Purwadhika
Digital Technology School

Full Stack Web Development

Database ORM

Job Connector Program

Database ORM

Purwadhika
Digital Technology School

ORM stands for object-relational mapping, it might seem complex, but its purpose is to make your life as a programmer easier. To get data out of a database, you need to write a query. Does that mean you have to learn SQL? Well, no. **Object relational mapping makes it possible for you to write queries in the language of your choice**.

There are many types of ORM: Knex.js, Sequelize, Mongoose, TypeORM, Prisma, etc

Database ORM

If you're building a small project, installing an ORM library isn't required. Using SQL statements to drive your application should be sufficient. **An ORM is quite beneficial for medium- to large-scale projects that source data from hundreds of database tables**. In such a situation, you need a framework that allows you to **operate and maintain your application's data layer in a consistent and predictable way**.

ORM Libraries

Purwadhika
Digital Technology School

ORM is commonly undertaken with help of a library. The term ORM most commonly refers to an actual ORM library — an object relational mapper — that carries out the work of object relational mapping for you.

Hence, using an ORM library to build your data layer helps ensure that the database will always remain in a consistent state. ORM libraries often contain many more essential features, such as:

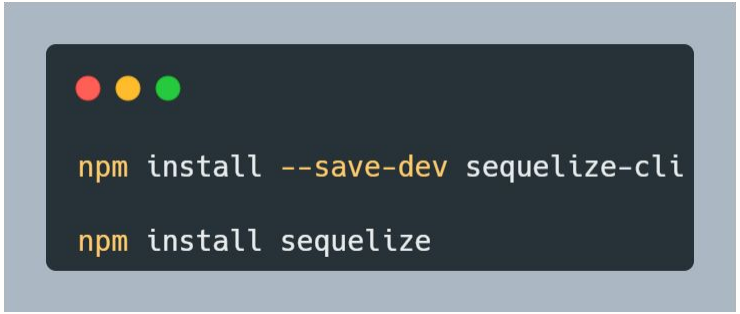
- Query builders
- Migration scripts
- CLI tool for generating boilerplate code
- Seeding feature for pre-populating tables with test data

Introduction to Sequelize

Purwadhika Digital Technology School

Sequelize is a modern TypeScript and Node.js ORM for Postgres, MySQL, MariaDB, SQLite and SQL Server, and more. Featuring solid transaction support, relations, eager and lazy loading, read replication and more.

Find out more: https://sequelize.org/

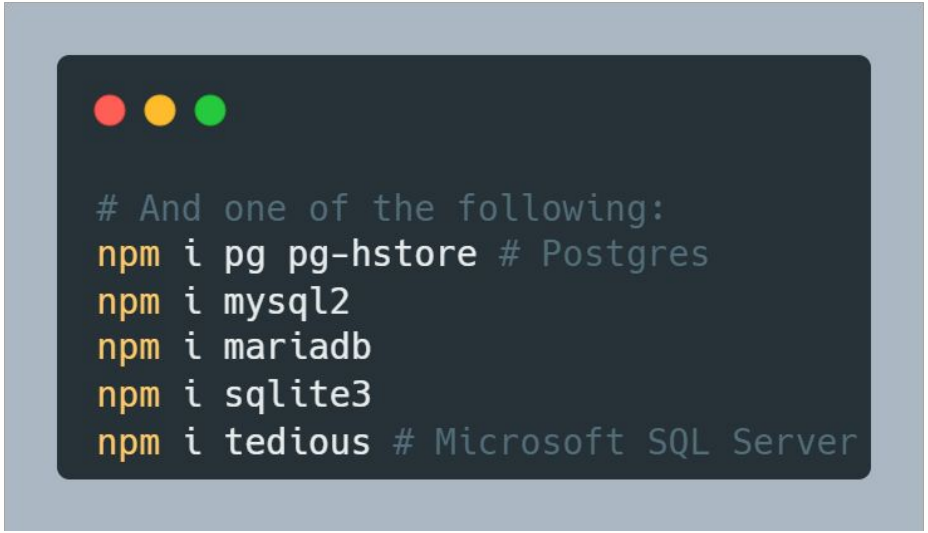


Sequelize - Getting Started

Purwadhika
Digital Technology School

Sequelize is a very mature and popular Node.js ORM library with excellent documentation containing well explained code examples. There are several database that could handle by Sequelize:

- Postgres
- Mysql
- Mariadb
- Sqlite
- tedious



Sequelize - Getting Started

Purwadhika
Digital Technology School

In the last command, it would create this following folder:

- **config**, contains config file, which tells CLI how to connect with database
- **models**, contains all models for your project
- **migrations**, contains all migration files
- **seeders**, contains all seed files

Getting Started

Purwadhika
Digital Technology School

```
{
  "development": {
    "username": "root",
    "password": null,
    "database": "database_development",
    "host": "127.0.0.1",
    "dialect": "mysql"
  },
}
```

Before continuing further we will need to tell the CLI how to connect to the database. To do that let's open default config file config/config.json.

There are several environment connection (development, testing, production). The keys of the objects (e.g. "development") are used on **model/index.js** for matching process.env.NODE_ENV (When undefined, "development" is a default value).

Getting Started

Purwadhika
Digital Technology School

```
{
  "test": {
    "username": "root",
    "password": null,
    "database": "database_test",
    "host": "127.0.0.1",
    "dialect": "mysql"
  },
  "production": {
    "username": "root",
    "password": null,
    "database": "database_production",
    "host": "127.0.0.1",
    "dialect": "mysql"
  }
}
```

Note that the Sequelize CLI assumes mysql by default. If you're using another dialect, you need to change the content of the "dialect" option.

Getting Started

Purwadhika
Digital Technology School

```
{
  "development": {
    "username": "root",
    "password": null,
    "database": "database_development",
    "host": "127.0.0.1",
    "dialect": "mysql"
  },
}
```

Adjust username, password, database, host, and dialect based on your own database. If the data is valid, you just connect your database into your projects successfully.

Sequelize - Migration

Purwadhika
Digital Technology School

Just like you use **version control systems** such as Git to manage changes in your source code, **you can use migrations to keep track of changes to the database**.

You will need the Sequelize Command-Line Interface (CLI). The CLI ships support for migrations and project bootstrapping.

A Migration in Sequelize is a javascript file which exports two functions, up and down, that dictates how to perform the migration and undo it.

Sequelize - Migration

Purwadhika
Digital Technology School

There are several command in migration could be used

```
sequelize-cli --version
sequelize-cli db:migrate
sequelize-cli db:migrate:status
sequelize-cli db:migrate:undo
sequelize-cli db:migrate:undo:all
sequelize-cli migration:create --name name-of-migration
sequelize-cli migration:generate --name name-of-migration
```

Sequelize - Migration

Purwadhika
Digital Technology School

Try to create your first migration. In this example we are would like to create a table name as branch through migration. Execute command below to generate the migration file. Open the file on migrations/xxxxxxxx-create-branch-table.js

```
sequelize-cli migration:generate --name create-branch-table
```

Sequelize - Migration

Purwadhika
Digital Technology School

Open migration file that we already create. There are two main points in migration:

- UP - executing or modify table(s).
- DOWN - revert back UP process.

```
'use strict';

module.exports = {
  async up (queryInterface, Sequelize) {
    /**
     * Add altering commands here.
     *
     * Example:
     * await queryInterface.createTable('users', { id: Sequelize.INTEGER });
     */
  },

  async down (queryInterface, Sequelize) {
    /**
     * Add reverting commands here.
     *
     * Example:
     * await queryInterface.dropTable('users');
     */
  }
};
```

Sequelize - Migration

Purwadhika
Digital Technology School

Modify your migration file, in this case we will create table named as Branchs where the table have id, branchName, address field.

Notice that **up** using createTable method while **down** using dropTable method. **Up** method would generate the table Branchs while **down** method would execute rollback action from create which is delete table Branchs.

```
'use strict';
module.exports = {
  async up(queryInterface, Sequelize) {
    await queryInterface.createTable('Branchs', {
      id: {
        allowNull: false,
        autoIncrement: true,
        primaryKey: true,
        type: Sequelize.INTEGER
      },
      branchName: {
        type: Sequelize.STRING
      },
      address: {
        type: Sequelize.STRING
      },
    });
  },
  async down(queryInterface, Sequelize) {
    await queryInterface.dropTable('Branchs');
  }
};
```

Sequelize - Migration

Purwadhika Digital Technology School

In order to execute your migration, run this command. It will executing **UP** method from your latest migration. In this case, you just successfully create branch table on your database.

```
sequelize-cli db:migrate
```

```
D:\jc-student\sample-sequelize>sequelize-cli db:migrate

Sequelize CLI [Node: 16.15.1, CLI: 6.4.1, ORM: 6.23.0]

Loaded configuration file "config\config.json".
Using environment "development".
== 20220921045809-create-branch-table: migrating =======
== 20220921045809-create-branch-table: migrated (0.055s)
```

Sequelize - Model

Purwadhika Digital Technology School

Models are the essence of Sequelize. **A model is an abstraction that represents a table in your database**. In Sequelize, it is a class that extends Model.

The model tells Sequelize several things about the entity it represents, such as the name of the table in the database and which columns it has (and their data types).

A model in Sequelize has a name. This name does not have to be the same name of the table it represents in the database. **Usually, models have singular names (such as User) while tables have pluralized names (such as Users)**, although this is fully configurable.

Sequelize - Model

Purwadhika
Digital Technology School

Models can be defined in two equivalent ways in Sequelize:

- Calling sequelize.define(modelName, attributes, options)
- Extending Model and calling init(attributes, options)

After a model is defined, it is available within **sequelize.models** by its model name.

Sequelize – Model sequelize.define

Purwadhika
Digital Technology School

To learn with an example, we will consider that we want to create a model to represent users, which have a firstName and a lastName. We want our model to be called User, and the table it represents is called Users in the database.

Both ways to define this model are shown. After being defined, we can access our model with **sequelize.models.User**.

/models/User.js

```
module.exports = (sequelize, Sequelize) => {
  const User = sequelize.define('User', {
    // Model attributes are defined here
    firstName: {
      type: Sequelize.STRING,
      allowNull: false,
    },
    lastName: {
      type: Sequelize.STRING,
      // allowNull defaults to true
    },
  });
  // `sequelize.define` also returns the model
  return User;
};
```

Sequelize - Model Extending model

Purwadhika Digital Technology School

Internally, sequelize.define calls Model.init, so both approaches are essentially equivalent.

/models/User.js

```
const { DataTypes, Model } = require('sequelize');

class User extends Model {}

module.exports = (sequelize, Sequelize) => {
  const user = User.init({
      // Model attributes are defined here
      firstName: {
        type: DataTypes.STRING,
        allowNull: false
      },
      lastName: {
        type: DataTypes.STRING
        // allowNull defaults to true
      }
    },
    {
    // Other model options go here
    sequelize, // We need to pass the connection instance
    modelName: 'User' // We need to choose the model name
    }
 );
  return user;
};
```

Sequelize - Model

Purwadhika
Digital Technology School

Observe that, in both methods, the table name (Users) was never explicitly defined. However, the model name was given (User).

By default, when the table name is not given, Sequelize automatically pluralizes the model name and uses that as the table name. This pluralization is done under the hood by a library called inflection, so that irregular plurals (such as person -> people) are computed correctly.

Sequelize - Model Synchronization

Purwadhika
Digital Technology School

When you define a model, you're telling Sequelize a few things about its table in the database. However, what if the table actually doesn't even exist in the database? What if it exists, but it has different columns, less columns, or any other difference?

A model can be synchronized with the database by calling model.sync(options), an asynchronous function (that returns a Promise). With this call, Sequelize will automatically perform an SQL query to the database. **Note that this changes only the table in the database, not the model in the JavaScript side**.

- **User.sync()** - This creates the table if it doesn't exist (and does nothing if it already exists)
- **User.sync({ force: true })** - This creates the table, dropping it first if it already existed
- **User.sync({ alter: true })** - This checks what is the current state of the table in the database (which columns it has, what are their data types, etc), and then performs the necessary changes in the table to make it match the model.

Sequelize – Model Synchronization

Purwadhika
Digital Technology School

index.js

```
const db = require('./models');

const User = db.User;

User.sync();
```

index.js

```
const db = require('./models');

const User = db.User;

User.sync({ alter: true });
```

index.js

```
const db = require('./models');

const User = db.User;

User.sync({ force: true });
```

Creating Model Through Sequelize-cli

Purwadhika
Digital Technology School

There are several ways to create a model, generate through the sequelize-cli command and generate by create model file directly under folder models.

In this case, we will use **model:generate** command. This command requires two options:

- **name**: the name of the model.
- **attributes**: the list of model attributes.

Let's create a model named User.

```
sequelize-cli model:generate --name User --attributes firstName:string,lastName:string,email:string
```

Creating the First Model Through Sequelize-cli

Purwadhika
Digital Technology School

Based on the command, this will:

- Create a model file user in models folder;
- Create a migration file with name like XXXXXXXXXXXXXX-create-user.js in migrations folder.

Note: Sequelize will only use Model files, it's the table representation. On the other hand, the migration file is a change in that model or more specifically that table, used by CLI. Treat migrations like a commit or a log for some change in database.

```
sequelize-cli model:generate --name User --attributes firstName:string,lastName:string,email:string
```

Run Migration

Purwadhika
Digital Technology School

We have just created the required model and migration files for our first model, User. Now to actually create that table in the database you need to run db:migrate command.

This command will execute these steps:

- Will ensure a table called SequelizeMeta in database. This table is used to record which migrations have run on the current database
- Start looking for any migration files which haven't run yet. This is possible by checking SequelizeMeta table. In this case it will run XXXXXXXXXXXXXX-create-user.js migration, which we created in last step.
- Creates a table called Users with all columns as specified in its migration file.

Model Querying - Basics

Purwadhika
Digital Technology School

Sequelize provides various methods to assist querying your database for data.

Important notice: to perform production-ready queries with Sequelize, make sure you have read the Transactions guide as well. Transactions are important to ensure data integrity and to provide other benefits.

Model Querying Basics – Insert

Purwadhika
Digital Technology School

index.js

```
const db = require('./models');

const User = db.User;

const createUser = async () => {
  const result = await User.create({ firstName: 'jon', lastName: 'wood' });
  console.log(result);
};
createUser();
```

```
INSERT INTO `Users` (`id`,`firstName`,`lastName`,`createdAt`,`updatedAt`) VALUES (DEFAULT,?,?,?,?);
```

Model Querying Basics – Select

Purwadhika
Digital Technology School

index.js

```
const db = require('./models');

const User = db.User;

const findUsers = async () => {
  const result = await User.findAll();
  console.log(result);
};
findUsers();
```

```
SELECT `id`, `firstName`, `lastName`, `createdAt`, `updatedAt` FROM `Users` AS `User`;
```

Model Querying Basics – Select Specifying Attributes

Purwadhika
Digital Technology School

index.js

```
const db = require('./models');

const User = db.User;

const findUsers = async () => {
  const result = await User.findAll({ attributes: ['firstName'] });
  console.log(result);
};
findUsers();
```

```
SELECT `firstName` FROM `Users` AS `User`;
```

Model Querying Basics – Select Specifying Attributes

Purwadhika
Digital Technology School

index.js

```
const db = require('./models');

const User = db.User;

const findUsers = async () => {
  const result = await User.findAll({ attributes: { exclude: ['firstName'] } });
  console.log(result);
};
findUsers()
```

```
SELECT `id`, `lastName`, `createdAt`, `updatedAt` FROM `Users` AS `User`;
```

Model Querying Basics – Select Specifying Attributes

Purwadhika
Digital Technology School

index.js

```
const db = require('./models');
const sequelize = require('sequelize');
const User = db.User;

const findUsers = async () => {
  const result = await User.findAll({
    attributes: [sequelize.fn('COUNT', sequelize.col('id'))],
  });
  console.log(result);
};
findUsers();
```

```
SELECT COUNT(`id`) FROM `Users` AS `User`;
```

Model Querying Basics – Select Where Clause

Purwadhika
Digital Technology School

index.js

```
const db = require('./models');
const User = db.User;

const findUsers = async () => {
  const result = await User.findAll({
    where: {
      id: 1,
    },
  });
  console.log(result);
};
findUsers();
```

```
SELECT `id`, `firstName`, `lastName`, `createdAt`, `updatedAt` FROM `Users` AS `User` WHERE `User`.`id` = 1;
```

Model Querying Basics – Select Where Clause

Purwadhika
Digital Technology School

index.js

```
const db = require('./models');
const sequelize = require('sequelize');
const User = db.User;

const findUsers = async () => {
  const result = await User.findAll({
    where: {
      firstName: {
        [sequelize.Op.eq]: 'j',
      },
    },
  });
  console.log(result);
};
findUsers()
```

```
SELECT `id`, `firstName`, `lastName` FROM `Users` AS `User` WHERE `User`.`firstName` = 'j';
```

Query Operators

Purwadhika
Digital Technology School

```
[Op.and]: [{ a: 5 }, { b: 6 }],            // (a = 5) AND (b = 6)
[Op.or]: [{ a: 5 }, { b: 6 }],             // (a = 5) OR (b = 6)
// Basics
[Op.eq]: 3,                               // = 3
[Op.ne]: 20,                              // != 20
[Op.is]: null,                            // IS NULL
[Op.not]: true,                           // IS NOT TRUE
[Op.or]: [5, 6],                          // (someAttribute = 5) OR (someAttribute = 6)

// Using dialect specific column identifiers (PG in the following example):
[Op.col]: 'user.organization_id',         // = "user"."organization_id"

// Number comparisons
[Op.gt]: 6,                               // > 6
[Op.gte]: 6,                              // >= 6
[Op.lt]: 10,                              // < 10
[Op.lte]: 10,                             // <= 10
[Op.between]: [6, 10],                    // BETWEEN 6 AND 10
[Op.notBetween]: [11, 15],                // NOT BETWEEN 11 AND 15
```

Query Operators

Purwadhika
Digital Technology School

```
[Op.all]: sequelize.literal('SELECT 1'), // > ALL (SELECT 1)

[Op.in]: [1, 2],                         // IN [1, 2]
[Op.notIn]: [1, 2],                      // NOT IN [1, 2]

[Op.like]: '%hat',                       // LIKE '%hat'
[Op.notLike]: '%hat',                    // NOT LIKE '%hat'
[Op.startsWith]: 'hat',                  // LIKE 'hat%'
[Op.endsWith]: 'hat',                    // LIKE '%hat'
[Op.substring]: 'hat',                   // LIKE '%hat%'
[Op.iLike]: '%hat',                      // ILIKE '%hat' (case insensitive) (PG only)
[Op.notILike]: '%hat',                   // NOT ILIKE '%hat'  (PG only)
[Op.regexp]: '^[h|a|t]',                 // REGEXP/~ '^[h|a|t]' (MySQL/PG only)
[Op.notRegexp]: '^[h|a|t]',              // NOT REGEXP/!~ '^[h|a|t]' (MySQL/PG only)
[Op.iRegexp]: '^[h|a|t]',                // ~* '^[h|a|t]' (PG only)
[Op.notIRegexp]: '^[h|a|t]',             // !~* '^[h|a|t]' (PG only)

[Op.any]: [2, 3],                        // ANY (ARRAY[2, 3]::INTEGER[]) (PG only)
[Op.match]: Sequelize.fn('to_tsquery', 'fat & rat') // match text search for strings 'fat' and 'rat' (PG only)

// In Postgres, Op.like/Op.iLike/Op.notLike can be combined to Op.any:
[Op.like]: { [Op.any]: ['cat', 'hat'] }  // LIKE ANY (ARRAY['cat', 'hat'])
```

Model Querying Basics – Update

Purwadhika
Digital Technology School

index.js

```
const db = require('./models');
const User = db.User;

const updateUser = async () => {
  const result = await User.update(
    {
      firstName: 'Tommy',
    },
    {
      where: {
        id: 1,
      },
    },
  );
};
updateUser();
```

```
UPDATE `Users` SET `firstName`=?,`updatedAt`=? WHERE `id` = ?
```

Update queries also accept the where option, just like the read queries shown above.

Model Querying Basics – Delete

Purwadhika
Digital Technology School

index.js

```
const db = require('./models');
const User = db.User;

const deleteUser = async () => {
  await User.destroy({
    where: {
      id: 1,
    },
  });
};
deleteUser();
```

```
DELETE FROM `Users` WHERE `id` = 1
```

Delete queries also accept the where option, just like the read queries shown above.

Model Querying Basics – Delete

Purwadhika
Digital Technology School

index.js

```
const db = require('./models');
const User = db.User;

const deleteUser = async () => {
  await User.destroy({
    truncate: true
  });
};
```

```
TRUNCATE `Users`
```

To destroy everything the TRUNCATE SQL can be used this script

Association

Purwadhika
Digital Technology School

Sequelize supports the standard associations: One-To-One, One-To-Many and Many-To-Many.

To do this, Sequelize provides four types of associations that should be combined to create them:

- The HasOne association
- The BelongsTo association
- The HasMany association
- The BelongsToMany association

Association

Purwadhika
Digital Technology School

The four association types are defined in a very similar way. Let's say we have two models, A and B. Telling Sequelize that you want an association between the two needs just a function call:

```
const A = sequelize.define('A', /* ... */);
const B = sequelize.define('B', /* ... */);

A.hasOne(B); // A HasOne B
A.belongsTo(B); // A BelongsTo B
A.hasMany(B); // A HasMany B
A.belongsToMany(B, { through: 'C' }); // A BelongsToMany B through the junction table C
```

Association – Standard Relationship

Purwadhika
Digital Technology School

As mentioned, usually the Sequelize associations are defined in pairs. In summary:

- To create a One-To-One relationship, the hasOne and belongsTo associations are used together
- To create a One-To-Many relationship, the hasMany and belongsTo associations are used together
- To create a Many-To-Many relationship, two belongsToMany calls are used together.

Note: there is also a Super Many-To-Many relationship, which uses six associations at once, and will be discussed in the Advanced Many-to-Many relationships guide.

Transactions

Purwadhika
Digital Technology School

Sequelize does not use transactions by default. However, for production-ready usage of Sequelize, you should definitely configure Sequelize to use transactions.

Sequelize supports two ways of using transactions:

- **Unmanaged transactions**: Committing and rolling back the transaction should be done manually by the user (by calling the appropriate Sequelize methods).
- **Managed transactions**: Sequelize will automatically rollback the transaction if any error is thrown, or commit the transaction otherwise. Also, if CLS (Continuation Local Storage) is enabled, all queries within the transaction callback will automatically receive the transaction object.

Transactions – Unmanaged Transactions

Purwadhika
Digital Technology School

```
const db = require('./models');
const User = db.User;

const severalAction = async () => {
  const t = await db.sequelize.transaction();
  try {
    await User.create(
      {
        firstName: 'Bart',
        lastName: 'Simpson',
      },
      { transaction: t },
    );
    await User.create(
      {
        firstName: 'Petter',
        lastName: 'Griffin',
      },
      { transaction: t },
    );
    await t.commit();
  } catch (error) {
    await t.rollback();
  }
};
severalAction();
```

As shown, the unmanaged transaction approach requires that you commit and rollback the transaction manually, when necessary.

Transactions – Managed Transactions

Purwadhika
Digital Technology School

```
const db = require('./models');
const User = db.User;

const severalAction = async () => {
  try {
    const result = await db.sequelize.transaction(async () => {
      await User.create(
        {
          firstName: 'Bart',
          lastName: 'Simpson',
        },
        { transaction: t },
      );
      await User.create(
        {
          firstName: 'Petter',
          lastName: 'Griffin',
        },
        { transaction: t },
      );
    });
    // If the execution reaches this line, the transaction has been committed successfully
    // `result` is whatever was returned from the transaction callback (the `user`, in this case)
  } catch (error) {
    // If the execution reaches this line, an error occurred.
    // The transaction has already been rolled back automatically by Sequelize!
  }
};
severalAction();
```

Note that t.commit() and t.rollback() were not called directly (which is correct).

Eager Loading

Purwadhika
Digital Technology School

Eager Loading is the act of querying data of several models at once (one 'main' model and one or more associated models). At the SQL level, this is a query with one or more joins. Lets assume we have this setup:

```
const User = sequelize.define('user', { name: DataTypes.STRING }, { timestamps: false });
const Task = sequelize.define('task', { name: DataTypes.STRING }, { timestamps: false });
const Tool = sequelize.define('tool', {
  name: DataTypes.STRING,
  size: DataTypes.STRING
}, { timestamps: false });
User.hasMany(Task);
Task.belongsTo(User);
User.hasMany(Tool, { as: 'Instruments' });
```

Eager Loading

Purwadhika
Digital Technology School

```
const tasks = await Task.findAll({ include: User });
console.log(JSON.stringify(tasks, null, 2));
```

This will load all the task with their associated user.

```
[{
  "name": "A Task",
  "id": 1,
  "userId": 1,
  "user": {
    "name": "John Doe",
    "id": 1
  }
}]
```

Lazy Loading

Purwadhika
Digital Technology School

Lazy Loading refers to the technique of fetching the associated data only when you really want it

With this way we can save time and memory by only fetching it when necessary.

```
const awesomeCaptain = await Captain.findOne({
  where: {
    name: "Jack Sparrow"
  }
});
// Do stuff with the fetched captain
console.log('Name:', awesomeCaptain.name);
console.log('Skill Level:', awesomeCaptain.skillLevel);
// Now we want information about his ship!
const hisShip = await awesomeCaptain.getShip();
// Do stuff with the shi
console.log('Ship Name:', hisShip.name);
console.log('Amount of Sails:', hisShip.amountOfSails);
```

Thank You!

Purwadhika
Digital Technology School