Database

Nest is database agnostic, allowing you to easily integrate with any SQL or NoSQL database. You have a number of options available to you, depending on your preferences. At the most general level, connecting Nest to a database is simply a matter of loading an appropriate Node.js driver for the database, just as you would with Express or Fastify.

You can also directly use any general purpose Node.js database integration **library** or ORM, such as MikroORM (see MikroORM recipe), Sequelize (see Sequelize integration), Knex.js (see Knex.js tutorial), TypeORM, and Prisma (see Prisma recipe), to operate at a higher level of abstraction.

For convenience, Nest provides tight integration with TypeORM and Sequelize out-of-the-box with the `@nestjs/typeorm` and `@nestjs/sequelize` packages respectively, which we'll cover in the current chapter, and Mongoose with `@nestjs/mongoose`, which is covered in this chapter. These integrations provide additional NestJS-specific features, such as model/repository injection, testability, and asynchronous configuration to make accessing your chosen database even easier.

TypeORM Integration

For integrating with SQL and NoSQL databases, Nest provides the `@nestjs/typeorm` package. TypeORM is the most mature Object Relational Mapper (ORM) available for TypeScript. Since it's written in TypeScript, it integrates well with the Nest framework.

To begin using it, we first install the required dependencies. In this chapter, we'll demonstrate using the popular MySQL Relational DBMS, but TypeORM provides support for many relational databases, such as PostgreSQL, Oracle, Microsoft SQL Server, SQLite, and even NoSQL databases like MongoDB. The procedure we walk through in this chapter will be the same for any database supported by TypeORM. You'll simply need to install the associated client API libraries for your selected database.

```
$ npm install --save @nestjs/typeorm typeorm mysql2
```

Once the installation process is complete, we can import the `TypeOrmModule` into the root `AppModule`.

```
@@filename(app.module)
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';

@Module({
  imports: [
    TypeOrmModule.forRoot({
      type: 'mysql',
      host: 'localhost',
      port: 3306,
      username: 'root',
      password: 'root',
      database: 'test',
      entities: [],
```

```
      synchronize: true,
    }),
  ],
})
export class AppModule {}
```

> warning **Warning** Setting `synchronize: true` shouldn't be used in production - otherwise you can lose production data.

The `forRoot()` method supports all the configuration properties exposed by the `DataSource` constructor from the TypeORM package. In addition, there are several extra configuration properties described below.

`retryAttempts`	Number of attempts to connect to the database (default: `10`)
`retryDelay`	Delay between connection retry attempts (ms) (default: `3000`)
`autoLoadEntities`	If `true`, entities will be loaded automatically (default: `false`)

> info **Hint** Learn more about the data source options here.

Once this is done, the TypeORM `DataSource` and `EntityManager` objects will be available to inject across the entire project (without needing to import any modules), for example:

```
@@filename(app.module)
import { DataSource } from 'typeorm';

@Module({
  imports: [TypeOrmModule.forRoot(), UsersModule],
})
export class AppModule {
  constructor(private dataSource: DataSource) {}
}
@@switch
import { DataSource } from 'typeorm';

@Dependencies(DataSource)
@Module({
  imports: [TypeOrmModule.forRoot(), UsersModule],
})
export class AppModule {
  constructor(dataSource) {
    this.dataSource = dataSource;
  }
}
```

Repository pattern

TypeORM supports the **repository design pattern**, so each entity has its own repository. These repositories can be obtained from the database data source.

To continue the example, we need at least one entity. Let's define the `User` entity.

```
@@filename(user.entity)
import { Entity, Column, PrimaryGeneratedColumn } from 'typeorm';

@Entity()
export class User {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  firstName: string;

  @Column()
  lastName: string;

  @Column({ default: true })
  isActive: boolean;
}
```

> info **Hint** Learn more about entities in the TypeORM documentation.

The `User` entity file sits in the `users` directory. This directory contains all files related to the `UsersModule`. You can decide where to keep your model files, however, we recommend creating them near their **domain**, in the corresponding module directory.

To begin using the `User` entity, we need to let TypeORM know about it by inserting it into the `entities` array in the module `forRoot()` method options (unless you use a static glob path):

```
@@filename(app.module)
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
import { User } from './users/user.entity';

@Module({
  imports: [
    TypeOrmModule.forRoot({
      type: 'mysql',
      host: 'localhost',
      port: 3306,
      username: 'root',
      password: 'root',
      database: 'test',
      entities: [User],
      synchronize: true,
    }),
  ],
})
export class AppModule {}
```

Next, let's look at the `UsersModule`:

```
@@filename(users.module)
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
import { UsersService } from './users.service';
import { UsersController } from './users.controller';
import { User } from './user.entity';

@Module({
  imports: [TypeOrmModule.forFeature([User])],
  providers: [UsersService],
  controllers: [UsersController],
})
export class UsersModule {}
```

This module uses the `forFeature()` method to define which repositories are registered in the current scope. With that in place, we can inject the `UsersRepository` into the `UsersService` using the `@InjectRepository()` decorator:

```
@@filename(users.service)
import { Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { User } from './user.entity';

@Injectable()
export class UsersService {
  constructor(
    @InjectRepository(User)
    private usersRepository: Repository<User>,
  ) {}

  findAll(): Promise<User[]> {
    return this.usersRepository.find();
  }

  findOne(id: number): Promise<User | null> {
    return this.usersRepository.findOneBy({ id });
  }

  async remove(id: number): Promise<void> {
    await this.usersRepository.delete(id);
  }
}
@@switch
import { Injectable, Dependencies } from '@nestjs/common';
import { getRepositoryToken } from '@nestjs/typeorm';
import { User } from './user.entity';
```

```
@Injectable()
@Dependencies(getRepositoryToken(User))
export class UsersService {
  constructor(usersRepository) {
    this.usersRepository = usersRepository;
  }

  findAll() {
    return this.usersRepository.find();
  }

  findOne(id) {
    return this.usersRepository.findOneBy({ id });
  }

  async remove(id) {
    await this.usersRepository.delete(id);
  }
}
```

> warning **Notice** Don't forget to import the `UsersModule` into the root `AppModule`.

If you want to use the repository outside of the module which imports `TypeOrmModule.forFeature`, you'll need to re-export the providers generated by it. You can do this by exporting the whole module, like this:

```
@@filename(users.module)
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
import { User } from './user.entity';

@Module({
  imports: [TypeOrmModule.forFeature([User])],
  exports: [TypeOrmModule]
})
export class UsersModule {}
```

Now if we import `UsersModule` in `UserHttpModule`, we can use `@InjectRepository(User)` in the providers of the latter module.

```
@@filename(users-http.module)
import { Module } from '@nestjs/common';
import { UsersModule } from './users.module';
import { UsersService } from './users.service';
import { UsersController } from './users.controller';

@Module({
  imports: [UsersModule],
  providers: [UsersService],
```

```
  controllers: [UsersController]
})
export class UserHttpModule {}
```

Relations

Relations are associations established between two or more tables. Relations are based on common fields from each table, often involving primary and foreign keys.

There are three types of relations:

`One-to-one`	Every row in the primary table has one and only one associated row in the foreign table. Use the `@OneToOne()` decorator to define this type of relation.
`One-to-many / Many-to-one`	Every row in the primary table has one or more related rows in the foreign table. Use the `@OneToMany()` and `@ManyToOne()` decorators to define this type of relation.
`Many-to-many`	Every row in the primary table has many related rows in the foreign table, and every record in the foreign table has many related rows in the primary table. Use the `@ManyToMany()` decorator to define this type of relation.

To define relations in entities, use the corresponding **decorators**. For example, to define that each `User` can have multiple photos, use the `@OneToMany()` decorator.

```
@@filename(user.entity)
import { Entity, Column, PrimaryGeneratedColumn, OneToMany } from
'typeorm';
import { Photo } from '../photos/photo.entity';

@Entity()
export class User {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  firstName: string;

  @Column()
  lastName: string;

  @Column({ default: true })
  isActive: boolean;

  @OneToMany(type => Photo, photo => photo.user)
  photos: Photo[];
}
```

> info **Hint** To learn more about relations in TypeORM, visit the TypeORM documentation.

Auto-load entities

Manually adding entities to the `entities` array of the data source options can be tedious. In addition, referencing entities from the root module breaks application domain boundaries and causes leaking implementation details to other parts of the application. To address this issue, an alternative solution is provided. To automatically load entities, set the `autoLoadEntities` property of the configuration object (passed into the `forRoot()` method) to `true`, as shown below:

```
@@filename(app.module)
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';

@Module({
  imports: [
    TypeOrmModule.forRoot({
      ...
      autoLoadEntities: true,
    }),
  ],
})
export class AppModule {}
```

With that option specified, every entity registered through the `forFeature()` method will be automatically added to the `entities` array of the configuration object.

> warning **Warning** Note that entities that aren't registered through the `forFeature()` method, but are only referenced from the entity (via a relationship), won't be included by way of the `autoLoadEntities` setting.

Separating entity definition

You can define an entity and its columns right in the model, using decorators. But some people prefer to define entities and their columns inside separate files using the "entity schemas".

```
import { EntitySchema } from 'typeorm';
import { User } from './user.entity';

export const UserSchema = new EntitySchema<User>({
  name: 'User',
  target: User,
  columns: {
    id: {
      type: Number,
      primary: true,
      generated: true,
    },
    firstName: {
```

```
      type: String,
    },
    lastName: {
      type: String,
    },
    isActive: {
      type: Boolean,
      default: true,
    },
  },
  relations: {
    photos: {
      type: 'one-to-many',
      target: 'Photo', // the name of the PhotoSchema
    },
  },
});
```

> warning error **Warning** If you provide the `target` option, the `name` option value has to be the same as the name of the target class. If you do not provide the `target` you can use any name.

Nest allows you to use an `EntitySchema` instance wherever an `Entity` is expected, for example:

```
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
import { UserSchema } from './user.schema';
import { UsersController } from './users.controller';
import { UsersService } from './users.service';

@Module({
  imports: [TypeOrmModule.forFeature([UserSchema])],
  providers: [UsersService],
  controllers: [UsersController],
})
export class UsersModule {}
```

TypeORM Transactions

A database transaction symbolizes a unit of work performed within a database management system against a database, and treated in a coherent and reliable way independent of other transactions. A transaction generally represents any change in a database (learn more).

There are many different strategies to handle TypeORM transactions. We recommend using the `QueryRunner` class because it gives full control over the transaction.

First, we need to inject the `DataSource` object into a class in the normal way:

```
@Injectable()
export class UsersService {
```

```
    constructor(private dataSource: DataSource) {}
}
```

> info **Hint** The `DataSource` class is imported from the `typeorm` package.

Now, we can use this object to create a transaction.

```
async createMany(users: User[]) {
  const queryRunner = this.dataSource.createQueryRunner();

  await queryRunner.connect();
  await queryRunner.startTransaction();
  try {
    await queryRunner.manager.save(users[0]);
    await queryRunner.manager.save(users[1]);

    await queryRunner.commitTransaction();
  } catch (err) {
    // since we have errors lets rollback the changes we made
    await queryRunner.rollbackTransaction();
  } finally {
    // you need to release a queryRunner which was manually instantiated
    await queryRunner.release();
  }
}
```

> info **Hint** Note that the `dataSource` is used only to create the `QueryRunner`. However, to test this class would require mocking the entire `DataSource` object (which exposes several methods). Thus, we recommend using a helper factory class (e.g., `QueryRunnerFactory`) and defining an interface with a limited set of methods required to maintain transactions. This technique makes mocking these methods pretty straightforward.

Alternatively, you can use the callback-style approach with the `transaction` method of the `DataSource` object (read more).

```
async createMany(users: User[]) {
  await this.dataSource.transaction(async manager => {
    await manager.save(users[0]);
    await manager.save(users[1]);
  });
}
```

Subscribers

With TypeORM subscribers, you can listen to specific entity events.

```
import {
  DataSource,
  EntitySubscriberInterface,
  EventSubscriber,
  InsertEvent,
} from 'typeorm';
import { User } from './user.entity';

@EventSubscriber()
export class UserSubscriber implements EntitySubscriberInterface<User> {
  constructor(dataSource: DataSource) {
    dataSource.subscribers.push(this);
  }

  listenTo() {
    return User;
  }

  beforeInsert(event: InsertEvent<User>) {
    console.log(`BEFORE USER INSERTED: `, event.entity);
  }
}
```

> error **Warning** Event subscribers can not be request-scoped.

Now, add the `UserSubscriber` class to the `providers` array:

```
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
import { User } from './user.entity';
import { UsersController } from './users.controller';
import { UsersService } from './users.service';
import { UserSubscriber } from './user.subscriber';

@Module({
  imports: [TypeOrmModule.forFeature([User])],
  providers: [UsersService, UserSubscriber],
  controllers: [UsersController],
})
export class UsersModule {}
```

> info **Hint** Learn more about entity subscribers here.

Migrations

Migrations provide a way to incrementally update the database schema to keep it in sync with the application's data model while preserving existing data in the database. To generate, run, and revert migrations, TypeORM provides a dedicated CLI.

Migration classes are separate from the Nest application source code. Their lifecycle is maintained by the TypeORM CLI. Therefore, you are not able to leverage dependency injection and other Nest specific features with migrations. To learn more about migrations, follow the guide in the TypeORM documentation.

Multiple databases

Some projects require multiple database connections. This can also be achieved with this module. To work with multiple connections, first create the connections. In this case, data source naming becomes **mandatory**.

Suppose you have an `Album` entity stored in its own database.

```
const defaultOptions = {
  type: 'postgres',
  port: 5432,
  username: 'user',
  password: 'password',
  database: 'db',
  synchronize: true,
};

@Module({
  imports: [
    TypeOrmModule.forRoot({
      ...defaultOptions,
      host: 'user_db_host',
      entities: [User],
    }),
    TypeOrmModule.forRoot({
      ...defaultOptions,
      name: 'albumsConnection',
      host: 'album_db_host',
      entities: [Album],
    }),
  ],
})
export class AppModule {}
```

warning **Notice** If you don't set the `name` for a data source, its name is set to `default`. Please note that you shouldn't have multiple connections without a name, or with the same name, otherwise they will get overridden.

warning **Notice** If you are using `TypeOrmModule.forRootAsync`, you have to **also** set the data source name outside `useFactory`. For example:

```
TypeOrmModule.forRootAsync({
  name: 'albumsConnection',
  useFactory: ...,
```

```
    inject: ...,
  }),
```

See this issue for more details.

At this point, you have `User` and `Album` entities registered with their own data source. With this setup, you have to tell the `TypeOrmModule.forFeature()` method and the `@InjectRepository()` decorator which data source should be used. If you do not pass any data source name, the `default` data source is used.

```
@Module({
  imports: [
    TypeOrmModule.forFeature([User]),
    TypeOrmModule.forFeature([Album], 'albumsConnection'),
  ],
})
export class AppModule {}
```

You can also inject the `DataSource` or `EntityManager` for a given data source:

```
@Injectable()
export class AlbumsService {
  constructor(
    @InjectDataSource('albumsConnection')
    private dataSource: DataSource,
    @InjectEntityManager('albumsConnection')
    private entityManager: EntityManager,
  ) {}
}
```

It's also possible to inject any `DataSource` to the providers:

```
@Module({
  providers: [
    {
      provide: AlbumsService,
      useFactory: (albumsConnection: DataSource) => {
        return new AlbumsService(albumsConnection);
      },
      inject: [getDataSourceToken('albumsConnection')],
    },
  ],
})
export class AlbumsModule {}
```

Testing

When it comes to unit testing an application, we usually want to avoid making a database connection, keeping our test suites independent and their execution process as fast as possible. But our classes might depend on repositories that are pulled from the data source (connection) instance. How do we handle that? The solution is to create mock repositories. In order to achieve that, we set up custom providers. Each registered repository is automatically represented by an `<EntityName>Repository` token, where `EntityName` is the name of your entity class.

The `@nestjs/typeorm` package exposes the `getRepositoryToken()` function which returns a prepared token based on a given entity.

```
@Module({
  providers: [
    UsersService,
    {
      provide: getRepositoryToken(User),
      useValue: mockRepository,
    },
  ],
})
export class UsersModule {}
```

Now a substitute `mockRepository` will be used as the `UsersRepository`. Whenever any class asks for `UsersRepository` using an `@InjectRepository()` decorator, Nest will use the registered `mockRepository` object.

Async configuration

You may want to pass your repository module options asynchronously instead of statically. In this case, use the `forRootAsync()` method, which provides several ways to deal with async configuration.

One approach is to use a factory function:

```
TypeOrmModule.forRootAsync({
  useFactory: () => ({
    type: 'mysql',
    host: 'localhost',
    port: 3306,
    username: 'root',
    password: 'root',
    database: 'test',
    entities: [],
    synchronize: true,
  }),
});
```

Our factory behaves like any other asynchronous provider (e.g., it can be `async` and it's able to inject dependencies through `inject`).

```
TypeOrmModule.forRootAsync({
  imports: [ConfigModule],
  useFactory: (configService: ConfigService) => ({
    type: 'mysql',
    host: configService.get('HOST'),
    port: +configService.get('PORT'),
    username: configService.get('USERNAME'),
    password: configService.get('PASSWORD'),
    database: configService.get('DATABASE'),
    entities: [],
    synchronize: true,
  }),
  inject: [ConfigService],
});
```

Alternatively, you can use the `useClass` syntax:

```
TypeOrmModule.forRootAsync({
  useClass: TypeOrmConfigService,
});
```

The construction above will instantiate `TypeOrmConfigService` inside `TypeOrmModule` and use it to provide an options object by calling `createTypeOrmOptions()`. Note that this means that the `TypeOrmConfigService` has to implement the `TypeOrmOptionsFactory` interface, as shown below:

```
@Injectable()
export class TypeOrmConfigService implements TypeOrmOptionsFactory {
  createTypeOrmOptions(): TypeOrmModuleOptions {
    return {
      type: 'mysql',
      host: 'localhost',
      port: 3306,
      username: 'root',
      password: 'root',
      database: 'test',
      entities: [],
      synchronize: true,
    };
  }
}
```

In order to prevent the creation of `TypeOrmConfigService` inside `TypeOrmModule` and use a provider imported from a different module, you can use the `useExisting` syntax.

```
TypeOrmModule.forRootAsync({
  imports: [ConfigModule],
  useExisting: ConfigService,
});
```

This construction works the same as `useClass` with one critical difference - `TypeOrmModule` will lookup imported modules to reuse an existing `ConfigService` instead of instantiating a new one.

> info **Hint** Make sure that the `name` property is defined at the same level as the `useFactory`, `useClass`, or `useValue` property. This will allow Nest to properly register the data source under the appropriate injection token.

Custom DataSource Factory

In conjunction with async configuration using `useFactory`, `useClass`, or `useExisting`, you can optionally specify a `dataSourceFactory` function which will allow you to provide your own TypeORM data source rather than allowing `TypeOrmModule` to create the data source.

`dataSourceFactory` receives the TypeORM `DataSourceOptions` configured during async configuration using `useFactory`, `useClass`, or `useExisting` and returns a `Promise` that resolves a TypeORM `DataSource`.

```
TypeOrmModule.forRootAsync({
  imports: [ConfigModule],
  inject: [ConfigService],
  // Use useFactory, useClass, or useExisting
  // to configure the DataSourceOptions.
  useFactory: (configService: ConfigService) => ({
    type: 'mysql',
    host: configService.get('HOST'),
    port: +configService.get('PORT'),
    username: configService.get('USERNAME'),
    password: configService.get('PASSWORD'),
    database: configService.get('DATABASE'),
    entities: [],
    synchronize: true,
  }),
  // dataSource receives the configured DataSourceOptions
  // and returns a Promise<DataSource>.
  dataSourceFactory: async (options) => {
    const dataSource = await new DataSource(options).initialize();
    return dataSource;
  },
});
```

> info **Hint** The `DataSource` class is imported from the `typeorm` package.

Example

A working example is available here.

Sequelize Integration

An alternative to using TypeORM is to use the Sequelize ORM with the `@nestjs/sequelize` package. In addition, we leverage the sequelize-typescript package which provides a set of additional decorators to declaratively define entities.

To begin using it, we first install the required dependencies. In this chapter, we'll demonstrate using the popular MySQL Relational DBMS, but Sequelize provides support for many relational databases, such as PostgreSQL, MySQL, Microsoft SQL Server, SQLite, and MariaDB. The procedure we walk through in this chapter will be the same for any database supported by Sequelize. You'll simply need to install the associated client API libraries for your selected database.

```
$ npm install --save @nestjs/sequelize sequelize sequelize-typescript
mysql2
$ npm install --save-dev @types/sequelize
```

Once the installation process is complete, we can import the `SequelizeModule` into the root `AppModule`.

```
@@filename(app.module)
import { Module } from '@nestjs/common';
import { SequelizeModule } from '@nestjs/sequelize';

@Module({
  imports: [
    SequelizeModule.forRoot({
      dialect: 'mysql',
      host: 'localhost',
      port: 3306,
      username: 'root',
      password: 'root',
      database: 'test',
      models: [],
    }),
  ],
})
export class AppModule {}
```

The `forRoot()` method supports all the configuration properties exposed by the Sequelize constructor (read more). In addition, there are several extra configuration properties described below.

`retryAttempts`	Number of attempts to connect to the database (default: `10`)
`retryDelay`	Delay between connection retry attempts (ms) (default: `3000`)
`autoLoadModels`	If `true`, models will be loaded automatically (default: `false`)
`keepConnectionAlive`	If `true`, connection will not be closed on the application shutdown (default:

	`false`)
`synchronize`	If `true`, automatically loaded models will be synchronized (default: `true`)

Once this is done, the `Sequelize` object will be available to inject across the entire project (without needing to import any modules), for example:

```
@@filename(app.service)
import { Injectable } from '@nestjs/common';
import { Sequelize } from 'sequelize-typescript';

@Injectable()
export class AppService {
  constructor(private sequelize: Sequelize) {}
}
@@switch
import { Injectable } from '@nestjs/common';
import { Sequelize } from 'sequelize-typescript';

@Dependencies(Sequelize)
@Injectable()
export class AppService {
  constructor(sequelize) {
    this.sequelize = sequelize;
  }
}
```

Models

Sequelize implements the Active Record pattern. With this pattern, you use model classes directly to interact with the database. To continue the example, we need at least one model. Let's define the `User` model.

```
@@filename(user.model)
import { Column, Model, Table } from 'sequelize-typescript';

@Table
export class User extends Model {
  @Column
  firstName: string;

  @Column
  lastName: string;

  @Column({ defaultValue: true })
  isActive: boolean;
}
```

> info **Hint** Learn more about the available decorators here.

The `User` model file sits in the `users` directory. This directory contains all files related to the `UsersModule`. You can decide where to keep your model files, however, we recommend creating them near their **domain**, in the corresponding module directory.

To begin using the `User` model, we need to let Sequelize know about it by inserting it into the `models` array in the module `forRoot()` method options:

```
@@filename(app.module)
import { Module } from '@nestjs/common';
import { SequelizeModule } from '@nestjs/sequelize';
import { User } from './users/user.model';

@Module({
  imports: [
    SequelizeModule.forRoot({
      dialect: 'mysql',
      host: 'localhost',
      port: 3306,
      username: 'root',
      password: 'root',
      database: 'test',
      models: [User],
    }),
  ],
})
export class AppModule {}
```

Next, let's look at the `UsersModule`:

```
@@filename(users.module)
import { Module } from '@nestjs/common';
import { SequelizeModule } from '@nestjs/sequelize';
import { User } from './user.model';
import { UsersController } from './users.controller';
import { UsersService } from './users.service';

@Module({
  imports: [SequelizeModule.forFeature([User])],
  providers: [UsersService],
  controllers: [UsersController],
})
export class UsersModule {}
```

This module uses the `forFeature()` method to define which models are registered in the current scope. With that in place, we can inject the `UserModel` into the `UsersService` using the `@InjectModel()` decorator:

```
@@filename(users.service)
import { Injectable } from '@nestjs/common';
import { InjectModel } from '@nestjs/sequelize';
import { User } from './user.model';

@Injectable()
export class UsersService {
  constructor(
    @InjectModel(User)
    private userModel: typeof User,
  ) {}

  async findAll(): Promise<User[]> {
    return this.userModel.findAll();
  }

  findOne(id: string): Promise<User> {
    return this.userModel.findOne({
      where: {
        id,
      },
    });
  }

  async remove(id: string): Promise<void> {
    const user = await this.findOne(id);
    await user.destroy();
  }
}
@@switch
import { Injectable, Dependencies } from '@nestjs/common';
import { getModelToken } from '@nestjs/sequelize';
import { User } from './user.model';

@Injectable()
@Dependencies(getModelToken(User))
export class UsersService {
  constructor(usersRepository) {
    this.usersRepository = usersRepository;
  }

  async findAll() {
    return this.userModel.findAll();
  }

  findOne(id) {
    return this.userModel.findOne({
      where: {
        id,
      },
    });
  }
```

```
  async remove(id) {
    const user = await this.findOne(id);
    await user.destroy();
  }
}
```

> warning **Notice** Don't forget to import the `UsersModule` into the root `AppModule`.

If you want to use the repository outside of the module which imports `SequelizeModule.forFeature`, you'll need to re-export the providers generated by it. You can do this by exporting the whole module, like this:

```
@@filename(users.module)
import { Module } from '@nestjs/common';
import { SequelizeModule } from '@nestjs/sequelize';
import { User } from './user.entity';

@Module({
  imports: [SequelizeModule.forFeature([User])],
  exports: [SequelizeModule]
})
export class UsersModule {}
```

Now if we import `UsersModule` in `UserHttpModule`, we can use `@InjectModel(User)` in the providers of the latter module.

```
@@filename(users-http.module)
import { Module } from '@nestjs/common';
import { UsersModule } from './users.module';
import { UsersService } from './users.service';
import { UsersController } from './users.controller';

@Module({
  imports: [UsersModule],
  providers: [UsersService],
  controllers: [UsersController]
})
export class UserHttpModule {}
```

Relations

Relations are associations established between two or more tables. Relations are based on common fields from each table, often involving primary and foreign keys.

There are three types of relations:

`One-to-one`	Every row in the primary table has one and only one associated row in the foreign

	table
`One-to-many / Many-to-one`	Every row in the primary table has one or more related rows in the foreign table
`Many-to-many`	Every row in the primary table has many related rows in the foreign table, and every record in the foreign table has many related rows in the primary table

To define relations in models, use the corresponding **decorators**. For example, to define that each `User` can have multiple photos, use the `@HasMany()` decorator.

```
@@filename(user.model)
import { Column, Model, Table, HasMany } from 'sequelize-typescript';
import { Photo } from '../photos/photo.model';

@Table
export class User extends Model {
  @Column
  firstName: string;

  @Column
  lastName: string;

  @Column({ defaultValue: true })
  isActive: boolean;

  @HasMany(() => Photo)
  photos: Photo[];
}
```

info **Hint** To learn more about associations in Sequelize, read this chapter.

Auto-load models

Manually adding models to the `models` array of the connection options can be tedious. In addition, referencing models from the root module breaks application domain boundaries and causes leaking implementation details to other parts of the application. To solve this issue, automatically load models by setting both `autoLoadModels` and `synchronize` properties of the configuration object (passed into the `forRoot()` method) to `true`, as shown below:

```
@@filename(app.module)
import { Module } from '@nestjs/common';
import { SequelizeModule } from '@nestjs/sequelize';

@Module({
  imports: [
    SequelizeModule.forRoot({
      ...
```

```
      autoLoadModels: true,
      synchronize: true,
    }),
  ],
})
export class AppModule {}
```

With that option specified, every model registered through the `forFeature()` method will be automatically added to the `models` array of the configuration object.

> warning **Warning** Note that models that aren't registered through the `forFeature()` method, but are only referenced from the model (via an association), won't be included.

Sequelize Transactions

A database transaction symbolizes a unit of work performed within a database management system against a database, and treated in a coherent and reliable way independent of other transactions. A transaction generally represents any change in a database (learn more).

There are many different strategies to handle Sequelize transactions. Below is a sample implementation of a managed transaction (auto-callback).

First, we need to inject the `Sequelize` object into a class in the normal way:

```
@Injectable()
export class UsersService {
  constructor(private sequelize: Sequelize) {}
}
```

> info **Hint** The `Sequelize` class is imported from the `sequelize-typescript` package.

Now, we can use this object to create a transaction.

```
async createMany() {
  try {
    await this.sequelize.transaction(async t => {
      const transactionHost = { transaction: t };

      await this.userModel.create(
          { firstName: 'Abraham', lastName: 'Lincoln' },
          transactionHost,
      );
      await this.userModel.create(
          { firstName: 'John', lastName: 'Boothe' },
          transactionHost,
      );
    });
  } catch (err) {
    // Transaction has been rolled back
```

```
    // err is whatever rejected the promise chain returned to the
transaction callback
  }
}
```

> info **Hint** Note that the `Sequelize` instance is used only to start the transaction. However, to test this class would require mocking the entire `Sequelize` object (which exposes several methods). Thus, we recommend using a helper factory class (e.g., `TransactionRunner`) and defining an interface with a limited set of methods required to maintain transactions. This technique makes mocking these methods pretty straightforward.

Migrations

Migrations provide a way to incrementally update the database schema to keep it in sync with the application's data model while preserving existing data in the database. To generate, run, and revert migrations, Sequelize provides a dedicated CLI.

Migration classes are separate from the Nest application source code. Their lifecycle is maintained by the Sequelize CLI. Therefore, you are not able to leverage dependency injection and other Nest specific features with migrations. To learn more about migrations, follow the guide in the Sequelize documentation.

Multiple databases

Some projects require multiple database connections. This can also be achieved with this module. To work with multiple connections, first create the connections. In this case, connection naming becomes **mandatory**.

Suppose you have an `Album` entity stored in its own database.

```
const defaultOptions = {
  dialect: 'postgres',
  port: 5432,
  username: 'user',
  password: 'password',
  database: 'db',
  synchronize: true,
};

@Module({
  imports: [
    SequelizeModule.forRoot({
      ...defaultOptions,
      host: 'user_db_host',
      models: [User],
    }),
    SequelizeModule.forRoot({
      ...defaultOptions,
      name: 'albumsConnection',
      host: 'album_db_host',
      models: [Album],
```

```
    }),
  ],
})
export class AppModule {}
```

> warning **Notice** If you don't set the `name` for a connection, its name is set to `default`. Please note that you shouldn't have multiple connections without a name, or with the same name, otherwise they will get overridden.

At this point, you have `User` and `Album` models registered with their own connection. With this setup, you have to tell the `SequelizeModule.forFeature()` method and the `@InjectModel()` decorator which connection should be used. If you do not pass any connection name, the `default` connection is used.

```
@Module({
  imports: [
    SequelizeModule.forFeature([User]),
    SequelizeModule.forFeature([Album], 'albumsConnection'),
  ],
})
export class AppModule {}
```

You can also inject the `Sequelize` instance for a given connection:

```
@Injectable()
export class AlbumsService {
  constructor(
    @InjectConnection('albumsConnection')
    private sequelize: Sequelize,
  ) {}
}
```

It's also possible to inject any `Sequelize` instance to the providers:

```
@Module({
  providers: [
    {
      provide: AlbumsService,
      useFactory: (albumsSequelize: Sequelize) => {
        return new AlbumsService(albumsSequelize);
      },
      inject: [getDataSourceToken('albumsConnection')],
    },
  ],
})
export class AlbumsModule {}
```

Testing

When it comes to unit testing an application, we usually want to avoid making a database connection, keeping our test suites independent and their execution process as fast as possible. But our classes might depend on models that are pulled from the connection instance. How do we handle that? The solution is to create mock models. In order to achieve that, we set up custom providers. Each registered model is automatically represented by a `<ModelName>Model` token, where `ModelName` is the name of your model class.

The `@nestjs/sequelize` package exposes the `getModelToken()` function which returns a prepared token based on a given model.

```
@Module({
  providers: [
    UsersService,
    {
      provide: getModelToken(User),
      useValue: mockModel,
    },
  ],
})
export class UsersModule {}
```

Now a substitute `mockModel` will be used as the `UserModel`. Whenever any class asks for `UserModel` using an `@InjectModel()` decorator, Nest will use the registered `mockModel` object.

Async configuration

You may want to pass your `SequelizeModule` options asynchronously instead of statically. In this case, use the `forRootAsync()` method, which provides several ways to deal with async configuration.

One approach is to use a factory function:

```
SequelizeModule.forRootAsync({
  useFactory: () => ({
    dialect: 'mysql',
    host: 'localhost',
    port: 3306,
    username: 'root',
    password: 'root',
    database: 'test',
    models: [],
  }),
});
```

Our factory behaves like any other asynchronous provider (e.g., it can be `async` and it's able to inject dependencies through `inject`).

```
SequelizeModule.forRootAsync({
  imports: [ConfigModule],
  useFactory: (configService: ConfigService) => ({
    dialect: 'mysql',
    host: configService.get('HOST'),
    port: +configService.get('PORT'),
    username: configService.get('USERNAME'),
    password: configService.get('PASSWORD'),
    database: configService.get('DATABASE'),
    models: [],
  }),
  inject: [ConfigService],
});
```

Alternatively, you can use the `useClass` syntax:

```
SequelizeModule.forRootAsync({
  useClass: SequelizeConfigService,
});
```

The construction above will instantiate `SequelizeConfigService` inside `SequelizeModule` and use it to provide an options object by calling `createSequelizeOptions()`. Note that this means that the `SequelizeConfigService` has to implement the `SequelizeOptionsFactory` interface, as shown below:

```
@Injectable()
class SequelizeConfigService implements SequelizeOptionsFactory {
  createSequelizeOptions(): SequelizeModuleOptions {
    return {
      dialect: 'mysql',
      host: 'localhost',
      port: 3306,
      username: 'root',
      password: 'root',
      database: 'test',
      models: [],
    };
  }
}
```

In order to prevent the creation of `SequelizeConfigService` inside `SequelizeModule` and use a provider imported from a different module, you can use the `useExisting` syntax.

```
SequelizeModule.forRootAsync({
  imports: [ConfigModule],
```

```
    useExisting: ConfigService,
  });
```

This construction works the same as `useClass` with one critical difference - `SequelizeModule` will lookup imported modules to reuse an existing `ConfigService` instead of instantiating a new one.

Example

A working example is available here.