

# Domain Driven Design aplicado a NestJs

# ¿Qué es NestJs?

*"A progressive Node.js framework for building efficient, reliable and scalable ~~server-side~~ applications."*

- **Sencillo** para principiantes.
- **Potente** para usuarios avanzados.
- Hecho en **TypeScript**.
- Inyector de dependencias.
- CLI.
- Muy bien **documentado**.



# Building blocks

## Principales

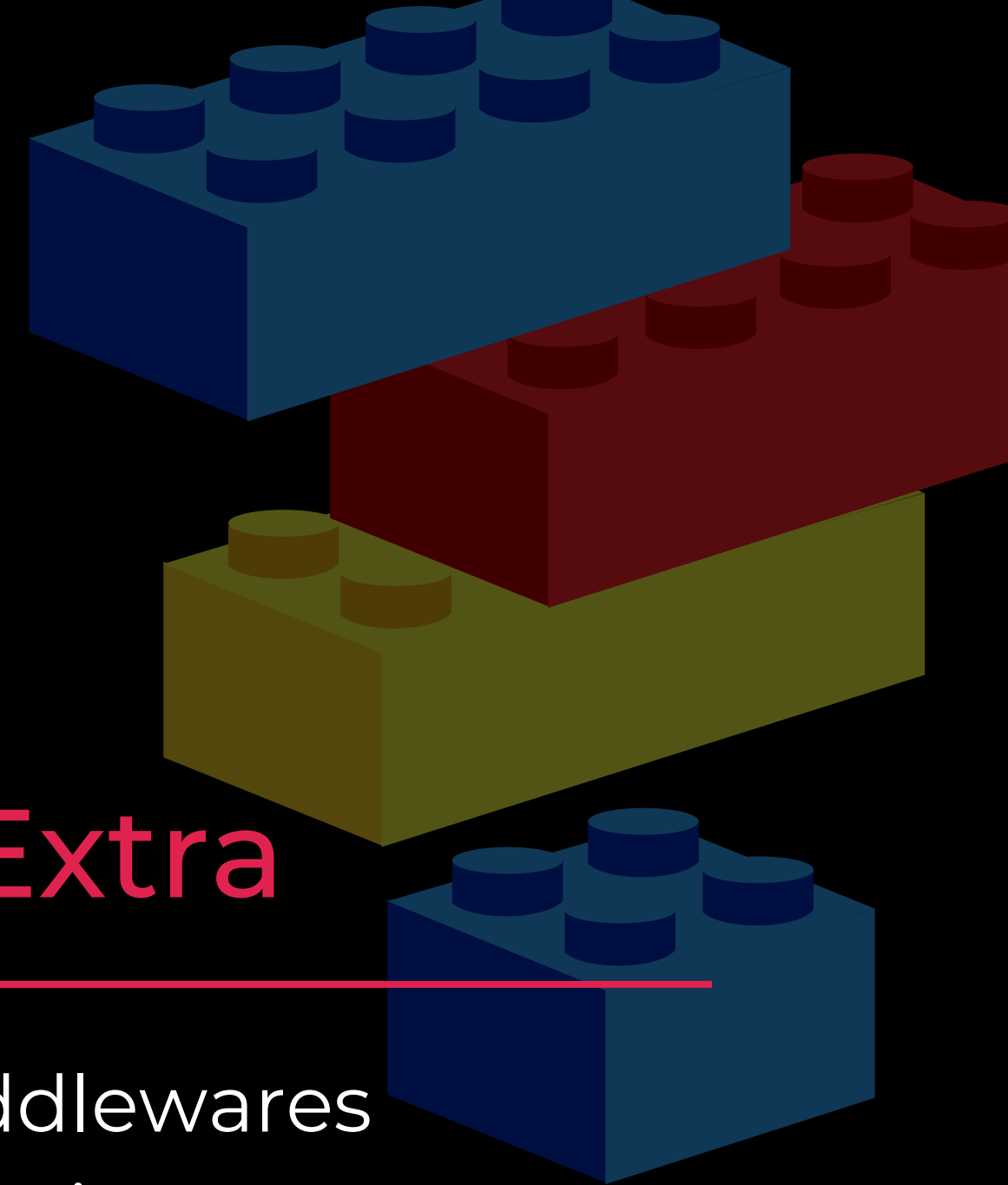
---

Controllers  
Providers  
Modules

## Extra

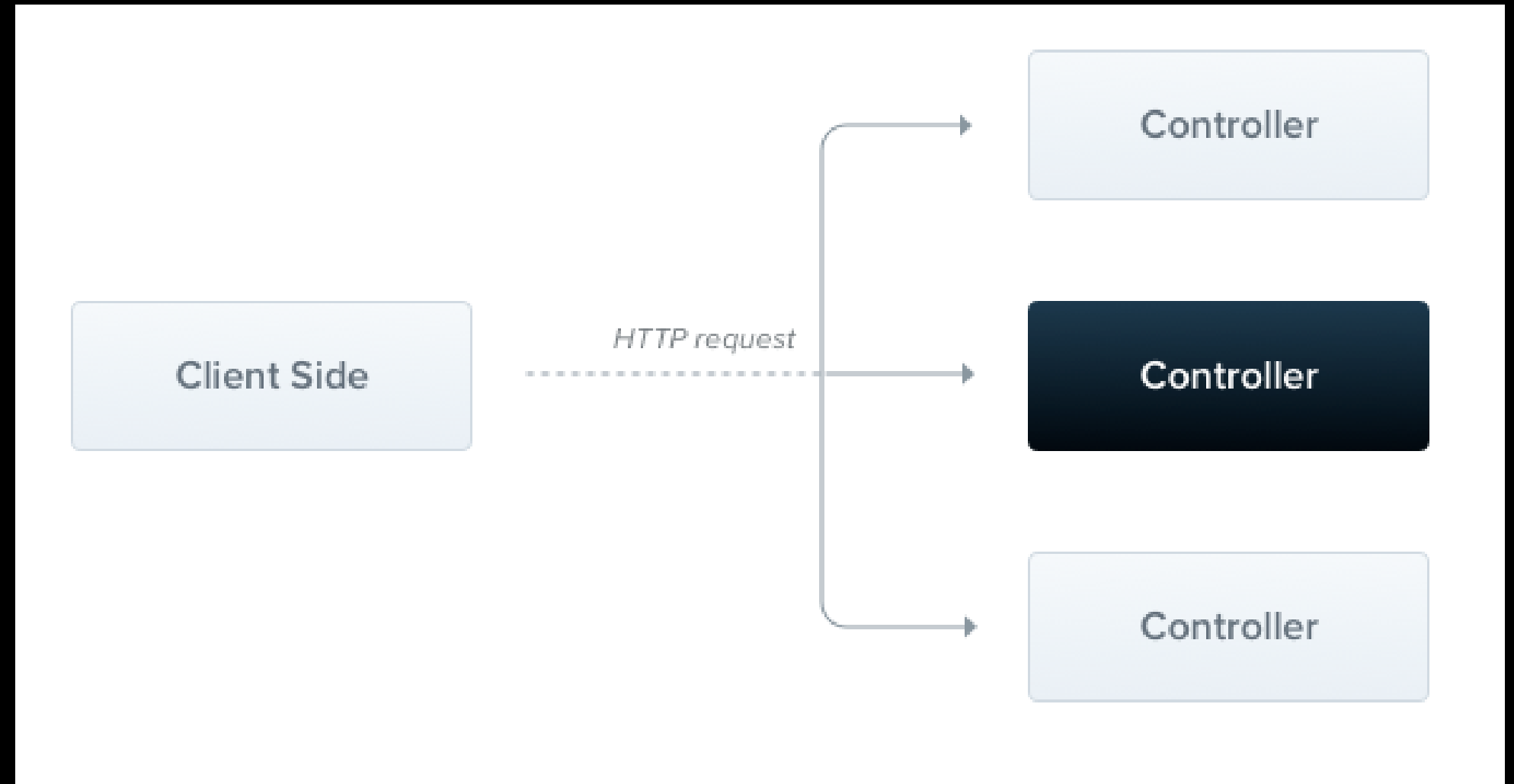
---

Middleware  
Pipes  
Guards  
Exception Filters  
Interceptors




# Controllers

- Son la capa más **externa** de nuestra aplicación.
- Se encargan de recibir **requests** y devolver una **response**.



# Controllers



```
@Controller('cars')
export class CarController {
  constructor(private readonly carService: CarService) {}

  @Get()
  getCars(): Car[] {
    return this.carService.getCars();
  }

  @Get('/:brand')
  getCarsByBrand(@Param('brand') brand: string): Car[] {
    return this.carService.getCarsByBrand(brand);
  }

  @Post()
  addCar(@Body() car: Car): void {
    this.carService.addCar(car);
  }
}
```



```
@Body() // Captura el body de una request
@Query() // Captura la query de una request
@Param() // Captura los parámetros de ruta de una request
@Req() // Captura la request
```

# Providers

- Se pueden **inyectar** como **dependencia**.
- Pueden ser **clases**, **objetos** o **factorias**.
- Muchas de las clases de Nest son providers.

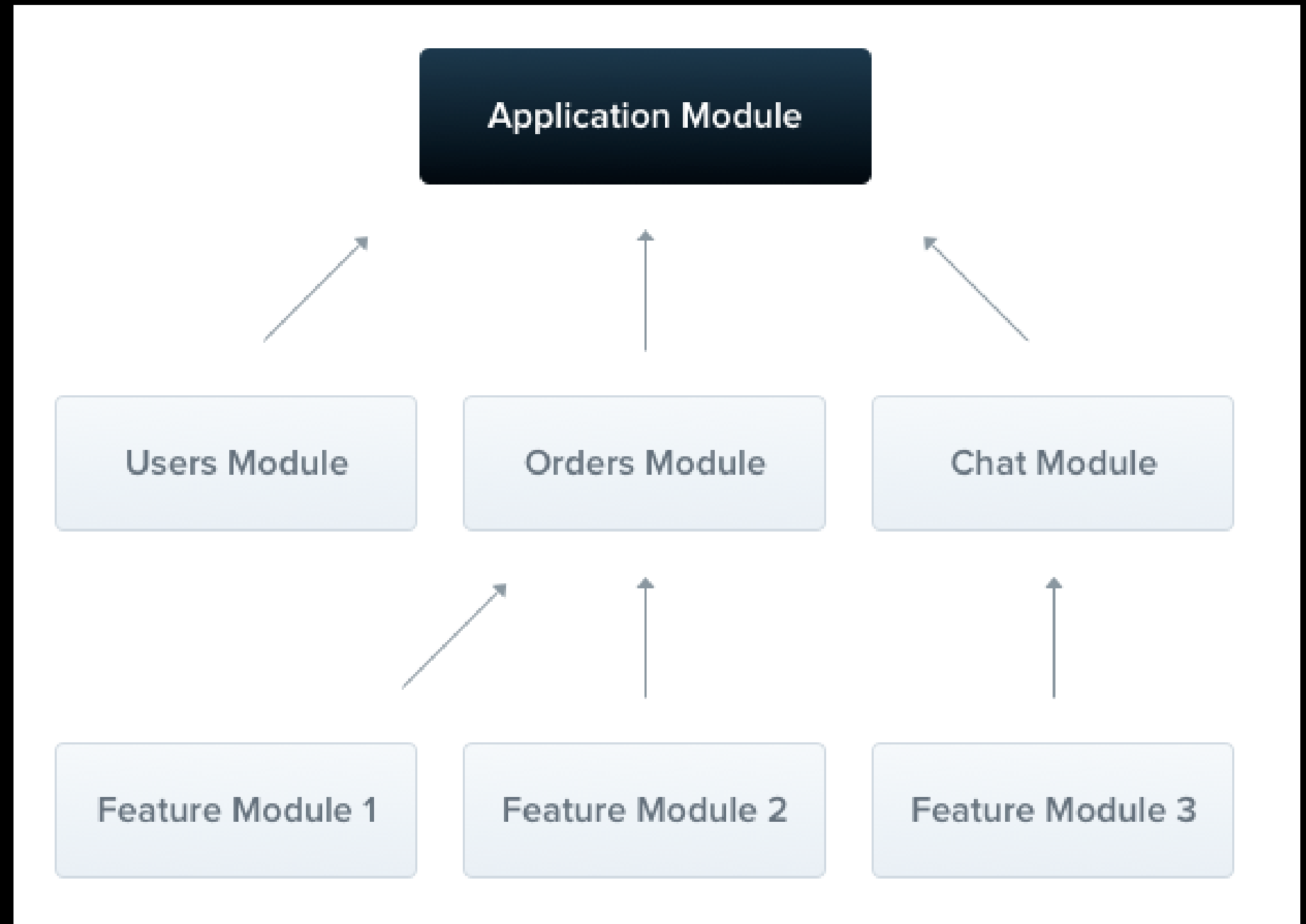
```
@Injectable()
export class CarService {
  getCars(): Car[] {
    return [];
  }

  addCar(car: Car): void {
    return;
  }

  getCarsByBrand(brand: string): Car[] {
    return [];
  }
}
```

# Modules

- Definen la **estructura** de nuestra aplicación.
- La idea es **encapsular** funcionalidades del mismo tipo.
- Todas las aplicaciones tienen **al menos un módulo**.



# Modules

```
@Module({
  imports: [],
  controllers: [CarController],
  providers: [CarService],
})
export class AppModule {}
```

<code>providers</code>	the providers that will be instantiated by the Nest injector and that may be shared at least across this module
<code>controllers</code>	the set of controllers defined in this module which have to be instantiated
<code>imports</code>	the list of imported modules that export the providers which are required in this module
<code>exports</code>	the subset of <code>providers</code> that are provided by this module and should be available in other modules which import this module. You can use either the provider itself or just its token ( <code>provide</code> value)



# Modules en profundidad

Hay 4 formas de indicarle a Nest como inyectar una dependencia.

useClass

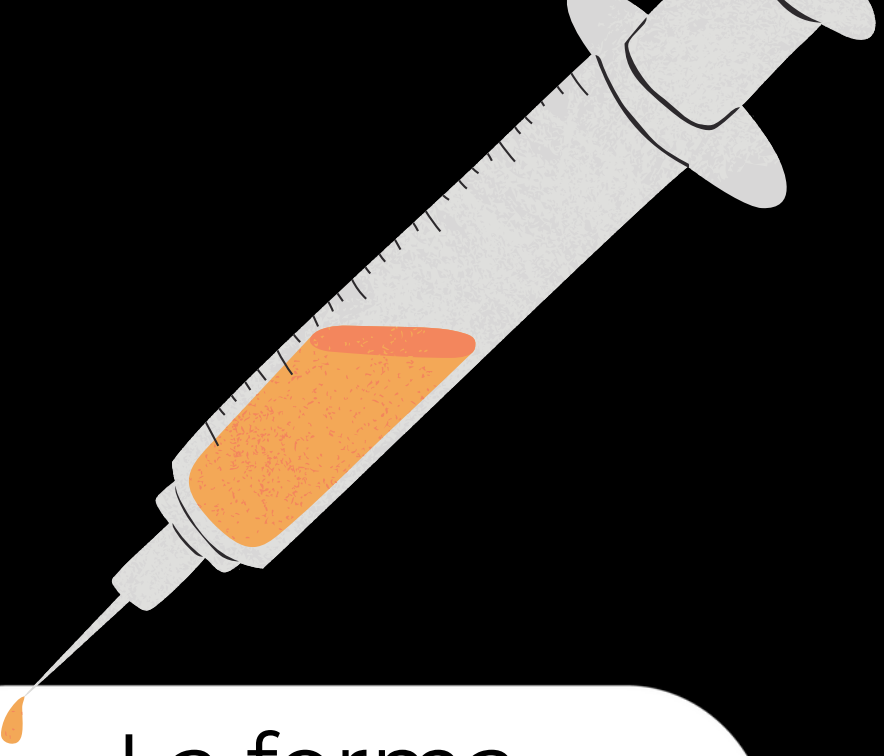
useValue

useFactory

useExisting

```
@Module({
  imports: [],
  controllers: [CarController],
  providers: [CarService],
})
export class AppModule {}
```

```
@Module({
  imports: [],
  controllers: [CarController],
  providers: [
    {
      provide: CarService,
      useClass: CarService,
    },
  ],
})
export class AppModule {}
```




La forma estándar es un shorthand de useClass

NOTE: La propiedad provide puede recibir una clase, un string o un Symbol

# useClass

Es el caso de uso más común, permite inyectar una clase para el token provisto.



```
@Module({
  imports: [],
  controllers: [CarController],
  providers: [
    {
      provide: CarService,
      useClass: CarService,
    },
  ],
})
export class AppModule {}
```

# useValue

Permite inyectar un valor para el token provisto, siendo este valor un valor primitivo, objeto, instancia o función.

```
const myServiceConfig = Symbol('config');

@Module({
  imports: [],
  controllers: [CarController],
  providers: [
    {
      provide: myServiceConfig,
      useValue: {
        url: 'http://hello.com',
      },
    },
  ],
})
export class AppModule {}
```

# useFactory

Permite ejecutar una función que será invocada para resolver una dependencia.

```
const myServiceConfig = Symbol('config');

@Module({
  imports: [],
  controllers: [CarController],
  providers: [
    {
      provide: myServiceConfig,
      useValue: {
        url: 'http://hello.com',
      },
    },
    {
      provide: MyService,
      useFactory: (config: { url: string }) => {
        return new SomeService(config);
      },
      inject: [myServiceConfig],
    },
  ],
})
export class AppModule {}
```

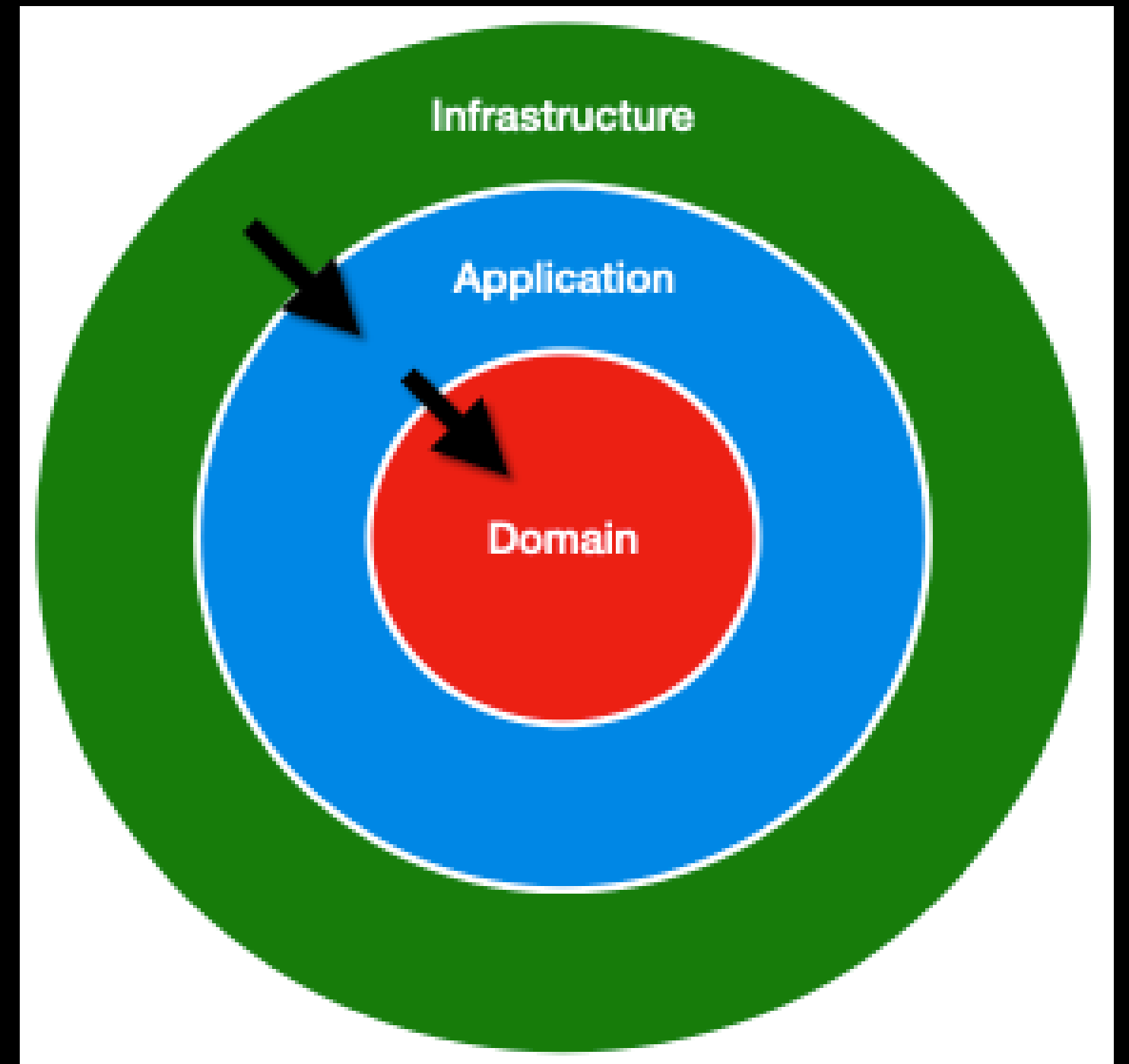
# useExisting

Es un simple alias, permite a una clase ser inyectada mediante un token diferente

```
@Global()  
@Module({  
  imports: [CqrsModule],  
  providers: [  
    {  
      provide: CommandBus,  
      useExisting: NestCommandBus,  
    },  
    {  
      provide: QueryBus,  
      useExisting: NestQueryBus,  
    },  
  ],  
  exports: [CqrsModule, CommandBus, QueryBus],  
})  
export class CqrsWrapperModule {}
```

# DDD

DDD (Domain driven design) es un conjunto de **patrones** y **buenas prácticas** que nos permiten desarrollar software de forma **eficiente, elegante, resistente al cambio** y **testeable**.



# Domain Layer

Es el **núcleo** de nuestra aplicación, no conoce ni depende de nada de lo que hay fuera de este. Contiene principalmente **lógica de negocio**.

- Value objects.
- Entities.
- Aggregates.
- Events
- Queries.
- Commands.
- Domain exceptions.
- Repository abstractions.

```


└─ domain
  └─ exceptions
    ├── book-not-found.domain-exception.ts
    └── some-other.domain-exception.ts
  └─ queries
    └── get-books.query.ts
  └─ repositories
    └── get-books.repository.ts
  └─ value-objects
    └── book.value-object.ts
```

# Application layer

**Orquesta** las piezas de dominio para comunicarse con el **exterior**. Ej. Manejo de transacciones en base de datos, envío de emails o manejo de ficheros.

A partir del nombre de los casos de uso se pueden conocer las funcionalidades que tiene una aplicación

- Use cases.
- Transaction-related management.
- Orchestrating domain entities.



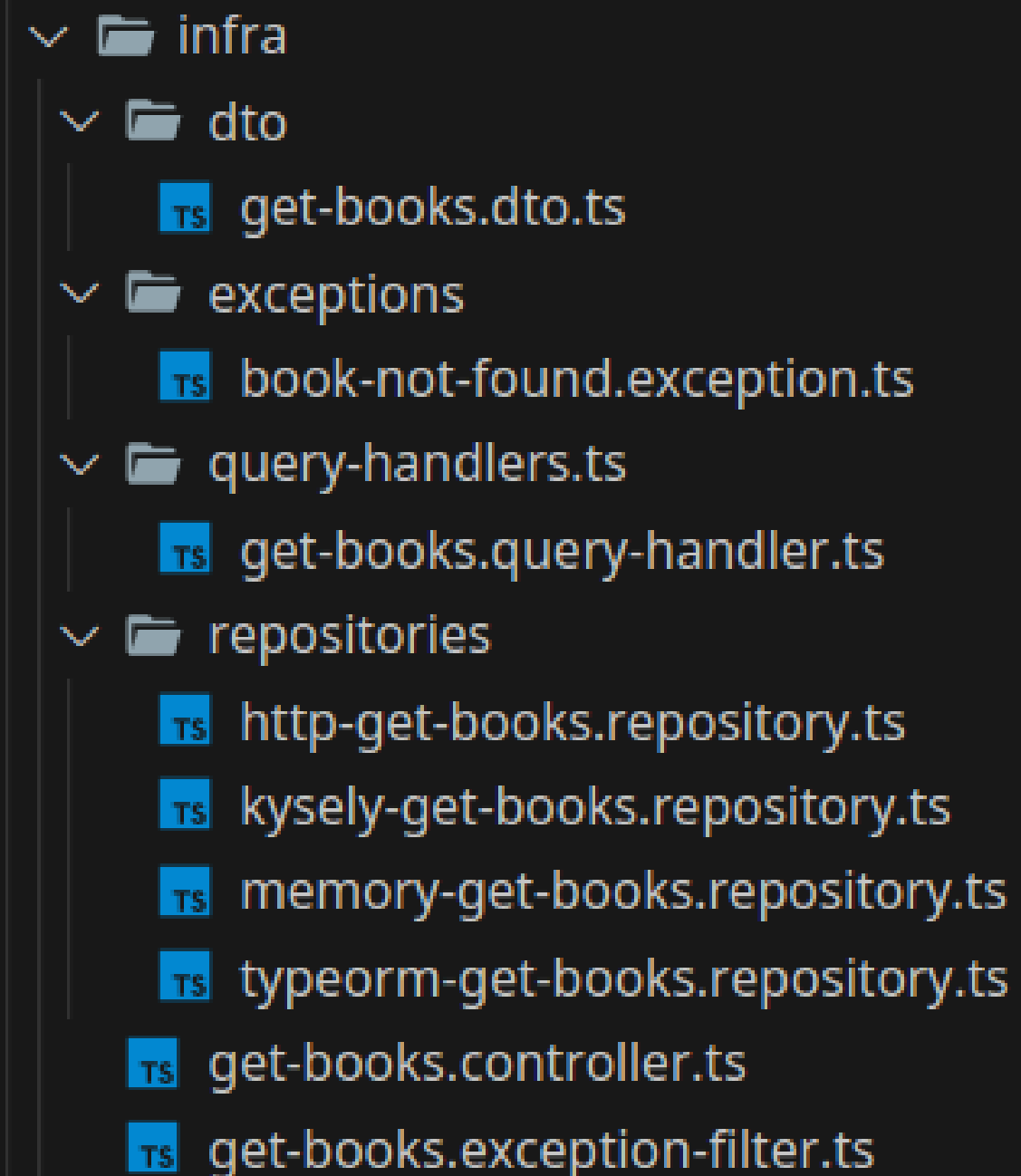
```
application / use-cases
  TS get-books.use-case.ts
```



# Infrastructure Layer

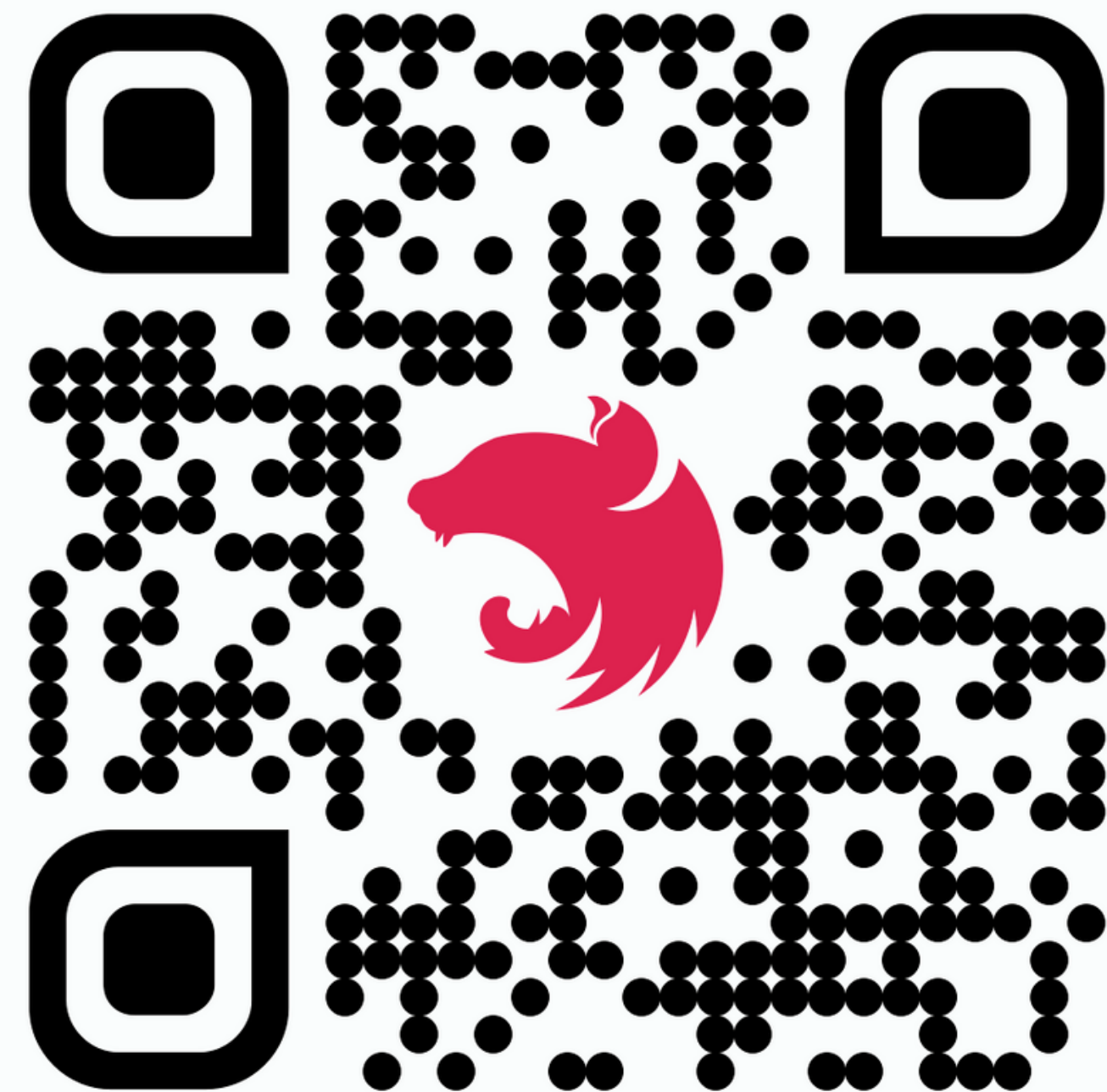
Es la capa más **externa**. En esta capa vamos a encontrar principalmente **implementaciones** concretas de las abstracciones que tenemos en nuestras capas de aplicación y dominio.

- Platform-related.
- Persistence-related.
- 3rd party and framework.
- Repository implementation.
- Dtos and validation.
- Http exceptions.
- Query handlers.
- Command handlers.



```
infra
├── dto
│   └── get-books.dto.ts
├── exceptions
│   └── book-not-found.exception.ts
├── query-handlers.ts
│   └── get-books.query-handler.ts
├── repositories
│   ├── http-get-books.repository.ts
│   ├── kysely-get-books.repository.ts
│   ├── memory-get-books.repository.ts
│   └── typeorm-get-books.repository.ts
├── get-books.controller.ts
└── get-books.exception-filter.ts
```

# Gracias!



REPO