionsService {

  constructor() { }
}
```

Look familiar? This code is nearly the same as our initial Nest service (the lone difference being the `providedIn` option passed in the Injector, which Nest doesn't support yet).

In our service, we make the HTTP request to `/missions` and return the data. Update the class in `missions.service.ts` with the following:

```
@Injectable({
  providedIn: 'root'
})
export class MissionsService {
  constructor(private httpClient: HttpClient) { }

  getMissions() {
    return this.httpClient.get('http://localhost:3000/missions');
  }
}
```

Since we are using `HttpClient`, make sure to add `HttpClientModule` (from `@angular/common/http`) in the app module's list of imports.

Next, modify the `home.page.ts` file to call into the service and save the results to a local observable:

```
export class HomePage implements OnInit {
  missions: Observable<any>;

  constructor(private missionsService: MissionsService) {}

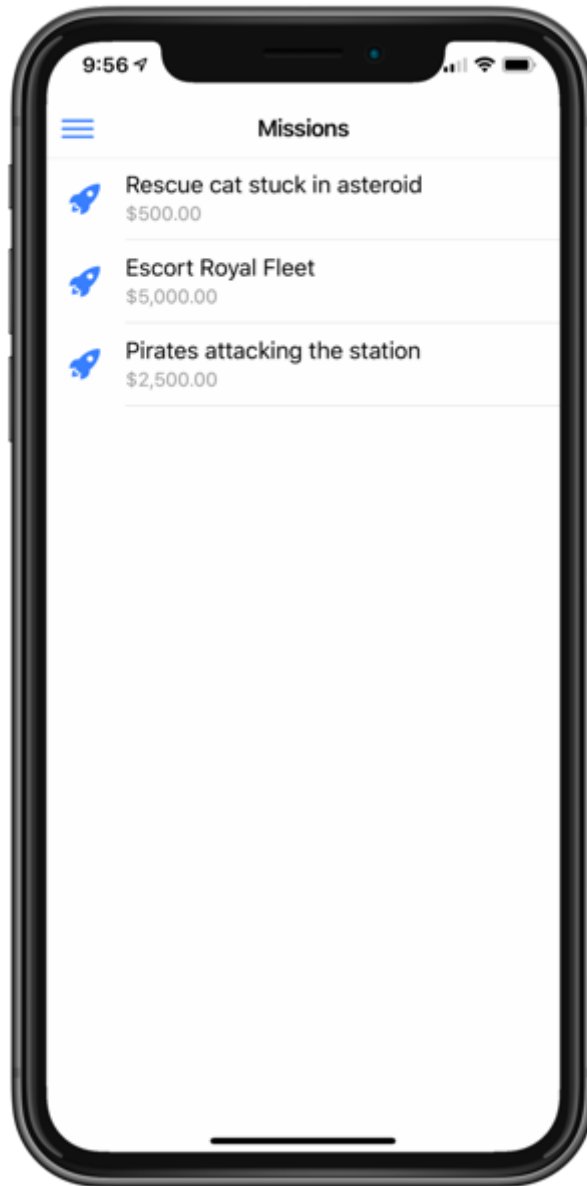
  ngOnInit() {
    this.missions = this.missionsService.getMissions();
  }
}
```

And to wrap it all up, replace the `home.page.html` template with the following:

```
<ion-header>
  <ion-toolbar>
    <ion-buttons slot="start">
      <ion-menu-button></ion-menu-button>
    </ion-buttons>
    <ion-title>Missions</ion-title>
  </ion-toolbar>
</ion-header>

<ion-content>
  <ion-list>
    <ion-item *ngFor="let mission of (missions | async)">
      <ion-label>
        <h2>{{ mission.title }}</h2>
        <p>{{ mission.reward | currency }}</p>
      </ion-label>
      <ion-icon slot="start" icon="rocket" color="primary"></ion-icon>
    </ion-item>
  </ion-list>
</ion-content>
```

Here, we have a reasonably simple Ionic page with a list that displays a few list items, each one with one of the missions coming back from the Nest API.



Now that our space rangers can view their missions, we will wrap part one of this series. So far, we've done a lot: We learned why TypeScript is beneficial not only on the client side but also on the server side. We also learned how to leverage the power of TypeScript on Node by using the Nest framework. And, we built a simple yet functional Ionic app that displays data served up from the Nest API. Exciting stuff!

If you would like to download what we have completed so far, grab the code on GitHub (<https://github.com/ionic-team/demo-ionic-nest-tutorial>).



In part two (<https://ionicframework.com/blog/full-stack-typescript-with-ionic-angular-and-nestjs-part-2/>), we dive deeper into the various building blocks Nest gives us and build out more of our API.

Happy Coding!

Sign up for the Ionic Newsletter to get the latest news and updates!

Email address



### Featured Comment



**Nmuta** • 3 years ago • edited

**Phenomenal** resource and great article. I never new about NestJS but it sounds a lot like a better version of SailsJS , and that fact that it's Angular-esque is amazing.... it's like Nest/Angular is a new 'stack' , so to speak. Can't wait to try out the sockets and the fluid communication with Angular 7 / Ionic.

2 ^ | v • Share ›

11 Comments

Ionic Framework

Disqus' Privacy Policy

Login ▾

Favorite 3

Tweet

Share

Sort by Best ▾



Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS

Name

**Max** • 3 years ago

This is absolutely phenomenal!!! Ionic + Angular + NestJS is exactly the stack that I am planning for and intending to use for my next project development. Really looking forward to your next blog posts on this topic! 👍

6 ^ | v • Reply • Share ›

**Nmuta** • 3 years ago • edited

🏆 Featured by Ionic Framework

**Phenomenal** resource and great article. I never new about NestJS but it sounds a lot like a better version of SailsJS , and that fact that it's Angular-esque is amazing.... it's like Nest/Angular is a new 'stack' , so to speak. Can't wait to try out the sockets and the fluid communication with Angular 7 / Ionic.

2 ^ | v • Reply • Share ›

**Ryan Dwi** • 2 years ago

hi, i got the error:

ERROR Error: Uncaught (in promise): TypeError: this.missionsService.getMissions is not a function

TypeError: this.missionsService.getMissions is not a function

please help:)

^ | v • Reply • Share ›

**Ryan Dwi** ➔ Ryan Dwi • 2 years ago

Its solved for me, the last problem is = Error: Uncaught (in promise): NullInjectorError:

solved when i put in (app.module.ts)

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

and

```
i add it also = imports: [BrowserModule, IonicModule.forRoot(), AppRoutingModule, HttpClientModule],
```

^ | v • Reply • Share ›

**Ryan Dwi** • 2 years ago

when i create ionic start gsr-client, i have an error->please help:)

Unexpected end of JSON input while parsing near '..."dist":{"shasum":"c7d'

C:\Users\LP-NATA-064\AppData\Roaming\npm-cache\\_logs\2020-10-18T12\_08\_33\_929Z-debug.log

^ | v • Reply • Share ›

**Ryan Dwi** ➔ Ryan Dwi • 2 years ago

SOLVED : I run npm cache clean --force

^ | v • Reply • Share ›

**Adebanke Alabi** • 3 years ago

I'm such a fan of this. Thank you for sharing. I feel much more comfortable diving into typescript/angular as a full stack framework. It will really help a lot of us create better apps instead of half learning multiple different frameworks/languages.

^ | v • Reply • Share ›

**Patrick Cotter** • 3 years ago • edited

I had to remove the @Injectable decorator from missions.service.ts & import 'src/models/mission.model' then in mission.controller.ts change import { Controller } to { Controller, Get } from nestjs import & also import './missions.service' to get past a 404 error on the missions URL.

Nest 6.0.0

Angular 8.3.0

Ionic 5.2.6

Node 12.9.1

Update - I'm a newbie. I had a ton of problem with 404 errors so I just rolled back to:

Node 10.16.3

then Nest 6.6.4

@angular/cli 8.1.3

@ionic/angular 4.9.0

also installed @ionic/angular-toolkit 2.0.0

**Newbie advice** I'm using Windows & VS Studio Code. Always terminate (Ctrl-C) any running servers in the terminals in VS Studio before closing VS studio or you will experience 404 errors when everything in the code appears to be correct because the port is still in use when you return.

^ | v • Reply • Share ›

**Marcos Ribeiro** • 3 years ago • edited

Hello, thanks for the post! How can I use Ionic DevApp with Nest? Everything works on my notebook and Ionic DevApp detects the app running on my Wifi network but I can't login on my app. The console don't show any errors... Thanks. =)

^ | v • Reply • Share ›

**Idrees Khan** • 3 years ago

Showing shared codebase approach would be more nicer and useful.

^ | v • Reply • Share ›

**elylucas** ➔ Idrees Khan • 3 years ago

Hi Idrees,

I'm actually working on a shared codebase approach as well outside of this post. I'll post here to the repo when its ready.

5 ^ | v • Reply • Share ›




Subscribe



Add Disqus to your site



Do Not Sell My Data

An Ionic (<https://ionicframework.com/>) creation.  RSS  
(<https://ionicframework.com/blog/feed/>)