POLITÉCNICO
DO PORTO
ESMAD

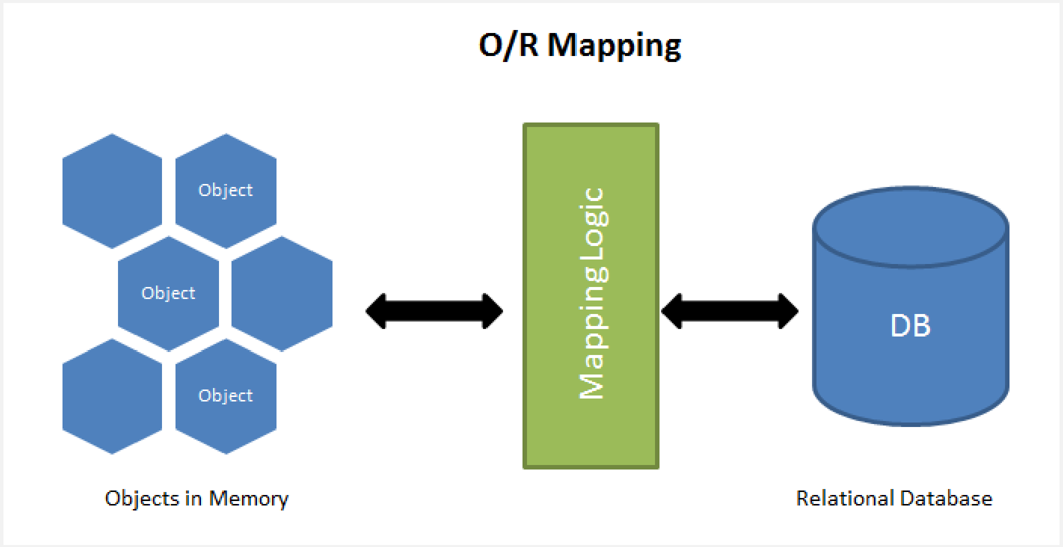
PROGRAMAÇÃO WEB II
TSIW

SUMMARY

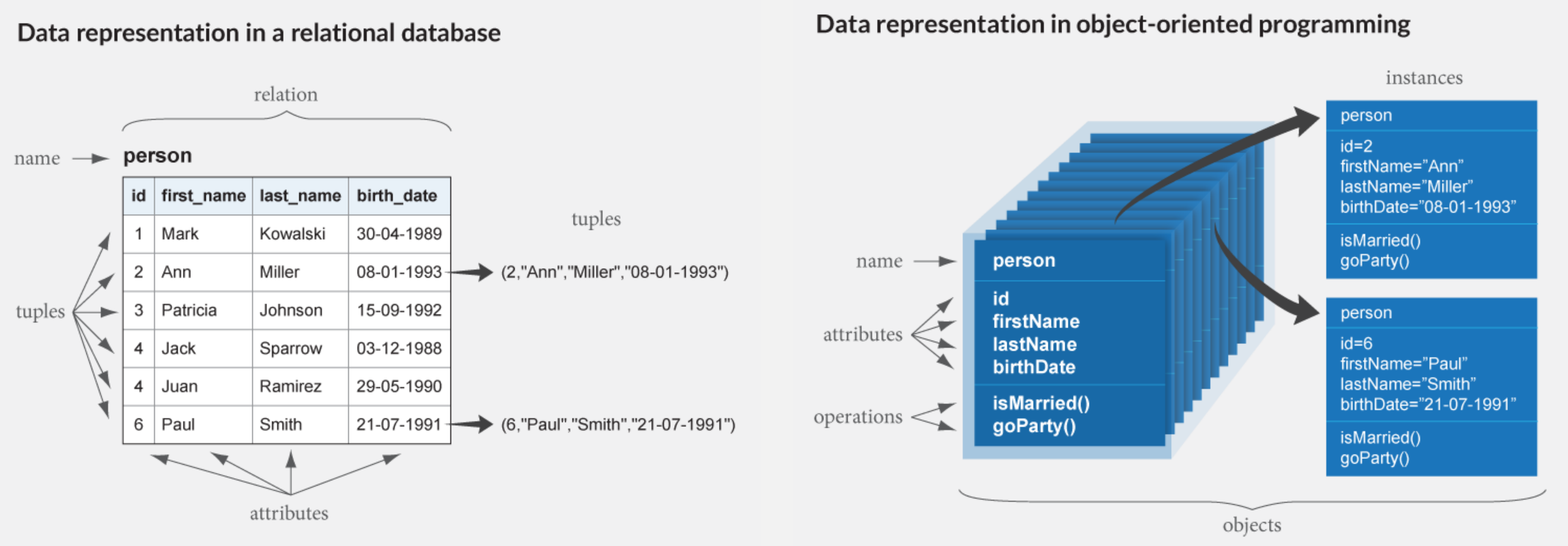
- ORM - Object-Relational Mapper
- Sequelize: promise-based Node.js ORM with support to MySQL DBs
- Relatioships with Sequelize

ORM

- Object-Relational Mapper: library that allows to write queries using the object-oriented paradigm of your preferred programming language
  - programming technique for converting data between incompatible type systems using object-oriented programming languages
  - ORM sets the mapping between the set of objects which are written in a programming language like JavaScript and a relational database like MySQL
  - It hides and encapsulates the SQL queries into "virtual database objects" and, instead of SQL queries, one can directly use the objects' methods to implement the SQL query



ORM



id	first_name	last_name	birth_date
1	Mark	Kowalski	30-04-1989
2	Ann	Miller	08-01-1993
3	Patricia	Johnson	15-09-1992
4	Jack	Sparrow	03-12-1988
4	Juan	Ramirez	29-05-1990
6	Paul	Smith	21-07-1991

ORM: pros and cons

- Advantages of using ORMs:
  - let the developer think in terms of **objects rather than tables**
  - **no need to write SQL code**
  - **advanced features** like eager or lazy loading, soft deletion, ...
  - **database independent**: no need to write code specific to a particular database
  - reduces code and allows developers to **focus on their business logic**, rather than complex database queries
  - most ORMs provide a **rich query interface**

ORM: pros and cons

- Disadvantages of using ORMs:
  - **complex** (because they handle a bidirectional mapping); their complexity implies a grueling learning curve – they have a special query language which developers must learn
  - provides only a **leaky abstraction** over a relational data store
  - systems using an ORM can perform badly if, due to naive interactions with the underlying database, **one uses ORM without knowing SQL**
  - ORM, by adding a layer of abstraction, speeds up the development but **adds overhead to the application**

ORM - Sequelize

- Sequelize: promise-based Node.js ORM for relational databases like Postgres, MySQL, MariaDB, SQLite, Microsoft SQL Server, among others
  - ➢ Has been around for a long time – 2011, has thousands of GitHub stars and is used by many applications
  - ➢ It is stable and has plenty of documentation available online
  - ➢ Sequelize has a large feature set that covers: queries, scopes, relations, transactions, raw queries, migrations, read replication, etc.
- **Installation**: sequelize is available via npm

```
npm install --save sequelize
```

You'll also have to manually install the driver for your database of choice:

```
npm install --save mysql2    #for MySQL databases
```

Comparison between mysql and mysql2 node modules: https://npmcompare.com/compare/mysql,mysql2

Sequelize

- Simple example: connection to a MySQL database and create a DB entry

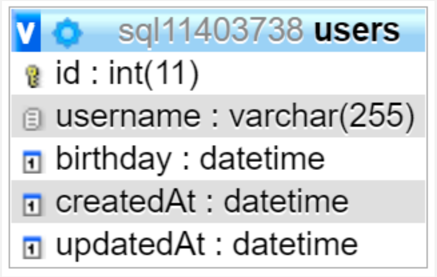
```
const { Sequelize, DataTypes } = require('sequelize');
const sequelize = new Sequelize('database', 'username', 'password',
{
  host: 'host',
  dialect: 'mysql'
});

const User = sequelize.define("user",
{
  username: DataTypes.STRING,
  birthday: DataTypes.DATE
});

(async () => {
  await sequelize.sync();
  const jane = await User.create({
    username: 'janedoe',
    birthday: new Date(1980, 6, 20)
  });
  console.log(jane.toJSON());
})();
```

1) Database connection

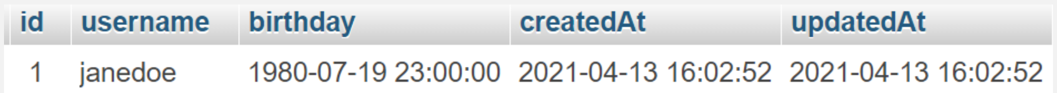
2) Model definition

creates the table if it doesn't exist (and does nothing if it already exists)

3) Querying: instantiate object and save it in database



sql11403738 **users**
- id : int(11)
- username : varchar(255)
- birthday : datetime
- createdAt : datetime
- updatedAt : datetime



id	username	birthday	createdAt	updatedAt
1	janedoe	1980-07-19 23:00:00	2021-04-13 16:02:52	2021-04-13 16:02:52

Sequelize

- **Connecting to a database**: to connect to the database, you must create a Sequelize instance
  - ➢ Example for MySQL databases

```
const sequelize = new Sequelize('database', 'username', 'password', {
    host: 'hostname',
    dialect: 'mysql'
});
```

Replace with your MySQL database credentials

  - ➢ **Terminology convention**: observe that Sequelize refers to the library itself while sequelize refers to an instance of Sequelize, which represents a connection to one database
  - ➢ This is the recommended convention, and is followed throughout the ORM documentation

Sequelize

- You can test the connection using authenticate(), which creates an instance to check whether the connection is working:

```
try {
    await sequelize.authenticate();
    console.log('Connection has been established successfully.');
 } catch (error) {
    console.error('Unable to connect to the database:', error);
 }
```

- **Closing the connection:** Sequelize will keep the connection open by default and use the same connection for all queries. If you need to close the connection, call sequelize.close()
  - Once called, it's impossible to open a new connection
  - For that, one will need to create a new Sequelize instance to access the database again

Sequelize: promises

- **Promises:** most of the methods provided by Sequelize are asynchronous and therefore return Promises
  - ➢ you can use the Promise API (using `then`, `catch`, `finally`)
  - ➢ or you can use `async` and `await` as well

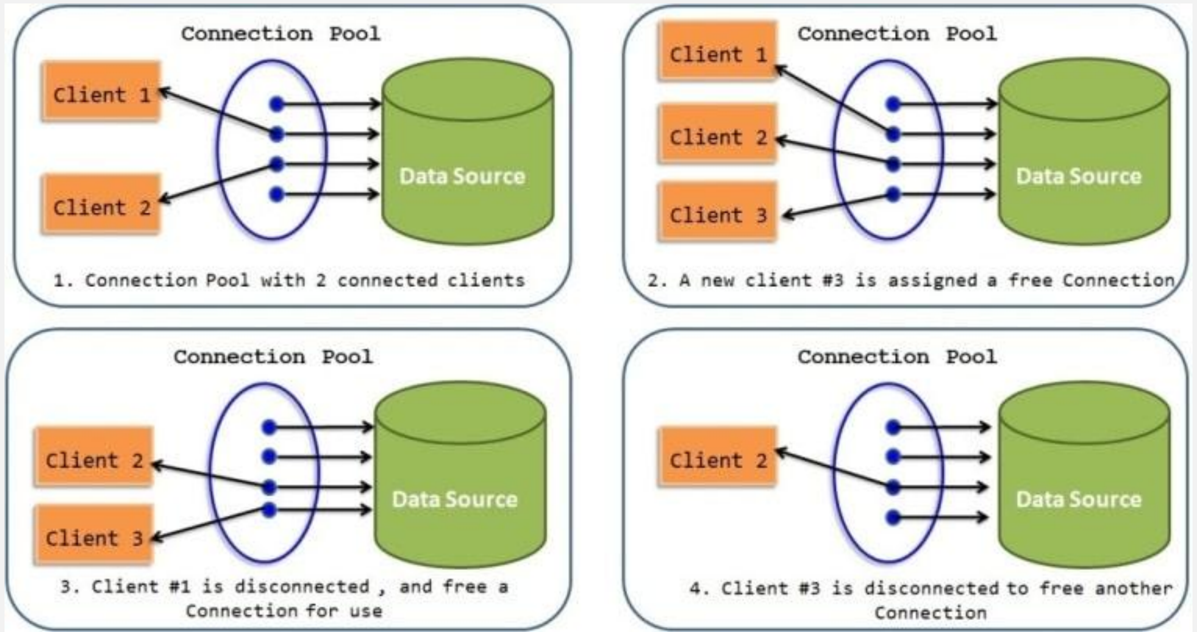
```
sequelize.authenticate()
    .then(() => {
        console.log('Connection has been established successfully.');
    })
    .catch(err => {
        console.error('Unable to connect to the database:', err);
    });
```

Two ways of using promises

```
(async () => {
    try {
        await sequelize.authenticate();
        console.log('Connection has been established successfully.');
    } catch (error) {
        console.error('Unable to connect to the database:', error);
    }
})();
```

Connection pooling

- Database connection pooling is a method used to keep database connections open so they can be reused by others
  - When an application starts, several connections are created and added to the pool
  - Then, the pool starts to assign connections for different requests. After it reached its maximum number of connections at the same time (something configured in the app), it waits until one of the connections is free again


Connection Pool
Client 1
Client 2
Data Source
1. Connection Pool with 2 connected clients
Connection Pool
Client 1
Client 2
Client 3
Data Source
2. A new client #3 is assigned a free Connection
Connection Pool
Client 2
Client 3
Data Source
3. Client #1 is disconnected , and free a Connection for use
Connection Pool
Client 2
Data Source
4. Client #3 is disconnected to free another Connection

Sequelize

- **Connection pooling:** Sequelize helps in the configuration of a connection pool

```
const sequelize = new Sequelize('database', 'username', 'password', {
     host: 'hostname',
     dialect: 'mysql',
     pool: {
          max: 5,   //maximum number of connections in pool
          min: 1,   //minimum number of connections in pool
          acquire: 30000, // maximum time (ms) that pool will try to get
connection before throwing error
          idle: 10000 // maximum time (ms) that a connection can be idle before
being released
          }
});
```

Sequelize - models basics

- Models are the essence of Sequelize, since a model is an abstraction that represents a table in the database
  - ➢ The model tells Sequelize several things about the entity it represents, such as the table name in the database and which columns it has (and their data types)
- Models can be defined in two equivalent ways in Sequelize:
  1. Calling `sequelize.define(modelName, attributes, options)`
  2. Extending class Model and calling `init(attributes, options)`
- After a model is defined, it is available within `sequelize.models` by its model name

Sequelize - models basics

- Model definition examples

```
const User = sequelize.define("User", {
      // ... (attributes)
}, {
      // other model options here
});
// `sequelize.define` also returns the model
console.log(User === sequelize.models.User); // true
```

Model name

```
class User extends Model {}
User.init({
  // ... (attributes)
}, {
  // other model options go here
  sequelize, // it is required to pass the connection instance
  modelName: 'User' // it is required to choose the model name
});

// the defined model is the class itself
console.log(User === sequelize.models.User); // true
```

Sequelize - models basics

- In the previous examples, the table name was not explicitly defined
- The model name in Sequelize 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*)
  - **Sequelize automatically pluralizes the model name** and uses that as the table name
  - And it is smart since models named *Person* will correspond to tables named *People*
- This behaviour however is fully configurable:

```
sequelize.define("User", {
      // ... (attributes)
}, {
      // table name is equal to the model name
      freezeTableName: true
});
```

```
sequelize.define("User", {
      // ... (attributes)
}, {
      // tell Sequelize the table name directly
      tableName: 'Employees'
});
```

Sequelize - models basics

- When you define a model, you're telling Sequelize a few things about its table in the database
  - ➢ 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?
- **Model synchronization**: call `model.sync(options)`, to synchronize the database with the model(s) definition(s)

```
// creates the table if it doesn't exist (and does nothing if it already exists)
User.sync();

// creates the table, dropping it first if it already existed
User.sync({ force: true });

// checks the table in the database (which columns it has, what are their data types, etc.),
and then performs the necessary changes to make it match the model
User.sync({ alter: true });

// automatically synchronize ALL models
sequelize.sync({ … });
```

Sequelize - models basics

```
// example showing synchronization of all models in a database
(async () => {
    try {
        await db.sequelize.sync();
        console.log('DB is successfully synchronized')
    } catch (error) {
        console.log(e)
    }
})();
```

- Model synchronization can perform **destructive operations**
  - they are not recommended for production-level software
- Synchronization should be done with the advanced concept of Migrations (keep track of changes to the database), with the help of the Sequelize CLI

Sequelize - models basics

- **Timestamps**: by default, Sequelize automatically adds the fields `createdAt` and `updatedAt` to every model, using the data type `DataTypes.DATE`
  - `createdAt`: timestamp representing the moment of creation
  - `updatedAt`: will contain the timestamp of the latest update
- This behaviour can be disabled when defining a model:

```
sequelize.define("User", {
     // ... (attributes)
}, {
     // table will NOT contain createdAt or updatedAt fields
     timestamps: false
});
```

Sequelize - models basics

- **Column declaration**: if the only thing being specified about a column is its data type, the syntax can be shortened

```
sequelize.define("tutorial", {
        title: {
            type: DataTypes.STRING
        }
});
```

```
sequelize.define("tutorial", {title: DataTypes.STRING});
```

column name   column data type

- Sequelize assumes that the default value of a column is NULL
  - ➢ This behavior can be changed by passing a specific `defaultValue` to the column definition

```
sequelize.define("User", {
        name: {
            type: DataTypes.STRING,
            defaultValue: "John Doe"
        }
});
```

```
sequelize.define("Meeting", {
        date: {
            type: DataTypes.DATETIME,
            defaultValue: Sequelize.NOW
        }
});
```

Sequelize datatypes

- Every column defined in the model must have a data type
- Import DataTypes to access a Sequelize built-in data type

```
const { DataTypes } = require("sequelize"); // Import the built-in data types
```

```
DataTypes.STRING                 // VARCHAR(255)
DataTypes.STRING(1234)           // VARCHAR(1234)
DataTypes.TEXT                   // TEXT
DataTypes.TEXT('tiny')           // TINYTEXT
DataTypes.BOOLEAN                // TINYINT(1)
DataTypes.INTEGER                // INTEGER
DataTypes.BIGINT                 // BIGINT
DataTypes.FLOAT                  // FLOAT
DataTypes.DOUBLE                 // DOUBLE
DataTypes.DECIMAL                // DECIMAL
DataTypes.INTEGER.UNSIGNED       // UNSIGNED INT
DataTypes.DATE                   // DATETIME
```

There are other data types, covered here

Sequelize datatypes

- When defining a column, apart from specifying its type, there are a lot more options that can be used

```
const Foo = sequelize.define("Foo", {
      // automatically set the flag to true if not set (allowNull: adds NOT NULL to the column)
     flag: { type: DataTypes.BOOLEAN, allowNull: false, defaultValue: true },

     // set primary key as autoincrementing integer column
     identifier: { type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true },

     // an attempt to insert a username that already exists will throw an error
     username: { type: DataTypes.TEXT, allowNull: false, unique: true }

     // create a foreign key
     bar_id: {
           type: DataTypes.INTEGER,
           references: {
                 model: Bar, // This is a reference to another model
                 key: 'id' // This is the column name of the referenced model
    }
});
```

Sequelize validators

- Validations are checks performed in the Sequelize level, in pure JavaScript
  - ➢ They can be arbitrarily complex if you provide a **custom validator function** or can be one of the **built-in validators** offered by Sequelize
  - ➢ Validations are automatically run on **create**, **update** and **save**. You can also call **validate()** to manually validate an instance
  - ➢ If a validation fails, no SQL query will be sent to the database at all

```
const Foo = sequelize.define("Foo", {
      // validates username length to be between 5 and 10 characters
     username: { type: DataTypes.STRING, validate: { len: [5, 10] } },

     // checks for email format (foo@bar.com)
     email: { type: DataTypes.STRING, validate: { isEmail: true } },

     // age must be >= 18 and set up a custom error message
     age: { type: DataTypes.INT, validate: { min: { args: 18, msg: "Must be of legal age" } } },

     language: {type: DataTypes.STRING, validate: { isIn: [['en', 'fr']] } }
});
```

Sequelize validators

- Examples of custom validators:

```
const Foo = sequelize.define("Foo", {
     bar: {
          type: DataTypes.INTEGER,

          // CUSTOM VALIDATORS
          isEven(value) {
               if (parseInt(value) % 2 !== 0) {
                    throw new Error('Only even values are allowed!');
               }
          }
          isGreaterThanOtherField(value) {
               if (parseInt(value) <= parseInt(this.otherField))
                    throw new Error('Bar must be greater than otherField.');
          }
     }
});
```

Sequelize: model querying

- **INSERT queries**: method `create` of Sequelize class `Model`

```
// Create a new user
const jane = await User.create({ firstName: "Jane", lastName: "Doe" });
console.log("Jane's auto-generated ID:", jane.id);
```

`create()` is a shorthand for building an instance of the model object and saving it into the DB

```
User.create({ username: 'alice123', isAdmin: true },
            { fields: ['username'] })
     .then( data => {
          console.log(data.username); // 'alice123'
          console.log(data.isAdmin); // false
     });
```

It is also possible to define which attributes can be set in the create method: in the example the `isAdmin` attribute is set with the default value (false)

Sequelize: model querying

- **INSERT queries**: method create of Sequelize class `Model`

```
User.create( req.body ) // use request body data to create a new User instance
      .then( data => {
            res.status(201).json(data.id); // ID of the new instance is retrieved in data.id
      })
      .catch(err => {
             // if model has validations and data from request does not fulfill them
            if (err.name === 'SequelizeValidationError')
                 // loop over err.errors array and get their messages
                 res.status(400).json({ msg: err.errors.map(e => e.message) });
            else
                 res.status(500).json({ msg: "Some error occurred while creating User."});
     });
```

Example of creating an instance using the HTTP request body data

Sequelize: model querying

- **SELECT queries**: method findAll of Sequelize class Model

```
// Find all users
const users = await User.findAll();
console.log("All users:", JSON.stringify(users));
```

Read the whole table from the database:
SELECT * FROM users;

```
User.findAll({ attributes: ['username', 'age'] },
      .then( data => {
            //...
      });
```

Read only the **listed attributes**:
SELECT username, age FROM users;

```
User.findAll({ attributes: [['username','name'], 'age'] });
```

Attributes can be **renamed** using a nested array:
SELECT username AS name, age FROM users;

Sequelize: model querying

- **SELECT queries**: method findAll of Sequelize class Model

```
let users = User.findAll({ attributes: {exclude: ['age'] }});
```

Read all attributes, except those **listed in exclude**

```
// Count hats column
const users = await User.findAll( {
      attributes: ['foo',
                   [sequelize.fn('COUNT', sequelize.col('hats')), 'n_hats'],
                   'bar'
                  ]
      });
);
```

Use sequelize.fn to do **aggregations** (when using aggregation function, you must give it an alias to be able to access it from the model)
SELECT foo, COUNT(hats) AS n_hats, bar FROM …….

Sequelize: model querying

- **SELECT queries** with **WHERE** clauses: there are lots of operators to use for the where clause

```
// SELECT * FROM post WHERE authorId = 2
Post.findAll({
     where: {
          authorId: 2
     }
});
```

No operator (from Op) was explicitly passed, so **Sequelize assumed an equality comparison by default**.

The promise is resolved with an array of Model instances if the query succeeds

```
// SELECT * FROM post WHERE authorId = 2
AND status = 'active';
Post.findAll({
     where: {
          authorId: 12
          status: 'active'
     }
});
```

same as

```
const { Op } = require("sequelize");
Post.findAll({
     where: {
          [Op.and]: [
               { authorId: {[Op.eq]: 12} },
               { status: {[Op.eq]: 'active'}
          ]
     }
});
```

In multiple checks, if no operator (from Op) is explicitly passed, **Sequelize infers that the caller wanted an AND**

Check here for more examples!

Sequelize: model querying

- SELECT queries provided PRIMARY KEY: method findByPk

```
// SELECT * FROM project WHERE id = 123
const project = await Project.findByPk(123);
if (project === null) {
     console.log('Not found!');
} else {
     console.log(project instanceof Project); // true
}
```

The promise is resolved with one model instance if the query succeeds; otherwise returns null

- Method findOne: obtains the first entry it finds (that fulfills the optional query options, if provided)

```
const project = await Project.findOne({ where: { title: 'My Title' } });
if (project === null) {
     console.log('Not found!');
} else {
     console.log(project.title); // 'My Title'
}
```

Returns the first instance found, or null if none can be found

Sequelize: model querying

- Method findOrCreate: create an entry in the table unless it can find one fulfilling the query options
  - In both cases, it will return an instance (either the found instance or the created instance) and a boolean indicating whether that instance was created or already existed

```
const [user, created] = await User.findOrCreate({
  where: { username: 'sdepold' },
  defaults: { job: 'Technical Lead' }
});

console.log(user.username); // 'sdepold'

// The boolean indicating whether this instance was just created
if (created) {
  console.log(user.job); // This will certainly be 'Technical Lead'
}
```

The **where** option is considered for finding the entry, and the **defaults** option is used to define what must be created in case nothing was found.

If the defaults do not contain values for every column, Sequelize will take the values given in where

Sequelize: model querying

- Method findAndCountAll: combines findAll and count
  - useful when dealing with queries related to pagination where you want to retrieve data with a **limit** and **offset** but also need to know the total number of records that match the query

```
const { count, rows } = await Project.findAndCountAll({
     where: {
          title: { [Op.like]: 'foo%' }
  },
  offset: 10,
  limit: 2
});
// the total number records matching the query
console.log(count);
// the obtained records (array of objects): rows.length = 2
console.log(rows);
```

In the example, `result.rows` will contain rows 11 and 12, while `result.count` will return the total number of rows that matched the query

Sequelize: model querying

- UPDATE queries: method update

```
// Change everyone without a last name to "Doe"
await User.update({ lastName: "Doe" }, {
     where: {
          lastName: null
     }
});
```

The promise returns an array with the number of actual **affected rows** (if no entry is found on DB or no changes were made, the return value is [0])

```
// Use instance method update for single row updates
let user = User.find(…);
await user.update({ lastName: "Doe" });
```

In both **update** methods, only the specified fields are updated!

- DELETE queries: method destroy

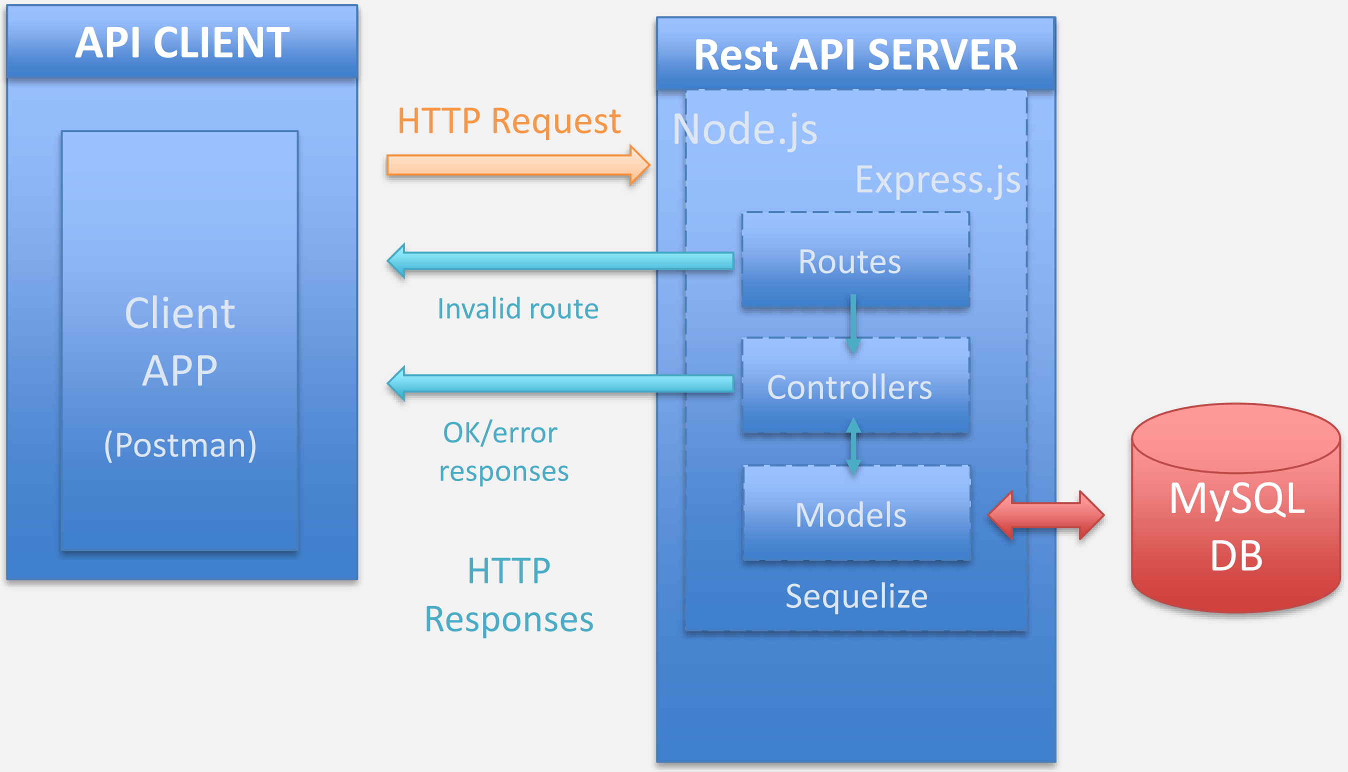
```
// Delete everyone named "Jane"
User.destroy({ where: {firstName: "Jane" } })
     .then (num => {
          if (num == 0)
               console.log("No Janes in DB");
     });
```

```
// Use instance method to delete a single row
let user = User.find(…);
await user.destroy();
```

The promise returns the number of destroyed rows

EXERCISE: using Sequelize in a REST API

- System architecture



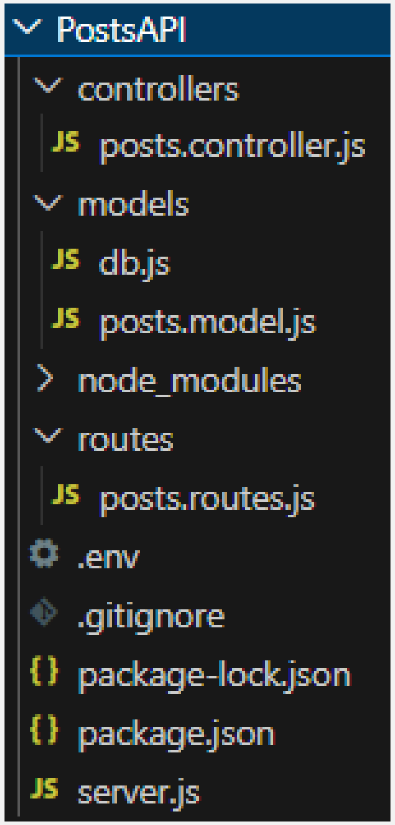
EXERCISE

- API routes

Verb	URI	Description
GET	posts	Gets the list of all posts and their details. **Optionally**, the posts can be filtered by title (partial string) and/or published status and ordered by the number of visualizations. Use **pagination**, to retrieve only one page per response
GET	posts/{id}	Gets details on a particular post. Return 404 error if post does not exist.
POST	posts	Creates a new post with the details provided in the request body. Response contains the URI for the newly created resource. Return 400 error if insuficiente body data.
PUT	posts/{id}	Modifies a particular post. Response contains the URI for the updated resource. Return 404 error if post does not exist. Return 400 error if insufficient body data.
DELETE	posts/{id}	Delete a particular post. Return 404 error if post does not exist.

EXERCISE

- Directory structure: let's structure the project in the following manner, so that the files are laid out logically in folders



```
PostsAPI
  controllers
    posts.controller.js
  models
    db.js
    posts.model.js
  node_modules
  routes
    posts.routes.js
  .env
  .gitignore
  package-lock.json
  package.json
  server.js
```

controllers folder: responsible for request data validation and for sending the API responses to clients

models folder: house the definition of the tutorials table in DB; file db.js is used to talk with the MySQL database

routes folder: define routes handlers

server.js file: sets up an Express web server

.env file: stores all the environment variables (API hostname, API port, DB credentials,…)

.gitignore file: list of files or folders to be ignored when saving your project into your local GIT repository (like node_modules folder or .env file)

EXERCISE: using Sequelize in a REST API

- Create a directory for the API Rest and build the package.json file with the necessary dependencies

  `dotenv` `express` `sequelize` `mysql2`

  loads **environment variables** from a `.env` file into `process.env`

- Set up environment variables on .env file
  - Like that, configuration variables and DB credentials are not hardcoded and would not be saved into your code repository

File *.env* under the project root folder

```
NODE_ENV=development
# Server configuration
PORT=3000
HOST='127.0.0.1'
# Database connection information
DB_HOST: 'yourDBhost'
DB_USER: 'yourDBuser'
DB_PASSWORD: 'yourDBpasswd'
DB_NAME: 'yourDBname'
```

EXERCISE

- Define the **post model**

File *posts.model.js* on *models* folder

```
module.exports = (sequelize, DataTypes) => {
    const Post = sequelize.define("Post", {
        title: {
            type: DataTypes.STRING,
            allowNull: false,
            validate: {len: { args: [5, 50], msg: "Title must have between 5 to 50 characters."}}
        },
        description: {type: DataTypes.STRING, allowNull: false},
        published: {
            type: DataTypes.BOOLEAN, defaultValue: false,
            validate: {
                isBoolean: function (val) { // custom validation function
                    if (typeof (val) != 'boolean')
                        throw new Error('Published must contain a boolean value!');
                }
            }
        },
        // TODO: complete model - views (number >=0, default 0), publishedAt (default NOW)
    }, {
         timestamps: false
    });
    return Post;
};
```

Defines the model name: **Post**

Defines the model attributes (and their data types and validations)

Disables the Sequelize default behavior of automatically adding fields createdAt and updatedAt to every model

EXERCISE

- Create a database using Sequelize

File *db.js* on *models* folder

```
const { Sequelize} = require('sequelize');

const sequelize = new Sequelize(process.env.DB_NAME, process.env.DB_USER, process.env.DB_PASSWORD, {
    host: process.env.DB_HOST, dialect: process.env.DB_DIALECT,
    pool: {
        max: 5, min: 0,
        acquire: 30000, idle: 10000
    }
});

// test the DB connection
(async () => {
    try {
        await sequelize.authenticate();
        console.log('Connection to the database has been established successfully.');
    } catch (err) {
        console.error('Unable to connect to the database:', err);
        process.exit(1); // exit the process with a failure code
    }
})();

...
```

EXERCISE

- Create a database using Sequelize

File *db.js* on *models* folder

```
...

const db = {}; //object to be exported
db.sequelize = sequelize; //save the Sequelize instance (actual connection pool)

//save the POST model (and add here any other models defined within the API)
db.Post = require("./posts.model.js")(sequelize, Sequelize.DataTypes);


// OPTIONAL: synchronize the DB with the sequelize model
(async () => {
    try {
        await db.sequelize.sync();
        console.log('DB is successfully synchronized')
    } catch (error) {
        console.log(error)
    }
})();

module.exports = db; //export the db object with the sequelize instance and Post model
```

EXERCISE

- Define the routes

File *posts.routes.js* on *routes* folder

```
const express = require('express');
let router = express.Router();
const postsController = require('../controllers/posts.controller');

// middleware for all routes related with posts
router.use((req, res, next) => {
    const start = Date.now();
    res.on("finish", () => { // finish event is emitted once the response is sent to the client
        const diffSeconds = (Date.now() - start) / 1000; // calculate #seconds to answer back to client
        console.log(`POST: ${req.method} ${req.originalUrl} completed in ${diffSeconds} seconds`);
    });
    next()
})

router.route('/')
    .post(postsController.addPost);


//…  TODO: add the other routes


// EXPORT ROUTES (required by APP)
module.exports = router;
```

EXERCISE

- Implement the controller functions
  - ➢ Example of the `create` function, for creating a new post

File *posts.controller.js* on *controllers* folder

```
const db = require("../models/index.js");
const Post = db.Post;

const { ValidationError } = require('sequelize'); //necessary for model validations using sequelize

exports.addPost = (req, res, next) => {
    Post.create(req.body) // Save POST in the DB (IF request body data is validated by Sequelize)
        .then(data => {
            res.status(201).json({ msg:"Post successfully created.",
                links: [  //add HATEOAS links to the created post
                           { rel: "self", href: `/posts/${post.id}`, method: "GET" },
                           { rel: "delete", href: `/posts/${post.id}`, method: "DELETE" },
                           { rel: "modify", href: `/posts/${post.id}`, method: "PUT" },
                        ]
           });
        })
        .catch(err => {
            if (err instanceof ValidationError) // Handle validation errors from Sequelize
                res.status(400).json({ error: err.errors.map(e => e.message) });
            next(err); // Handle other errors  (pass them to the error handler middleware)
    });
};
```

Using *then… catch*

EXERCISE

- Implement the controller functions
  - ➢ Example of the `create` function, for creating a new post

File *posts.controller.js* on *controllers* folder

```
const db = require("../models/index.js");
const Post = db.Post;

const { ValidationError } = require('sequelize'); //necessary for model validations using sequelize

exports.create = async (req, res, next) => {
    try {
        Post.create(req.body) // Save POST in the DB (IF request body data is validated by Sequelize)
        res.status(201).json({ msg:"Post successfully created.",
                links: [  //add HATEOAS links to the created post
                           { rel: "self", href: `/posts/${post.id}`, method: "GET" },
                           { rel: "delete", href: `/posts/${post.id}`, method: "DELETE" },
                           { rel: "modify", href: `/posts/${post.id}`, method: "PUT" },
                         ]
           });
    } catch (err) {
        if (err instanceof ValidationError)
            res.status(400).json({ error: err.errors.map(e => e.message) });
        next(err); // Handle other errors  (pass them to the error handler middleware)
   }};
};
```

Using *async … await*

EXERCISE

File *server.js* in the root project folder

```
require('dotenv').config();          // read environment variables from .env file
const express = require('express');

const app = express();
const port = process.env.PORT ;  // use environment variables
const host = process.env.HOST ;

app.use(express.json()); //enable parsing JSON body data

// routing middleware for resource POSTS
app.use('/posts', require('./routes/posts.routes.js'))

// handle invalid routes
app.use((req, res, next) => {
      res.status(404).json({message:`The API does not recognize the request: ${req.method}
${req.originalUrl}`});
})

// error middleware (always at the end of the file)
app.use((err, req, res, next) => {
    // error thrown by express.json() middleware when the request body is not valid JSON
    if (err.type === 'entity.parse.failed')
        return res.status(400).json({ message: 'Invalid JSON payload! Check if your body data is a valid
JSON.'});
    res.status(err.statusCode || 500).json({ message: err.message || 'Internal Server Error' });
});

app.listen(port, host, () => console.log(`App listening at http://${host}:${port}/`));
```

EXERCISE

- Complete the REST API, with the routes to:
  - Get all posts (with pagination, filtering and ordering)
  - Read just one post, providing its ID
  - Update a post, providing its ID
  - Delete a post, providing its ID

Relationships in Sequelize

- Sequelize supports the standard associations: 1:1, 1:N and N:M
- Creating associations in Sequelize is done by calling one of the above functions on a model (the `source - A`), and providing another model as the first argument to the function (the `target - B`):
  - ➢ A.hasOne(B): 1:1 relationship between A and B models; adds a foreign key to the B
  - ➢ A.belongsTo(B): 1:1 relationship; add a foreign key to the A
  - ➢ A.hasMany(B) : 1:N relationship; adds a foreign key to target (B)
  - ➢ A.belongsToMany(B, { through: 'C' }): N:M association between A and B, through the junction table C

Relationships in Sequelize

- They all accept an options object as a second parameter
  - optional for the first three, mandatory for belongsToMany containing at least the through property

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

A.hasOne(B, { /* options */ });
A.belongsTo(B, { /* options */ });
A.hasMany(B, { /* options */ });
A.belongsToMany(B, { through: 'C', /* options */ });
```

- Sequelize **automatically adds foreign keys** to the appropriate models (unless they are already present)
- For the N:M association, the junction table C is also created (unless it already exists) with the appropriate foreign keys on it
- Foreign key **constraints** are also created: all associations use CASCADE on update and SET NULL on delete, except for N:M, which also uses CASCADE on delete

Relationships in Sequelize

- Usually, the Sequelize associations are defined in pairs:
  - To create a 1:1 relationship, the hasOne and belongsTo associations are used together;
  - To create a 1:N relationship, the hasMany and belongsTo associations are used together;
  - To create a N:M relationship, two belongsToMany calls are used together
- The advantages of using these pairs instead of one single association will be discussed latter

1:1 relationships

- Create a 1:1 relationship
  - In a relational database, this will be done by establishing a foreign key in one of the tables
  - Which one? Sequelize will infer what to do from the source and target models

Setup a 1:1 relationship between models Foo and Bar

```
// since no option was passed, Sequelize adds a fooId FK column into Bar model
Foo.hasOne(Bar);
Bar.belongsTo(Foo);
```

Calling `Bar.sync()` after the above will yield the following SQL:

```
CREATE TABLE IF NOT EXISTS "foos" ( /* ... */);
CREATE TABLE IF NOT EXISTS "bars" (
        /* ... */
        "fooId" INTEGER REFERENCES "foos" ("id") ON DELETE SET NULL ON UPDATE CASCADE
);
```

Relationships in Sequelize: options

- Various options can be passed as a second parameter of the association call:
  - to configure the ON DELETE and ON UPDATE behaviors

Setup a 1:1 relationship between models Foo and Bar, such that Bar gets a `fooId` column (FK)

```
Foo.hasOne(Bar, {
  onDelete: 'SET DEFAULT', // default would be 'SET NULL'
  onUpdate: 'NO ACTION'    // default would be 'CASCADE'
}););
Bar.belongsTo(Foo);
```

Calling `Bar.sync()` after the above will yield the following SQL:

```
CREATE TABLE IF NOT EXISTS "foos" ( /* ... */);
CREATE TABLE IF NOT EXISTS "bars" (
    /* ... */
    "fooId" INTEGER REFERENCES "foos" ("id") ON DELETE SET DEFAULT ON UPDATE NO ACTION
);
```

Relationships in Sequelize: options

- Various options can be passed as a second parameter of the association call:
    - to customize the foreign key name

Setup a 1:1 relationship between models Foo and Bar, with 4 options to set FK name as myFooId

```
// Option 1: Bar gets a myFooId column (FK)
Foo.hasOne(Bar, {
  foreignKey: 'myFooId'
});
Bar.belongsTo(Foo);

// Option 2 (longer)
Foo.hasOne(Bar, {
  foreignKey: {
    name: 'myFooId'
  }
});
Bar.belongsTo(Foo);
```

```
// Option 3: FK set up in belongTo
Foo.hasOne(Bar);
Bar.belongsTo(Foo, {
  foreignKey: 'myFooId'
});

// Option 4 (longer)
Foo.hasOne(Bar);
Bar.belongsTo(Foo, {
  foreignKey: {
    name: 'myFooId'
  }
});
```

Relationships in Sequelize: options

- Various options can be passed as a second parameter of the association call:
  - ➢ to alter the default optional association to a mandatory one

Setup a 1:1 relationship between Foo and Bar models, such the fooId column (FK) is not allowed to be null, meaning that one Bar cannot exist without a Foo

```
Foo.hasOne(Bar, {  // a fooId column (FK) must be added to Bar
  foreignKey: {
    allowNull: false // means that one Bar cannot exist without a Foo
  }
});
```

Calling sync() after the above will yield the following SQL:

```
    CREATE TABLE IF NOT EXISTS "foos" ( /* ... */);
    CREATE TABLE IF NOT EXISTS "bars" (
        /* ... */
        "fooId" INTEGER NOT NULL REFERENCES "bars" ("id") ON DELETE RESTRICT ON UPDATE RESTRICT
    );
```

1:M relationships

- Create a 1:M relationship
  - In a relational database, this will be done by establishing a foreign key in the M side

Setup a 1:M relationship between Team and Player models

```
// Sequelize knows that a teamId FK column is added to Player
Team.hasMany(Player);
Player.belongsTo(Team);
```

Calling `sync()` after the above will yield the following SQL:

```
CREATE TABLE IF NOT EXISTS "teams" ( /* ... */);
CREATE TABLE IF NOT EXISTS "players" (
    /* ... */
    "teamId" INTEGER REFERENCES "team" ("id") ON DELETE SET NULL ON UPDATE CASCADE
);
```

N:M relationships in Sequelize

- Many-To-Many associations cannot be represented by just adding one foreign key to one of the tables, as the other relationships did
  - ➢ Instead, an **extra model is needed** (and extra table in the database) which will have two foreign key columns and will keep track of the associations - the junction table is also sometimes called join table or through table

```
const Movie = sequelize.define('Movie', { name: DataTypes.STRING });
const Actor = sequelize.define('Actor', { name: DataTypes.STRING });
Movie.belongsToMany(Actor, { through: 'ActorMovies' });
Actor.belongsToMany(Movie, { through: 'ActorMovies' });
```

In N:M relationships, the string given in the **through** option of the **belongsToMany** call, will automatically create the **ActorMovies** model which will act as the junction model

```
CREATE TABLE IF NOT EXISTS "ActorMovies" (
        "createdAt" TIMESTAMP WITH TIME ZONE NOT NULL,
        "updatedAt" TIMESTAMP WITH TIME ZONE NOT NULL,
        "MovieId" INTEGER REFERENCES "Movies" ("id") ON DELETE CASCADE ON UPDATE CASCADE,
        "ActorId" INTEGER REFERENCES "Actors" ("id") ON DELETE CASCADE ON UPDATE CASCADE,
        PRIMARY KEY ("MovieId","ActorId")
);
```

N:M relationships in Sequelize

- The **junction model** can also be defined by **passing a model directly**
  - ➢ no extra model will be created automatically

```
const Movie = sequelize.define('Movie', { name: DataTypes.STRING });
const Actor = sequelize.define('Actor', { name: DataTypes.STRING });
const ActorMovies = sequelize.define('ActorMovies', {
  MovieId: {
    type: DataTypes.INTEGER,
    references: {
      model: Movie, key: 'id'
    }
  },
  ActorId: {
    type: DataTypes.INTEGER,
    references: {
      model: Actor, key: 'id'
    }
  }
});
Movie.belongsToMany(Actor, { through: 'ActorMovies' });
Actor.belongsToMany(Movie, { through: 'ActorMovies' });
```

Relationships with aliases

- When creating associations, it is possible to provide an alias, using the as option
  - ➢ This is useful if the same model is associated twice, or you want your association to be called something other than the name of the target model

Example: consider the case where users have many pictures, one of which is their profile picture. All pictures have a `userId`, but in addition the user model also has a `profilePictureId`, to be able to easily load the user's profile picture

```
User.hasMany(Picture) // all pictures now have a userId attribute, as foreign key
User.belongsTo(Picture, { as: 'ProfilePicture', constraints: false }) // all users
now have a ProfilePicture attribute, as foreign key

// MIXINS: special methods to interact between models of an association
user.getPictures() // gets all user's pictures
user.getProfilePicture() // gets only the user's profile picture

User.findAll({
  where: ...,
  include: [
    { model: Picture }, // load all user's pictures
    { model: Picture, as: 'ProfilePicture' }, // load the user's profile picture
  ]
})
```

constraints: to sync the models correctly and avoid circular dependencies between User and Picture

Sequelize: queries in associations

- For creating, updating and deleting, you can either:
  - ➢ Use the standard model queries directly

```
Foo.hasOne(Bar);
Bar.belongsTo(Foo); // Bar as FK fooID

// This creates a Bar belonging to the Foo of ID 5
Bar.create({
  name: 'My Bar',
  fooId: 5
});
```

Sequelize: queries in associations

- For creating, updating and deleting, you can either:
  - ➢ Use **mixins functions:** when an association is defined between two models, their instances gain special methods to interact with their associated counterparts
  - ➢ For example, if two models, Foo and Bar, are associated, their instances will have the following mixins available, depending on the association type:

Association	Mixins

Foo.hasOne(Bar)
Foo.belongsTo(Bar)
	fooInstance.getBar()
	fooInstance.setBar()
	fooInstance.createBar()

Foo.hasMany(Bar)
Foo.belongsToMany(Bar, { through: Baz })
	fooInstance.getBars()
	fooInstance.countBars()
	fooInstance.hasBar()
	fooInstance.hasBars()
	fooInstance.setBars()
	fooInstance.addBar()
	fooInstance.addBars()
	fooInstance.removeBar()
	fooInstance.removeBars()
	fooInstance.createBar()

Sequelize: queries in associations

- For creating, updating and deleting, you can either:
  - ➢ Use **mixins functions**

```
// for now, only Foo knows about Bar, so mixins are created for the Foo model
Foo.hasOne(Bar);
// now, also Bar knows about Foo, so also for this model mixins are created
Bar.belongsTo(Foo);

// mixins for hasOne & belongsTo associations
const foo = await Foo.create({ name: 'the-foo' });
const bar1 = await Bar.create({ name: 'some-bar' });
     console.log(await foo.getBar()); // null

await foo.setBar(bar1);
     console.log((await foo.getBar()).name); // 'some-bar’

await foo.createBar({ name: 'yet-another-bar' });
const newlyAssociatedBar = await foo.getBar();
     console.log(newlyAssociatedBar.name); // 'yet-another-bar'

await foo.setBar(null); // Un-associate
      console.log(await foo.getBar()); // null
```

Mixins are formed by a prefix (e.g. get, add, set) concatenated with the model name

Sequelize: queries in associations

- Consider a Ships and Captains models, with a 1:1 relationship between them:

```
const Ship = sequelize.define('ship', {
  name: DataTypes.TEXT,
  crewCapacity: DataTypes.INTEGER,
  amountOfSails: DataTypes.INTEGER
}, { timestamps: false });

const Captain = sequelize.define('captain', {
  name: DataTypes.TEXT,
  skillLevel: { type: DataTypes.INTEGER, validate: { min: 1, max: 10 } }
}, { timestamps: false });

// the associations below create the `captainId` foreign key in Ship, while allowing
for null values, meaning that a Ship can exist without a Captain and vice-versa
Captain.hasOne(Ship);
Ship.belongsTo(Captain);
```

Sequelize: queries in associations

- **Lazy Loading**: technique of fetching the associated data only when you really want it
  - ➢ save time and memory by only fetching it when necessary

getShip() is a **mixin function**: when an association is defined between 2 models, their instances gain special methods to interact with their associated counterparts

```
const awesomeCaptain = await Captain.findOne({
  where: { name: "Jack Sparrow" }
});

// Do stuff with the fetched captain (from the previous query)
console.log('Name:', awesomeCaptain.name);
console.log('Skill Level:', awesomeCaptain.skillLevel);

// LAZY LOADING (2nd query): now we want information about his ship!
const hisShip = await awesomeCaptain.getShip();

// Do stuff with the ship
console.log('Ship Name:', hisShip.name);
console.log('Amount of Sails:', hisShip.amountOfSails);
```

Sequelize: queries in associations

- **Eager Loading**: brings the associated data with only one query
  - ➢ when Sequelize fetches associated models, they are added to the output object as model instances
  - ➢ in Sequelize, eager loading is done by using the **`include`** option on a model finder query

```
const awesomeCaptain = await Captain.findOne({
  where: { name: "Jack Sparrow" },
  include: Ship // EAGER LOADING, by providing the model object
 (creates a left outer join in the query)
});

// Now the ship comes with it
console.log('Name:', awesomeCaptain.name);
console.log('Skill Level:', awesomeCaptain.skillLevel);
console.log('Ship Name:', awesomeCaptain.ship.name);
console.log('Amount of Sails:', awesomeCaptain.ship.amountOfSails);

// EAGER LOADING (providing the model name)
const ship = await Ship.findAll({ include: 'captain' })
```

OUTPUT example:

```
{
   "id": 1,
   "name": "Jack Sparrow",
   "skillLevel": "10",
   "shipId": 1,
   "ship": {
          "id": 1
          "name": "John Doe",
          "crewCapacity": "100"
          "amountOfSails": "50"
    }
}
```

Sequelize: queries in associations

- Eager Loading: if an association is aliased (using the as option) the alias must be specified when including the model
    - ➢ Consider this next example, where 1:N associations are defined between User and Task and between User and Tool models

```
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);
User.hasMany(Tool, { as: 'Instruments' }); //aliased association


const users = await User.findAll({ include: Task }); // #1
const users = await User.findAll({                   // #2
  include: { model: Tool, as: 'Instruments' }
});
//OR
const users = await User.findAll({                   // #2
  include: 'Instruments'
});
```

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

OUTPUT #2:
```
[{
 "name": "John Doe",
 "id": 1,
 "Instruments": [{
  "name": "Scissor",
  "id": 1,
  "userId": 1
 }]
}]
```

Sequelize: queries in associations

- Eager Loading: one can force the query to return only records which have an associated model, effectively converting the query from the default OUTER JOIN to an INNER JOIN
  - This is done with the `required: true` option

```
const users = await User.findAll({
  include: { model: Tool, as: 'Instruments', required: true }
});
```

Generated SQL:
SELECT * FROM \`users\` AS \`user\`
INNER JOIN \`tools\` AS \`Instruments\` ON \`user\`.\`id\` = \`Instruments\`.\`userId\`

Sequelize: queries in associations

- Eager Loading: one can also filter the associated model using the `where` option
  - when the where option is used inside an include, Sequelize **automatically sets the `required` option to true**. So, instead of the default OUTER JOIN, an INNER JOIN is done, returning only the parent models with at least one matching children

```
const users = await User.findAll({
  include: { model: Tool, as: 'Instruments',
      where: {
         size: {
           [Op.ne]: 'small'
         }
       } }
});
```

Generated SQL:
SELECT * FROM `users` AS `user`
INNER JOIN `tools` AS `Instruments` ON `user`.`id` = `Instruments`.`userId`
AND `Instruments`.`size` != 'small';

Sequelize: queries in associations

- Eager Loading: to obtain top-level WHERE clauses that involve nested columns, the '`$nested.column$`' syntax of Sequelize provides a way to reference nested columns
  - In SQL, the default OUTER JOIN is used

```
User.findAll({
  where: {
    '$Instruments.size$': { [Op.ne]: 'small' }
  },
  include: [{
    model: Tool,
    as: 'Instruments'
  }]
});
```

Generated SQL:
SELECT * FROM `users` AS `user`
LEFT OUTER JOIN `tools` AS `Instruments` ON `user`.`id` = `Instruments`.`userId`
WHERE `Instruments`.`size` != 'small';

Sequelize: queries in associations

- Multiple eager loading: the `include` option can receive an array in order to fetch multiple associated models at once

```
Foo.findAll({
  include: [
    {
      model: Bar,
      required: true
    },
    {
      model: Baz,
      where: /* ... */
    },
    /* ... */
  ]
})
```

Sequelize: queries in associations

- Eager loading N:M associations: when performing eager loading on a model with a Belongs-to-Many relationship, Sequelize fetchs the junction table data as well, by default

```
const Foo = sequelize.define('Foo', { name: DataTypes.TEXT });
const Bar = sequelize.define('Bar', { name: DataTypes.TEXT });
Foo.belongsToMany(Bar, { through: 'Foo_Bar' });
Bar.belongsToMany(Foo, { through: 'Foo_Bar' });

await sequelize.sync();
const foo = await Foo.create({ name: 'foo' });
const bar = await Bar.create({ name: 'bar' });
await foo.addBar(bar); // mixins
const fetchedFoo = await Foo.findOne({ include: Bar });
console.log(JSON.stringify(fetchedFoo, null, 2));
```

```
OUTPUT:
{
  "id": 1,
  "name": "foo",
  "Bars": [
    {
      "id": 1,
      "name": "bar",
      "Foo_Bar": {
        "FooId": 1,
        "BarId": 1
      }
    }
  ]
}
```

Sequelize: queries in associations

- Eager loading N:M associations: to remove the extra data from the junction table, one can explicitly provide an empty array to the `attributes` option inside the `through` option of the `include` option

```
const Foo = sequelize.define('Foo', { name: DataTypes.TEXT });
const Bar = sequelize.define('Bar', { name: DataTypes.TEXT });
Foo.belongsToMany(Bar, { through: 'Foo_Bar' });
Bar.belongsToMany(Foo, { through: 'Foo_Bar' });


await sequelize.sync();
const foo = await Foo.create({ name: 'foo' });
const bar = await Bar.create({ name: 'bar' });
await foo.addBar(bar); // mixins
const fetchedFoo = await Foo.findOne({
  include: {
    model: Bar,
    through: { attributes: [] }
  }
});
console.log(JSON.stringify(fetchedFoo, null, 2));
```

OUTPUT:
```
{
  "id": 1,
  "name": "foo",
  "Bars": [
    {
      "id": 1,
      "name": "bar",
    }
  ]
}
```

READ MORE ABOUT EAGER LOADING:
https://sequelize.org/docs/v6/advanced-association-concepts/eager-loading/

Sequelize: associations to fields not PK

- Sequelize allows to define an association that uses another field, instead of the primary key field, to establish the association
  - ➢ This other field must have a unique constraint on it (otherwise, it wouldn't make sense)
  - ➢ In the association options define the **target/source** key (depending on the relation type) and the foreign key name

```
const Ship = sequelize.define('ship', { name: DataTypes.TEXT });
const Captain = sequelize.define('captain', { name: { type: DataTypes.TEXT, unique: true }});

// This creates a foreign key called `captainName` in the source model (Ship)
// which references the `name` field from the target model (Captain)
Ship.belongsTo(Captain, { targetKey: 'name', foreignKey: 'captainName' });

await Captain.create({ name: "Jack Sparrow" });
const ship = await Ship.create({ name: "Black Pearl", captainName: "Jack Sparrow" });
console.log((await ship.getCaptain()).name); // "Jack Sparrow"
```

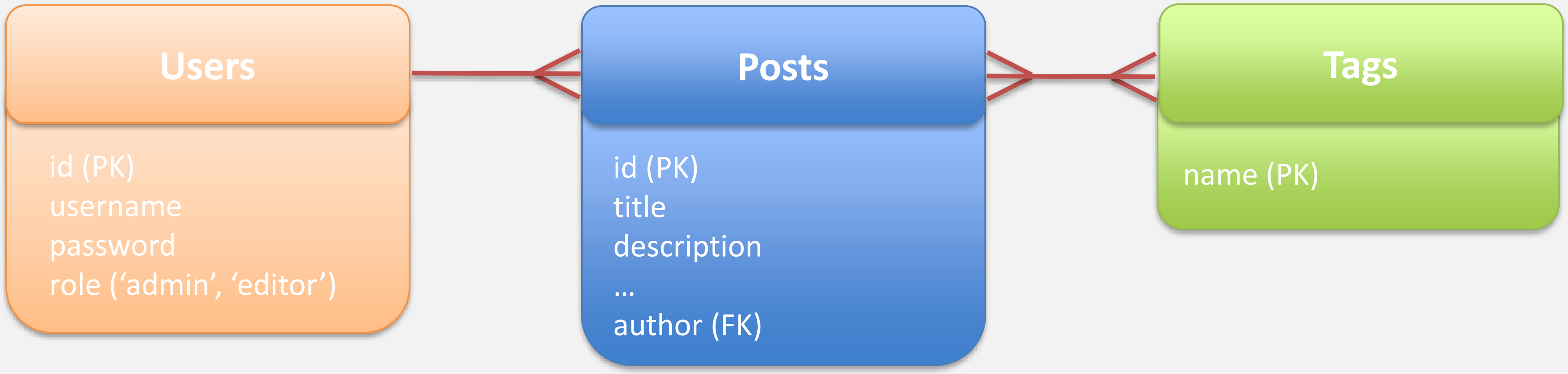
Sequelize: associations to fields not PK

- Sequelize allows to define an association that uses another field, instead of the primary key field, to establish the association
  - ➢ This other field must have a unique constraint on it (otherwise, it wouldn't make sense)
  - ➢ In the association options define the **target/source** key (depending on the relation type) and the foreign key name

```
const Foo = sequelize.define('foo', { name:  { type: DataTypes.TEXT, unique: true }});
const Bar = sequelize.define('bar', { title: { type: DataTypes.TEXT, unique: true }});

// This creates a junction table `foo_bar` with fields `fooName` and `barTitle`
Foo.belongsToMany(Bar, { through: 'foo_bar', sourceKey: 'name', targetKey: 'title' });
```

Relationships in Sequelize

- Let's continue our Posts REST API and present how to implement 1-N and N-N relationships and include some more routes to manipulate the new models
  - Include a **Users** model, where a Post is created by 1 user, but one user can create many posts: **one-to-many** relationship
  - Assume also that one Post has many **Tags**, and one Tag can point to many Posts: **many-to-many** relationship



Relationships in Sequelize

Start by adding the new models in the `models` folder

File *User.model.js* on *models* folder

```
module.exports = (sequelize, DataTypes) => {
    const User = sequelize.define("User", {
        username: {
            type: DataTypes.STRING, allowNull: false
        },
         password: {
            type: DataTypes.STRING, allowNull: false
        },
         role: {
            type: DataTypes.ENUM('admin', 'editor’), allowNull: false,
            validate: {
                isIn: {
                     args: [['admin', 'editor']],
                     msg: "Role must be one of the following: admin or editor"
                }
         }
    }, {
        timestamps: false
    });
    return User;
};
```

Relationships in Sequelize

Start by adding the new models in the `models` folder

File *tags.model.js* on *models* folder

```
module.exports = (sequelize, DataTypes) => {
    const Tag = sequelize.define("tag", {
        name: {
            type: DataTypes.STRING,
            primaryKey: true
        }
    }, {
        timestamps: false
    });
    return Tag;
};
```

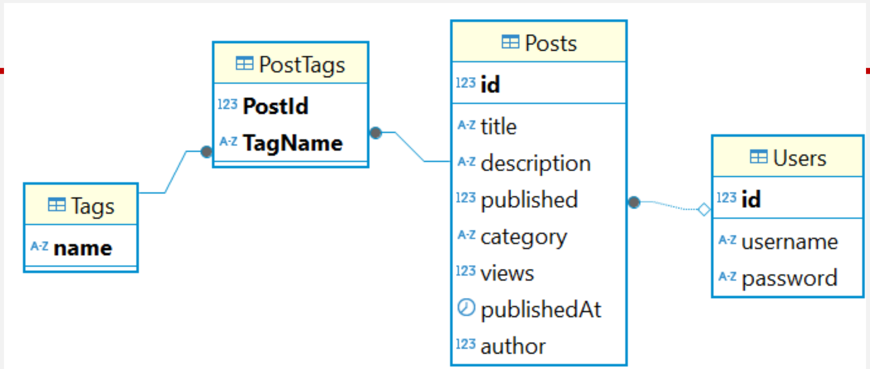
Relationships in Sequelize

Define the relationships between models

– synchronize the database if necessary

File *db.js* on *models* folder

```
…
//export the new models: COMMENT and TAG
db.User = require("./users.model.js")(sequelize, Sequelize.DataTypes);
db.Tag = require("./tags.model.js")(sequelize, Sequelize. DataTypes);

//define the relationship 1:N between POST and USER models: define foreign key name and
// if a user is deleted, delete all posts associated to him/her
db.User.hasMany(db.Post, {foreignKey: 'author', onDelete: 'CASCADE', allowNull: false});
db.Post.belongsTo(db.User, { foreignKey: 'author', onDelete: 'CASCADE', allowNull: false});

//define the relationship N:M between TUTORIAL and TAG models
db.Post.belongsToMany(db.Tag, { through: 'PostTags', timestamps: false });
db.Tag.belongsToMany(db.Post, { through: 'PostTags', timestamps: false });

…
```



Relationships in Sequelize

Add the following new routes to the API:

Verb	URI	Description
POST	/tags	Creates a new tag (validate that the new tag is unique)
GET	/tags	List all tags
PUT	/posts/:id/tags/:idT	Adds a tag to a post (check if post and tag exists and that post does already has that tag)
DELETE	/posts/:idP/tags/:idT	Deletes a given tag from a given post (check if post and tag exists and that post actually has that tag)
GET	/users/{id}/posts	Get all posts created by a given user (include the user data in the response (without sensitive data or repetitions)

Also, alter some of the 'old' routes to the API:

Verb	URI	Description
GET	/posts/:id	Include the author and tags names
POST	/posts	Include the author id when creating a new post