

HOLA! Ya he hecho toda la API en nestJS de una página web de tareas en local. También tiene su BBDD montada en Oracle. Ahora tengo que hacer el FRONT en angular. Pero lo primero es lo primero. ¡Tengo que instalar ANGULAR y prepararlo todo para empezar a trabajar. ¿Podrás asistirme en la creación de mi sitio web en local?

¡Por supuesto! Estaré encantado de ayudarte a configurar tu entorno de Angular para comenzar con el desarrollo del frontend de tu aplicación de tareas. Vamos a hacerlo paso a paso:

1. Instalar Node.js y npm

Angular requiere Node.js y npm (que viene incluido con Node.js). Si no los tienes instalados:

- Descarga Node.js desde <https://nodejs.org/> (versión LTS recomendada).
- Instálalo y verifica la instalación con:

```
bash
```

- `node -v`
- `npm -v`

2. Instalar Angular CLI

El Angular CLI es la herramienta oficial para crear y gestionar proyectos Angular. Instálalo globalmente con:

```
bash
npm install -g @angular/cli
```

Verifica la instalación:

```
bash
ng version
```

3. Crear un nuevo proyecto Angular

Ejecuta el siguiente comando para generar un nuevo proyecto (reemplaza `my-task-app` con el nombre de tu proyecto):

```
bash
ng new my-task-app
```

- El CLI te preguntará si quieres añadir **routing** (elige **Sí** si tu app tendrá múltiples vistas).
- Selecciona **CSS** como preprocesador (a menos que prefieras Sass/SCSS).

4. Ejecutar el proyecto

Navega a la carpeta del proyecto y levanta el servidor de desarrollo:

```
bash
cd my-task-app
ng serve
```

Abre tu navegador en <http://localhost:4200>. ¡Verás la app Angular por defecto!

5. Configuración adicional importante

Conectar con tu API en NestJS

- Crea un servicio Angular para consumir la API:

```
bash
```

- `ng generate service api/task`

- Edita el archivo `src/app/api/task.service.ts` para incluir llamadas HTTP a tu backend (ejemplo básico):

```
typescript
```

- ```
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';
```

```
@Injectable({
 providedIn: 'root'
})
export class TaskService {
 private apiUrl = 'http://localhost:3000'; // Ajusta el puerto de tu
 API NestJS

 constructor(private http: HttpClient) { }

 getTasks() {
 return this.http.get(`${this.apiUrl}/tasks`);
 }
}
```

- Asegúrate de importar `HttpClientModule` en `src/app/app.module.ts`:

```
typescript
```

- ```
import { HttpClientModule } from '@angular/common/http';
```
- ```
@NgModule({
```
- ```
  imports: [HttpClientModule],
```
- ```
})
```

### Proxy para evitar problemas de CORS

Si tu API NestJS corre en otro puerto (ej: 3000), crea un archivo `proxy.conf.json` en la raíz del proyecto Angular:

```
json
{
 "/api": {
 "target": "http://localhost:3000",
 "secure": false,
 "pathRewrite": {"^/api": ""}
 }
}
```

Luego, modifica `ng serve` para usar el proxy:

```
bash
ng serve --proxy-config proxy.conf.json
```

Y actualiza la URL en el servicio a `/api/tasks`.

---

## 6. Estructura básica del proyecto

Organiza tu frontend según las mejores prácticas:

```
text
src/app/
├── components/ # Componentes reutilizables
├── pages/ # Vistas principales (ej: home, tasks)
├── services/ # Lógica de API (TaskService)
├── models/ # Interfaces (ej: Task.model.ts)
└── shared/ # Módulos compartidos (pipes, directives)
```

## 7. Ejemplo de componente para listar tareas

Genera un componente:

```
bash
ng generate component pages/task-list
```

Edita `task-list.component.ts`:

```
typescript
import { Component, OnInit } from '@angular/core';
import { TaskService } from '../../../services/task.service';

@Component({
 selector: 'app-task-list',
 templateUrl: './task-list.component.html'
})
export class TaskListComponent implements OnInit {
 tasks: any[] = [];

 constructor(private taskService: TaskService) {}
```

```
ngOnInit() {
 this.taskService.getTasks().subscribe(data => {
 this.tasks = data;
 });
}
```

---

## 8. Extras recomendados

- **Instalar Bootstrap o Angular Material** para estilos:

```
bash
```

```
npm install bootstrap
```

Luego importa los estilos en `angular.json`:

```
json
```

- `"styles": ["node_modules/bootstrap/dist/css/bootstrap.min.css"]`

- **Habilitar CORS en NestJS** (si no usas proxy):

En tu `main.ts` de NestJS:

```
typescript
```

- `app.enableCors();`
- 

## ¿Qué necesitas a continuación?

- ¿Quieres ayuda para crear un formulario de tareas?
- ¿Necesitas configurar rutas en Angular?
- ¿O prefieres profundizar en la conexión API-frontend?

¡Dime en qué quieres enfocarte y seguimos avanzando! ☐

## Es necesario instalar angular globalmente con -g?

¡Buena pregunta! No, **no es estrictamente necesario instalar Angular CLI globalmente** (con el flag `-g`), pero hacerlo tiene ventajas importantes que vale la pena considerar. Aquí te explico los detalles:

---

## ¿Por qué se recomienda instalarlo globalmente?

1. **Acceso directo al comando `ng` desde cualquier carpeta:**

Si lo instalas global (`npm install -g @angular/cli`), podrás ejecutar comandos como `ng new`, `ng generate`, etc., en cualquier proyecto sin tener que reinstalar el CLI en cada uno.

## 2. **Facilidad para mantener actualizado el CLI:**

Actualizarlo es más sencillo (con `npm update -g @angular/cli`), y todos tus proyectos usarán la misma versión global (aunque cada proyecto puede tener su propia versión local si es necesario).

## 3. **Es la práctica estándar:**

La documentación oficial de Angular asume que lo tienes instalado globalmente.

---

## **Alternativa: Instalación local (sin -g)**

Puedes instalarlo **solo en tu proyecto** (o incluso usar `npx` para ejecutarlo sin instalación), pero tiene desventajas:

- **Ejemplo de instalación local:**

```
bash
```

- `npm install @angular/cli --save-dev`

- **Problemas:**

- Tendrás que usar `npx ng` o `scripts` de `package.json` para ejecutar comandos (ej: `npx ng new my-app`).
  - No tendrás acceso directo al comando `ng` fuera del proyecto.
  - Puede generar conflictos si trabajas con múltiples proyectos Angular.
- 

## **Recomendación práctica**

### 1. **Instálalo globalmente** si planeas trabajar con Angular frecuentemente:

```
bash
```

1. `npm install -g @angular/cli`

### 2. **Cada proyecto Angular tendrá su propia versión local del CLI** (en `node_modules`), aunque uses la global para generar el proyecto.

- El CLI global solo se usa para comandos de scaffolding (`ng new`, `ng generate`).
  - La versión local del CLI se usa al ejecutar `ng serve` o `ng build` dentro del proyecto.
- 

## **¿Qué pasa si no quiero instalarlo globalmente?**

Puedes usar **npx** (que viene con npm) para ejecutar el CLI sin instalación global:

```
bash
```

```
npx @angular/cli new my-task-app
```

Pero deberás hacer esto cada vez que quieras crear un proyecto o ejecutar comandos fuera de uno existente.

---

## Conclusión

- **Para desarrollo profesional:** Instálalo globalmente (-g).
- **Para proyectos puntuales o entornos restrictivos:** Usa `npmx @angular/cli`.

Si ya lo instalaste local (sin -g) y quieres pasarlo a global, desinstálalo primero localmente:

```
bash
npm uninstall @angular/cli
```

Luego instálalo globalmente:

```
bash
npm install -g @angular/cli
```

¿Necesitas ayuda con algún otro paso de la configuración? ☐

? Which stylesheet format would you like to use? (Use arrow keys) ☐ CSS [ <https://developer.mozilla.org/docs/Web/CSS> ] Sass (SCSS) [ <https://sass-lang.com/documentation/syntax#scss> ] Sass (Indented) [ <https://sass-lang.com/documentation/syntax#the-indented-syntax> ] Less [ <http://lesscss.org> ]

When setting up your Angular project, you'll be asked to choose a stylesheet format. Here's a breakdown of each option to help you decide:

### 1. CSS (Recommended for Beginners)

- **Best for:** Simple projects or if you're new to frontend development.
- **Pros:**
  - No extra setup needed (works out of the box).
  - Standard web technology (no compilation step).
- **Cons:**
  - No advanced features like variables or nesting.

### 2. Sass (SCSS) (Recommended for Most Projects)

- **Best for:** Professional projects where you want maintainable styles.
- **Pros:**
  - Supports **variables, nesting, mixins, and functions**.
  - More structured than plain CSS.
  - `.scss` syntax is similar to CSS (easy to learn).
- **Cons:**
  - Requires a Sass compiler (handled automatically by Angular CLI).

### 3. Sass (Indented Syntax)

- **Best for:** Developers who prefer a minimal syntax (like Python-style indentation).
- **Pros:**
  - Less verbose than SCSS (no `{ }` or `;` needed).
- **Cons:**
  - Less common (harder to find examples/documentation).

### 4. Less

- **Best for:** Legacy projects or if you're already familiar with Less.
- **Pros:**
  - Similar to SCSS (variables, mixins).
- **Cons:**
  - Less popular than Sass in the Angular ecosystem.

---

### Recommendation

- **Choose `css`** if you want the simplest setup.
- **Choose `scss`** (Sass) if you plan to build a scalable app (it's the most popular choice in Angular projects).

You can always add **Sass later** if you start with CSS by running:

```
bash
npm install sass
```

And renaming your `.css` files to `.scss`.

---

### After Selection

The Angular CLI will:

1. Set up the necessary build configuration.
2. Generate `.css` or `.scss` files accordingly.

Would you like guidance on structuring your styles? ☐

✓ Which stylesheet format would you like to use? Sass (SCSS) [ <https://sass-lang.com/documentation/syntax#scss> ] ? Do you want to enable Server-Side Rendering (SSR) and Static Site Generation (SSG/Prerendering)? (y/N)

**About Server-Side Rendering (SSR) and Static Site Generation (SSG) in Angular**

Since you've selected **Sass (SCSS)** for styling, now Angular CLI is asking if you want to enable:

- **Server-Side Rendering (SSR)** via `@angular/ssr`
- **Static Site Generation (SSG/Prerendering)**

### Should You Enable It?

| Option                                               | Best For                                                                  | Pros                                                                                      | Cons                                     |
|------------------------------------------------------|---------------------------------------------------------------------------|-------------------------------------------------------------------------------------------|------------------------------------------|
| <input type="checkbox"/> <b>Yes (Enable SSR/SSG)</b> | SEO-friendly apps, marketing sites, or apps needing fast initial load.    | Better SEO, faster First Contentful Paint (FCP), improved performance on low-end devices. | Adds complexity, slightly slower builds. |
| <input type="checkbox"/> <b>No (Skip SSR/SSG)</b>    | Internal dashboards, apps where SEO doesn't matter, or simple prototypes. | Simpler setup, faster builds.                                                             | Worse SEO, slower initial load.          |

### Recommendation

- **Choose `no` (N)** if:
  - You're building an **internal admin panel** (like a task management app).
  - You don't care about SEO.
  - You want a simpler setup for now.
- **Choose `yes` (Y)** if:
  - Your app is **public-facing** (e.g., a blog, portfolio, or e-commerce site).
  - You need **better SEO** (Google crawlers prefer SSR).
  - You want **optimized performance** for slow networks.

---

### What Happens If You Enable It?

1. Angular CLI installs `@angular/ssr`.
2. Adds:
  - A server entry point (`server.ts`)
  - Prerendering config (`prerender.ts`)
3. Updates `angular.json` for SSR/SSG support.

---

### Can You Add SSR Later?

**Yes!** You can always add it manually later by running:

```
bash
ng add @angular/ssr
```

---



## Final Decision

- For a **task management app**, **no** (**n**) is usually sufficient.
- For a **public website**, **yes** (**y**) is better.

Press **n** (default) to skip, or **y** to enable.

Want help deciding based on your specific project? ☐

## ¿Cómo funcionaba SCSS?

Es un **preprocesador de CSS** que añade superpoderes a tu código CSS normal. Aquí las diferencias clave:

### ☐ Ventajas de SCSS sobre CSS puro

1. **Variables** → Evitas repetir colores/fuentes.

SCSS

- `$primary-color: #3498db;`  
`.boton { background: $primary-color; }`

- **Anidamiento (Nesting)** → Estructura más clara.

SCSS

- `.lista-tareas {`  
  `li {`  
    `color: blue;`  
    `&:hover { color: red; } // & = parent selector (.lista-tareas`  
`li:hover)`  
  `}`  
`}`

- **Mixins** → Reutiliza bloques de estilos.

SCSS

- `@mixin flex-center {`  
  `display: flex;`  
  `justify-content: center;`  
`}`  
`.contenedor { @include flex-center; }`

- **Importar archivos** → Modulariza tu CSS.

SCSS

4. `@import 'variables'; // _variables.scss`
5. `@import 'components/buttons';`

### ☐ ¿Es obligatorio usar estas features?

¡No! Puedes usar SCSS como si fuera CSS normal (simplemente cambiando la extensión de `.css` a `.scss`). Angular lo compilará automáticamente.

---

## ¿Se puede usar SCSS + Bootstrap?

¡Sí! De hecho, es una combinación muy común. Bootstrap se puede integrar de dos formas:

### ❑ Opción 1: Usar Bootstrap como CSS clásico (Recomendado para tu caso)

1. Instala Bootstrap:

```
bash
```

- `npm install bootstrap`
- Importa su CSS en `styles.scss`:

scss

```
2. @import '~bootstrap/dist/css/bootstrap.min.css';
```

#### 3. **Ventaja:**

- No necesitas aprender Sass.
- Puedes sobrescribir estilos de Bootstrap con tus propias clases en SCSS.

### ❑ Opción 2: Usar Bootstrap + Sass (Personalización avanzada)

Si quisieras modificar variables de Bootstrap (ej: colores primarios):

1. Instala Bootstrap + Sass:

```
bash
```

- `npm install bootstrap @ng-bootstrap/ng-bootstrap`
- Crea un archivo `_custom-bootstrap.scss` donde defines tus variables antes de importar Bootstrap:

scss

```
2. $primary: #3498db; // Cambia el color primario de Bootstrap
```

```
3. @import '~bootstrap/scss/bootstrap';
```

#### 4. **Ventaja:**

- Personalización total de Bootstrap.

---

## Ejemplo práctico para tu app de tareas

### Estructura SCSS recomendada

```

text
src/
├── styles.scss # Estilos globales + importa Bootstrap
├── app/
│ ├── auth/
│ │ ├── login.component.scss
│ │ └── register.component.scss
│ └── tasks/
│ ├── task-list.component.scss
│ └── task-form.component.scss

```

## Uso básico en un componente

```

SCSS
// task-list.component.scss
@import 'src/styles/variables'; // Si tienes variables globales

.lista-tareas {
 padding: 1rem;
 &-item { // Compila a .lista-tareas-item
 border-bottom: 1px solid gray;
 &:hover { background: #f8f9fa; }
 }
}

```

---

## Conclusión

- **SCSS** es CSS con superpoderes, pero puedes usarlo como CSS normal si prefieres.
- **Bootstrap + SCSS** funcionan perfectamente juntos. Para tu proyecto, la **Opción 1** (Bootstrap como CSS) es la más rápida.
- Si más adelante quieres profundizar en Sass, puedes empezar con variables y nesting.

¿Necesitas un ejemplo concreto de cómo estructurar tus estilos? ☐

```

PS C:\Users\Usuario\Desktop\Proyecto A3SATEL\Frontend\3tt-client> ng serve
Component HMR has been enabled, see https://angular.dev/hmr for more info.
Application bundle generation failed. [1.477 seconds] X [ERROR] Could not
resolve "~bootstrap/dist/css/bootstrap.min.css" src/styles.scss:2:8: 2 | @import
'~bootstrap/dist/css/bootstrap.min.css'; |
~~~~~ You can mark the path
"~bootstrap/dist/css/bootstrap.min.css" as external to exclude it from the bundle,
which will remove this error and leave the unresolved path in the bundle. Watch
mode enabled. Watching for file changes... /* You can add global styles to this file,
and also import other style files */ @import
'~bootstrap/dist/css/bootstrap.min.css';

```

El error ocurre porque **Angular (usando el compilador ESBuild)** no reconoce el **prefijo ~ para importar Bootstrap**. Esto es común en versiones recientes de Angular (v17+). Aquí te muestro cómo solucionarlo rápidamente:

---

## Solución 1 (Recomendada): Usar la ruta directa de `node_modules`

Reemplaza la línea en `src/styles.scss`:

```
SCSS
/* Antes (error) */
@import '~bootstrap/dist/css/bootstrap.min.css';

/* Ahora (correcto) */
@import 'bootstrap/dist/css/bootstrap.min.css'; // Sin el ~
```

---

## Solución 2: Configurar `angular.json` para alias `~`

Si prefieres mantener el `~` (como en versiones antiguas de Angular), edita `angular.json`:

```
json
{
  "projects": {
    "tu-proyecto": {
      "architect": {
        "build": {
          "options": {
            "stylePreprocessorOptions": {
              "includePaths": ["node_modules"]
            }
          }
        }
      }
    }
  }
}
```

Luego usa en `styles.scss`:

```
SCSS
@import '~bootstrap/dist/css/bootstrap.min.css'; // Ahora funcionará
```

---

## Solución 3: Importar desde `src/assets` (opcional)

1. Copia manualmente el CSS de Bootstrap a `src/assets/css/`:

bash

- `cp node_modules/bootstrap/dist/css/bootstrap.min.css src/assets/css/`

- Importa desde ahí:

SCSS

```
2. @import '../assets/css/bootstrap.min.css';
```

---

## ¿Por qué ocurre esto?

- Angular 17+ usa **ESBuild** (más rápido que Webpack), que no soporta el alias ~ por defecto.
  - Las soluciones anteriores son compatibles con cualquier versión de Angular.
- 

## Verificación

Después del cambio, ejecuta:

```
bash
ng serve
```

¡Debería funcionar sin errores! Verifica que Bootstrap se aplique correctamente (ej: un botón con clase `btn btn-primary`).

---

## Extra: Si usas SASS/SCSS

Aprovecha para personalizar Bootstrap (opcional):

```
scss
// styles.scss
@import 'bootstrap/scss/bootstrap'; // Importa todo Bootstrap (SCSS)

// Sobrescribe variables
$primary: #3498db;
$danger: #ff4136;

// Luego importa el resto de tu SCSS
@import 'custom-styles';
```

¿Necesitas ayuda con otro error o configuración? ☐