

Guide

Purpose

Provides developers a guide on the steps needed to access and utilize the Nestre API, for the purposes of making RESTful API calls.

Prerequisite: Creating An Authorized API Account

Many of the options in our API require signing in as an authorized user of the Nestre app, which requires knowing your **username** and **password** used to create a new user in the Nestre mobile app.

However, this also requires that your account was created from within a **'Development'** build. The version of the app from the App Store is a **"Production"** build, so the database of users on it is different, and those credentials won't help you authorize in our API.

You will need to follow the Nestre Frontend guide series up to the point where you can deploy a developer build to either your Android or iOS device, and then from there create an account from within the app.

Keep note of the username & password for this "Developer" mode user account. We cover how to authorize your session in a section of this guide.

Alternatively, you can use our default **Development** username and password to test this API, located later in this document.

Accessing the RESTful API

You can find the Nestre RESTful API at the link below.

Both options work the same, it's a matter of preference on the style.

<https://appservices.dev.nestreapp.com/docs> - The 'OpenAPI' style

<https://appservices.dev.nestreapp.com/redoc> - The 'Redoc' style

For the purposes of this guide, we will assume you are using the '**OpenAPI**' style.

Logging Into the Nestre OpenAPI

In order to log in, you must first be **invited by an administrator**. This usually will be sent during onboarding to your **name@nestrehealthandperformance.com** email account.

You will have to change your password and enable a two-factor authorization to gain full access after initial login.















If you are missing login information, contact your PM and/or Daniel from the Nestre team to receive your credentials.


Navigating the OpenAPI

Once you have logged in, you are presented with a OpenAPI styled series of API calls, here's an overview of how they work.

Authorize 

User

GET	/v2/user/get-by-email	Get user by email address	 
GET	/v2/user/{user_id}	Get user profile	 
PATCH	/v2/user/{user_id}	Update the user	 
GET	/v2/user/{user_id}/profile	Get user profile with assessment and streaks	 
POST	/v2/user	Create new user account	 
DELETE	/v2/user	Delete user account	 
POST	/v2/user/{user_id}/referral-code	Create referral code	 

[**Authorize** ] - Used to pass in an session token, authorizing you to use additional API calls that are normally locked to non-authorized users.

GET - Returns data from the backend

POST - Creates data in the backend

PATCH - Updates data in the backend

DELETE - Deletes data in the backend

How To Authorize

Authorizing provides access to several API calls that are normally unavailable to unauthorized users.

Authorization requires a valid session token, which we receive after logging in.

For the purposes of testing, you can sign in and get a session token from within our OpenAPI.

Navigate to near the bottom of the OpenAPI page,
and find the **POST** **/dev/app-user-cognito-login** option.

Click on this method to expand it.

Development

POST	/dev/app-user-cognito-login	Authenticate an app user using AWS Cognito credentials
GET	/dev/staff-access-token	Get access token for logged-in staff user

POST /dev/app-user-cognito-login Authenticate an app user using AWS Cognito credentials

This endpoint handles user authentication by validating the provided username and password against the AWS Cognito user pool for the Nestre application.

Args:

username (str): The user's username for authentication
password (str): The user's password for authentication

Returns:

dict: Authentication tokens from AWS Cognito including access token,
refresh token, and ID token upon successful authentication

Raises:

HTTPException: When authentication fails due to invalid credentials
or Cognito service errors

Parameters

No parameters

Try it out

Request body required

application/json

Example Value | Schema

```
{  "username": "string",  "password": "string"}
```

Responses

Code	Description	Links
200	Successful Response	No links

Media type

application/json

Controls Accept header.

Example Value | Schema

Click on the **Try It Out** button.

Change the values of the username and password to match your account and press **Execute** to continue.

After the API call succeeds, you can scroll down and see the response from the server.

```
curl -X 'POST' \
  https://appservices.dev.nestreapp.com/dev/app-user-cognito-login' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "username": "derrick@nestrehealthandperformance.com",
    "password": "Nestrel@2025"
  }'
```

<https://appservices.dev.nestreapp.com/dev/app-user-cognito-login>

Code	Details
------	---------

[illegible]

Now that you have the **AccessToken** in your clipboard, scroll back to the top and click on the [\[Authorize](#)

[/openapi.json](#)

Authorize 

Paste your AccessToken into the Value input field and click [\[Authorize 🔒\]](#).

Available authorizations

HTTPBearer (http, Bearer)

Value:

AuthorizeClose

Available authorizations

HTTPBearer (http, Bearer)

Authorized

Value: *****

LogoutClose

`/profile` Get user profile with assessment and streaks

And now your session while your viewing the API from this browser tab is authorized. This allows you to execute most functions as its usually required.

When your working in the codebase, you have to do something similar to use the API, first you sign in using a service, which sends you an AuthToken, and then you pass that into the Nestre API before you start making requests.

Default Development Account Credentials

If you need a default set credentials, there should always be a default account available to use within the **Development** server.

Feel free to directly **copy+paste** the JSON below to authenticate using the default Development server account.

```
{
  "username": "derrick@nestrehealthandperformance.com",
  "password": "Nestre!@2025"
}
```

And in case you need it during development, here's this user's unique identifier. Most of our internal API tests will have this UID filled out already, but it's here for you to copy+paste as needed.

- **UID:** 4839e58f-9d9c-4065-a864-8da46121c174

What Is A RESTful API?

Think of a RESTful API as a bridge to communicate between the backend and frontend of our Nestre apps.

Using the example of a restaurant, our frontend is a customer, and our backend is the cook. The RESTful API is the waiter.

Your HTTP requests are delivered to the backend, which is then returned with the response data.

The HttpClient is used to 'talk' to the API, which involves making requests to specific URLs (endpoints) using standard methods to perform actions.

Think of this as selecting your order from a menu, telling your waiter, who tells the chef, and then returns with your food.

Communication Concepts

Continuing with our restaurant metaphor, our OpenAPI documentation outlines our “menu”.

To communicate with our API from the frontend, we combine four key concepts.

HTTP Method: The action you want to perform (e.g., `GET`, `POST`, `PATCH`, `DELETE`).

Endpoint: The URL for the specific resource you want to interact with.

Parameters: Identifiers or options you pass in the URL (e.g., `{user_id}`).

Body: The data payload you send to the API, usually in JSON format (for `POST` or `PATCH` requests).

How To Communicate Using Angular's HttpClient

As a best practice, we use a dedicated service in Angular to handle our API calls. That way our implementation details are handled in only one place and can be called from anywhere in our application.

If your coming from a C-style language, think of this as a **static singleton**.

```
//#region IMPORTS

// src/app/services/api.service.ts
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

//#endregion
```

```

/**
 * Provides Nestre API access from anywhere in the application
 */
@Injectable({
  providedIn: 'root'
})
export class ApiService
{

  // #region PRIVATE - VARIABLES

  /**
   * base of our API URL, this should be pulled from an environment file
   * instead of being hardcoded
   */
  private baseUrl = 'https://api.project.com/v2';

  // #endregion

  constructor(private http: HttpClient) { }

  /**
   * Returns a user profile
   * @param {string} userId - The id of the user.
   */
  //-----//
  getUserProfile(userId: string): Observable<any>
  //-----//
  {
    const url = `${this.baseUrl}/user/${userId}/profile`;
    return this.http.get(url);
  }

  // END getUserProfile Method

  /**
   * Creates a new user
   * @param {any} userData - The data of the new user.
   */
  //-----//
  createUser(userData: any): Observable<any>
  //-----//
  {
    const url = `${this.baseUrl}/user`;

```

```

    // userData is an object that matches the backend's expected schema
    return this.http.post(url, userData);

} //END createUser Method

/*
 * Updates a user
 * @param {string} userId - The id of the user.
 * @param {any} updates - data object following the backends expected
schema
 */
//-----//
updateUser(userId: string, updates: any): Observable<any>
//-----//
{
    const url = `${this.baseUrl}/user/${userId}`;
    return this.http.patch(url, updates);

} //END updateUser Method

/*
 * Deletes a user
 * @param {string} userId - The id of the user
 */
//-----//
deleteUser(userId: string): Observable<any>
//-----//
{
    const url = `${this.baseUrl}/user/${userId}`;
    return this.http.delete(url);

} //END deleteUser Method

} //END ApiService Class

```

And then we can use our ApiService in another Angular script...

```

// Within another Angular component
this.apiService.getUserProfile('some-user-123').subscribe

```

```
(
  profileData => { console.log('User Profile:', profileData); }
);
```

What's happening:

- We construct the full URL by inserting the `userId`.
- `http.get(url)` tells the "waiter" to fetch the data from that specific table (URL).
- `.subscribe()` is how Angular handles asynchronous operations. It waits for the data to come back from the server and then executes the code inside.

POST: Creating New Data

`POST` requests are used to create a new resource.

- **Example Endpoint:** `POST /v2/user`
- **Goal:** Create a new user account.

`POST` requests almost always include a **request body**—the data for the new item. The structure of this body is defined by the `Schemas` in your documentation (e.g., `UserCreate`).

```
// In your Angular component
const newUser = {
  email: 'test@example.com',
  password: 'a-strong-password',
  firstName: 'John',
  // ... other fields defined in the UserCreate schema
};

this.apiService.createUser(newUser).subscribe(createdUser => {
  console.log('Successfully created user:', createdUser);
});
```

What's happening:

- We pass the `newUser` object as the second argument to `http.post()`. This is the "order" you're giving the waiter. 🍷
- The backend receives this object, creates the user in the database, and typically returns the newly created user's data as a confirmation.

PATCH: Updating Existing Data

`PATCH` is used for making partial updates to an existing resource.

- **Example Endpoint:** `PATCH /v2/user/{user_id}`
- **Goal:** Update a specific user's information (e.g., their email).

Like `POST`, this uses a body, but it often only contains the fields that need changing. It also requires a path parameter to know *which* user to update.

```
// In your Angular component
const userUpdates = {
  maritalStatus: 'Married' // We only want to change this one field
};

this.apiService.updateUser('some-user-123',
userUpdates).subscribe(updatedUser => {
  console.log('Update successful:', updatedUser);
});
```

DELETE: Removing Data

`DELETE` requests are used to remove a resource.

- **Example Endpoint:** `DELETE /v2/user/{user_id}/frame/{frame_id}`
- **Goal:** Delete a specific "frame" belonging to a user.

This often just requires an ID in the URL and doesn't need a body.

```
// In your Angular component
this.apiService.deleteFrame('some-user-123', 'frame-abc-456').subscribe(()
```

```
=> {  
  console.log('Frame was successfully deleted.');
```

// The backend might return an empty response or a success message.

```
});
```

Whitelisted Addresses

For Our Development Server

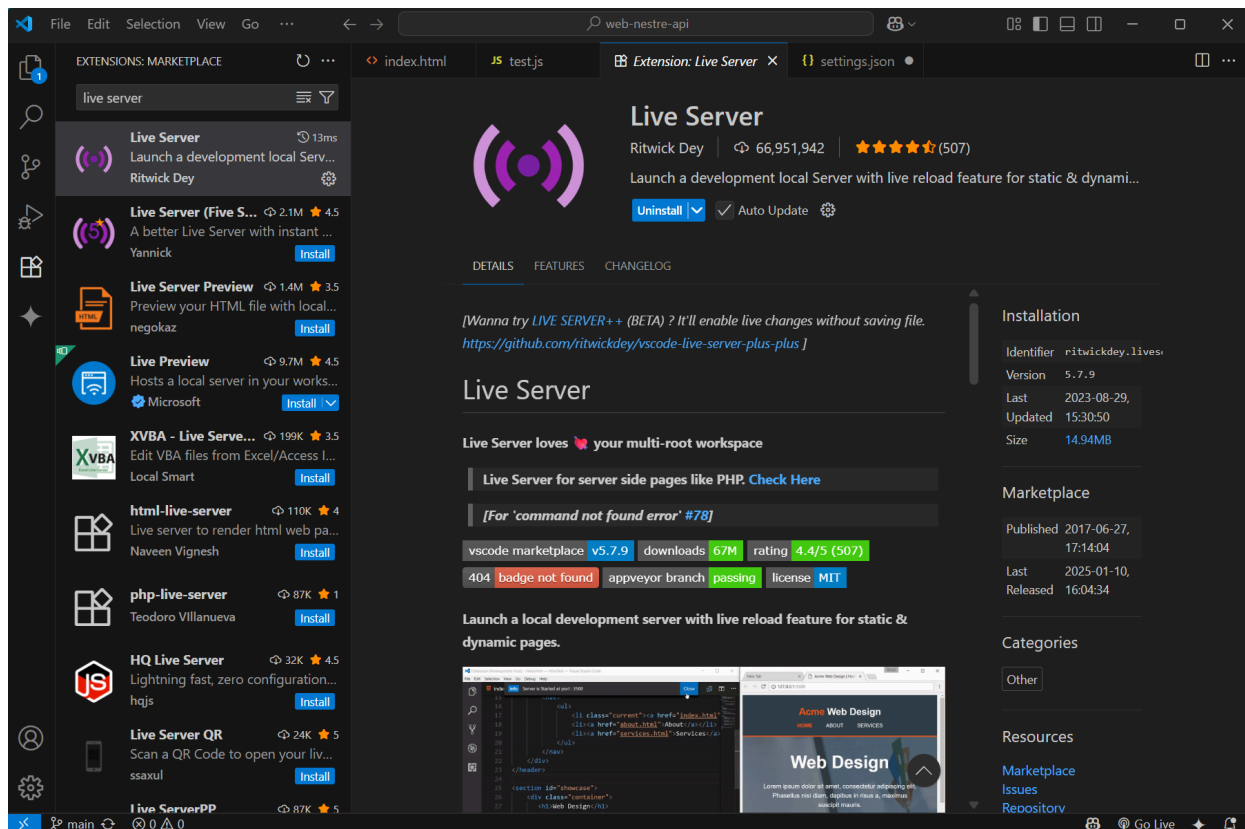
When testing API calls to our development server from a local project, oftentimes you'll test code that runs using a local live server plugin in VSCode, which avoids us having to deploy our code to a server to test or use a tool like WebPack or Vite to pass our API requests through a hosted server.

In order to make it past CORS and have your incoming request not blocked, you'll need to run your live server extension from one of the following ports.

```
ALLOWED_ORIGINS = [  
  "http://localhost:8100",  
  "http://localhost:8080",  
  "http://localhost:4200",  
  "http://localhost",  
  "https://localhost",  
  "capacitor://localhost",  
  "ionic://localhost",  
  "https://admin-dev.nestreapp.com",  
  "https://admin.nestreapp.com",  
]
```

Setting Up The Live Server Plugin In VSCode

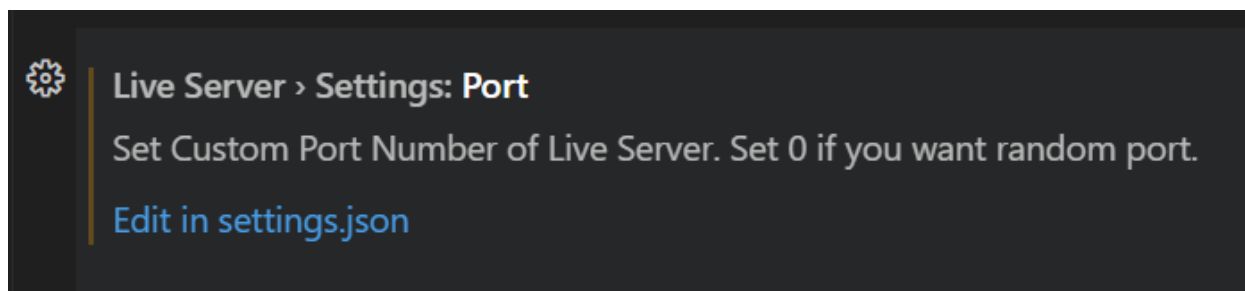
In VSCode, go to the extensions marketplace, and search for and install the 'Live Server' plugin from Ritwick Dey.



Now you can right click any HTML file and open it in a live server. The default port is 5500, which isn't part of default whitelist the Nestre Backend allows. So we'll need to change the port number.

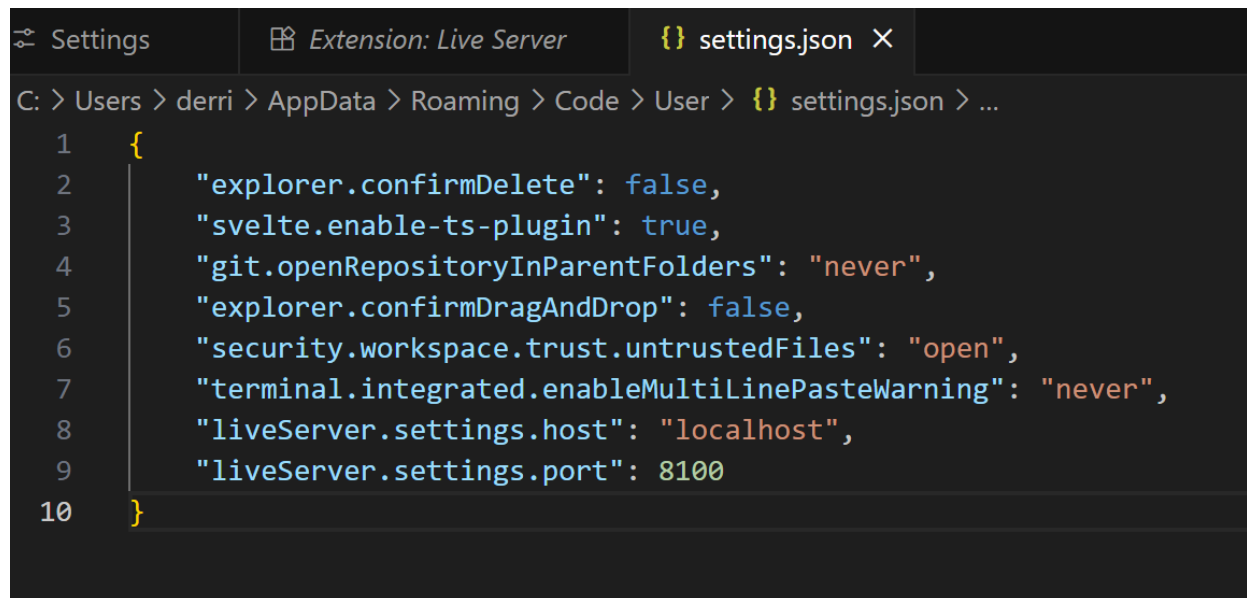
This is done from the Live Server extension page in the marketplace. To the right of the **Uninstall** button is a settings gear icon . Click it and open the plugin settings.

Scroll down until you see the Port setting



Click the **edit** button, then change it from '0' to '8100' so its one of our whitelisted

options and not random.

A screenshot of the Visual Studio Code editor showing the settings.json file. The top bar shows tabs for 'Settings', 'Extension: Live Server', and 'settings.json'. The breadcrumb path is 'C: > Users > derri > AppData > Roaming > Code > User > settings.json'. The file content is a JSON object with various settings, including 'explorer.confirmDelete', 'svelte.enable-ts-plugin', 'git.openRepositoryInParentFolders', 'explorer.confirmDragAndDrop', 'security.workspace.trust.untrustedFiles', 'terminal.integrated.enableMultiLinePasteWarning', 'liveServer.settings.host', and 'liveServer.settings.port'.

```
1  {
2    "explorer.confirmDelete": false,
3    "svelte.enable-ts-plugin": true,
4    "git.openRepositoryInParentFolders": "never",
5    "explorer.confirmDragAndDrop": false,
6    "security.workspace.trust.untrustedFiles": "open",
7    "terminal.integrated.enableMultiLinePasteWarning": "never",
8    "liveServer.settings.host": "localhost",
9    "liveServer.settings.port": 8100
10 }
```

We also should change our **host** variable to **localhost**, that way our live server will directly match our first whitelisted url origin from the list posted in the previous section.

```
"http://localhost:8100"
```

Here's the JSON snippet for you to copy+paste into your settings.json file

```
"liveServer.settings.host": "localhost",
"liveServer.settings.port": 8100
```

Also, if you for some reason can't directly open the link to the settings.json file from VSCode (this can happen due to permissions in Windows). You can always navigate to the file directly to open it up for edit.

The file path is **%APPDATA%/Code/User/settings.json**

This usually translates to

C:/Users/<username>/AppData/Roaming/Code/User/settings.json

Save the **settings.json** file, and next time you launch the live server you'll be using the correct whitelisted port.

