UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Object Relational Mapping and API with Loopback 4

Final Degree Thesis

Escola Tècnica Superior
d'Enginyeria de Telecomunicacions de Barcelona

by:

Guillem Llucià i Turu

Supervised by:
Jose Luis Muñoz

Barcelona, September 2019

Abstract

Nowadays API are a part of our day-to-day. They are so integrated to our daily life we use them without even thinking about it. And we could not think of a world without them. Search engines, digital commerce, digital forums, are just an example of our routinary use of API.

Object Relational Mapping is a technique to ’persist’ data stored in this API applications into databases.

LoopBack 4 is a new API framework supported by StroongLoop (IBM). It is the fourth version of this framework. It is still on developing ways.

The goal of this project is to understand how does LoopBack implements the other two concepts above. To have an insight on how a *back-end* works, how databases work, and how the framework persist the data on them. And finally to create a complete documentation so anyone could understand how ORM is implemented in LoopBack 4 from scratch.

In the course of the project we studied and implemented some explanatory examples on how relations work in LoopBack 4. This involves not just the models, but datasources, controllers and repositories. And afterwards we have inspected how this relations get stored inside a relational and a non-relational database.

Resum

Avui en dia les API formen part del nostre dia a dia. Estan tan integrades en les nostres rutines que les utilitzem sense ser-ne conscients. Buscadors d'internet, comerços digitals, fòrums digitals, són només alguns dels exemples de l'ús rutinari que tenen les API.

L'*Object Relational Mapping* és una tècnica usada per fer *persistir* les dades usades per aquestes API dins de bases de dades.

LoopBack 4 és un *framework* per a desenvolupar API amb el suport de StroongLoop (IBM). És la quarta versió del framework i la més nova. És per això que encara s'està desenvolupant.

L'objectiu d'aquest projecte és entendre com LoopBack 4 implementa els dos conceptes anteriors. D'aconseguir una percepció de com funciona un *back-end*, de com funcionen les bases de dades i com el *framework* hi guarda allà les dades. Finalment crear una documentació completa des d'on qualsevol pugui entendre com implementar ORM en LoopBack 4 des de zero.

Durant el projecte hem estudiat i implementat uns exemples per explicar com funcionen les relacions en LoopBack 4. Això no només implica models; també inclou *datasources*, controladors i repositoris. Un cop implementats en LoopBack hem mirat aquestes relacions queden reflectides dins d'una base de dades relacional, i una de no-relacional.

Resumen

Actualmente les API forman parte de nuestro día a día. Están tan integradas en nuestras rutinas que las usamos sin ser ni tan solo conscientes de ello. Buscadores de internet, comercios digitales o foros digitales, son solo alguno de los ejemplos de este uso rutinario de las API.

El Object Relational Mapping es una técnica usada con para persistir los datos usados por estas API.

LoopBack 4 es un framework para desarollar API con el soporte de StrongLoop (IBM). Esta es la cuarta versión del software y la más nueva. Aún sé está desarrollando.

El objetivo de este proyecto es entender como LoopBack 4 implementa los dos conceptos mencionados anteriormente. De conseguir una percepción de como funciona un back-end, de como funcionan las bases de datos, y de como el framework guarda allí los datos. Finalmente crearemos una documentación completa para que alguien pueda entender el ORM en Loopback 4 des de cero.

Durante el proyecto hemos estudiado e implementado unos ejemplos para explicar como funcionan las relaciones en LoopBack4. Esto no solo implica modelos; también incluye datasources, controladores y repositorios. Una vez implementados en Loopback, veremos como estas relaciones quedar plasmadas dentro de una base de datos relacional y una de no relacional.

Acknowledgments

This thesis would not have been possible without the assistance of my tutor Jose Luis Muñoz.

Thanks to the friends who have crossed their path with mine during these years, without them I would not arrived this far.

Also thanks to my family specially both my parents, without their unwavering support I definitely would be here.

Revision history and approval record

Revision	Date	Objective
0	20/09/2019	Document Creation
1	3/10/2019	Document Revison
2	06/10/2019	Document Final Revision

Document distribution list:

Name	e-mail
Guillem Llucià i Turu	guillucia@gmail.com
Jose Luis Muñoz Tapia	jose.munoz@entel.upc.edu

Written by:		Reviewed and approved by:	
Date	20/09/2019	**Date**	06/05/2019
Name	Guillem Llucià	**Name**	Jose Luis Muñoz
Role	Author	**Role**	Supervisor

Contents

List of Figures

Chapter 1

Introduction

1.1 Objectives

This project aims to study, understand and write down a documentation on how Object Relational Mapping works in the Loopback 4 framework. The main objective is to create a documentation guide, and didactic material that will help someone understand how Object Relational Mapping works in Loopback 4 almost without previous knowledge about databases nor Loopback.

In order to do so there are some middle steps or secondary but necessary objectives. We must acquire a certain level of general knowledge about databases. Also we must learn not just about the ORM ways of the framework but also a more generic scope about how loopback 4 works. And to achieve the last point we will also have to learn a bit about JavaScript and TypeScript.

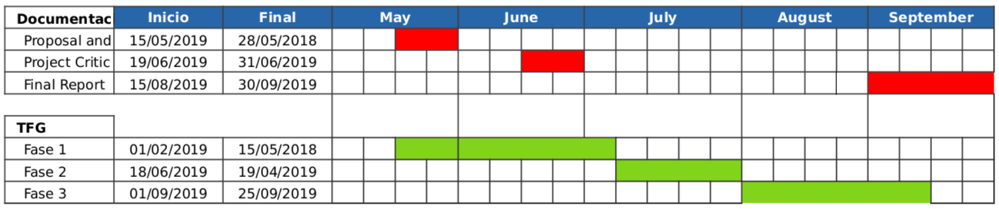
1.2 Requirements and specifications

To develop this project we have used the following technologies:

- TypeScript programming language, which compiles to JavaScript.
- Loopback 4 framework.
- Docker to run databases images.
- MongoDb database.
- PostgreSQL database.
- LaTeX, XeLaTeX, to write documentation.

1.3 Work Plan

This project has been structured in three phases as seen in the figure 1.1.



Documentac	Inicio	Final	May				June				July					August				September			
Proposal and	15/05/2019	28/05/2018			🟥	🟥																	
Project Critic	19/06/2019	31/06/2019							🟥	🟥													
Final Report	15/08/2019	30/09/2019																		🟥	🟥	🟥	🟥
TFG																							
Fase 1	01/02/2019	15/05/2018			🟩	🟩	🟩	🟩	🟩	🟩	🟩												
Fase 2	18/06/2019	19/04/2019										🟩	🟩	🟩	🟩								
Fase 3	01/09/2019	25/09/2019														🟩	🟩	🟩	🟩	🟩	🟩		

Figure 1.1: Gantt Diagram

- **Phase one:** Studying and understanding the different technologies used in this project (seen in the Requirements and Specifications section). This requires a lot of time since there are a lot of new technologies to learn from scratch.
- **Phase two:** The second phase is dedicated to, once understood the technologies, designing and programming the environments and examples to later document them.
- **Phase three:** The third phase is dedicated to documenting all the knowledge gathered during the first phase and illustrate it with examples programmed during the second phase.

1.4 Deviation and setbacks

During the realization of this thesis we have encountered some incidents that complicated it's development. The setbacks and its consequences are listed below:

- **Poor Loopback 4 documentation:** As Loopback 4 is still on developing ways, its documenation is poor, unstable, and confusing. There are lots of omissions and gaps that made the learning and understanding of the framework rough and time consuming. This leaded us to shorten the scope of the project.
- **Sick leave:** I had an accident that lead to surgery with a long-time rehabilitation that took me away from the project for almost two months.

Chapter 2

Concepts

2.1 Object Relational Mapping

Object-relational mapping (ORM, O/RM, and O/R mapping tool) [2] in computer science is a programming technique for converting data between incompatible type systems using object-oriented programming languages. This creates, in effect, a "virtual object database" that can be used from within the programming language.

The heart of the problem involves translating the logical representation of the objects into an atomized form that is capable of being stored in the database while preserving the properties of the objects and their relationships so that they can be reloaded as objects when needed. If this storage and retrieval functionality is implemented, the objects are said to be persistent.

2.2 Databases

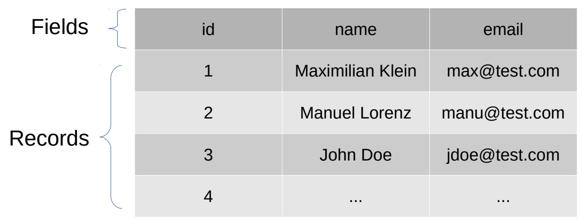
A database [1] is a collection of information that is organized so that it can be easily accessed, managed and updated.

There are two big families of Databases: Relational and Non-Relational databases.

We will proceed to explain them and discuss them so we can understand which one to use in every occasion.

2.2.1 Relational Databases

- In a relational database [7] we have **tables** where we find **fields** (columns) and we can fill it with **records** (Rows). It can be seen in figure 2.1.
- The fields create the **schema**
- Each Record must have a value (it can be null) for every field.
- All records must follow the schema.

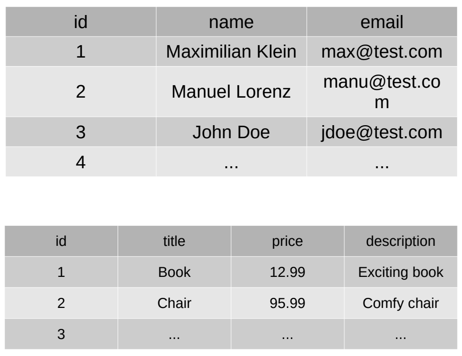

Fields
Records

id	name	email
1	Maximilian Klein	max@test.com
2	Manuel Lorenz	manu@test.com
3	John Doe	jdoe@test.com
4	...	...

Figure 2.1: SQL Structure Example

But what happens if we have two different Tables with different objects and we would want to link them?

For instance we have a table with customers and a table with products and we want to relate them.



id	name	email
1	Maximilian Klein	max@test.com
2	Manuel Lorenz	manu@test.co m
3	John Doe	jdoe@test.com
4	...	...

id	title	price	description
1	Book	12.99	Exciting book
2	Chair	95.99	Comfy chair
3	...	...	...

Figure 2.2: Relations Example

There are different kind of relations:

2.2.1.1 One to One Relation

- The ’source’ table contains a field where we store the id of the ’target’ table.
  - When a key from another table is stored, it is called a foreign key.
  - While we call the id from our tables local keys.

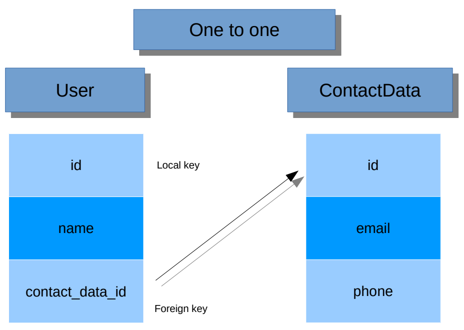


Figure 2.3: One to One Relation

2.2.1.2 One to Many Relation

- The ’target’ table contains a field where we store the id of the ’source’ table.

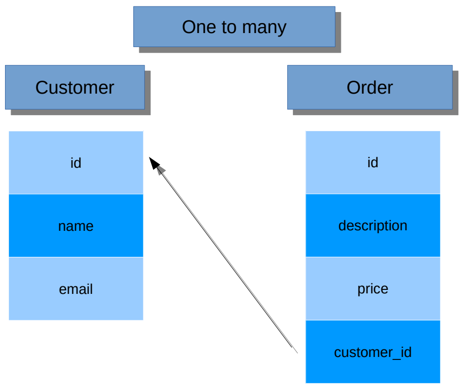


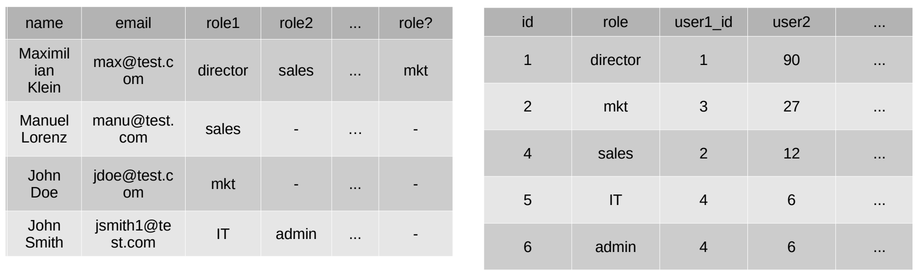
Figure 2.4: One to Many Relation

2.2.1.3 Many to Many Relation

When trying to implement a ’Many to Many Relation’ we find the following problem:

Following the user and roles example:

- We can not predict how many roles will a user have nor how many users will hold one specific role, there’s no way we can create a fitting schema.
- Creating the tables with "blank" spaces would be highly inefficient and it would be limited.



name	email	role1	role2	...	role?
Maximilian Klein	max@test.com	director	sales	...	mkt
Manuel Lorenz	manu@test.com	sales	-	...	-
John Doe	jdoe@test.com	mkt	-	...	-
John Smith	jsmith1@test.com	IT	admin	...	-

id	role	user1_id	user2	...
1	director	1	90	...
2	mkt	3	27	...
4	sales	2	12	...
5	IT	4	6	...
6	admin	4	6	...

Figure 2.5: Many to Many Problem

The solution is to create a third table where we store the relations between the other two using their id's.

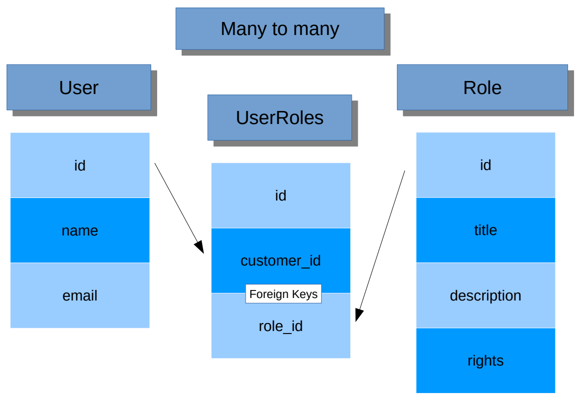


Figure 2.6: Many to Many Relation correctly implemented

2.2.2 Non-Relational Databases

2.2.2.1 Motivation

Relational databases are designed to be **orthogonal** and to not repeat data. This can be really 'storage-efficient' but when it comes to data reading it can be very process consuming and slow as it may have to access multiple tables.

Later, on the internet era, the need of processing queries in a faster way appeared. If we know which queries will be the most asked, we can just store them directly in a "table" so we do not need to access all the different tables each time.

Under this idea non-Relational databases [7] are born.

2.2.2.2 Structure

In non relational databases we have 'collections'. In a collection we store 'documents'. **Documents do not need to follow a schema.**

Another important change is that we do not use relations.

We just create another collection with the information of the two other collections we want to relate.

- That way we can serve much more many requests.
- It duplicates data, besides the obvious problem, there is also some complications if we ever want to modify some data as we will to change more than one collection.

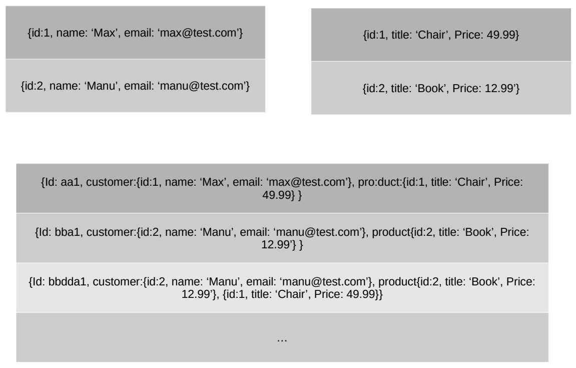

{id:1, name: ‘Max’, email: ‘max@test.com’}
{id:2, name: ‘Manu’, email: ‘manu@test.com’}
{id:1, title: ‘Chair’, Price: 49.99}
{id:2, title: ‘Book’, Price: 12.99’}
{Id: aa1, customer:{id:1, name: ‘Max’, email: ‘max@test.com’}, pro:duct:{id:1, title: ‘Chair’, Price: 49.99} }
{Id: bba1, customer:{id:2, name: ‘Manu’, email: ‘manu@test.com’}, product{id:2, title: ‘Book’, Price: 12.99’} }
{Id: bbdda1, customer:{id:2, name: ‘Manu’, email: ‘manu@test.com’}, product{id:2, title: ‘Book’, Price: 12.99’}, {id:1, title: ‘Chair’, Price: 49.99}}
…

Figure 2.7: Non-Relational Database Structure

2.2.3 Relational Databses vs Non-Relational Databases

Relational

- First kind of databases.
- Orthogonality of data does not duplicate data but it makes the queries difficult and slow.
- Has a schema.
- Uses relations.
- Can not escalate horizontally.

Non-Relational

- Not orthogonal, duplicates data, sacrificing storage in order to serve faster.
- Does not use a schema.
- Does not use relations.
- Allows both vertical and horizontal escalation.

2.2.3.1 Which is better?

It will depend on your project. There is no "absolute" winner. You must understand both ways and choose whichever adapts better to your necessities.

2.3 Docker

As Docker is just a tool we used in order to carry out our project, that's why we won't study it very thoroughly but instead just give a general notion of what it is in this section.

2.3.0.1 What is Docker?

Docker [3] is what is called a **Container**. And a container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another.

A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, run-time, system tools, system libraries and settings.

2.3.0.2 Comparing Containers and Virtual Machines

Containers and virtual machines have similar resource isolation and allocation benefits, but function differently because containers virtualize the operating system instead of hardware. Containers are more portable and efficient.

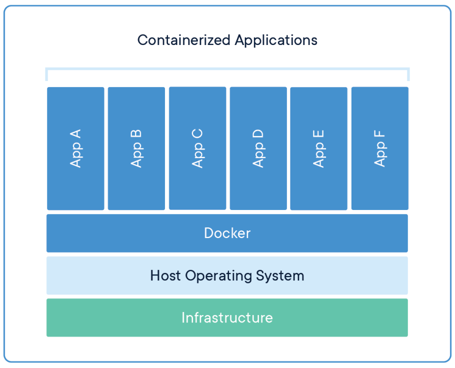


Figure 2.8: Container Schema

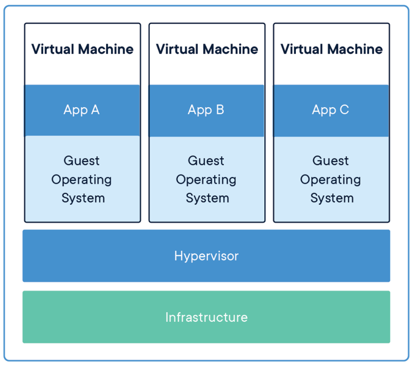


Figure 2.9: Virtual Machine Schema

2.4 Loopback 4

LoopBack 4 [6] is a framework oriented to design Web APIs. It is built over NodeJs and Typescript technology. It is ready for production environments and to serve REST API. The framework is recent and it lacks documentation, our way of study has been through tutorials and peaking into it's font code.

Loopback has a modular and extensible structure with lots of characteristics which make him innovative and competitive. We can design an API as a component, for example: activity registry, messaging, authentication, monotorization, etc. It is a good practice in LB4 to separate the functionalities of our API in components.

Loopback has integrations with data fonts. Some of them are maintained by StrongLoop and others by the community. As we can see in the figure 2.10, it has connectors to databases of the SQL and the non-SQL type, it also has connectors to exterior services as GPRC, SOAP and REST, among others.

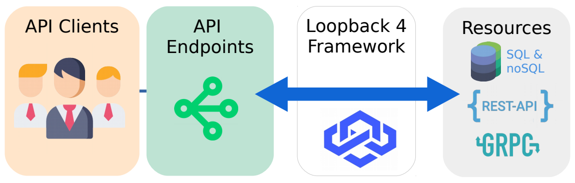


Figure 2.10: Loopback Schema

We will proceed to explain the most useful characteristics and some software patterns used.

2.4.1 CLI

Loopback has a CLI which auto-generates code. Here are some of its commands:

- $ lb4 model: It is used to generate a model. We specify its name, attributes, types, and properties. It also generates documentation for the model.
- $ lb4 datasource: Adds the configuration for *datasources* and installs the required packages to establish the connection with them.
- $ lb4 controller: Generates the controllers and auto-completes the stubs with the CRUD methods.
- $ lb4 repository: Generates the repositories from the selected *datasources*.

2.4.2 Dependency Injection

Loopback 4 uses a technique called *Dependency Injection*. We can inject dependencies using the constructor or the setter. This way, the class is not responsible for instantiating the dependency. The main perks of using this pattern are:

- High-level modules do not depend on low-level modules.
- Code is separated from the low-level implementation.
- Depends on abstractions (interfaces).
- Reusable modules.
- Easy to test.

2.4.3 DataSource

Datasources are LoopBacks way of connecting to sources of data. Data can be in a database, memory, other APIs, message queues and more.

The *datasource* class represents the connector once it is configured. This inherits from the Juggler and contains the connector.

The functionality of the *datasource* is to configure the connector using a configuration file.

2.4.4 Connectors

The *Connector* is the agent responsible to communicate with the database, API REST or SOAP, and allows us to abstract from the low level implementation.

There are many connectors supported by StrongLoop that allows to connect to almost every database.

2.4.5 Juggler

The *Juggler* is defined as ORM which allows to interact with the databases, REST APIs and other kind of data.

In other words: it is a **common interface between all connectors** that allows to abstract the *Datasource* from the application's functionality. We can use the same model for different *datasources*.

Sadly the ***Juggler* is not aware of *relations***. This adds a lot of complexity to the project if it requires lots of them as they need to be specified not only in the code of the models, but in the code of the repositories and controllers.

We can see it in the figure 2.11.

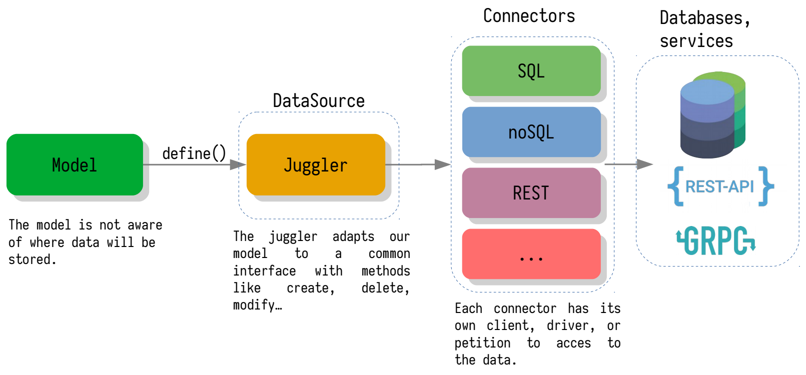

Connectors
Databases, services
DataSource
SQL
noSQL
REST
...
Model
define()
Juggler
REST-API
GRPC
The model is not aware of where data will be stored.
The juggler adapts our model to a common interface with methods like create, delete, modify…
Each connector has its own client, driver, or petition to acces to the data.

Figure 2.11: Juggler

2.4.6 Repositories

The class *repository* is an abstraction of the Juggler and the connector. It is usually called from the *controller* to access (create, edit, delete, consult...) the data.

2.4.7 Inversion of Control

Loopback is designed with Inversion of Control.

The IoC is a software design principle in which the flux of programming gets inverted respect other traditional methods.

The workflow of the application is not controlled by the programmer. Instead of calling libraries when we need them, the *framework* 'calls our code'.

Therefore, the operation of our application is already implemented. We just need to implement the 'details'.

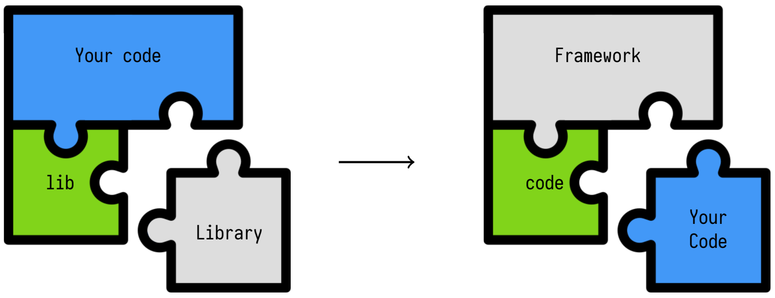


Figure 2.12: Inversion of Control Concept

2.4.8 Models

2.4.8.1 Model Concept

A model describes the **shape** of business domain objects like a mold.

For example: Customer, Address, Order...

Defines a list of properties with name, type, and other constraints.

2.4.8.2 Features of Models

Models **describe only the shape** of data. Behavior such as CRUD operations are not defined in the model, but in the repositories. Also, a single model can be used with multiple different repositories.

When mapped in a relational database, the model could be seen as the "table". As in relational databases, in LB4 apps we can define relationships between models (more later).

2.4.8.3 Types of Models

There are two types of Models:

- **Value Object:** Which **dont have an identity (ID)** because its equality is based on the structural value. Two Value Objects will be the same if they share the same properties. A postal address would fit this case, two addresses are the same if they share the same street number, street name, city, and zip code values.

- **Entity:** They **require an object identity (ID)** and its equality is based on this identity. A Customer would fit this case as two people might have the same name but they will be the same if they are referencing to the same ID.

2.4.9 Controller

A *Controller* is a class that implements the operations (business logic) of an HTTP/REST API.

Here we specify the *paths* or endpoints where we will find the operations. Usually the *Controller* calls other modules such as repositories (which contain connectors), but also third party applications, or services.

2.4.10 Relations

Relations are the way of linking models. Loopback 4 currently supports 3 kinds of relations:

2.4.10.1 hasOne Relation

A hasOne [5] relation denotes a one-to-one connection of a model to another model through referential integrity. The referential integrity is enforced by a foreign key constraint on the target model which usually references a primary key on the source model and a unique constraint on the same column/key to ensure one-to-one mapping. This relation indicates that each instance of the declaring or source model has exactly one instance of the target model. Lets take an example where an application has models Supplier and Account and a Supplier can only have one Account on the system as illustrated in the diagram below.

The diagram in figure 2.13 shows target model Account has property supplierId as the foreign key to reference the declaring model Suppliers primary key id. supplierId needs to also be used in a unique index to ensure each Supplier has only one related Account instance. To add a hasOne relation to your LoopBack application and expose its related routes, you need to perform the following steps

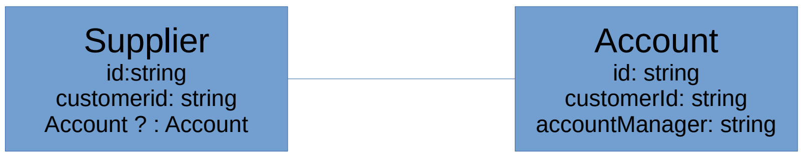


Figure 2.13: hasOne Relation in LoopBack 4

2.4.10.2 hasMany Relation

A hasMany [4] relation denotes a one-to-many connection of a model to another model through referential integrity. The referential integrity is enforced by a foreign key constraint on the target model which usually references a primary key on the source model. This relation indicates that each instance of the declaring or source model has zero or more instances of the target model. For example, in an application with customers and orders, a customer can have many orders as illustrated in the diagram below.

The diagram shows target model Order has property customerId as the foreign key to reference the declaring model Customers primary key id.

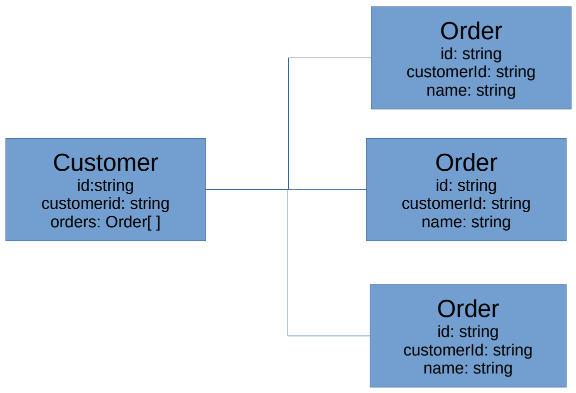


Figure 2.14: hasMany Relation in LoopBack 4

2.4.10.3 belongsTo Relation

As its name suggests this relation denotes that the model is a part of a bigger model. The bigger model can contain one or more than one of the small model. This relation is used in the ’other side’ of the two relations to reinforce the referential integrity.

Chapter 3

Development

3.1 Introduction

In the previous section we have explained the basic concepts to understand what will follow.

In this section we will discuss the code used in LoopBack 4 to correctly implement some ORM functionalities , and then we will proceed to see how do they affect to the databases. We will also see how to create our databases with docker and how to peak inside them.

We will study a *PostgreSQL* as an example of a relational database and *MongoDB* as an example of a non-relational database.

For more information about this contents you can reference to the annex under the same title.

3.2 PostgreSQL

3.2.1 Creating our Database

First of all we must have a database to analyze. To create it we use dockers as it is a simple and clean way to run services in our machine. Here is how we created our PostgreSQL database:

```
$ docker run --name some-postgres\
-v "$(pwd)"/databases/postgres:/var/lib/postgresql/data \
-e POSTGRES_PASSWORD=password -e POSTGRES_USER=username \
-p 5432:5432 -d postgres
```

Once we have the image running:

```
$ docker exec -it container_id psql -U username
psql (11.5 (Debian 11.5-1.pgdg90+1))
Type "help" for help.

username=# CREATE DATABASE database;
```

The previous command will execute `psql` in our container and create a database.

3.2.2 DataSource configuration

The simplest way to configure our datasource is using the CLI functionality: *lb4 datasource* in our app directory.

3.2.3 Model

For our first approach we will create a simple model without any relation, just to see how it gets translated in our SQL database.

To do so, the easiest way is, again, using the CLI.

3.2.4 Repository and Controller

As we don't include any relations in our first example and we just want to see how the API maps the information in our database, we will just go with the default options given by the CLI.

3.2.5 Migrate Schema

Before we start the application, we will modify our *index.ts* file so it automatically creates a schema in our database based on the models we have. This is important as we start our application without an schema in the database, we will get errors. This will be seen later.

3.2.6 Implementing a UUID for Our Models

When working with a database we will need an ID to identify the data we store.

As we do not want the user of our application to be 'annoyed' by this kind of task, we will implement the default generation of uuid.

First we install the package with:

```
$ npm install --save @types/uuid
```

To do so we will use a third-party library named 'uuid'. So in all our models we shall include it:

```
import {v4 as uuid} from 'uuid';
```

Then we should go to the 'id' property and set it to string (uuid generates strings!) and write the following lines:

```
@property({
    type: 'string',
    id: true,
    //add this line
    default: () => uuid(),
})
id?: string;
```

This has to be made before creating the other classes, or else, we will have to modify a few lines in other parts of the code.

3.2.7 Running the App

We can run our application with the following command:

```
myapp$ npm start
```

Then we connect to the docker.

```
$ docker exec -it ec psql -U user
```

We check for existing databases

```
user=# \l
List of databases

Name     | Owner | Encoding |  Collate   |   Ctype    | Access privileges
-----------+-------+----------+------------+------------+-------------------
database  | user  | UTF8     | en_US.utf8 | en_US.utf8 |
template0 | user  | UTF8     | en_US.utf8 | en_US.utf8 | =c/user          +
          |       |          |            |            | user=CTc/user
user      | user  | UTF8     | en_US.utf8 | en_US.utf8 |
```

We switch to 'database'

```
user=# \c database
You are now connected to database "database" as user "user".
database=#
```

And then check for relations

```
database=# \dt
List of relations
Schema | Name | Type  | Owner
--------+------+-------+-------
public | todo | table | user
(1 row)
```

But the table is empty!

```
database=# SELECT * FROM todo;
id | name
----+------
 (0 rows)
```

Now if we have our server running we can send a post request and LoopBack will create a todo object for us.

To do so LoopBack has a web interface but we can do it also with a curl command:

```
$ curl -X POST "http://[::1]:3000/todos" -H "accept:
 application/json" \
-H "Content-Type: application/json" \
-d "{\"id\":0,\"title\":\"Database Tutorial\",\"desc\":\
"Make a tutorial about databases and lb4\",\"isComplete\":false}"

{"id":0,"title":"Database Tutorial","desc":"Make a tutorial about\
 databases and lb4","isComplete":false}
```

Now if we check our database:

```
database=# SELECT * FROM todo;
id | title  |  desc  | iscomplete
----+--------+--------+------------
0 | string | string | t
```

The info is correctly stored there.

If we try to insert the same value twice we get a 500 error.

```
{"error":{"statusCode":500,"message":"Internal Server Error"}}
```

3.3 MongoDB

As LoopBack is designed in a modular way, the steps are practically identical to what we have seen for PostgreSQL. We will just mention what's different.

3.3.1 Creating our database

We will also use Docker

```
$ docker run -d --name some-mongo \
-v "$(pwd)"/databases/mongo:/data/db -p 27017:27017 -d mongo
```

3.3.2 Configuring the Datasource

We will also use the CLI but we will choose the MongoDB option when asked.

3.3.3 Running the App

We start our server:

```
npm start
```

Then we connect to the docker.

```
$ docker exec -it db mongo -U user
```

Then we check for existing databases

```
> show dbs
admin    0.000GB
config   0.000GB
local    0.000GB
```

We proceed to send a POST request.

```
$ curl -X POST "http://[::1]:3000/todos" -H "accept:\
application/json" -H "Content-Type: application/json" \
-d "{\"id\":0,\"title\":\"Database Tutorial\",\"desc\":\"Make a\
tutorial about databases and lb4\", \"isComplete\":false}"



{"id":0,"title":"Database Tutorial","desc":"Make a tutorial about
databases and lb4", "isComplete":false}
```

And then check our database:

```
> show dbs
admin    0.000GB
config   0.000GB
example  0.000GB
local    0.000GB
database 0.000GB
```

Now the database exists!

And we can find the 'todo' object we stored:

```
db.Todo.find()
{ "_id" : 0, "title" : "Database Tutorial", "desc" : "Make a tutorial about databases and lb4","isComplete" : false }
```

We could also test it with the GET request:

And we can find the 'todo' object we stored:

```
$ curl -X GET "http://[::1]:3000/todos/0" -H "accept:\
application/json"

{"id":0,"title":"Database Tutorial","desc":"Make a tutorial about\
databases and lb4", "isComplete":false}
```

3.4 Relations

We will illustrate some kinds of relations by using two examples:

1. **Customer-Order example:** it will illustrate the relations HasMany, and Belongto.
2. **Category Example:** it will illustrate the *recursive* relation.

We will not talk about the HasOne, as it is implemented in a very similar way as the HasMany, Belongs to.

3.4.1 Customer-Order Example

We will follow a didactic way to easy understand the meaning of the new lines we are adding.

3.4.1.1 Customer Model

```
// src/models/customer.model.ts
import {Order} from './order.model';
import {Entity, property, hasMany} from '@LoopBack/repository';

@model()
export class Customer extends Entity {
    @property({ type: 'number', id: true }) id: number;
    @property({ type: 'string', required: true }) name: string;
    @hasMany(() => Order) orders?: Order[];

    constructor(data: Partial<Customer>) { super(data); }
}
```

The definition of the `hasMany` relation is inferred by using `@hasMany`

The decorator takes in a function resolving the target model class constructor and optionally a custom foreign key to store the relation metadata.

3.4.1.2 Order and OrderWithRelations

```
// src/models/order.model.ts
import {Entity, model, property} from '@LoopBack/repository';
@model()
export class Order extends Entity {
    @property({ type: 'number', id: true, required: true }) id: number;
    @property({ type: 'string', required: true })  name: string;
    @belongsTo(() => Customer) customerId: number;
    constructor(data?: Partial<Order>) { super(data); }
}
export interface OrderRelations {
    // describe navigational properties here
    customer?: Customer;
}
export type OrderWithRelations = Order & OrderRelations;
```

Notice that:

- The foreign key property (customerId) in the target model is added via a `belongsTo` relation.
- To be able to navigate from the target (order) to the source (customer) we need to add a new property.
- This property is added with the help of an interface and the type `OrderWithRelations` is who includes the navigational property.

3.4.2 CustomerWithRelations

Now we also add a navigational property to create the `CustomerWithRelations` type:

```
// src/models/customer.model.ts
import {Order, OrderWithRelations} from './order.model';
import {Entity, property, hasMany} from '@LoopBack/repository';

@model()
export class Customer extends Entity {
    @property({ type: 'number', id: true }) id: number;
    @property({ type: 'string', required: true }) name: string;
    @property({ type: 'boolean', default: false }) isDelivered: boolean;
    @hasMany(() => Order) orders?: Order[];
    constructor(data: Partial<Customer>) { super(data); }
}

export interface CustomerRelations {
    // navigational properties
    orders?: OrderWithRelations[];
}

export type CustomerWithRelations = Customer & CustomerRelations;
```

Finally, replace `Customer` by our new type `CustomerWithRelations` in the property of the interface `OrderRelations`.

3.4.2.1 Repositories with Relations

To configure the relation we need to do the following on the source repository:

1. In the constructor of your source repository class, use Dependency Injection to receive a getter[1] function for obtaining an instance of the target repository.
2. Declare a property with the factory function type:

   `HasManyRepositoryFactory<targetModel, typeof sourceModel.prototype.id>`
3. Call the `createHasManyRepositoryFactoryFor` function in the constructor of the source repository class with the relation name (decorated relation property on the source model) and target repository instance and assign it the previously declared property.

[1]We need a getter function, accepting a string repository name instead of a repository constructor, or a repository instance, in order to break a cyclic dependency between a repository with a `hasMany` relation and a repository with the matching `belongsTo` relation.

3.4.2.2 Target (Order) Repo CRUD API

The following CRUD APIs are now available in the constrained target repository factory orders for instances of customerRepository:

- `create` for creating a target model instance belonging to customer model instance.
- `find` finding target model instance(s) belonging to customer model instance.
- `delete` for deleting target model instance(s) belonging to customer model instance.
- `patch` for patching target model instance(s) belonging to customer model instance.

- For example, we can create an order where the `orderData` is created in a "constrained repository" for the `customerId`:

  ```
      customerRepositoryInstance.orders(customerId).create(orderData);
  ```

- For **updating** (full replace of all properties on a PUT endpoint for instance) a target model:
  - You have to directly use the target model repository.
  - In this case, the caller must provide both the `foreignKey` value and the primary key (id).
  - Since the caller already has access to the primary key of the target model, there is no need to go through the relation repository.
  - The operation can be performed directly on `DefaultCrudRepository` of the target model (`OrderRepository` in our example).
  - In our example this full replacement can be used to change the customer of an order:

    ```
    orderRepositoryInstance.replaceById(id: 2, data: {name: "tv", isDelivered: false, customerId: 124 });
    ```

3.4.2.3 Source (Customer) Repository CRUD API Modification

The current code (called `juggler`) that auto-generates the CRUD API for repositories is not currently aware of relations. As a result, we will have to override the `find()` and `findById()` functions created by default by the `juggler` to get the related Order object for each `Customer`.

```
// After the constructor:
async find( filter?: Filter<Customer>, options?: Options): Promise<CustomerWithRelations[]> {
    // Prevent juggler for applying "include" filter, it is not aware of LB4 relations
    const include = filter && filter.include;
    filter = {...filter, include: undefined};
    const result = await super.find(filter, options);
    // poor-mans inclusion resolver, this is a temporary implementation
    await Promise.all( result.map(async r => { r.orders = await this.orders(r.id).find(); }));

    return result;
}
```

```
async findById( id: typeof Customer.prototype.id, filter?: Filter<Customer>, options?: Options)
: Promise<CustomerWithRelations> {

    const include = filter && filter.include;
    filter = {...filter, include: undefined};
    const result = await super.findById(id, filter, options);
    // poor-mans inclusion resolver, this is a temporary implementation
    result.orders = await this.orders(result.id).find();

    return result;
}
```

Now when you get a Customer, a orders property will be included that contains the related Orders.

Notice that the inclusion resolver is a temporary implementation, this should be handled by `DefaultCrudRepo`

3.4.2.4 Target (Order) Repository CRUD API Modification

We will have to do the same on the Target (order) Repository so it includes the source (customer).

3.4.2.5 Source (Customer) Controller

We can create the default REST Controller with CRUD functions with the CLI.

We will modify the `delete` method to erase all the orders inside a customer in case it gets deleted. This way we will not have objects in our database pointing to a non-existent object.

```
export class CustomerController {
    constructor(
    @repository(CustomerRepository)
    public customerRepository : CustomerRepository,
    @repository(OrderRepository) public orderRepository : OrderRepository,
    ) {}
    // other methods
    // ...
    // ...
    @del('/customers/{id}', {
    responses: {
        '204': {
        description: 'Customer DELETE success',
        },
    },
    })
    async deleteById(@param.path.string('id') id: string): Promise<void> {
    await this.orderRepository.deleteAll({customerId : id});
    await this.customerRepository.deleteById(id);
    }
```

3.4.2.6 Target (Order) Controller

In the Target controller we can just go with the default REST *Controller* with the CRUD functions generated with the CLI.

3.4.2.7 Relation Controller: CustomerOrderController

LoopBack 4 good practices urges us to create a separate Controller to manage the constrained repositories.

In the `CustomerOrderController` we can call the underlying constrained repository CRUD APIs and expose this related models in our routes

3.4.3 Category Example

Consider an e-commerce application that has Categories, each Category may have several sub-categories and each sub-category may have also sub sub-categories

3.4.3.1 Category Model

The model for category can be defined like this:

```typescript
// src/models/category.model.ts
export class Category extends Entity {
    @property({ type: 'number', id: true, generated: true }) id?: number;
    @hasMany(() => Category, {keyTo: 'parentId'}) categories?: Category[];
    @belongsTo(() => Category) parentId?: number;
    constructor(data?: Partial<Category>) { super(data); }
}

export interface CategoryRelations {
    categories?: CategoryWithRelations[];
    parent?: CategoryWithRelations;
}

export type CategoryWithRelations = Category & CategoryRelations;
```

3.4.3.2 Category Repository

The natural repository one that code would be:

```typescript
// src/repositories/category.repository.ts (bad definition)
export class CategoryRepository extends DefaultCrudRepository<Category,typeof Category.prototype.id,CategoryRelations> {
    public readonly parent: BelongsToAccessor< Category, typeof Category.prototype.id >;
    public readonly categories: HasManyRepositoryFactory< Category, typeof Category.prototype.id >;

    constructor(
    @inject('datasources.db') dataSource: DbDataSource,
    @repository.getter(CategoryRepository) protected categoryRepositoryGetter: Getter<CategoryRepository> ) {
    super(Category, dataSource);
    this.parent = this.createBelongsToAccessorFor( 'parent', categoryRepositoryGetter );
    this.categories = this.createHasManyRepositoryFactoryFor( 'categories', categoryRepositoryGetter );
    }
}
```

But if we keep this code and carry on with the example we will get a Circular dependency error. To fix it:

```typescript
// src/repositories/category.repository.ts (fix)
export class CategoryRepository extends DefaultCrudRepository<Category,typeof Category.prototype.id,CategoryRelations> {
    public readonly parent: BelongsToAccessor< Category, typeof Category.prototype.id >;
    public readonly categories: HasManyRepositoryFactory< Category, typeof Category.prototype.id >;

    constructor(@inject('datasources.db') dataSource: DbDataSource) {
        super(Category, dataSource);
        this.parent = this.createBelongsToAccessorFor( 'parent', Getter.fromValue(this) ); // to break circular dep
        this.categories = this.createHasManyRepositoryFactoryFor( 'categories', Getter.fromValue(this) );
    }
}
```

3.4.3.3 Category Controller

We can generate the controller with the CLI, and then add a modification to de @del method to erase all the subcategories inside a category, in case the parent gets deleted. We do this to avoid having objects in our database linked to a non-existent object. We could just "clear" the link between them, that would be a plausible solution too.

```
// src/controller/category.parent.controller.ts
export class CategoryParentController {
    constructor(@repository(CategoryRepository) protected categoryRepository: CategoryRepository) { }
    @get('/categories/{id}/parent')
    async getCustomer( @param.path.number('id') id: number): Promise<Category> {
    return await this.categoryRepository.parent(id);
    }

    @del('/categories/{id}', {
        responses: {
            '204': {
                description: 'Category DELETE success',
            },
        },
    })
    async deleteById(@param.path.string('id') id: string): Promise<void> {
        // @param.query.object.

        await this.categoryRepository.deleteAll({parentId: id}); // To delete all subcategories.
        await this.categoryRepository.deleteById(id);
    }
}
```

3.4.4 Testing Relations in our Databases

We will proceed to test our code and see how the relations affect in our database.

3.4.4.1 PostgreSQL

The first step will be start our server and create a customer with a post request:

```
$ npm start
```

```
$ curl -X POST "http://[::1]:3000/customers" -H "accept: application/json" -H "Content-Type:
application/json" -d "{\"name\":\"John Smith\"}"
```

And then we will check in our database to see if the object has been created.

```
database=# SELECT * FROM customer WHERE name = 'John Smith';
id                    |    name
--------------------------------------+------------
99e554eb-d699-4e49-b4b6-73d2cd0f3b2f | John Smith
(1 row)
```

LoopBack will create the relation in the database by storing the ID of a customer inside an order, so it is normal that the schema in this database does not contain anything about orders.

Once we have a Customer we will check if the get request is working

```
$ curl -X GET "http://[::1]:3000/customers" -H "accept: application/json"

[{"id":"99e554eb-d699-4e49-b4b6-73d2cd0f3b2f","name":"John Smith","orders":[]}]
```

We can see how orders is an array, but it is empty.

Now let's add a order associated with this customer through the CustomerOrder controller.

```
$ curl -X POST "http://[::1]:3000/customers/99e554eb-d699-4e49-b4b6-73d2cd0f3b2f/order"\
-H "accept: */*" -H "Content-Type:
application/json" -d "{\"name\":\"Earl Grey\"}"

{
"id": "7dc908c0-9838-4d36-b912-0086962dbff6", "name": "Earl Grey",
"customerId": "99e554eb-d699-4e49-b4b6-73d2cd0f3b2f"
}
```

If we check the database:

```
database=# SELECT  * FROM "order" WHERE customerId = '99e554eb-d699-4e49-b4b6-73d2cd0f3b2f';
id                   |   name    |             customerid
--------------------------------------+-----------+--------------------------------------
7dc908c0-9838-4d36-b912-0086962dbff6 | Earl Grey | 99e554eb-d699-4e49-b4b6-73d2cd0f3b2f
(1 row)
```

Here we can see how the object has two ID's stored, one of its own, and the other from the object it belongs to (foreign key).

Now we will check if customers will accept more orders.

```
$ curl -X POST "http://[::1]:3000/customers/99e554eb-d699-4e49-b4b6-73d2cd0f3b2f/order" \
-H "accept: */*" -H "Content-Type: application/json" -d "{\"name\":\"Kettle\"}"
```

And we will also check if the get request in /customers/id works fine and returns an array of orders:

```
$ curl -X GET "http://[::1]:3000/customers/99e554eb-d699-4e49-b4b6-73d2cd0f3b2f" \
-H "accept: application/json"
{"id":"99e554eb-d699-4e49-b4b6-73d2cd0f3b2f","name":"John Smith","orders":
[{"id":"7dc908c0-9838-4d36-b912-0086962dbff6","name":"Earl Grey",
"customerId":"99e554eb-d699-4e49-b4b6-73d2cd0f3b2f"},{"id":"d78fb7d5-f769-4a11\
-aff4-a298f2997c86","name":"Kettle",
"customerId":"99e554eb-d699-4e49-b4b6-73d2cd0f3b2f"}]}
```

The app returns the array with all the orders that are related to our customer!

Now let's test the recursive relation. First we will create a 'Category' object.

```
$ curl -X POST "http://[::1]:3000/categories" -H "accept: application/json"\
-H "Content-Type:
application/json" -d "{\"name\":\"Humour\"}"


{
"id": "5d764111-fbd3-455c-b183-0147e9bbf39a",
"name": "Humour"
}
```

And then we will create another category which will be a subcategory of the first one.

```
$ curl -X POST "http://[::1]:3000/categories" -H "accept:
application/json" -H "Content-Type: application/json" -d "{\"name\":\"Dark Humour\",\
"parentId\":\"5d764111-fbd3-455c-b183-0147e9bbf39a\"}"

{
"id": "9c6fd1e3-00a6-4f99-b169-2f22f187c18a",
"name": "Dark Humour",
"parentId": "5d764111-fbd3-455c-b183-0147e9bbf39a"
}
```

In our database:

```
\dt
List of relations
Schema |   Name   | Type  |  Owner
--------+----------+-------+----------
public | category | table | username
public | customer | table | username
public | order    | table | username
(3 rows)
```

```
database=# SELECT * FROM "category";
id                   |    name     |               parentid
--------------------------------------+-------------+--------------------------------------
5d764111-fbd3-455c-b183-0147e9bbf39a | Humour      |
9c6fd1e3-00a6-4f99-b169-2f22f187c18a | Dark Humour | 5d764111-fbd3-455c-b183-0147e9bbf39a
```

We can see here, how lb4 manages this recursive relation in Postgres.

It creates the field parentId where it will store the id with the foreign key of the parent category. If it is empty it means it does not have a parent id.

3.4.4.2 MongoDB

Now we will create the same objects in a Mongo database to see how LoopBack manages relations in non relational databases.

The Customers will look like this:

```
> db.Customer.find()
{ "_id" : "1f22dbc4-9534-43c5-a4d5-7906c98fb1e3", "name" : "John Smith" }
```

And the orders will look like this:

```
> db.Order.find()
{ "_id" : "a3709b12-e16a-40a4-add1-46365ce59359", "name" : "Earl Grey",
"customerId" : "1f22dbc4-9534-43c5-a4d5-7906c98fb1e3" }
```

And the recursive relations look like this:

```
> show collections
Category
> db.Category.find()
{ "_id" : "e711b72d-94ff-46d9-9b7b-0b141202b870", "name" : "Humour" }
{ "_id" : "535c3e21-5d54-4533-839f-22738ae77386", "name" : "Dark Humour",
"parentId" : "e711b72d-94ff-46d9-9b7b-0b141202b870" }

{ "_id" : "0af20cff-593c-4778-8fcc-8230b6d1cae5", "name" : "Absurd Humour",
"parentId" : "e711b72d-94ff-46d9-9b7b-0b141202b870" }
```

We can see that in a non-relational db lb4 decides to store the data in a very similar way than in the relational database. The only difference is that the empty fields in a relational database (ex: parentId when the category is not a subcategory) will not be found in the mongo database.

3.4.4.3 Nested Relations

A nested relation would be a model which had a hasOneRelation or hasManyRelation with an other model already having a relation of this kind with a third model.

Due to lack of both time and documentation we were not able to make work a nested relation.

3.5 Migrations

The 'app.migrateSchema()' function is used to make migrations.

This functiona accepts an object of the shape 'SchemaMigrationOptions'. It is described like this:

```
export interface SchemaMigrationOptions extends Options {
    /**
    * When set to 'drop', schema migration will drop existing tables and recreate
    * them from scratch, removing any existing data along the way.
    *
    * When set to 'alter', schema migration will try to preserve current schema
    * and data, and perform a non-destructive incremental update.
    */
    existingSchema?: 'drop' | 'alter';

    /**
    * List of model names to migrate.
    *
    * By default, all models are migrated.
    */
    models?: string[];
}
```

So if we call our migrateSchema function like this:

```
await app.migrateSchema({existingSchema: 'drop',  models:['Customer', 'Order']});
```

We are explicitly calling auto-migrate function. Otherwise we should call:

```
await app.migrateSchema({existingSchema: 'alter',  models:['Customer', 'Order']});
```

Although it seems that this last feature it's not fully implemented yet, at least for the connectors we have seen.

3.5.1 Migration at Boot Time

The entry point for the application is the index.ts file in which you have the following lines of code:

```
// src/index.ts
await app.boot(); // searches and binds artifacts
await app.start();
```

We can call migrateSchema to make migrations at boot time:

```
// src/index.ts
await app.boot(); // searches and binds artifacts
await app.migrateSchema(); // migrates the Models to the databases creating the tables
await app.start();
```

However, it is usually better to have more control about the database migration and trigger the updates explicitly.

3.5.2 Database Explicit Migration

The order of table creation is important to make the migrations:

```
    // src/migrate.ts
    // Replace the following line:
    await app.migrateSchema({existingSchema});

    //with:
    await app.migrateSchema({
        existingSchema,
        // The order of table creation is important.
        // A referenced table must exist before creating a foreign key constraint.
        // For PostgreSQL connector, it does not create tables in the right order.
        // Therefore, this change is needed.
        models: ['Customer', 'Order'],existingSchema: {'alter'} //. 'alter' or 'drop'
    });
```

Then, to create the tables for `Customer` and `Order`, execute the following:

```
myproject$ npm run build
myproject$ npm run migrate
```

Chapter 4

Budget

.

4.1 Equipment

Object	Price
Laptop	950 €

4.2 Salary

To calculate an approximation of the salary we will assume that a junior engineer earns about 10€ per hour. The duration of this project has been about 19 weeks with a weekly dedication of 25h. This gives us the approximate amount of 5750€ .

4.3 Total

Concepto	Coste
Laptop	950 €
Salary	4750 €
Total	5700 €

Chapter 5

Conclusions

This project has been about documenting how the LoopBack 4 framework manages its ORM, by doing so we have touched some areas and technologies other than LoopBack itself. Here are the conclusions extracted from them:

- **SQL vs Non-SQL:** There is no absolute winner between them. The better choice will depend on the use you plan to give to them.
- **Docker** is a clean and quite simple solution to run services in a machine witout worrying about breaking things.
- **JavaScript** can be used not just in the front-end but also in the back-end.

About LoopBack 4 we can say:

- **It lacks documentation:** And the little you can find is very chaotic. This is a serious problem when talking about a framework. It is really difficult to make any progress in this disinformation situation.
- It lacks support on the ORM:
  - Even though there are a lot of connectors (every principal database has support from LoopBack), they lack on functionalities, **migrations are poorly resolved** as the migration without data-loss does not work. This could end up in production data loss.
  - **The Juggler is not aware of relations**. This means we have to manage them on each controller and repository. By each relation we must edit a bunch of files and it is tedious. This could be "easily" automatized by the framework.
  - There are problems with nested relations as commented before.

That being said, we must consider this framework is still developing, and it has lots of other functionalities which are better resolved than its ORM. If we aim to have a simple API it is a good option to consider.

Also I would like to give my personal evaluation. It has been a great opportunity to learn about SQL, MongodB, API, JavaScript, TypeScript, ORM, and LoopBack. I knew near to nothing about all them at the beginning of this project and now I have a very valuable insight about them.

5.1 Future Lines of Work

This project could be extended in the following ways:

- Continuing the documentation on LoopBack 4. There so many more great functionalities offered by the framework yet to be explored, and documented.
- Contributing the LoopBack community by designing a support for the migrations for LoopBack 4.
- Exploring other frameworks and compare them.

Bibliography

[1] Database . https://searchsqlserver.techtarget.com/definition/database . Last access 2019.

[2] ORM . en.wikipedia.org/wiki/Object-relational-mapping. Last access 2019.

[3] Docker. https://www.docker.com/. Last access 2019.

[4] Has Many Relation. https://github.com/strongloop/loopback-next/blob/master/docs/site/HasMany-relation.md. Last access 2019.

[5] Has One Relation. https://github.com/strongloop/loopback-next/blob/master/docs/site/hasOne-relation.md. Last access 2019.

[6] LoopBack 4. https://loopback.io/doc/en/lb4/. Last access 2019.

[7] Academind. SQL and non-SQL . https://youtu.be/ZS_kXvOeQ5Y, 2018. Last access 2019.