Objection.js and Knex.js Guidebook

Lars Albert

0.0 Introduction

NodeJS is often associated with the popular NoSQL database MongoDB which is, by default, the most JavaScript friendly database. It uses a JSON similar data storage type (called BSON). NoSQL databases have the advantage that you don't have to "waste" too much time, defining data structures and possible relationships. Additionally, they are great getting your project quickly production up and running and don't enforce field constraints.

When starting a new NodeJS project, however, you shouldn't just accept MongoDB as the default choice. Instead, the type of database you choose should depend on your project's requirements. Sometimes you want to use an ACID compliant database. **ACID**, in short, means that a full database task either succeeds or fails. There are no missing values because some parts of a transaction threw an error. These are some crucial advantages needed by banks and eCommerce websites. The most popular ACID compliant databases are SQL databases.

Why Knex.js and Objection.js

Node.js doesn't provide a simple way out of the box to connect to databases due to its general-purpose nature. Of course, there are packages like **pg** (for PostgreSQL - SQL Database) or **mongoose** (for MongoDB - NoSQL Database) to interact directly to your database, but if your application gets big enough, managing connection and queries through these packages can become a pain quickly.

Sequelize is another popular package that creates a connection between NodeJS and a SQL database; however, the code of it is so much abstracted that it doesn't feel like SQL anymore. Additionally, some people online complain that Sequelize can become an annoyance, and the package is not well documented.

Knex.js and **Objection.js** combined are an alternative to many **SQL database** connection and ORM packages (Knex.js and Objection.js are only used with SQL databases). It probably has a steeper learning curve, but many people online swear that after they migrated from Sequelize to Knex.js and Objection.js, they never looked back. Knex.js is the package used to execute database queries. Objection.js is the package used to create a unified representation of a database table with many features, without actually defining the database schema.

As you've already noticed, the choice of using Knex.js and Objection.js for SQL databases here is made based on third party opinions. Still, those opinions should be highlighted to give some hints about the advantages these packages could provide.

Choice of SQL database

Knex.js, the tool which manages the database (and not Objection.js), supports PostgreSQL, MSSQL, MySQL, MariaDB, SQLite3, Oracle, and Amazon Redshift.

After I worked some time Knex.js and Objection.js, I would personally recommend using these tools only in combination with PostgreSQL databases. Only PostgreSQL enables all features of Knex.js and Objection.js where other SQL databases have problems.

For example, when inserting multiple rows, only PostgreSQL can return all of the inserted rows in the same query. Other SQL databases only returns the first inserted row.

This whole guidebook is based on MySQL simplicity reasons.

Requirements

For you to continue, you should have a general knowledge of REST APIs, NodeJS, Express, and MySQL.

What you need

Make sure that you have the following tools installed on your computer:

- **Node.js with NPM**
- A MySQL database, I recommend using **XAMPPs MySQL Database** (Install the whole program, but we only need the MySQL database of it)
- **Postman to test the API**

The project

We are going to build a todo system with users and todo's. You should be able to:

- request all users from the REST API
- request all todos from the REST API
- request specific todos and the related user from the REST API I.e., we have two tables with relationships between them.
- send a POST request with user input to create a new user

0.1 Project Setup

We first create a basic REST API using `Express`, before we start using the database, Knex.js, and Objection.js.

Setup

1. Create a project directory

Create a project directory that you will work with

2. Initialize the project

Run in your terminal

```
npm init -y
```

3. Install express

Express is a minimal, open-source, and flexible Node.js web app framework designed to make developing websites, web apps, & API's much easier.

Run in your terminal

```
npm install express --save
```

4. Create the NodeJS server file

Create a file named `server.js`.

> **Note:** You can always run the app writing in your terminal `node server.js`, which starts the server. Alternatively, you can also use the **nodemon (Install guide here)** package to auto re-start the server on file change using the `nodemon server.js` command.

0.2 API Endpoints Setup

Express is the framework we use to create the API endpoints (called routes) and handle requests & responses.

There are two ways to create routes in express using the:

- route methods -> `app.get(...)`
- router class -> `router.get(...)`

The difference between them is that the second option is used to make the routes more maintainable in more significant projects.

Instead of having all routes inside the main NodeJS file, you can categorize and separate them into their own **mini-app**.

Using the router class, you define in your main NodeJS file some *base endpoints*, which then is extended in more route files. E.g.

- *www.restaurant.com/food* -> everything food-related will be handled by the `/routes/food.js` file
- *www.restaurant.com/order* -> everything order-related will be handled by the `/routes/order.js` file
- *www.restaurant.com/reservation* -> everything table reservation-related will be handled by the `/routes/reservation.js` file

> **Note:** If you know the Python framework called *Django*, then you can think of it as splitting up a project in multiple apps. Each route file, in this case, is an app and has its very own functionality.

Let's have a look at both approaches with hardcoded data.

Example using route methods

/server.js

```
// Route Methods
const express = require("express");
const app = express();

// Create endpoint of all users
app.get("/api/users/", (req, res) => {
  res.json([
    {
      id: 1,
      name: "Marc",
      age: 19
    },
    {
      id: 2,
      name: "Ben",
      age: 31
    },
    {
      id: 3,
      name: "Jessica",
      age: 27
    }
  ]);
});

// Create endpoint of all todos
// Done status: 0 - false ; 1 - true
app.get("/api/todos/", (req, res) => {
  res.json([
    {
      id: 1,
      user_id: 1, // -> Marc
      todo: "Buy Milk",
      done: 0
    },
    {
      id: 2,
      user_id: 1, // -> Marc
      todo: "Walk the dog",
      done: 0
    },
    {
      id: 3,
      user_id: 3, // -> Jessica
      todo: "Call grandma",
      done: 0
    }
  ]);
});

// Start the server
const server = app.listen(8080, error => {
  if (error) {
    console.log("Error running Express");
  }
  console.log("Server is running on port", server.address().port);
});
```

This is the fastest and easy way to create routes.

> **To return or not return** `app.get("/", (req,res) => { ... })` does **not** expect a value to be returned. There's no

> difference with using `return` or not, except using it will add additional overhead. Only use `return` if you want to stop the functions execution (e.g. nested if else statements.)

Router class

Considering that we, later on, create large and complex web applications, we use the router class.

For this, we have to create new directories and files. Create the:

- **/routes/** directory. This directory contains all route files.
- **/routes/api.js** file. This file handles all route files related to the `example.com/api/` endpoint.
- **/routes/api/** directory. This directory contains all API route related files.
- **/routes/api/users.js** file. This file handles all `api/users/` endpoints, which could be getting all users or updating a single one.
- **/routes/api/todos.js** file. This file handles all `api/todos/` endpoints, which could be getting all todos or updating a single one.

After that, we have to update the *server.js* file and remove the logic from it. The only purpose of this file should be to include the route handlers. **server.js**

```
// Route Class
const express = require("express");
const app = express();

// Require router files
const apiRoutes = require("./routes/api");

// Include the routes to express
app.use("/api", apiRoutes);

// Start the server
const server = app.listen(8080, error => {
  if (error) {
    console.log("Error running Express");
  }
  console.log("Server is running on port", server.address().port);
});
```

Next, our */routes/api.js* file should, once again, only combine and include the logic of each endpoint, which separates the */routes/api/users.js* and */routes/api/todos.js* files.

/routes/api.js

```
const express = require("express");
const app = express();

// Require router files
const usersRoutes = require("./api/users");
const todosRoutes = require("./api/todos");

// Include the routes to express
app.use("/users", usersRoutes);
app.use("/todos", todosRoutes);

// Export the file to be used in server.js
module.exports = app;
```

/routes/api/users.js

```
const express = require("express");
const router = express.Router();

// Create endpoint of all users
router.get("/", (req, res) => {
  res.json([
    {
      id: 1,
      name: "Marc",
      age: 19
    },
    {
      id: 2,
      name: "Ben",
      age: 31
    },
    {
      id: 3,
      name: "Jessica",
      age: 27
    }
  ]);
});

// Export to api.js
module.exports = router;
```

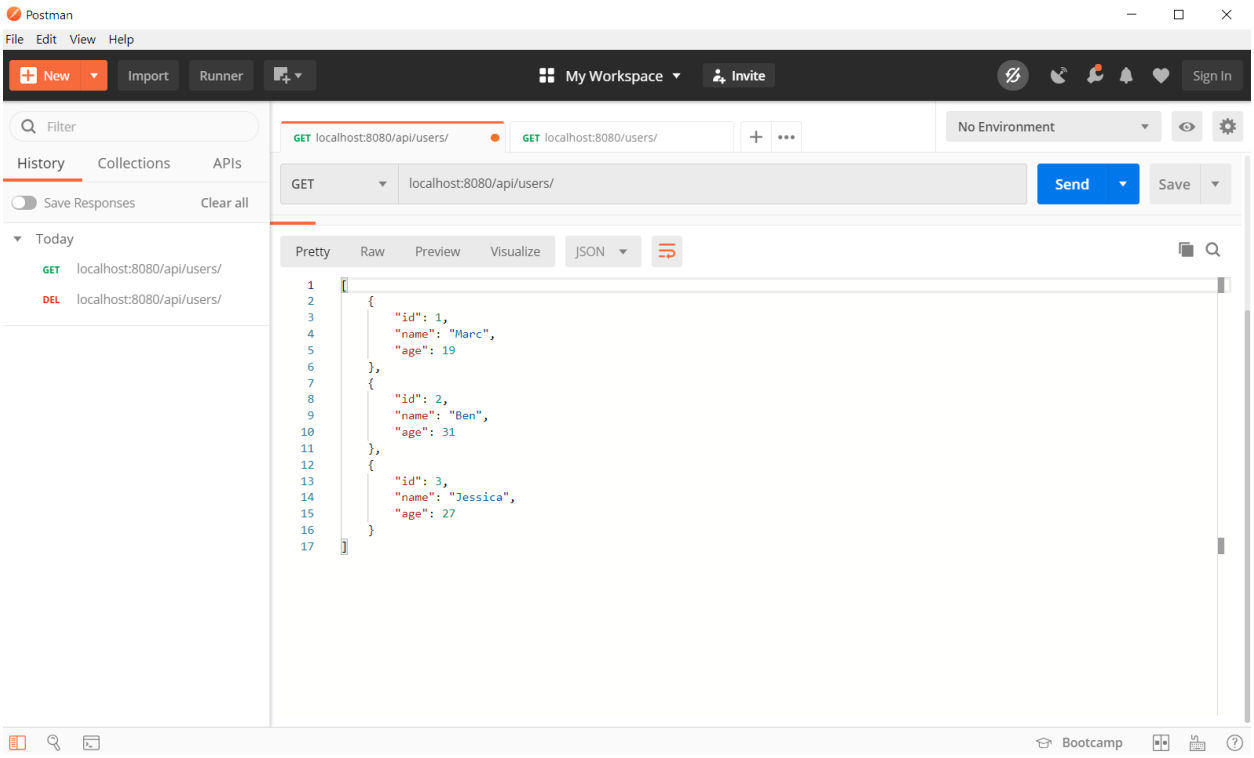
/routes/api/todos.js

```
const express = require("express");
const router = express.Router();

// Create endpoint of all todos
router.get("/", (req, res) => {
  res.json([
    {
      id: 1,
      user_id: 1, // -> Marc
      todo: "Buy Milk",
      done: 0
    },
    {
      id: 2,
      user_id: 1, // -> Marc
      todo: "Walk the dog",
      done: 0
    },
    {
      id: 3,
      user_id: 3, // -> Jessica
      todo: "Call grandma",
      done: 0
    }
  ]);
});

// Export to api.js
module.exports = router;
```

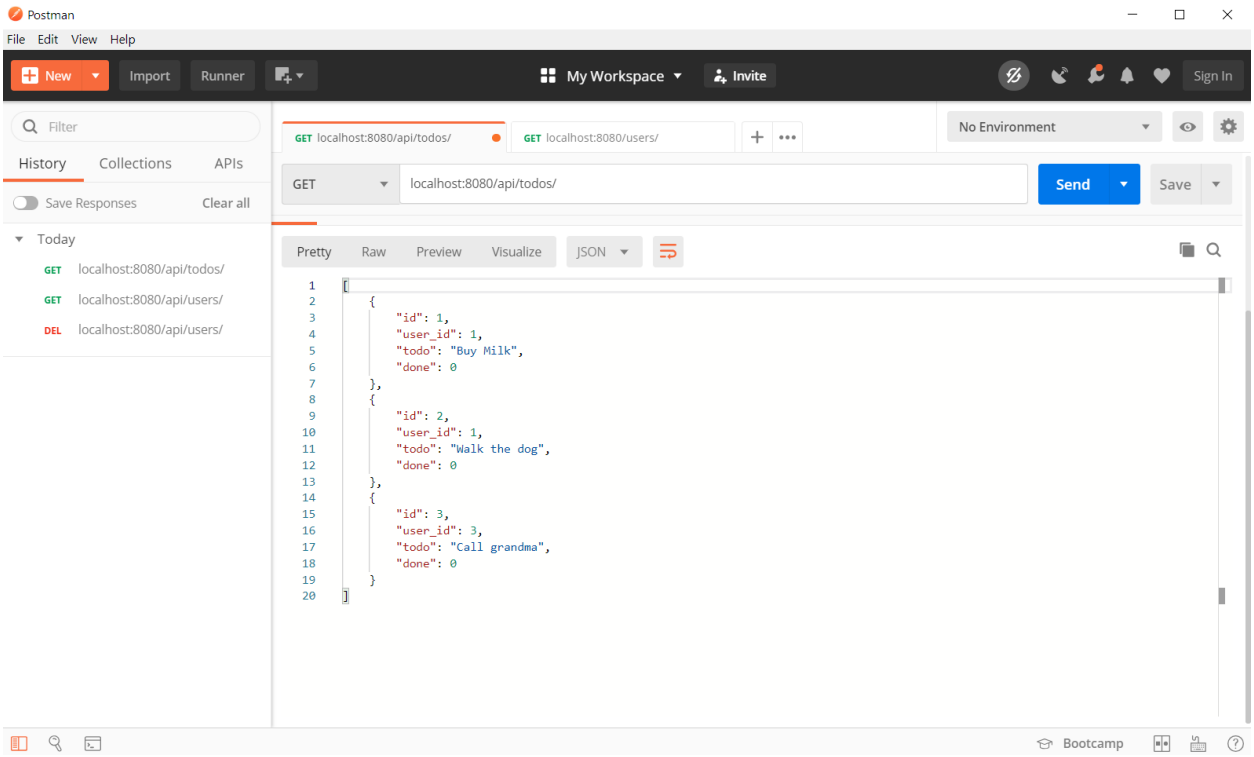
Testing the endpoints

If you would now visit the endpoints with Postman, you should see for

localhost:8080/api/users/



localhost:8080/api/todos/



Now we should have the underlying REST API setup and are ready to implement the database with Knex.js and Objection.js

> This setup is an overkill for this small and only REST API project, but hopefully, it is a good base for future projects.

1.0 Knex.js & Objection.js

Knex.js is the package that manages the connection between Node.JS and your SQL database. Knex.js provides a set of methods to execute CRUD operations on the database, which means you don't write actual SQL.

Additionally to these methods, Knex.js comes with more tools like:

- **migrations** - makes your life more comfortable managing the tables of your SQL database. Migrations allow us to quickly alter a table after we have already created all the tables.
- **seeds** - helps you to populate the database with initial values. Usually, you'll need to seed your database to set some pre-defined values or for testing purposes.

Objection.js is an ORM and built upon Knex.js and allows us to define models for our SQL database. It also comes with build-in validation using a JSON schema you define for each model.

Getting Started

A project which implements both packages should always start setting up Knex.js first and then Objection.js. The database should have the tables and some data first before working with Objection.js.

1.1 Knex.js Setup

In *Chapter 0.0 - Introduction*, we mentioned that we would use the MySQL database shipped with XAMPP, but you are not bound to use XAMPP or MySQL altogether. Knex.js supports Postgres, MSSQL, MySQL, MariaDB, SQLite3, Oracle, and Amazon Redshift.

1. Create the database

Before we can use Knex.js, we have to create the database manually. Our database name is going to be *node_todo*.

Run the XAMPP app and start the MySQL module. Then connect to your database either using a terminal or phpMyAdmin, which is delivered with the XAMPP installation.

Connecting using the terminal:

```
mysql -u root
```

You maybe have a user with a password. Connect accordingly.

After you connected in the terminal or phpMyAdmin, create a new database.

Creating a database using the terminal:

```
CREATE DATABASE node_todo CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
```

2. Install Knex.js

Head back to your project and run in your terminal:

```
npm install knex --save
```

If you haven't installed knex globally yet, do so.

```
npm install knex -g
```

The reason behind this is that you want to run `knex` commands in the terminal too. Installing it globally, you can do so without troubleshooting issues depending on your operating system.

3. Install MySQL client

The connection between MySQL and NodeJS is actually made by a different package (called the client). Therefore, we have to install the additional package called `mysql` for our MySQL database. In your terminal, type:

```
npm install --save mysql
```

> **`mysql` vs `mysql2` ?** If you search online, you'll find that there are two major mysql clients for NodeJS, `mysql` and `mysql2`. They are both supported in the same way, so it doesn't matter which one to use. Many implementations go with `mysql`.

4. Initialize Knex.js

Run in your terminal:

```
knex init
```

This initializes the project with knex and generates a file named **knexfile.js**, which holds the database connection configurations.

5. Create the database config file and directory

You want to separate the database connection credentials in a project and remove it from a version control system such as *git*. Only a template of how these credentials are stored should be published online, never the real credentials file.

Therefore are we going to create a new directory with two files: **/config/db_config.js** - contains the real credentials and should be ignored in .gitignore

```
module.exports = {
    host: "127.0.0.1",
    database: "node_todo",
    user: "root",
    password: ""
};
```

/config/db_template.js - contains the template of how credentials are stored, should be published with the project

```
module.exports = {
    host: "127.0.0.1",
    database: "my_dev_db",
    user: "username",
    password: "password"
};
```

6. Configure knexfile.js

If you open the **knexfile.js**, you'll see that there are many setups to connect to production, development, and staging database. However, we only have the MySQL database, which we use to develop the project locally.

Therefore, import the */config/db_config.js* credentials and configure the development database accordingly.

knexfile.js

```
const credentials = require('./config/db_config.js');

module.exports = {
  development: {
    client: 'mysql',
    connection: {
      database:   credentials.database,
      user:           credentials.user,
      password:   credentials.password
    }
  }
};
```

The client field uses the previously installed `mysql` package. It is the field that tells Knex.js which third party client it should use to connect to the database. Other databases, like PostgreSQL, need a different client, and you can find the correct packages on Knex.js documentation websites.

1.2 Knex.js migrations (tables)

We'll use Knex.js **migrations** to create our database tables and to track changes within our database schema. Migrations allow us to easily alter a table after we have already created all the tables.

We want to create the following tables using Knex.js instead of writing raw SQL. In regular SQL, the code to create the tables would look like the following

```
CREATE TABLE IF NOT EXISTS users (
    id INTEGER AUTO_INCREMENT,
    name VARCHAR(255) NOT NULL,
    age INTEGER,
    PRIMARY KEY(id)
);

CREATE TABLE IF NOT EXISTS todos (
    id INT AUTO_INCREMENT,
    todo VARCHAR(255) NOT NULL,
    done BOOLEAN NOT NULL DEFUALT false,
    FOREIGN KEY (user_id) REFERENCES users(id)
    ON UPDATE CASCADE
    ON DELETE CASCADE,
    PRIMARY KEY(id)
)  ENGINE = InnoDB;
```

Above is a basic schema, covering an OneToMany foreign key relationship in the todos tables (One user can have Many todos).

Creating migrations file

We are going to create two migration files, one to manage the **users** table and another for the **todos** table.

Create two migration files by executing the following code in your terminal

```
knex migrate:make create_tables
```

> **Note** Your migration name should be something descriptive, not simply the name of the table(s) that you are creating / modifying. Remember that throughout the lifetime of your app, there may be several migrations that all relate to the same database table. Hence, you want your migration names to inform other developers as to what that migration is doing.

The previous command generates a **/migrations** folder in your project directory with a migration file inside:

- **20200331182227_create_tables.js**

Notice the long number in the filename. It is a timestamp of the file creation time. Timestamps are necessary because Knex.js uses the timestamp to know which migrations to run first (This makes sense in a bit).

Opening the migrations file, you notice that they are almost empty except for a bit of boilerplate code.

There are two functions defined here

```
// Executed during a migration
exports.up = function(knex) {

};

// Executed during a rollback
exports.down = function(knex) {

};
```

In the `exports.up( )` function, we define table names and columns to create in the database during migration. We use the `exports.down( )` function to specify what changes we want to undo during a rollback (usually used to drop the table).

> **Note:** It is possible to create for each table one migration file. E.g. *20200331182227_create_users.js* and *20200331182227_create_todos.js*. This is also recommended for multiple application projects. Be careful choosing this approach as you **must** first create the users migration file before the todos due to the dependency and the migration order. Read more here: **Knex.js migrations, does each table has its own migration?**

Creating migrations

Our `exports.up( )` and `exports.down( )` functions should always return a promise, therefore is the `return` keyword necessary. We are going to tackle one after another.

exports.up( )

Inside of the `exports.down` function of your newly created migration file add the following code

/migrations/20200331182227_create_tables.js

```
// Executed during a migration
exports.up = function(knex) {
  return knex.schema
    .createTable("users", table => {
      table.increments("id");
      table.string("name").notNullable();
      table.integer("age");
    })
    .createTable("todos", table => {
      table.increments("id");
      table.string("todo").notNullable();
      table.boolean("done").defaultTo(false);
      table.integer("user_id")
        .unsigned()
        .notNullable();

      // Set the foreign key
      table.foreign("user_id")
        .references("id")
        .inTable("users")
        .onDelete("CASCADE")
        .onUpdate('CASCADE');
    });
};
};
```

In the code above:

- we create a table with using the `.createTable(tableName, callback)` method

- we can chain multiple `.createTable(tableName, callback)` methods to create an additional table.
- the callback function takes an argument. Using the argument we can chain field methods to create columns with or without constraints.
- `increments("id")` is the equivalent to `INTEGER AUTO_INCREMENT PRIMARY KEY` in SQL
- `.notNullable()` is the equivalent to `NOT NULL` in SQL
- `.defaultTo(value)` is the equivalent to `DEFAULT value` in SQL
- `.unsigned()` denies negative numerical values and is needed for foreign keys.
- `.foreign("user_id").references("id").inTable("users").onDelete("CASCADE").onUpdate('CASCADE');` is the equivalent to
  `FOREIGN KEY (user_id) REFERENCES users(id) ON UPDATE CASCADE ON DELETE CASCADE` in SQL

Documentation:

- **Schema Builder methods - Knex.js** e.g. `.createTable(tableName, callback)`
- **Schema Builder field methods - Knex.js** e.g. `.string("name")`
- **Schema Builder chainable field methods - Knex.js** e.g. `.notNullable()`

exports.down( )

Inside of the `exports.down( )` function of your newly created migration file add the following code

/migrations/20200331182227_create_tables.js

```
// Executed during a rollback
exports.down = function(knex) {
  return (
    knex.schema
      // Here, delete tables in reverse order because todos depends on users
      .dropTableIfExists("todos")
      .dropTableIfExists("users")
  );
};
```

In the code above:

- we delete a table with using the `.dropTableIfExists(tableName)` method, which also can be chained

Run migrations

Lastly, we have to apply the migration files, so that Knex.js creates the tables in our SQL database.

In your terminal, run

```
knex migrate:latest
```

You can log in to MySQL CLI or phpMyAdmin and check that the tables, alongside two knex tables, have been created.

```
MariaDB [node_todo]> show tables;
+----------------------+
| Tables_in_node_todo  |
+----------------------+
| knex_migrations      |
| knex_migrations_lock |
| todos                |
| users                |
+----------------------+
4 rows in set (0.001 sec)
```

Knex.js automatically create the tables *knex_migrations* and *knex_migrations_lock*. These tables maintain a running list of which migrations have been implemented. It is best not to touch those two.

Rollback migrations

If you ever wish to update the database tables, you either create an additional migrations file that alters the existing table, or you can run a rollback which deletes the tables, then do the required updates inside `exports.up( )` and `exports.down( )` functions and run a new migrations to re-create the updated migrations file.

However, best practice is that you should create a new migration file for each database change.

In your terminal, run

```
knex migrate:rollback
```

to rollback a migration.

1.3 Knex.js Seeds (initial values)

Seeding means setting the database tables with sample data to get a new development environment up and running. Knex makes it easy to seed a database. We seed our users and todos table with some initial values.

Creating Seed files

You can instruct Knex to create new seed scripts using the following command in your terminal

```
knex seed:make 001_create_data
```

You will see a newly created directory and file **/seeds/001_create_data.js** with a boilerplate.

> **Note** Unlike migrations, seeds don't have a timestamp. Your seed name should, therefore, be something descriptive and start with a version number, e.g., `001_`, `002_`. Seed files are executed sequentially (one after another) if used with a version number.
>
> Remember that throughout the lifetime of your app, there may be several seeds that all relate to the same database table, so you want your seeds names to inform other developers the version and action it does.

Lets edit the seed file to our needs **/seeds/001_create_data.js**

```
exports.seed = function(knex) {
  // Deletes ALL data of todos
  return knex("todos")
    .del()
    .then(() => {
      //Delete ALL data of users
      return knex("users").del();
    })
    .then(() => {
      //Inserts new data into users
      return knex("users").insert([
        {
          name: "Marc",
          age: 19
        },
        {
          name: "Ben",
          age: 31
        },
        {
          name: "Jessica",
          age: 27
        }
      ]);
    })
    .then(users => {
      /**
       * We can use the callback of the previous users inserts,
       * which returns a single item or an array of items (array only available in PostgreSQL), to
       * insert todos data and establish the relationship with users.
       */
      return knex("todos").insert([
        {
          user_id: 1, //-> Marc
          // alternatively to the fixed value one, you can also use
          // user_id: users[0]
          todo: "Buy Milk"
        },
        {
          user_id: 1, //-> Marc
          todo: "Walk the dog"
        },
        {
          user_id: 2, //-> Jessica
          todo: "Call grandma"
        }
      ]);
    });
};
```

Explanation: You notice that the seed file starts by deleting any rows in the database. This is so that we can ensure we're starting with a clean slate any time we run our seed file.

Because we are working with users and todos, we need to clear out both tables (todos first, as they depend on users existing).

Knex.js nature is promise based and expects a callback. The same goes for seeding data to the database. We not only perform one task but multiple ones. Chaining multiple tasks with a `.then( ()=>{ ... } )` method allows us to perform multiple tasks one after another.

`.insert()`, `.del()` and many other Knex.js methods return a callback with a single or multiple items (multiple items, except get

queries, are only available in Postgress) which then can be used in the following chained `.then()` method.

Documentation: **insert, delete, update, ... QueryBuilder methods**

Apply seeds

To apply the seed data to your database tables, type in your terminal

```
knex seed:run
```

Seeds issue

Running seeds inserts new rows in the tables. These tables have an *auto increment* id attribute. Each time new rows are added, the id is increased to a new +1 value.

The above seed file only works once, when no data has ever been added to the table before!

Trying to re-run those seeds (i.e., delete all data and re-insert), they will fail because the users ids starts at 4 after we inserted the data the first time. The hard coded users id 1 and 3 in line 38, 44, and 48 do not exist anymore.

And no, there is no logic build in rollback command as we know it for migrations.

Two solutions to this issues are shown below.

Solution A:

Doing a full reset, which includes: 1. `knex migrate:rollback` -> delete table 2. `knex migrate:latest` -> create table

makes sure that the tables *auto increment* value for the ids start again at 1.

Now you could run the seed without issues **one time** again.

Using this method, you have to redo a full reset every time you want to add new data to tables.

Solution B:

If it doesn't matter to use only one users id for all todos, then you can make use of the callback in line 26, which always returns the id of the first inserted user in the users table.

Here would then be the updated seeds file: **/seeds/001_create_data.js**

```
exports.seed = function(knex) {
  // Deletes ALL data of todos
  return knex("todos")
    .del()
    .then(() => {
      //Delete ALL data of users
      return knex("users").del();
    })
    .then(() => {
      //Inserts new data into users
      return knex("users").insert([
        {
          name: "Marc",
          age: 19
        },
        {
          name: "Ben",
          age: 31
        },
        {
          name: "Jessica",
          age: 27
        }
      ]);
    })
    .then(users => {
      console.log(JSON.stringify(users, null, 4));
      /**
       * We can use the callback of the previous users inserts,
       * which returns a single item or an array of items (array only available in PostgressSQL), to
       * insert todos data and establish the relationship to users.
       */
      return knex("todos").insert([
        {
          user_id: users[0], //-> Marc
          todo: "Buy Milk"
        },
        {
          user_id: users[0], //-> Marc
          todo: "Walk the dog"
        },
        {
          user_id: users[0], //-> Marc
          todo: "Call grandma"
        }
      ]);
    });
};
```

Because of database system natures, PostgreSQL is the only database which returns **all** inserted items. MySQL and the others can only return the first item.

1.4 Knex.js Queries

The database tables are created and have sample data. Now we can have a look at how we to query the database in NodeJS using Knex.js query methods.

For this exercise, create a **test.js** file in the project's root folder. We will delete this file later on but use it for now to run some examples.

Get all users

To retrieve all users, put the code below in the **test.js** file

```
// Include the knex package and config file
const Knex = require("knex");
const knexFile = require("./knexfile.js");

// Make the connection to the database
const knex = Knex(knexFile.development);

// Run queries
knex("todos")
  .where("user_id", 1)
  .then(rows => {
    for (row of rows) {
      console.log(row);
    }
  })
  .catch(err => {
    console.log(err);
    throw err;
  });
```

In the code above, we run all queries on the knex method, which takes a single argument, the table name to run the query on. Then by chaining multiple query methods, can we finally handle the returned data using the `.then( )` method, which takes the query result as a callback from the previous chained methods.

The variable `rows` contains now all query results, and we console.log them using a for a loop.

A full list of available query methods can be found here **Query Builder - Knex.js**

2.0 Objection.js Setup

Setting up Objection.js is fairly simple. The only requirement you need is to have `knex` alongside a database client, like `mysql` installed and initialized with migrations (and seeds if you want sample data).

Both requirements have been met by following along *Chapter 1*.

What's left is to install objection. Type in your terminal:

```
npm install objection
```

2.1 Objection.js Models

The first thing we want to define is our models, which represent the tables in our SQL database.

You need to understand that Objection.js models are not creating the database table structure as used to with other languages and frameworks. Objection.js only task is to retrieve (called *getters*) from or update (called *setters*) to **existing tables**. Database table structures are handled by Knex.js migration files.

Objection.js provides many getter and setter methods.

The configuration of Objection.js happens through model classes, and there is no global configuration file (unlike the *knexfile.js* config file of Knex.js).

Our first Model

We start by creating a working model with the minimum amount of code. Afterward, when we understand the basics, the code will be restructured and separated into their own files like **server.js**, **/models/User.js**, … and so on.

Reuse the previously created **test.js** and insert the following code:

```
// 1. Import and initialize Knex.js
const Knex = require("knex");
const knexFile = require("./knexfile.js");
const knex = Knex(knexFile.development);

// 2. Import Objection.js Model class
const { Model } = require("objection");

// 3. Bind all models to the knex instance
Model.knex(knex);

// 4. Create the User model class
class User extends Model {
  static get tableName() {
    return "users";
  }
}

// 5. Run the query in async/await
const getUsers = async () => {
  const users = await User.query();
  console.log(users);
};
getUsers();
```

Output

```
$ node test.js
[
  User { id: 1, name: 'Marc', age: 19 },
  User { id: 2, name: 'Ben', age: 31 },
  User { id: 3, name: 'Jessica', age: 27 }
]
```

In the above code: 1. Knex.js is initialized the same way as the **test.js** file of *Chapter 1.4 Knex.js Queries* 2. The Model class is imported from Objection.js 3. Bind all models to the knex instance 4. Create a minimal working `User` model class, which extends the `Model` object. The `static get tableName()` is one of many Objection.js native **static properties** and is **required** for a model. `tableName()` returns the name `users` : the name of the database table we want our `User` class to model. This is the property that tells Objection.js during a query what table to query on. I.e., you don't specify table names during a query as previously seen in *Chapter 1.4 Knex.js Queries*. 5. Run the query in an async/await (or you could alternatively use `.then()` functions) because the query returns a promise.

Restructuring our first Model

Now that we have the basic understanding of an Objection.js with Knex.js structure, we can split up the code, so it works in our initial project setup of NodeJS with Express.

1. Users Model

Each database table should have its own model, they should all live in a folder named `/models/` and the first character should be capital to indicate a class file.

Let's create the users model and add some code.

/models/Users.js

```
// Include Model
const { Model } = require('objection');

// Step 4 from test.js moved here
class User extends Model {
  static get tableName() {
    return 'users';
  }
}

// Export User class
module.exports = User;
```

2. Include Knex.js and Objection.js in our project

server.js is our project configuration file but is currently missing Knex.js and Objection.js. Let us change that

server.js

```
const express = require("express");
const app = express();

const apiRoutes = require("./routes/api");

// Step 1 from test.js moved here
const Knex = require("knex");
const knexFile = require("./knexfile.js");
const knex = Knex(knexFile.development);

// Step 2 from test.js moved here
const { Model } = require("objection");

// Step 3 from test.js moved here
Model.knex(knex);

app.use("/api", apiRoutes);

const server = app.listen(8080, error => {
  if (error) {
    console.log("Error running Express");
  }
  console.log("Server is running on port", server.address().port);
});
```

3. Update the Express routes

Earlier on, the **/routes/api/users.js** file returned hardcoded data. Now that our project is hooked up to Objection.js and Knex.js can we return data directly from our SQL database.

```
const express = require("express");
const router = express.Router();
const User = require("../../models/Users.s"); // Extra: Import the User

router.get("/", async (req, res) => { // Extra: add async to the function

  // Step 5 from test.js moved here
  const users = await User.query();
  res.json(users);


  // The old response
  /*
  res.json([
    {
      id: 1,
      name: "Marc",
      age: 19
    },
    {
      id: 2,
      name: "Ben",
      age: 31
    },
    {
      id: 3,
      name: "Jessica",
      age: 27
    }
  ]);
*/
});

module.exports = router;
```

Ensure that you import the User model and that the anonymous `(req,res) => { ... }` recieves the `async` keyword.

Todo model

So far we have a working project that only returns *users* using the database. Lets implement *todos* too, create a new file **/models/Todos.js** and place this code in it

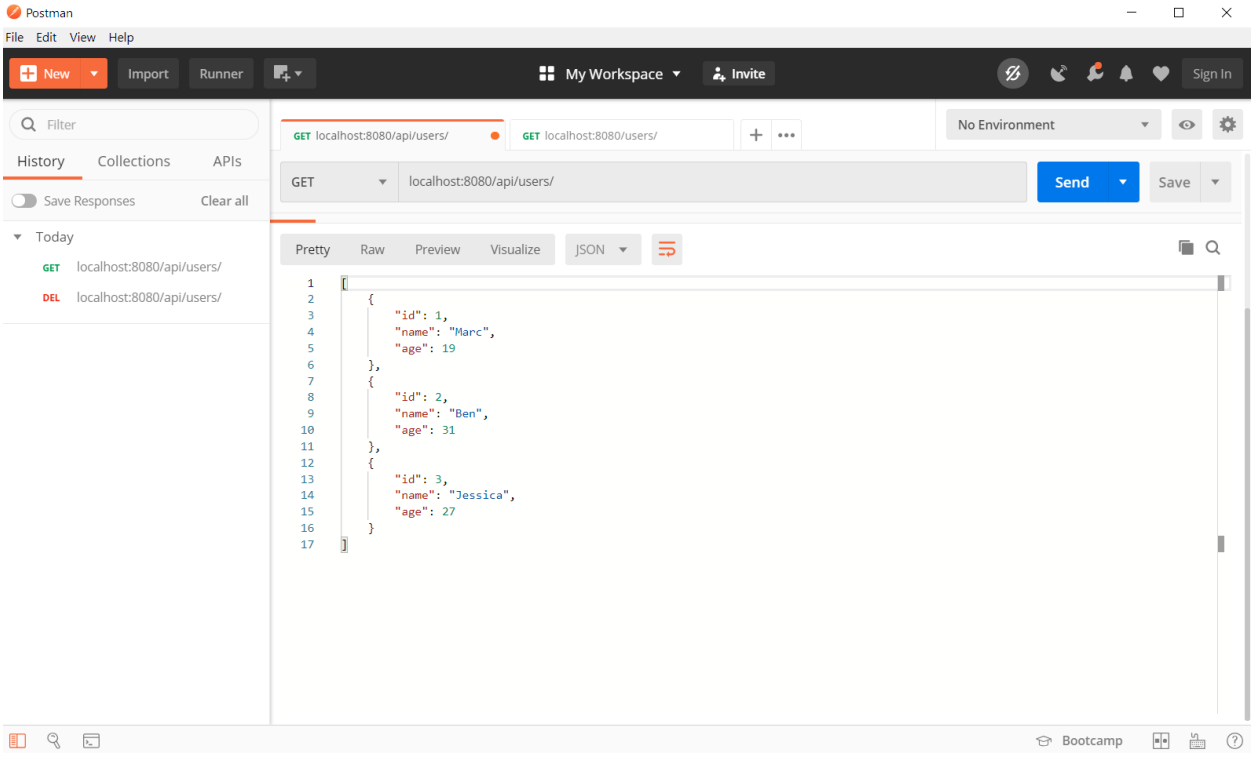
```
// Import Model class from Objection.js
const { Model } = require("objection");

// Create the Todo model class
class Todo extends Model {
  static get tableName() {
    return "todos";
  }
}

// Export the Todo to be used in routes
module.exports = Todo;
```

The project is configured with Objection.js and Knex.js; therefore the only thing missing is to update the **/routes/api/todos.js** file

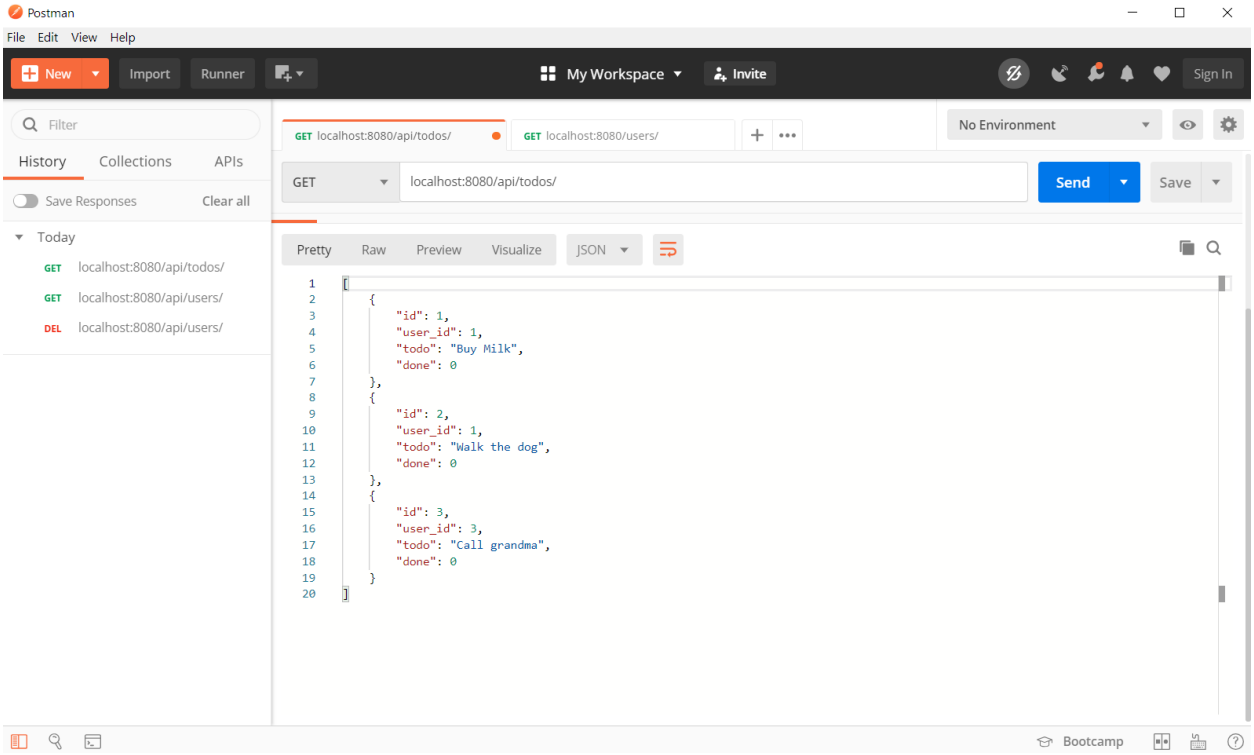
Testing the endpoints

Visiting the *users* endpoint with Postman, you should see for

localhost:8080/api/users/



localhost:8080/api/todos/



2.2 Objection.js Queries

Looking through **Objection.js Query Documentation**, it quickly becomes apparent that a large amount of the documented queries are **Knex.js Queries**.

The `.query()` method is used to start a query operation on a model and can be chained with **Query Builder** methods.

Using the **/routes/api/users.js** and **/routes/api/todos.js**, we will demonstrate some basic queries.

Todos - Get only the `todo` and `done` field

Using `.select()`, you can specify which fields to return. `Model.query()` and `Model.query().select("*")` are always the same, except if the query is done on multiple tables (e.g. for a JOIN).

/routes/api/todos.js

```
router.get("/", async (req, res) => {
  const todos = await Todo.query().select("todo","done")

  // Handle if no todos have been found
  if(!todos.length === 0){
    return res.status(404).json({message: "No todo found"})
  }

  res.json(todos);
});
```

Outputs

```
[
    {
        "todo": "Buy Milk",
        "done": 0
    },
    {
        "todo": "Walk the dog",
        "done": 0
    },
    {
        "todo": "Call grandma",
        "done": 0
    }
]
```

Users - Fetch user by id

The `.findById()` method returns a single object. If no item could be found, the query returns `undefined`.

/routes/api/users.js

```
router.get("/", async (req, res) => {
  const user = await User.query().findById(2); // returns a single user in an object, not in an arra
  res.json(users);
});
```

Outputs

```
{
    "id": 2,
    "name": "Ben",
    "age": 31
}
```

Todos - Get all todos where todo value is “Buy milk”

`.where( COLUMN , VALUE )` allows us to specify a search condition. An empty array is returned if nothing matched the condition . **/routes/api/todos.js**

```
router.get("/", async (req, res) => {
  const todos = await Todo.query().where("todo", "Buy Milk");

  // Handle if no todos have been found
  if(!todos.length === 0){
    return res.status(404).json({message: "No todo found"})
  }

  // If todos exist, return the todo set
  res.json(todos);
});
```

Outputs

```
[
    {
        "id": 1,
        "todo": "Buy Milk",
        "done": 0,
        "user_id": 1
    }
]
```

More query examples can be found on Objection.js **query example page** or **Query Builder API page**.

2.3 Objection.js Relationships

In *Chapter 1.2 - Knex.js migrations*, we defined an OneToMany relationship between the table `users` and `todos` with the help of foreign keys inside `todos` .

While these relationships usually are created between the primary key of one table and a foreign key reference of another table, Objection.js has no such limitations. You can create a relationship using any two columns (or any sets of columns). You can even create relations using values nested deep inside JSON columns.

This is helpful while querying for items from multiple tables (e.g., in a JOIN operation) as we can pre-define relationships and then create non-complex queries using these pre-defined relationships.

> **Reminder** Our Objection.js models **are not** the ones creating tables. They only *get/set* data *from/to* the tables. The following sections have nothing to do with defining and creating the database tables.

relationMappings

The `relationMappings()` **static property** defines the relations (relationships, associations) to other models.

Lets look how we would add `relationMappings()` to our Todo class in **/models/Todo.js**

```
const { Model } = require("objection");

// Import User class model for relationMappings()
const User = require("./Users.js");

class Todo extends Model{

    class Todo extends Model {
      static get tableName() {
        return "todos";
      }

    // defines the relations to other models.
    static get relationMappings() {
          return {
              user: {
                  relation: Model.BelongsToOneRelation,
                  modelClass: User,
                  join: {
                      from: "todos.user_id",
                      to: "users.id"
                  }
              }
          };
    }
}
```

In the syntax above:

- `relationMappings()` always returns an object
- the key `user` is how we will refer to it to the parent class
- the key `relation` has the relationship value `Model.BelongsToOneRelation` which says that each todo has one user.
- the key `modelClass` says that the whole `user` object comes from the `User` class
- the key `join` specifies the database table and column names to perform a SQL `JOIN` operation on, in this case, the `user_id` column in the `todos` table to the `id` column in the `users` table

Model.RELATION_TYPE

There are three RELATION_TYPE to choose from

`Model.BelongsToOneRelation` The parent class has one relation to another model. Example: The Todo class uses `.BelongsToOneRelation` because a todo only has one assigned user.

`Model.HasManyRelation` The parent class has many relations to another model. Example: The User class uses `.HasManyRelation` because a user can have many todos)

`Model.ManyToManyRelation` The parent class and another model have many relations with each other. Example: Currently, neither the User or Todo has a `ManyToManyRelation`, but if it was possible that multiple users could have the same todo task, then we could use the `ManyToManyRelation`.

Query the relation

Simply because the relation to users has is defined doesn't mean that it's included in the base `.query()` function. You still need to

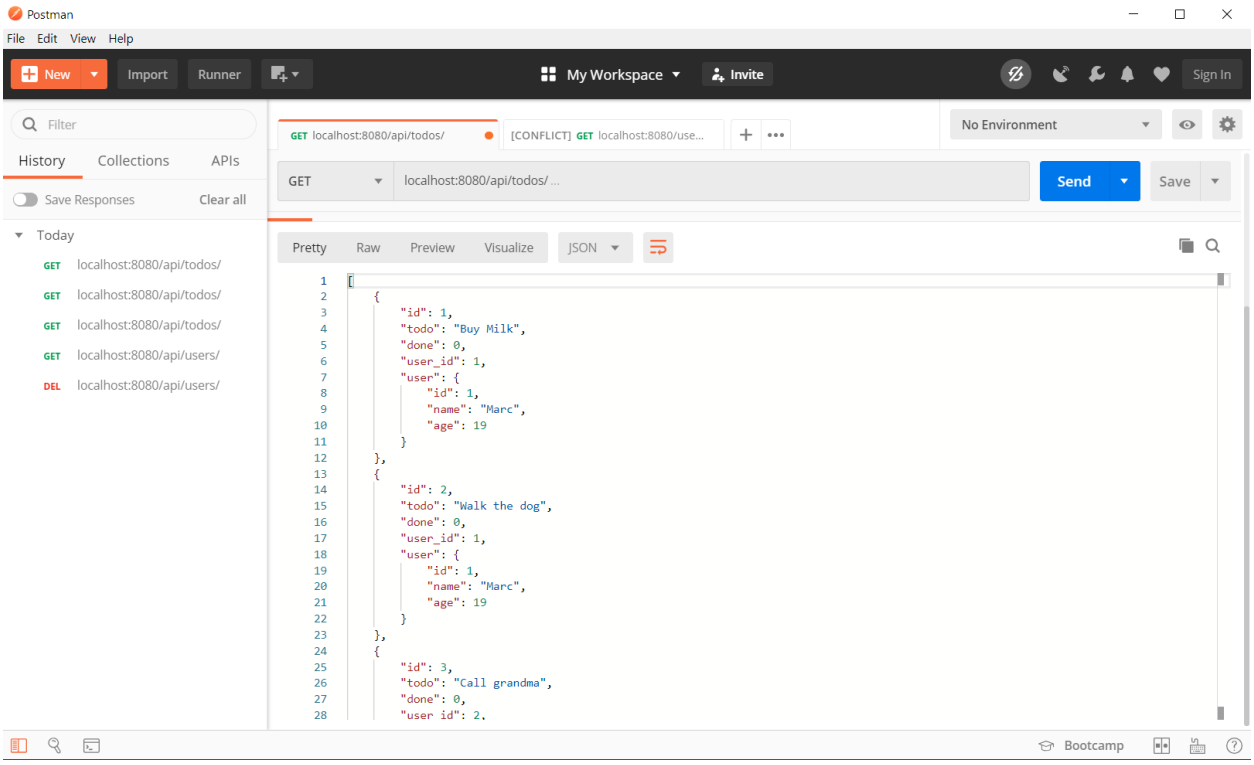
specify that you want to use it. This is achieved by using the `withGraphFetched()` function.

We can see it in action by updating our **/routes/api/todos.js** file

```
const express = require("express");
const router = express.Router();
const Todo = require("../../models/Todos.js");

// Create endpoint of all todos
router.get("/", async (req, res) => {

  // Get all todos WITH the assigned user
  const todos = await Todo.query().withGraphFetched("user");
  res.json(todos);
});

// Export to api.js
module.exports = router;
```

Here, the text `user` inside `.withGraphFetched` refers to our **custom key** in the Todo class `relationMappings()` and not to the users database table.

Testing the updated query in Postman shows



Why use this approach?

You might already have inspected Objection.js and Knex.js documentation and figured out that there is also a `.join()` **method**, which for our case would look like

```
// Get all todos WITH the assigned user
  const todos = await Todo.query()
    .select("*")
    .rightJoin("users", "todos.user_id", "=", "users.id");

  res.json(todos);
```

with the SQL equivalent

```sql
SELECT *
FROM todos
RIGHT JOIN users
ON todos.user_id = users.id;
```

Both approaches are totally valid, but when you create relation mappings to model (using `relationMappings()`), you don't have to write joins manually every time you need to query relations. Also, they enable many other objection features, which require information about how row relations goes in DB.

Eager loading

The whole above process is called **eager loading**. Eager loading is the process whereby a query for one type of entity also loads related entities as part of the query.

For example, when querying todos, eager-load their users. The todos and their users are retrieved in a single query.

Multiple methods can be used to load relations eagerly: `.withGraphFetched()` (the one we have just seen) and `.withGraphJoined()`. The main difference is that `.withGraphFetched()` uses multiple queries under the hood to fetch the result while `.withGraphJoined()` uses a single query and joins to fetch the results. Both methods allow you to do different things which we will go through in detail in the examples below and the examples of the `.withGraphJoined()` method.

Even though `.withGraphJoined()` sounds more performant, but that is not always true. 90% of the times you want to use `.withGraphFetched()`.

Why tell you this? You'll likely find online resources on Objection.js `relationMappings()` and eager loading where they use the `.eager()` method. However, this method is deprecated, and you should use `.withGraphFetched()`.

2.4 Objection.js Validation

At this stage, the REST API only returns data from the database and is missing the feature to create or update rows.

As soon as you handle user input, you have to validate the data. Objection.js has an in-house validator, the `jsonSchema()` **method**, where you define properties, data types, and additional constraints.

Lets have a look at our User model with such a validator:

/models/User.js

```
const { Model } = require("objection");

class User extends Model {
  static get tableName() {
    return "users";
  }

  // jsonSchema used for input validation. This is not the database schema!
  static get jsonSchema(){
    return {
      type: 'object',
      required : ['name'],

      properties:{
        id: {type:'integer'},
        name: {type: 'string', minLength:1, maxLength:255},
        age: {type: "number"} //optional
      }
    }
  }
}

module.exports = User;
```

In the code above:

- `jsonSchema()` returns always an object with a key/value pair of `type:'object'` This is a default to Objection.js.
- they key `required` accepts an array of key names that must have a value. A `required` key name needs to exist inside the `properties` key
- `name` and `age` key names have no direct link to the database columns but should have the same name as the column name.
- the key `properties` is used to specify key names and their respective data type (and constraints).
- `type` reflects what data type you expect to receive. Some basic types are `string`, `number`, `integer`, `boolean`, … The difference between `number` and `integer` is that `34` and `1.82` satisfy `number`, but only `34` satisfies `integer`. A more detailed reference can be found **here (JSON Schema type Reference)**.
- `minLength` and `maxLength` are data type-specific constraints and are self-explaining.
- `age` is not part of the required key and the key `type` is only used if a value exists

`jsonSchema()` uses the third party tool **JSON Schema** for validation - so you have to use that one for reference.

Validate user input

The validator is ready, so let's insert some data and see what happens.

First, we use static data. Later on, will we use data received from a `POST` request. Let's update our users route

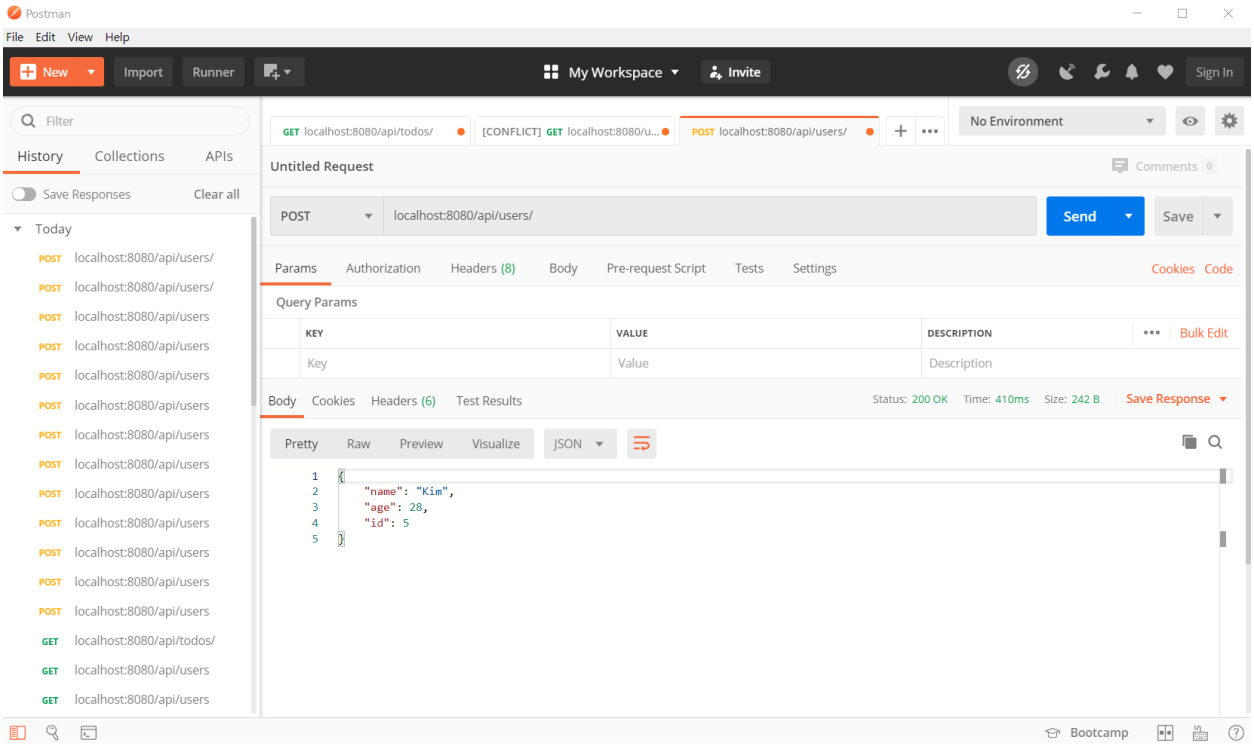
/routes/api/users.js

```
const router = express.Router();

const User = require("../../models/Users");

// get all users
router.get("/", async (req, res) => {
  const users = await User.query();
  res.json(users);
});

// Create new user using a POST request
router.post("/", async (req, res) => {

    // wrap in try/catch
    try {
        const user = await User.query().insert({ name: "Kim", age: 28 });
        res.json(user);
    } catch (err) {
        // VERY basic error handler, look in Error handling chapter for better solution
        res.status(400).json(err.data);
    }
});

// Export to api.js
module.exports = router;
```

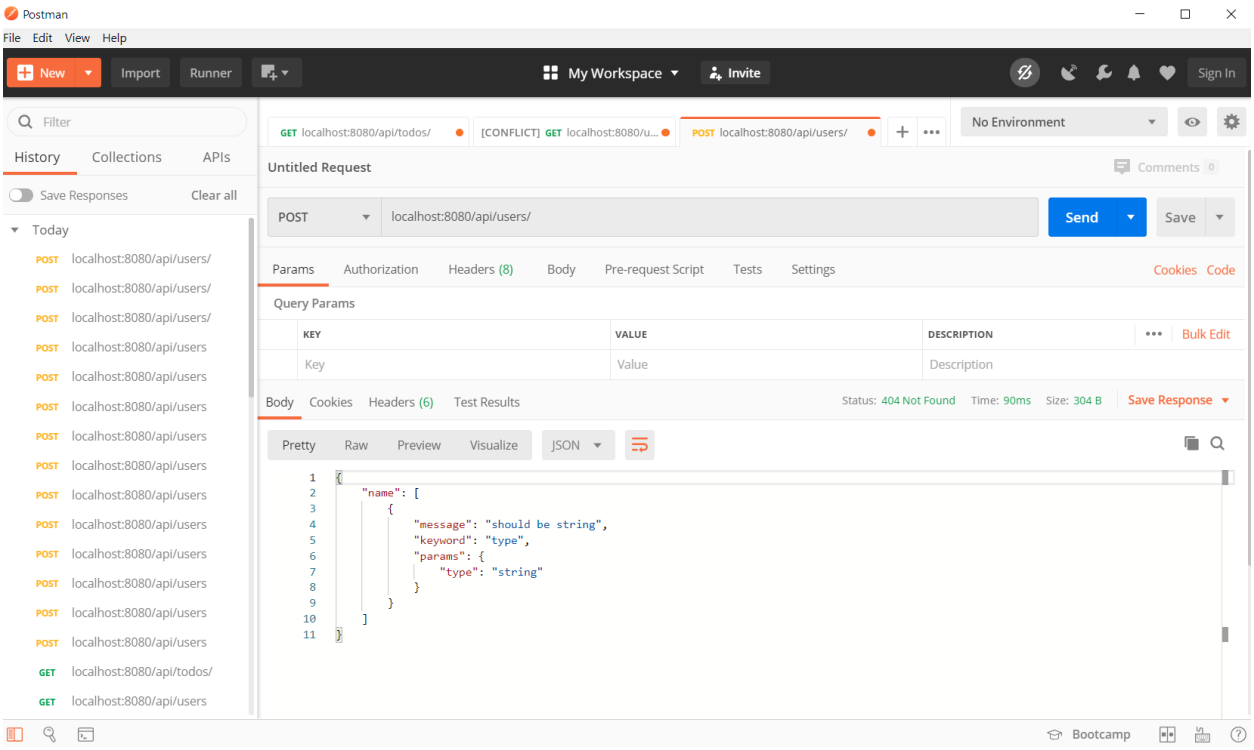
In the code above:

- wrap the insert method in a try/catch to handle validation.
- `insert()` is the method used to insert data. We're going to talk about the method in detail below.
- `catch(err)` catch if validation or insert errors occur and send an HTTP status 400 Bad Request back. Look in *Chapter 3.1 - Error handling* how errors should be dealt with.

Testing with Postman Visiting **localhost:8080/api/users/** with a POST request displays



A failed validation (E.g., using a number for name - `name: 33`) would have looked like this



insert() method

Using the `insert` method, the inserted data is validated against the `jsonSchema` method. The promise throws an error if the validation fails. You can either use `try/catch` or `.then().catch`.

The `insert` method accepts both *objects* and *arrays* (batch insert - for multiple items) **. Still, arrays are only supported if you use a PostgreSQL database** because it is the only database system that returns **all** inserted rows. Sounds familiar? We have already seen the same issue in *Chapter 1.3 - Knex.js Seeds*.

Workaround for MySQL and other non-PostgreSQL databases We have done batch inserts in *Chapter 1.3 - Knex.js Seeds* using plain Knex.js without Objection.js. Using `Model.knexQuery()` instead of `Model.query()` allows us to run queries directly through Knex.js (and not Objection.js models). Even with `knexQuery` there is still only one item returned with multiple inserts.

3.0 Custom user data

The REST API is ready to accept incoming POST requests with user input. The user will most probably send data in JSON format. Express accepts JSON, but it needs first to be configured to do so.

Updating our **server.js**

```
const express = require("express");
const app = express();

// Accept JSON data
app.use(express.urlencoded({ extended: false })); // -> add this
app.use(express.json()); // -> add this

const apiRoutes = require("./routes/api");

const Knex = require("knex");
const knexFile = require("./knexfile.js");
const knex = Knex(knexFile.development);

const { Model } = require("objection");

Model.knex(knex);

app.use("/api", apiRoutes);

const server = app.listen(8080, error => {
  if (error) {
    console.log("Error running Express");
  }
  console.log("Server is running on port", server.address().port);
});
```

By adding `app.use(express.urlencoded({ extended: false }));` and `app.use(express.json());`, express is now able to accept POST data in JSON format.

By editing the routes files like below, our user endpoint is ready to accept incoming JSON data.

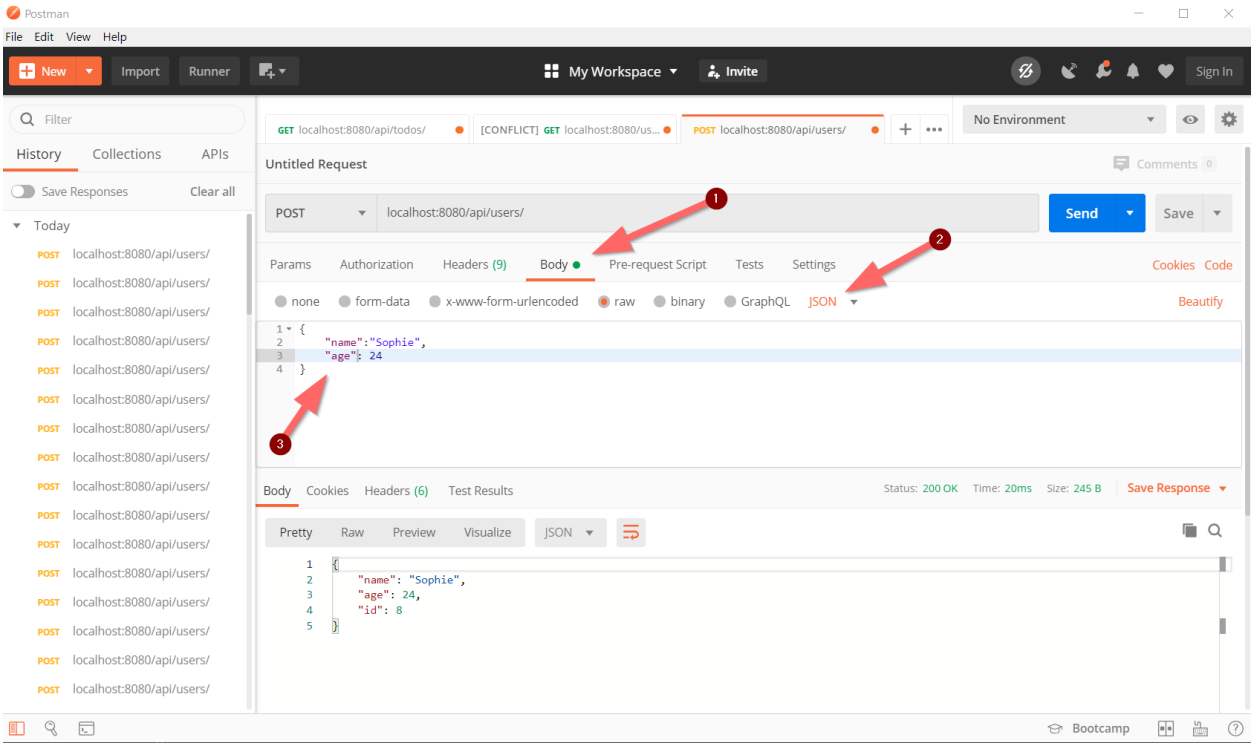
/routes/api/users.js

```
const router = express.Router();

const User = require("../../models/Users");

// get all users
router.get("/", async (req, res) => {
  const users = await User.query();
  res.json(users);
});

// Create new user using a POST request
router.post("/", async (req, res) => {
      // req.body contains the JSON
    const {name, age} = req.body; // destructure name and age out of the sent JSON

    try {
        const user = await User.query().insert({ name, age }); //instead of name:name, can we use na
        res.json(user);
    } catch (err) {
        res.status(400).json(err.data);
    }
});

module.exports = router;
```

FYI: The app will not fail if no age is provided.

Testing with Postman

In Postman, you can send JSON data by visiting the *Body* tab (point 1 in below image) and clicking on JSON (point 2 in below image). Make sure to format your data as actual JSON by using quotation marks (point 3 in below image) !



In the above image, you can see in the body response that the project returns the inserted user with the user id

3.1 Error handling

Error handling is an essential function of our project and is up to us to deal with it. Neither is Objection.js or Knex.js handling any errors.

Current state of error handling

We have seen in *Chapter 2.4 - Validation* that currently, we handle errors on a case level using a try/catch block. And honestly, we don't handle any errors here; we simply respond with the error message to the REST API.

Example: **/routes/api/users.js**

```
// ...

// Create new user
router.post("/", async (req, res) => {
    const {name, age} = req.body;
    try {
        const user = await User.query().insert({ name, age });
        res.json(user);
    } catch (err) {
        res.status(400).json(err.data);
    }
});

// ...
```

The way we currently handle errors is not optimized for a larger project.

First of all, how can we guarantee that we always use the same schema of returned errors? By way of an example, for the users route, *developer A* responds with the full `err.data` object to the API (just like in our code above). However, for the todos route, developer B feels that the response needs additional details and wraps the data in an object, like below

```
return res.status(400).json({status: "false", data: err.data});
```

Having no unified error response confuses the API user. There are no properties he/she can rely on to handle responses correctly.

Second, if you have a large project, how do you make sure that it runs properly and interfere when a significant error happen? You need a way to access errors that might be with logging each error in log files or using one of many monitoring products like **PM2** or **Sentry**. Right now, our project only displays the error to the API-user, and we have no way of accessing them.

Finally, handling errors and returning a response for every try/catch and if/else statement becomes very tedious and makes the codebase messy.

Improved error handling

The solution is to move error handling to a distinct error handling function, and this applies for every Express.js project, not only with Knex.js and Objection.js. A unified and consistent error structure is guaranteed with a distinct function.

> **Note** But you should never send the errors directly to the client as they may contain SQL and other information that reveals too much about the inner workings of your app.

Objection.js provides a set of case-specific errors. Some are:

- `ValidationError` - thrown if validation of any input fails. By input, we mean any data that can come from the outside world, like model instances (using *jsonSchema()*). Returns an HTTP 400 - Bad Request
- `NotFoundError` - has to be invoked manually. Best suited during a get/find query when nothing was found, then you can do `throw new NotFoundError(emptyResult)` to invoke this error. Returns a HTTP 404 Not found
- `DBError` - thrown if the database client, in our case `mysql`, throws an error

Objection.js documentation page provides an **error handler function** specifically made for Objection.js. We are about to implement and use a better, slightly modified version of it in our project.

We haven't used, nor created any helper functions yet (which error handlers are a part of). Therefore, we have to create a new folder called **/helpers** with the **error.js** file and import the following code

/helpers/error.js

```
/**
 * Custom Objection.js error handler
 */

// Import all Error types
const {
  ValidationError,
  NotFoundError,
  DBError,
  ConstraintViolationError,
  UniqueViolationError,
  NotNullViolationError,
  ForeignKeyViolationError,
  CheckViolationError,
  DataError
} = require("objection");

// Our custom Error Handler
// err and res are arguments coming from Express.js
function errorHandler(err, res) {
  // Handle Error with type ValidationError -> User input data was false
  if (err instanceof ValidationError) {
    switch (err.type) {
      case "ModelValidation":
        res.status(400).send({
          message: err.message,
          type: err.type,
          data: err.data
        });
        break;
      case "RelationExpression":
        res.status(400).send({
          message: err.message,
          type: "RelationExpression",
          data: {}
        });
        break;
```

```
      case "UnallowedRelation":
        res.status(400).send({
          message: err.message,
          type: err.type,
          data: {}
        });
        break;
      case "InvalidGraph":
        res.status(400).send({
          message: err.message,
          type: err.type,
          data: {}
        });
        break;
      default:
        res.status(400).send({
          message: err.message,
          type: "UnknownValidationError",
          data: {}
        });
        break;
    }
  } else if (err instanceof NotFoundError) {
    // Handle Error with type NotFoundError -> manually invoked using "throw new NotFoundError"
    res.status(404).send({
      message: err.message,
      type: "NotFound",
      data: {}
    });
  } else if (err instanceof UniqueViolationError) {
    // Handle Error with type UniqueViolationError -> database threw a constraint error
    res.status(409).send({
      message: err.message,
      type: "UniqueViolation",
      data: {
        columns: err.columns,
        table: err.table,
        constraint: err.constraint
      }
    });
  } else if (err instanceof NotNullViolationError) {
    // Handle Error with type NotNullViolationError -> database threw a constraint error
    res.status(400).send({
      message: err.message,
      type: "NotNullViolation",
      data: {
        column: err.column,
        table: err.table
      }
    });
  } else if (err instanceof ForeignKeyViolationError) {
    // Handle Error with type ForeignKeyViolationError -> database threw a constraint error
    res.status(409).send({
      message: err.message,
      type: "ForeignKeyViolation",
      data: {
        table: err.table,
        constraint: err.constraint
      }
    });
  } else if (err instanceof CheckViolationError) {
    // Handle Error with type ForeignKeyViolationError -> database threw a check constraint error; r
    res.status(400).send({
```

```
      message: err.message,
      type: "CheckViolation",
      data: {
        table: err.table,
        constraint: err.constraint
      }
    });
  } else if (err instanceof DataError) {
    // Handle Error with type DataError -> database threw a invalid data error
    res.status(400).send({
      message: err.message,
      type: "InvalidData",
      data: {}
    });
  } else if (err instanceof DBError) {
    // Handle Error with type DBError -> database threw an error too broad to handle specifically
    res.status(500).send({
      //   message: err.message, // -> modified: It could be possible that the err contains sensitiv
      message: "Unknown Error",
      type: "UnknownDatabaseError",
      data: {}
    });
  } else {
    // Handle every other error generally
    res.status(500).send({
      //   message: err.message, // -> modified: It could be possible that the err contains sensitiv
      message: "Unknown Error",
      type: "UnknownError",
      data: {}
    });
  }
}

module.exports = {
  errorHandler
};
```

In the code above, we check for every possible Objection.js error type, set the response status accordingly (e.g., 404 when no data was found during a find query) and return the message, the error type, and additional data (if possible/ required).

> **Note** Our errorHandler function is another middleware, just like Knex.js, Objection.js. Even Express.js is a kind of middleware. Middlewares are often used in the context of Express.js framework and are a fundamental concept for Node.js. In a nutshell, it's a function that has access to the request, response, and error objects of your application and does additional stuff on them.

Next step is to include our newly written errorHandler middleware to our project. For this, we have to update our **server.js** file

```
1  const express = require("express");
2  const app = express();
3
4  app.use(express.urlencoded({ extended: false }));
5  app.use(express.json());
6
7  const apiRoutes = require("./routes/api");
8
9  const Knex = require("knex");
10 const knexFile = require("./knexfile.js");
11 const knex = Knex(knexFile.development);
12
13 const { Model } = require("objection");
14
15 Model.knex(knex);
16
17 app.use("/api", apiRoutes);
18
19 // Include error handlers - must be the last one among other middleware or routes to function proper
20 const { errorHandler } = require("./helpers/error.js");
21 app.use((err, req, res, next) => {
22   errorHandler(err, res);
23 });
24
25 const server = app.listen(8080, error => {
26   if (error) {
27     console.log("Error running Express");
28   }
29   console.log("Server is running on port", server.address().port);
30 });
```

The error-handling middleware in line 22 must be the last among other middleware and routes for it to function correctly.

> **Note** Error-handling middleware always takes **four** arguments. You must provide four arguments to identify it as an error-handling middleware function. Even if you don't need to use the next object, you must specify it to maintain the signature. Otherwise, the next object will be interpreted as regular middleware and will fail to handle errors. Look below for an explanation of `next`

Use error handler

Now that we created and included our error handling middleware, we are ready to handle errors correctly

/routes/api/users.js

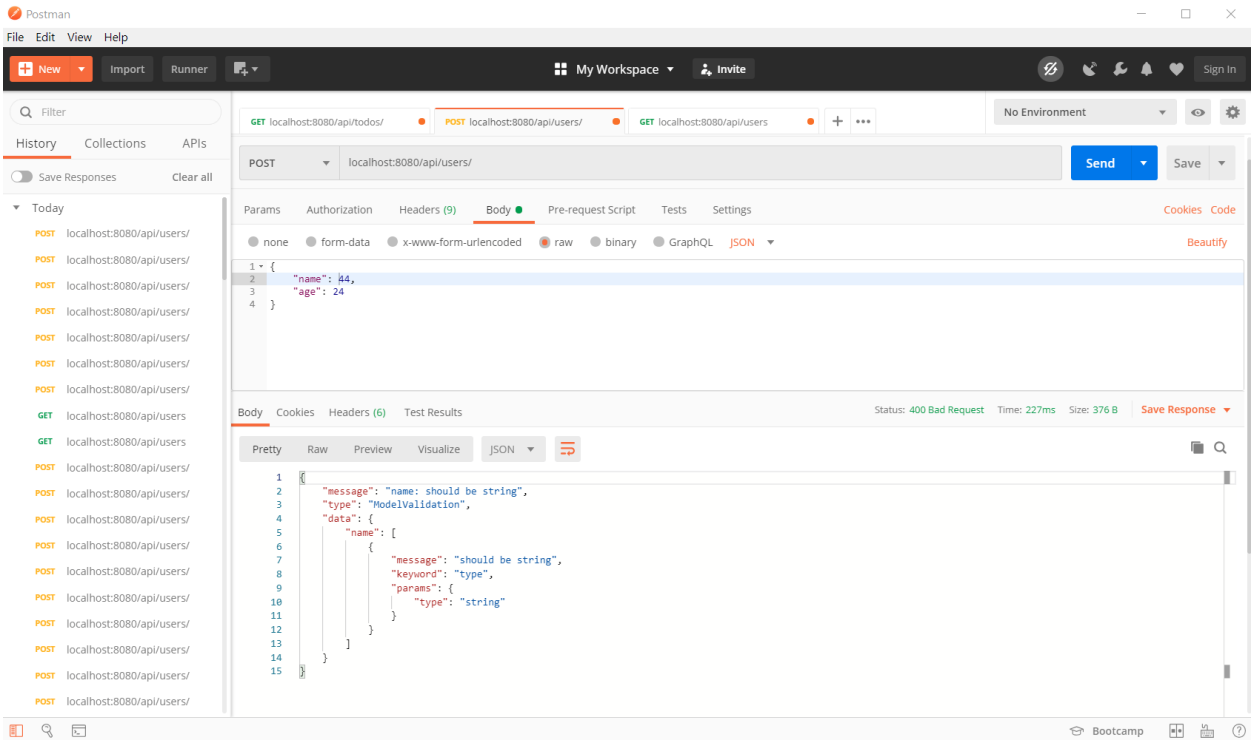
```
// Create endpoint of all users
router.get("/", async (req, res, next) => {
  try {
    const users = await User.query();
    // throw a not found error
    if (!users.length === 0) {
      throw new NotFoundError(users);
    }
    res.json(users);
  } catch (err) {
    next(err);
  }
});

router.post("/", async (req, res, next) => {
  const { name, age } = req.body;

  try {
    const user = await User.query().insert({ name, age });
    res.json(user);
  } catch (err) {
    next(err);
  }
});

// Export to api.js
module.exports = router;
```

In the code above:

- inside the `app.get`, we have added an if statement, which checks if our database query returns rows. If no rows are returned, we do a `throw new NotFoundError(users);`. Throwing an error inside a try block invokes the catch block.
- `next` is an Express.js native function and means to continue to the next matching route and pass our `err` to our error handling middleware (`errorHandler` function). **More information on the next function can be found here**
- inside the `app.post`, we don't do any custom `res.status(400).json(err.data);` but let our errorHandler function catch the error type and send a response accordingly.

Testing the error handler

Example validation error We can invoke a *validation error* inside the `app.post` by simply sending an integer as the name value (without quotation marks).



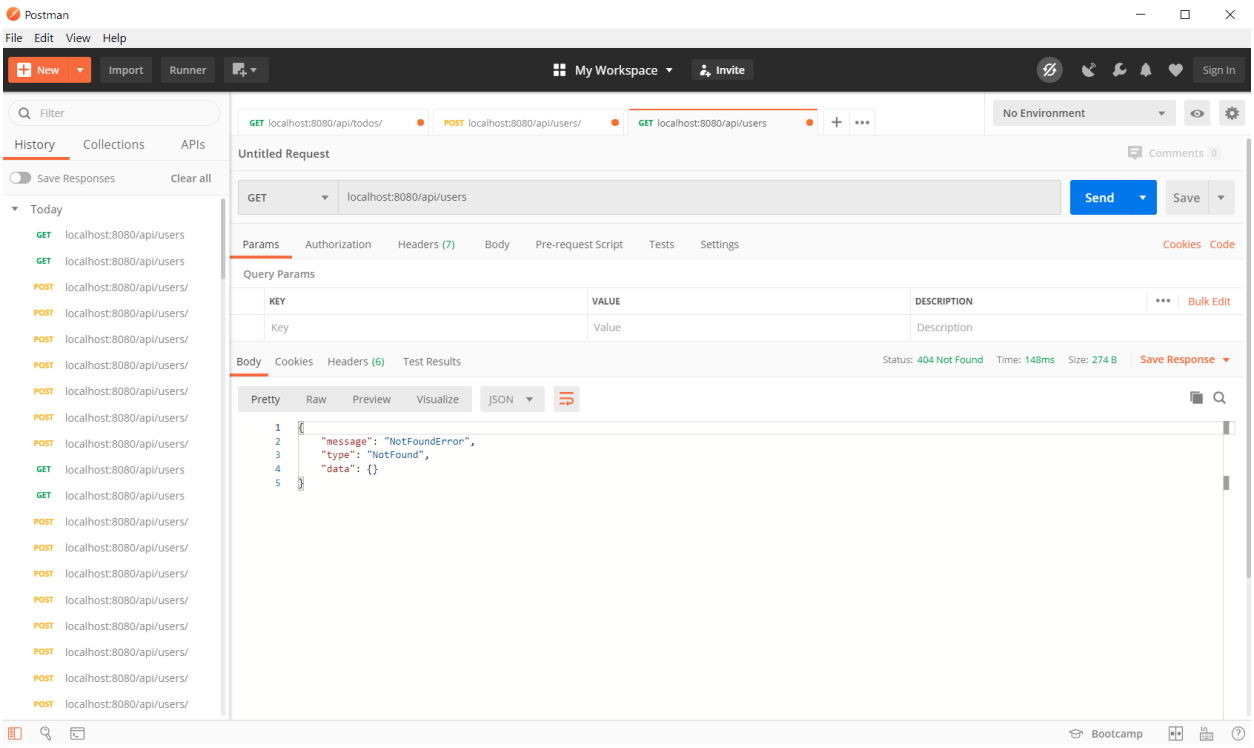
In the above image, we can see the returned error from our error handler function.

Example nothing found error We can invoke a *not found error* by slightly modifying our `app.get` block not to return any rows.

```
// Create endpoint of all users
router.get("/", async (req, res, next) => {
  try {
    const users = await User.query().where("name","Penguin Giraf");
    // throw a not found error
    if (!users.length === 0) {
      throw new NotFoundError(users);
    }
    res.json(users);
  } catch (err) {
    next(err);
  }
});
```

We are definitely sure that no name “Penguin Giraf” exists in the database, i.e. the query returns an empty array.

Requesting the URL in Postman displays the error message below



Now our project as an improved and unified error handling, honoring a particular structure.

About

This booklet is based on:

- Objection.js
- Knex.js - A SQL Query Builder for Javascript
- Knex cheatsheet
- Building and Running SQL Queries with Knex.js - DEV
- A definitive guide to SQL in NodeJS with Objection.js + Knex — Part 1 - DEV
- Express + Knex + Objection = Painless API with DB - ITNEXT
- Knex.js tutorial - programming databases with Knex.js
- Database Migrations with Knex
- Express with Knex - Front-End Engineering Curriculum - Turing School of Software and Design
- JavaScript Frameworks for Modern Web Development: The Essential Frameworks, Libraries, and Tools to Learn Right Now
- A Step Towards Simplified Querying in NodeJS - Velotio Perspectives - Medium
- Hacker Noon
- YouTube Central Error Handling in Express - DEV Community node.js - What is the parameter “next” used for in Express? - Stack Overflow Throw notFoundError by default · Issue #874 · Vincit/objection.js · GitHub Central Error Handling in Express

Appendix: Databases

Knex.Js officially supports the following databases:

- PostgreSQL
- MariaDB
- MySQL
- Oracle
- SQLite
- Amazon Redshift

Here, we are comparing MySQL/MariaDB (they are nearly the same), SQLite, and PostgreSQL.

Relational Database comparison

SQLite

The SQLite project's website describes it as a "serverless" database. This means that any process that accesses the database are reads and writes directly to the disk file, without the need for a database engine, compared to MySQL and PostgreSQL. This simplifies SQLite's setup process, since it eliminates any need to configure a server process. Likewise, there's no configuration necessary for programs that will use the SQLite database: all they need is access to the disk. This serverless architecture enables the database to be cross-platform compatible. SQLite is used for Android & iOS Mobile phones but also for McAfee antivirus programs, Skype program for Mac OSX and Windows, Firefox…

Advantages

SQLite is a good choice for low-to-medium traffic websites (~100k requests a day) due to its small footprint (600kb of space), user setup friendly, and portability.

Disadvantages

Although multiple processes can **access** and **query** an SQLite database at the same time, only one process can make **changes** to the database at any given time. This puts SQLite at a disadvantage to MySQL and PostgreSQL when working with lots of data or high write volumes.

MySQL

MySQL has been the most popular open-source RDBMS for websites in the past years, which powers many of the world's largest websites and applications, including Twitter, Facebook, Netflix, and Spotify. Getting started with MySQL is relatively straightforward, thanks in large part to its exhaustive documentation and a large community of developers, as well as the abundance of MySQL-related resources online.

MySQL was designed for speed and reliability at the expense of fully complying with the SQL standards. The MySQL developers continually work towards closer adherence to standard SQL, but it still lags behind other SQL implementations.

In 2008, *Sun Microsystems* acquired MySQL. 2 years later, MariaDB, a fork (a copy) of MySQL emerged because developers were not happy with the for-profit policy of *Sun Microsystems*. Both continue to work the same way. *Opinionated statement: MariaDB can be seen as the real "2008 MySQL" successor.*

Advantages

MySQL is a good choice for all kind of websites due to its popularity and ease of use, improved security (compared to SQLite), speed (by choosing not to implement certain SQL features) and the ability of database replication which is the practice of sharing information across two or more hosts (multiple databases on multiple computers) to help improve reliability, availability, and fault-tolerance

Disadvantages

MySQL, however, is known with certain SQL limitations, such as lacking support for `FULL JOIN` clauses. Also, if your application has lots of users writing data to it at once, another RDBMS like PostgreSQL might be a better choice of database (e.g., a chat application if it needs to be an SQL database).

PostgreSQL

PostgreSQL, also known as Postgres, bills itself as "the most advanced open-source relational database in the world." It was created to be highly extensible and SQL standards-compliant.

Postgres is capable of efficiently handling multiple tasks at the same time, a characteristic known as concurrency, **but** don't let this blind you as if PostgreSQL is more performant than MySQL (have a look at advantages/disadvantages). It achieves this without reading locks

thanks to its implementation of Multiversion Concurrency Control (MVCC), which ensures the atomicity, consistency, isolation, and durability of its transactions, also known as ACID compliance (FYI: SQLite and MySQL are also ACID compliant).

Advantages

PostgreSQL is a good choice for all kinds of websites for the same reasons as MySQL but additionally handles multi-user environments much better and has built-in support for JSON and Array data types.

Disadvantages

When compared to MySQL, PostgreSQL is more power-hungry, as it takes up 10MB RAM for each client connection. This model can take up much memory as concurrent client connection goes when compared to the thread-per-connection model of MySQL. Another big disadvantage can be seen during frequent UPDATEs, where due to no support for clustered indexes, PostgreSQL can have a huge adverse impact on performance compared to MySQL databases.

Which to pick?

Thanks to Knex.Js query methods, we don't have to think that much about database-specific syntaxes. We can switch between SQL databases in no time without having to rewrite code.

But the database behind a web application matters. SQLite should be used for single-user websites. Later on, when you have to build production-ready websites with multiple functionalities and users, then you should switch to MySQL or PostgreSQL.

MySQL and PostgreSQL are both excellent choices, and you should pick the one you're most comfortable with and used to. Many online platforms discuss which is better - but both are performant, one over the other depending on the operation.

Personally, if you are new to databases, go with MySQL since it's very versatile and beginner-friendly. If you've used MySQL before and have the time and want to use all features of Knex.js and Objection.js, then get started with PostgreSQL to get some additional relational database experience.

PDF Book from Markdown

Converts a bunch of markdown files made for a programming book into a pdf. Uses prism syntax highlighting and a modified version of github stylesheet.

Requirements:

You need to install:

- https://wkhtmltopdf.org/
- https://pandoc.org/installing.html
- LaTeX (See https://tug.org/mactex/ on OS X, https://miktex.org/ on Windows, or install the texlive package on Linux.
- Python

How to use

1. Install pdfkit `pip install pdfkit`
2. Place all your files in the root directory
3. Edit the metadata.yaml file
4. Make sure they are alphabetically sorted (0-9, then A - z)
5. Run `python create_book.py`
6. Enjoy