

```
import { Entity, PrimaryGeneratedColumn, Column } from 'typeorm';
```

```
@Entity({ name: 'CATEGORY' })
```

```
export class Category {
```

```
  @PrimaryGeneratedColumn({ name: 'ID' })
```

```
  id: number;
```

```
  @Column({ name: 'NAME' })
```

```
  name: string;
```

```
  @Column({ name: 'USER_ID' })
```

```
  user_id: number;
```

```
} Hay qué poner algo para clave foránea?
```

Sí, si `user_id` es una **clave foránea** que referencia a otra tabla (por ejemplo, `USER`), debes definir la relación adecuada en TypeORM. Aquí te explico cómo hacerlo según el tipo de relación:

1. Relación Muchos-a-Uno (Many-to-One)

Si una categoría pertenece a un solo usuario (y un usuario puede tener muchas categorías), usa `@ManyToOne`:

typescript

```
import { Entity, PrimaryGeneratedColumn, Column, ManyToOne, JoinColumn } from 'typeorm';
import { User } from '../user.entity'; // Asegúrate de importar la entidad User
```

```
@Entity({ name: 'CATEGORY' })
```

```
export class Category {
```

```
  @PrimaryGeneratedColumn({ name: 'ID' })
```

```
  id: number;
```

```
  @Column({ name: 'NAME' })
```

```
  name: string;
```

```
  @ManyToOne(() => User, (user) => user.categories) // Relación Many-to-One
```

```
  @JoinColumn({ name: 'USER_ID' }) // Columna FK en la BD
```

```
  user: User; // Nombre del campo para acceder al objeto User relacionado
```

```
}
```

En la entidad `User`:

```
typescript
import { Entity, OneToMany } from 'typeorm';
import { Category } from '../category.entity';

@Entity({ name: 'USER' })
export class User {
  // ... (otros campos)

  @OneToMany(() => Category, (category) => category.user) // Relación
  inversa
  categories: Category[]; // Lista de categorías del usuario
}
```

2. Si solo necesitas la FK sin cargar la relación

Si solo quieres almacenar el `user_id` **sin cargar el objeto `User` completo**, puedes dejarlo como está:

```
typescript
@Column({ name: 'USER_ID' })
user_id: number; // Solo el ID, sin relación
```

3. Clave foránea con restricciones

Para agregar restricciones de integridad (ej: `ON DELETE CASCADE`), configura la relación así:

```
typescript
@ManyToOne(() => User, (user) => user.categories, {
  onDelete: 'CASCADE', // Elimina categorías si se borra el usuario
})
@JoinColumn({ name: 'USER_ID' })
user: User;
```

4. Migración después de los cambios

Si usas migraciones, genera una nueva para aplicar los cambios:

```
bash
typeorm migration:generate -n AddUserRelationToCategory
```

Diferencia clave

Enfoque	Ventaja	Desventaja
Con	Accedes al objeto <code>User</code> completo	Carga datos adicionales (usa

Enfoque	Ventaja	Desventaja
<code>@ManyToOne</code>	<code>(category.user.name).</code>	<code>eager: false</code> para evitar esto).
Solo <code>user_id</code>	Más ligero (solo almacena el ID).	No puedes acceder a datos del usuario sin consulta adicional.

Recomendación

Usa `@ManyToOne` si necesitas acceder a los datos del usuario frecuentemente. Si solo necesitas el ID, mantén el `@Column` simple.

Para más detalles, consulta la [documentación de TypeORM](#).

```
@ManyToOne(() => User, (user) => user.categories) // Relación Many-to-One
```

```
@JoinColumn({ name: 'USER_ID' }) // Columna FK en la BD
```

```
user: User; // Nombre del campo para acceder al objeto User relacionado.
```

un momento no entiendo esto. En mis tabla category hay un user_id para saber el usuario pero no hay necesidad de tener un campo categories en la tabla user. Es un 1 a muchos clásico y típico. por qué entonces user.categories?

Entiendo tu confusión! Vamos a aclararlo con un ejemplo práctico y sencillo:

Explicación simplificada:

En tu caso, tienes una relación "1 a muchos" clásica:

1 Usuario (USER) → Muchas Categorías (CATEGORY).

- En la **tabla CATEGORY**: Ya tienes el campo `USER_ID` (la clave foránea). □
- En la **tabla USER**: **No necesitas** un campo `categories`. □

¿Por qué aparece `user.categories` en el código?

TypeORM usa este código para representar la **relación inversa** (opcional), pero **no afecta a la base de datos**. Es solo para acceder a los datos desde el código.

Solución ajustada a tu caso:

Si solo quieres la **FK en CATEGORY** sin la relación inversa en `USER`:

1. Entidad *Category* (así como la necesitas):

typescript

```
import { Entity, PrimaryGeneratedColumn, Column, ManyToOne, JoinColumn } from 'typeorm';
import { User } from '../user.entity'; // Asegúrate de tener esta entidad

@Entity({ name: 'CATEGORY' })
export class Category {
  @PrimaryGeneratedColumn({ name: 'ID' })
  id: number;

  @Column({ name: 'NAME' })
  name: string;

  // Relación Many-to-One (sin referencia inversa en User)
  @ManyToOne(() => User) // <-- Sin (user) => user.categories
  @JoinColumn({ name: 'USER_ID' }) // FK en la tabla CATEGORY
  user: User; // Para acceder al usuario: `category.user.name`
}
```

2. Entidad *User* (sin campo *categories*):

typescript

```
@Entity({ name: 'USER' })
export class User {
  @PrimaryGeneratedColumn({ name: 'ID' })
  id: number;

  // ... otros campos (email, password, etc.)
  // ;No necesitas declarar la relación inversa si no la usas!
}
```

¿Qué ganas con esto?

- **Desde una categoría:** Puedes acceder al usuario completo:

typescript

- `const category = await categoryRepository.findOne({ id: 1 });`
- `console.log(category.user.name);` // Nombre del usuario dueño de la categoría
- **La tabla `USER` en la BD no cambia** (no se agrega ningún campo).
- **La tabla `CATEGORY` sigue teniendo solo `USER_ID` como FK.**

Si NO necesitas acceder al usuario desde categoría:

Elimina por completo la relación y usa solo `user_id` como columna simple:

typescript

```
@Entity({ name: 'CATEGORY' })
```

```
export class Category {
  // ...
  @Column({ name: 'USER_ID' })
  user_id: number; // Solo el ID, sin relación TypeORM
}
```

Usa esto si:

- Solo necesitas almacenar el ID (ej. para filtros).
- No requieres cargar datos del usuario frecuentemente.

Resumen clave:

Enfoque	¿Necesitas <code>user.categories</code> en <code>User</code> ?	¿Puedes hacer <code>category.user</code> ?
Con <code>@ManyToOne()</code>	No	Sí
Solo <code>use user_id</code>	No	No

Elige la primera opción si quieres acceder a los datos del usuario fácilmente desde TypeORM. □

`@ManyToOne(() => User, (user) => user.categories) // Relación Many-to-One`

`@JoinColumn({ name: 'USER_ID' }) // Columna FK en la BD`

`user: User; // Nombre del campo para acceder al objeto User relacionado.`

un momento no entiendo esto. En mi tabla category hay un user_id para saber el usuario pero no hay necesidad de tener un campo categories en la tabla user. Es un 1 a muchos clásico y típico. por qué entonces user.categories?

¡Entiendo tu confusión! Vamos a aclararlo con un ejemplo práctico y sencillo:

Explicación simplificada:

En tu caso, tienes una relación "1 a muchos" clásica:

1 Usuario (USER) → Muchas Categorías (CATEGORY).

- En la **tabla CATEGORY**: Ya tienes el campo `USER_ID` (la clave foránea). ☐
- En la **tabla USER**: **No necesitas** un campo `categories`. ☐

¿Por qué aparece `user.categories` en el código?

TypeORM usa este código para representar la **relación inversa** (opcional), pero **no afecta a la base de datos**. Es solo para acceder a los datos desde el código.

Solución ajustada a tu caso:

Si solo quieres la FK en **CATEGORY** sin la relación inversa en **USER**:

1. Entidad `Category` (así como la necesitas):

typescript

```
import { Entity, PrimaryGeneratedColumn, Column, ManyToOne, JoinColumn } from 'typeorm';
import { User } from '../user.entity'; // Asegúrate de tener esta entidad

@Entity({ name: 'CATEGORY' })
export class Category {
  @PrimaryGeneratedColumn({ name: 'ID' })
  id: number;

  @Column({ name: 'NAME' })
  name: string;

  // Relación Many-to-One (sin referencia inversa en User)
  @ManyToOne(() => User) // <-- Sin (user) => user.categories
  @JoinColumn({ name: 'USER_ID' }) // FK en la tabla CATEGORY
  user: User; // Para acceder al usuario: `category.user.name`
}
```

2. Entidad `User` (sin campo `categories`):

typescript

```
@Entity({ name: 'USER' })
export class User {
  @PrimaryGeneratedColumn({ name: 'ID' })
  id: number;

  // ... otros campos (email, password, etc.)
  // ¡No necesitas declarar la relación inversa si no la usas!
}
```

¿Qué ganas con esto?

- **Desde una categoría:** Puedes acceder al usuario completo:

typescript

- `const category = await categoryRepository.findOne({ id: 1 });`
- `console.log(category.user.name);` // Nombre del usuario dueño de la categoría
- **La tabla USER en la BD no cambia** (no se agrega ningún campo).
- **La tabla CATEGORY** sigue teniendo solo `USER_ID` como FK.

Si NO necesitas acceder al usuario desde categoría:

Elimina por completo la relación y usa solo `user_id` como column simple:

typescript

```
@Entity({ name: 'CATEGORY' })
export class Category {
  // ...
  @Column({ name: 'USER_ID' })
  user_id: number; // Solo el ID, sin relación TypeORM
}
```

Usa esto si:

- Solo necesitas almacenar el ID (ej. para filtros).
- No requieres cargar datos del usuario frecuentemente.

Resumen clave:

Enfoque	¿Necesitas <code>user.category_id</code> en User?	¿Puedes hacer <code>category.user</code> ?
Con <code>@ManyToOne</code>	No	Sí
Solo <code>user_id</code>	No	No

Elige la primera opción si quieres acceder a los datos del usuario fácilmente desde TypeORM. □

```
@ManyToOne(() => Users, {users} => users.categories) // Relación Many-to-One
```

```
@JoinColumn({ name: 'USER_ID' }) // Columna FK en la BD
```

`user: Users;` // Nombre del campo para acceder al objeto User relacionado el problema es que `categories` me dice que no existe, y claro, veo una tontería declarar un `categories`. Más aún cuando resulta que, tengo toda la info que necesito del usuario en mi JWT y no es relevante para mostrar en la creación y obtención de categorías

Exacto! Si **no necesitas** la relación inversa (y además ya tienes los datos del usuario desde el JWT), puedes simplificar completamente el código. No hay necesidad de complicarlo con relaciones bidireccionales si no las usas.

Solución limpia y práctica para tu caso:

1. Entidad Category (solo con la FK como `user_id`):

typescript

```
import { Entity, PrimaryGeneratedColumn, Column } from 'typeorm';

@Entity({ name: 'CATEGORY' })
export class Category {
  @PrimaryGeneratedColumn({ name: 'ID' })
  id: number;

  @Column({ name: 'NAME' })
  name: string;

  @Column({ name: 'USER_ID' }) // Solo guarda el ID (sin relación
  TypeORM)
  user_id: number; // Tipo number (coincide con el ID del usuario)
}
```

Ventajas:

- **Simple:** No requiere decoradores complejos (`@ManyToOne`, `@JoinColumn`).
- **Eficiente:** No carga datos innecesarios del usuario.
- **JWT-friendly:** Como ya tienes el `user_id` desde el token, puedes usarlo directamente en consultas:

typescript

- // Ejemplo: Obtener categorías del usuario actual (usando el `user_id` del JWT)
- `const categories = await categoryRepository.find({`

- `where: { user_id: req.user.id }, // req.user.id viene del JWT`
 - `});`
-

¿Cuándo usar `@ManyToOne` entonces?

Solo es útil si:

1. Necesitas acceder a **datos completos del usuario** desde una categoría (ej: `category.user.email`).
2. Quieres cargar automáticamente al usuario al consultar categorías (con `eager: true`).

En tu caso, como ya tienes los datos del usuario en el JWT y no los usas en las respuestas de categorías, **evita la relación**. Es sobre-ingeniería.

Bonus: Validación con DTOs

Aprovecha para validar el `user_id` en tus DTOs (si lo recibes en el cuerpo de la petición):

```
typescript

import { IsNumber } from 'class-validator';

export class CreateCategoryDto {
  @IsString()
  name: string;

  @IsNumber()
  user_id: number; // Opcional (si no lo tomas del JWT)
}
```

Conclusión

- Usa `@Column({ name: 'USER_ID' })` si solo necesitas almacenar/referenciar el ID.
- **Omite `@ManyToOne`** si no vas a cargar datos del usuario desde TypeORM.
- **Aprovecha el JWT** para manejar el `user_id` en tus consultas.

Así mantienes tu código **limpio, eficiente y alineado con tus necesidades reales**. □

Ya pero por si a caso, quiero guardarlo con relación pero sin tener que indicarle un users de referencia. Quiero tener algo así como un campo que diga. Oye que este campo se llama users_id y es clave foránea de user. Ah! y si se borra el user, me borras las categorías.

Aquí tienes la implementación completa para el **POST** (create) de tu módulo Category, incluyendo el **controller**, **service**, **repository** e **interface**, siguiendo las mejores prácticas de NestJS y TypeORM:

1. Controller (category.controller.ts)

typescript

```
import { Controller, Post, Body } from '@nestjs/common';
import { CategoryService } from '../category.service';
import { CreateCategoryDto } from '../dto/create-category.dto';

@Controller('categories') // Ruta base: /categories
export class CategoryController {
  constructor(private readonly categoryService: CategoryService) {}

  @Post()
  async create(@Body() createCategoryDto: CreateCategoryDto) {
    return this.categoryService.create(createCategoryDto);
  }
}
```

2. Service (category.service.ts)

typescript

```
import { Injectable } from '@nestjs/common';
import { CategoryRepository } from '../category.repository';
import { CreateCategoryDto } from '../dto/create-category.dto';
import { Category } from '../entities/category.entity';

@Injectable()
export class CategoryService {
  constructor(private readonly categoryRepository: CategoryRepository) {}

  async create(createCategoryDto: CreateCategoryDto):
  Promise<Category> {
    const category =
    this.categoryRepository.create(createCategoryDto);
    return this.categoryRepository.save(category);
  }
}
```

3. Repository Interface (category.repository.interface.ts)

typescript

```
import { Category } from '../entities/category.entity';
import { CreateCategoryDto } from '../dto/create-category.dto';

export interface ICategoryRepository {
  create(categoryData: CreateCategoryDto): Promise<Category>;
  save(category: Category): Promise<Category>;
}
```

4. Repository Implementation (category.repository.ts)

typescript

```
import { Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { Category } from '../entities/category.entity';
import { CreateCategoryDto } from '../dto/create-category.dto';
import { ICategoryRepository } from '../category.repository.interface';

@Injectable()
export class CategoryRepository implements ICategoryRepository {
  constructor(
    @InjectRepository(Category)
    private readonly typeOrmRepository: Repository<Category>,
  ) {}

  async create(categoryData: CreateCategoryDto): Promise<Category> {
    return this.typeOrmRepository.create(categoryData);
  }

  async save(category: Category): Promise<Category> {
    return this.typeOrmRepository.save(category);
  }
}
```

5. DTO (create-category.dto.ts)

typescript

```
import { IsString, IsNotEmpty } from 'class-validator';

export class CreateCategoryDto {
  @IsString()
  @IsNotEmpty()
  name: string;

  @IsNotEmpty()
  user_id: number; // Asignado desde el JWT (sin decorador para evitar
  duplicidad)
}
```

6. Módulo (category.module.ts)

typescript

```
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
```

```
import { CategoryController } from '../category.controller';
import { CategoryService } from '../category.service';
import { CategoryRepository } from '../category.repository';
import { Category } from '../entities/category.entity';

@Module({
  imports: [TypeOrmModule.forFeature([Category])],
  controllers: [CategoryController],
  providers: [
    CategoryService,
    {
      provide: 'ICategoryRepository',
      useClass: CategoryRepository,
    },
  ],
})
export class CategoryModule {}
```

7. Uso con JWT (ejemplo en controller)

Si el `user_id` viene del token (JWT), modifica el controller para inyectarlo:

typescript

```
@Post()
async create(
  @Body() createCategoryDto: CreateCategoryDto,
  @Req() req: Request, // Requiere @nestjs/common
) {
  createCategoryDto.user_id = req.user.id; // Asigna el user_id desde
  el JWT
  return this.categoryService.create(createCategoryDto);
}
```

Flujo de trabajo:

1. **Cliente** envía POST a `/categories` con `{ "name": "Ejemplo" }`.
 2. **Controller** asigna el `user_id` desde el JWT.
 3. **Service** llama al repository para crear y guardar.
 4. **Repository** interactúa con TypeORM para persistir la categoría.
-

Notas clave:

- **Inyección de dependencias:** Usamos `ICategoryRepository` como abstracción (útil para testing).
- **TypeORM:** El `Repository<Category>` ya incluye métodos como `create` y `save`.
- **Seguridad:** El `user_id` se asigna desde el JWT, no desde el body de la petición (evita spoofing).

Si necesitas ajustar algo (como usar un `UUID` en lugar de `number`), ¡avísame! ☐

