Fundamentals of Web Development

Third Edition by Randy Connolly and Ricardo Hoar

Chapter 7

Working with Databases

Part 1

Technical notes

In this chapter we can use a local MySQL server.

Or use an online free MySQL service:

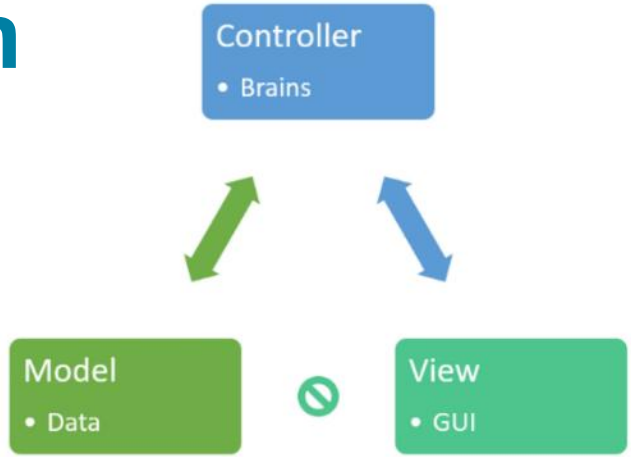
https://aiven.io/

Complex server application

- When we have many API endpoints we start having a complex file that will become difficult to maintain and debug.
- The solution to this problem is to apply a very popular pattern that is recognized to be the best solution to this issue: the MVC architectural pattern.

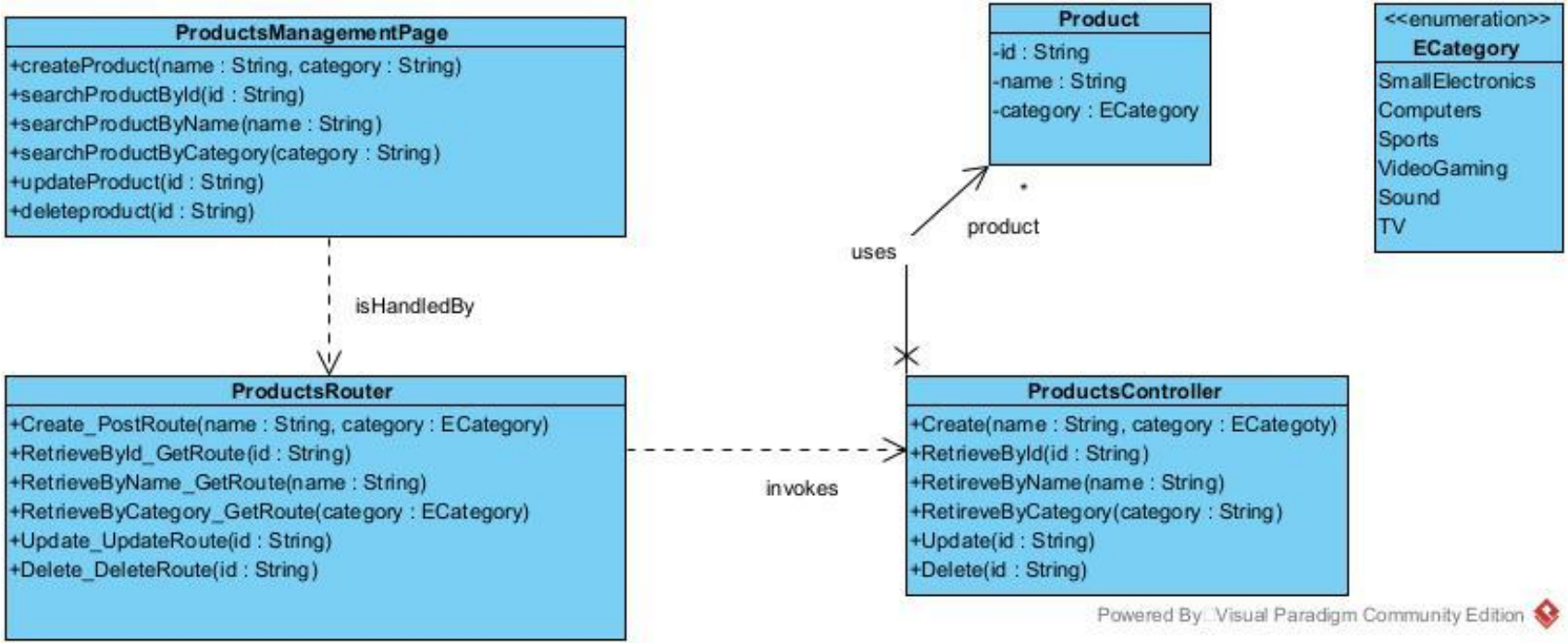
```
connect_mongo > JS app02.js > ...
app.get('/all-employees', (request, response) => {
 Employee.find()
   .then( (result) => {
     response.send(result);
   })
   .catch( (err) => {
     console.log(err)
   });
});

app.get('/find-employee/id/:id', (request, response) => {
 Employee.findById(request.params.id)
   .then( (result) => {
     response.send(result);
   })
   .catch( (err) => {
     console.log(err);
   });
});

app.get('/find-employee/position/:position', (request, response) => {
 Employee.find({ positions: request.params.position })
 .sort({ age : -1})
 .select('name positions')
   .then( (result) => {
     response.send(result);
   })
   .catch( (err) => {
     console.log(err);
   });
});

app.get('/find-employee2/position/:position', (request, response) => {
 Employee.find({ positions: request.params.position })
   .sort( { age: -1 } )
   .select( 'name positions' )
   .then( (result) => {
     response.send(result);
   })
   .catch( (err) => {
     console.log(err);
   });
```

MVC architectural pattern



- In the MVC pattern, the architecture of the code is decomposed as follows:
  - The model refers to the parts of your application that store (database) and manipulate the data (save, update, delete, retrieve operations).
  - The View is the code responsible for presenting the data to the user (web pages, mobile interface, GUI).
  - The Controller is responsible to link the model and the view. It selects the view that will present the data. It select the model operation to execute and eventually returns the results to the view (here the controller is the express router + business logic).
- MVC is an application of (probably) the most important principle in software engineering: Separation of Concerns. Which has many advantages such as: code is more maintainable, easier to provide different views for the same data (add a mobile app to an existing web app, etc.)

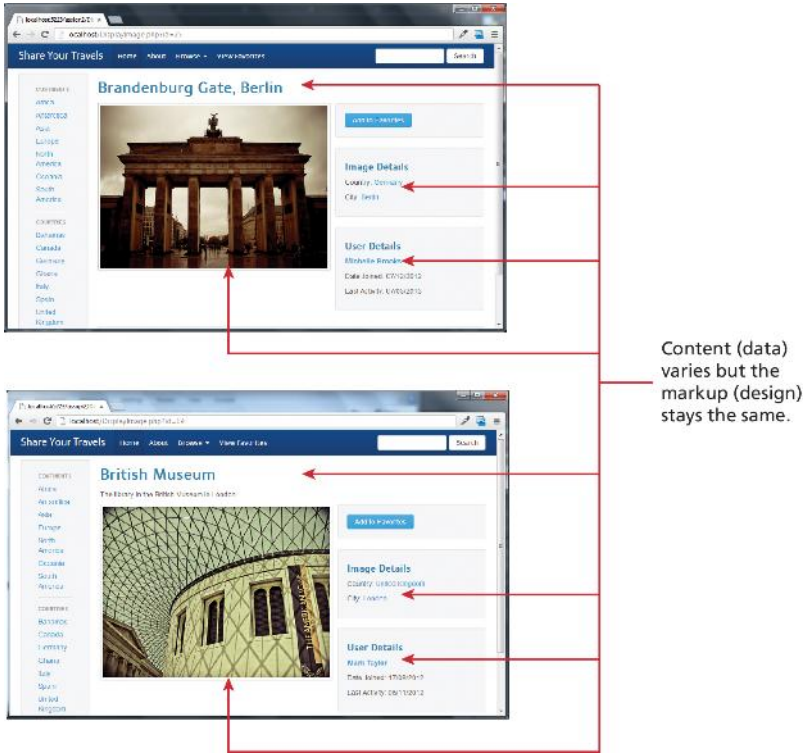
Example of MVC



The Role of Databases in Web Development

Databases provide a way to implement an important software design principle: separate static (doesn't change) content from dynamic content.

On the web the visual appearance (i.e., the HTML and CSS) is static, while the data content is dynamic (changes).



Databases and Web Development

To work with data, we can use different **relational** DBMS like **SQLite** or **MySQL**, **PostgreSQL, Oracle Database**, **IBM DB2**, and **Microsoft SQL Server**.

In addition to relational database systems, there are **non-relational** models for database systems. These systems are usually categorized with the term **NoSQL** and includes systems such as **Cassandra, Firebase** and **MongoDB.**

NoSQL Databases

NoSQL (which stands for Not-only-SQL) is a category of database software that describes a style of databases that doesn't use the relational table model of normal SQL databases.

NoSQL databases rely on a different set of ideas for data modeling that put **fast retrieval** ahead of other considerations like **consistency**.

Systems like DynamoDB, Firebase, and MongoDB now power thousands of sites including companies like eBay, Forbes, mckinsey, ericsson, and others.

DB engines ranking:
https://db-engines.com/en/ranking

Rank			DBMS	Database Model	Score		
Oct 2024	Sep 2024	Oct 2023			Oct 2024	Sep 2024	Oct 2023
1.	1.	1.	Oracle	Relational, Multi-model	1309.45	+22.85	+48.03
2.	2.	2.	MySQL	Relational, Multi-model	1022.76	-6.73	-110.56
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	802.09	-5.67	-94.79
4.	4.	4.	PostgreSQL	Relational, Multi-model	652.16	+7.80	+13.34
5.	5.	5.	MongoDB	Document, Multi-model	405.21	-5.02	-26.21
6.	6.	6.	Redis	Key-value, Multi-model	149.63	+0.20	-13.33
7.	7.	↑ 11.	Snowflake	Relational	140.60	+6.88	+17.36
8.	8.	↓ 7.	Elasticsearch	Multi-model	131.85	+3.06	-5.30
9.	9.	↓ 8.	IBM Db2	Relational, Multi-model	122.77	-0.28	-12.10
10.	10.	↓ 9.	SQLite	Relational	101.91	-1.43	-23.23

Why (and Why Not) Choose NoSQL?

NoSQL systems handle huge datasets better than relational systems. But they aren't the best answer for all scenarios. SQL databases use schemas for a very good reason: they ensure **data integrity and data consistency**.

Definitions:

Data integrity: The guarantee of all data constraints (primary and foreign keys, data types, etc…).

Data consistency: The guarantee that database constraints are not violated when executing transactions.

Key-Value Stores

Key-value stores alone are quite straightforward in that every value, whether an integer, string, or other data structure, has an associated key (i.e., they are analogous to Maps)

Here, every value has a key. This allows fast retrieval through means such as a hash function, so no need for indexes on multiple fields as is the case with SQL.

Examples: DBM, Berkeley DB.

Key	Value
Customer.Name	"Randy"
Price	200.00
ShippingAddress	"4825 Mount Royal Gate SW"
Countries	"Canada","France","Germany","United States"

Document Store

Document Stores (also called document-oriented databases) associate keys with values, but unlike key-value stores, they call that value a document.

- A document can be a binary file like a .doc or .pdf or a semi-structured XML or JSON document.
- Most NoSQL systems are of this type. MongoDB (ranking 5, 1 in no-SQL), AWS DynamoDB (16), Google FireBase (39), and Cloud Datastore (79) are popular examples.

MongoDB

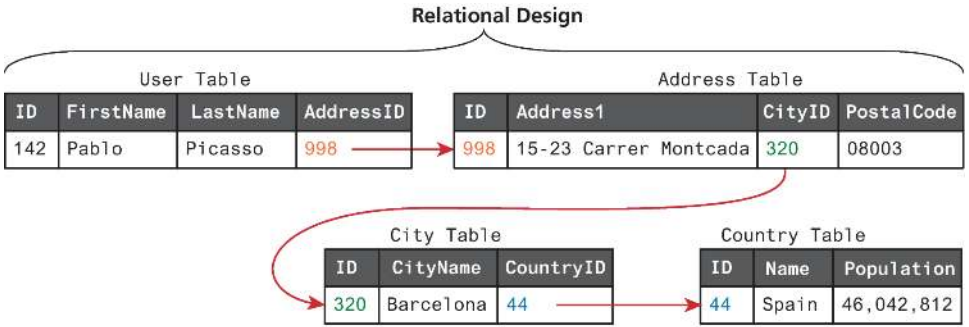
MongoDB is an open-source, NoSQL, document-oriented database. It can be used with any backend technology (JEE, PHP, etc.), it is much more commonly used with Node

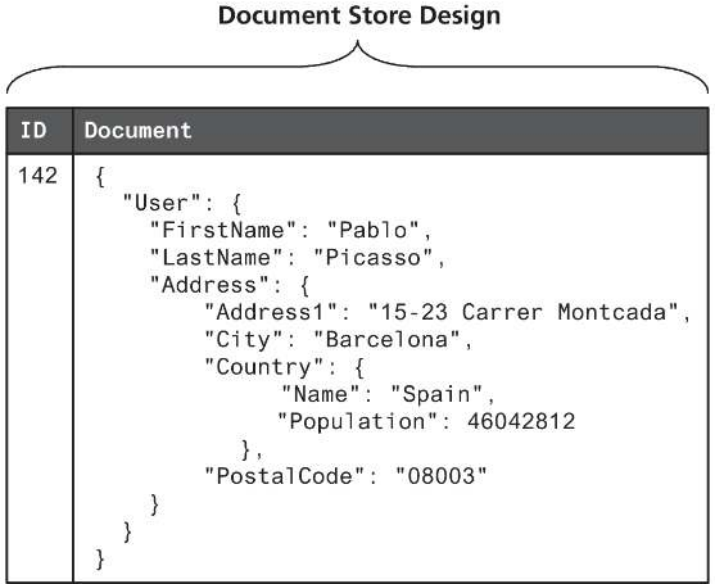
You simply package your data as a JSON object, give it to MongoDB, and it stores this object or document as a binary JavaScript object (BSON).

MongoDB does not support: ~~transactions (from 4.0)~~, joins (it uses nested documents instead).

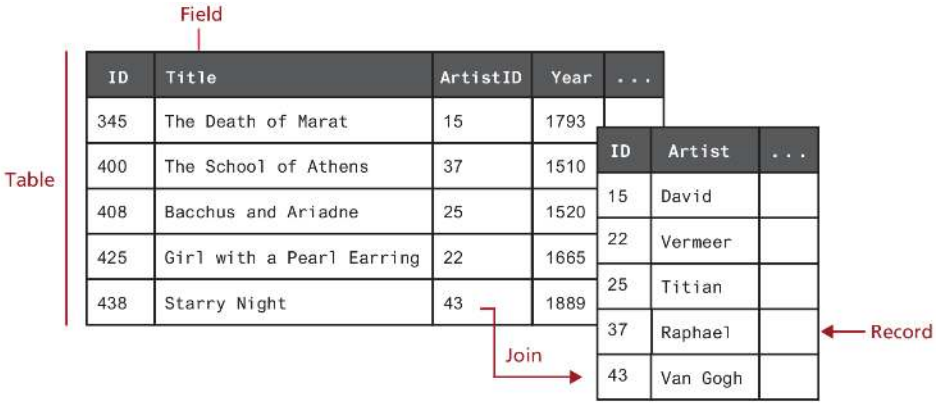
The ability to run on multiple servers means MongoDB can handle large datasets: **replication => redundancy + high availability**

Relational data versus document store data

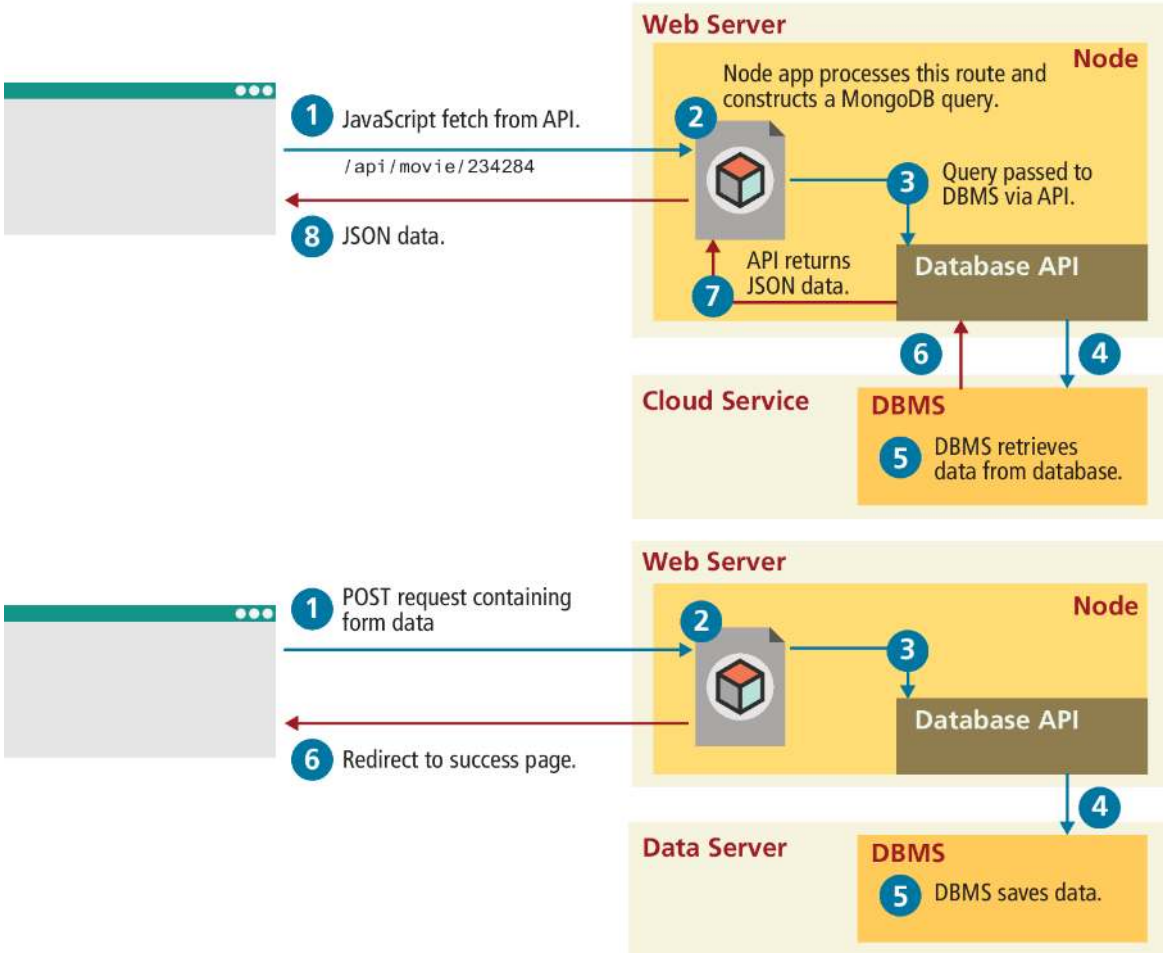




Comparing relational databases (MySQL) to document data model (MongoDB)



How websites use databases

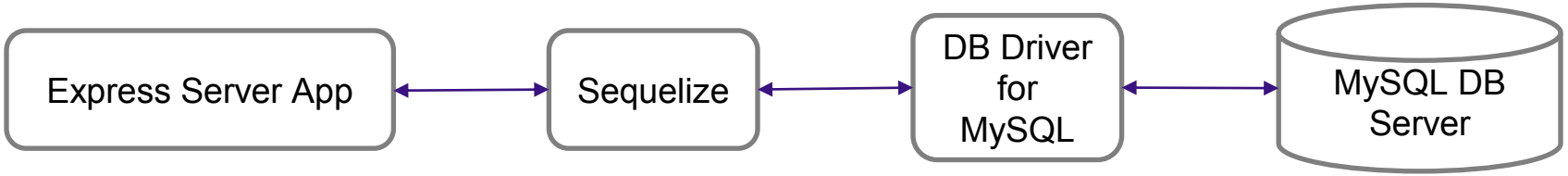


Object Relational Mapping libraries

- Object Relational Mapping libraries (ORM) allow the use of an API to query databases without using database-specific queries.
- They offer a layer of abstraction that makes your code independent of the subsequent database system used.
- Advantages:
  - Reduce security risks (SQL or non-SQL injections).
  - Simplify database communication.
  - Allow changing the database system without changing your code.

The *Sequelize* ORM library

- ***Sequelize*** is a popular ORM that supports MySQL, DB2, SQL Server and more.
- To use ***Sequelize***, we need to also install a database driver (API used by *Sequelize* to access a given database system such as MySQL)



- To install Sequelize and the mysql driver:

> *npm install sequelize mysql2*

Configure the database connection

```
// import the sequelize library
const Sequelize = require('sequelize');

// create a configured Sequelize object
const sequelize = new Sequelize(
  'se371db',   //database name
  'se371',     // database user
  'se371pwd',  // user password
  {
    // specify the database server used
    dialect: 'mysql',

    // using a local db so host is 'localhost'
    host: 'localhost'
  }
);
```

1- import sequelize library

2- create a sequelize object instance with our configuration data

Source code: chapter07/01

Connect to the database

```
const connectToDB = async () => {
  try {
    await sequelize.authenticate();
    console.log(`Successfully connected to database server...`);
  } catch(error) {
    console.log(error);
  }
}
```

Note: here the sequelize object will connect to the database using the given configuration.

Create the data model

```
const db = require("../config/database");
const { DataTypes } = require('sequelize');

const Employee = db.sequelize.define('Employee', {
  id: {
    type: DataTypes.INTEGER,
    primaryKey: true
  },
  name: {
    type: DataTypes.STRING,
    allowNull: false,
    validate: {
      max: 100
    }
  },
  position: {
    type: DataTypes.STRING,
    defaultValue: "Developer"
  }
});

db.sequelize.sync({ alter: true });
```

Import the database config connection objects
Import DataTypes from the sequelize library

Create a model named 'Employee'
Define a first property called 'id' of type 'Integer' that is a primary key. (the table in the DB will be automatically called employees)

Create a property named 'name' of type String that cannot be null with a max length limit of 100 characters

Create a property named 'position' of type 'String' and a default value of 'Developer'

Apply this model to the actual database. If the table Employees does not exist, it will be created. If it exists, any changes will be applied to the database table structure.

Which HTTP method to use: What is CRUD?

- In HTTP we have different request methods that should be used for the corresponding database operation:

Database operation	HTTP method	express function
Create	POST is used to create new data in the backend (in the database for example).	app.post()
Read	By default, GET is used, GET is used to get a resource from the server (webpage, css file, image, data, etc.)	app.get()
Update	PUT is used to update existing data in the backend.	app.put()
Delete	DELETE is used to delete data	app.delete()

Design the API endpoints

Page routes

Web API routes

Operation	Route	HTTP Method
To open the home page To open the form page	/ **/**employees/	GET
To retrieve articles	/**api/**employees**/v1/** /**api/**employees/v1/id/:id /**api/**employees/v1/position/:position	GET
To create a new article	/**api/**employees/v1/ // using a form /**api/**employees/v1/id/:id/name/:name/position/:position	POST
To update an article	/**api/**employees/v1/id/:id/name/:name/position/:position	PUT
To delete an article	/**api/**employees/v1/id/:id	DELETE

These two routes return web pages

These routes return json data only

- By using **/api/employees/v1/** with all operations related to employees, we can create a router that handles only employees related operations -> better modularity in our code.

Create a new element in the DB

```
// Create
app.post('/api/employees/v1/', async (request, response) => {
  const { id, name, position } = request.body;
  const newEMployee = Employee.build({
      "id": id, "name": name, "position": position
  });

  try {
    await newEmployee.save();
    response.status(201).json(newEmployee);
  } catch(error) {
    response.json(error);
  }
})
```

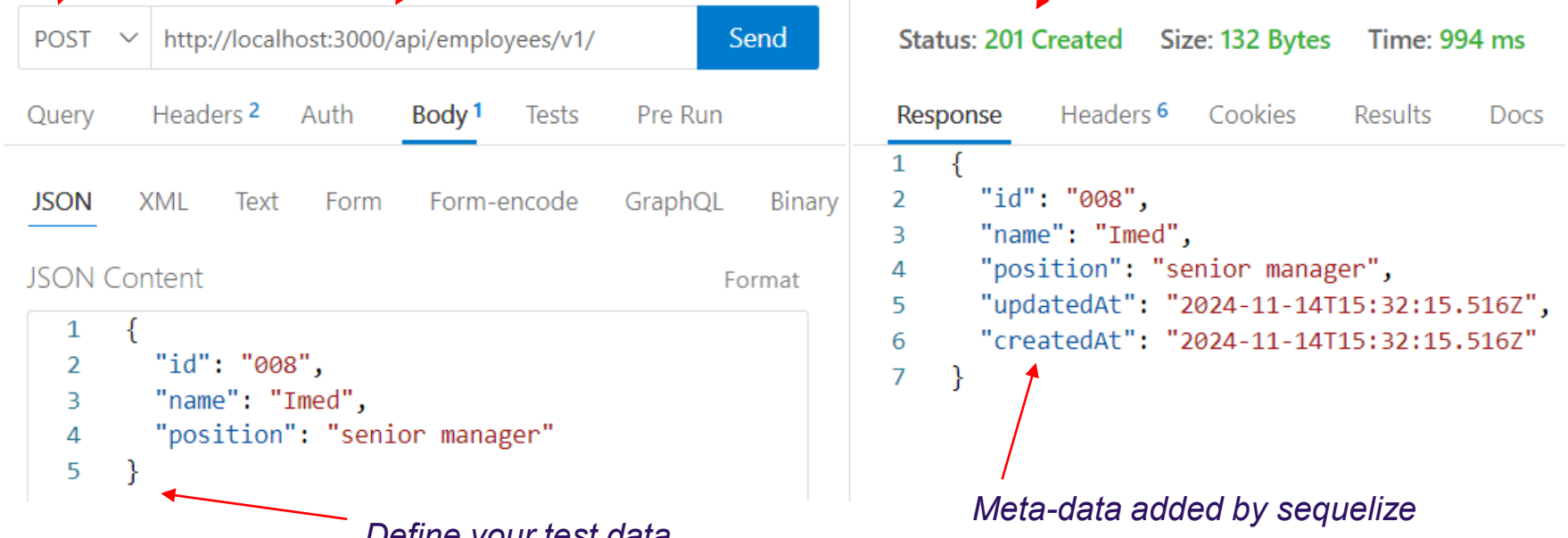
Source code: chapter07/01/app02.js

Calling the *add employee* api endpoint

Change the method to a POST request

Specify the URL of the create employee operation

The client received the 201 message meaning: "resource created"

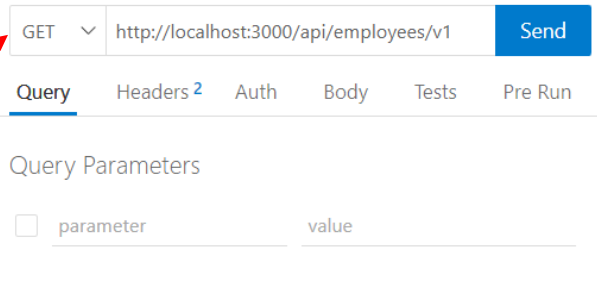


Define your test data as a JSON object

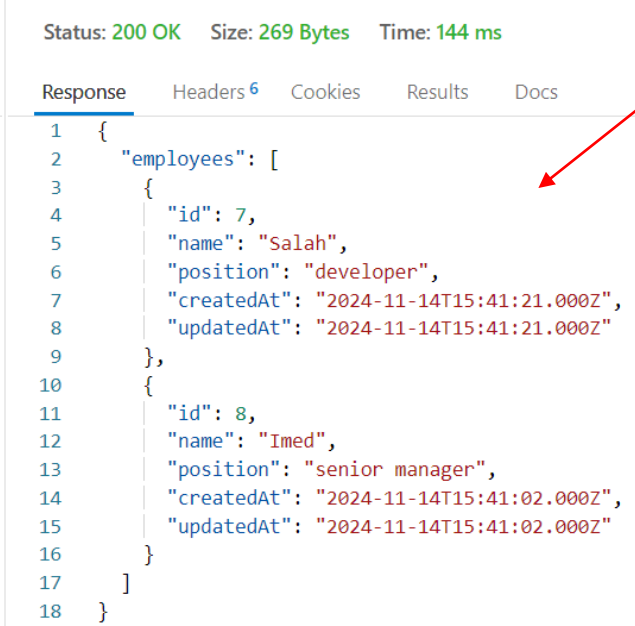
Meta-data added by sequelize

Retrieving data form the DB

```
// Select all
router.get('/api/employees/v1/', async (request, response) => {
  const employees = await Employee.findAll();
  response.status(200).json({ employees: employees });
})
```



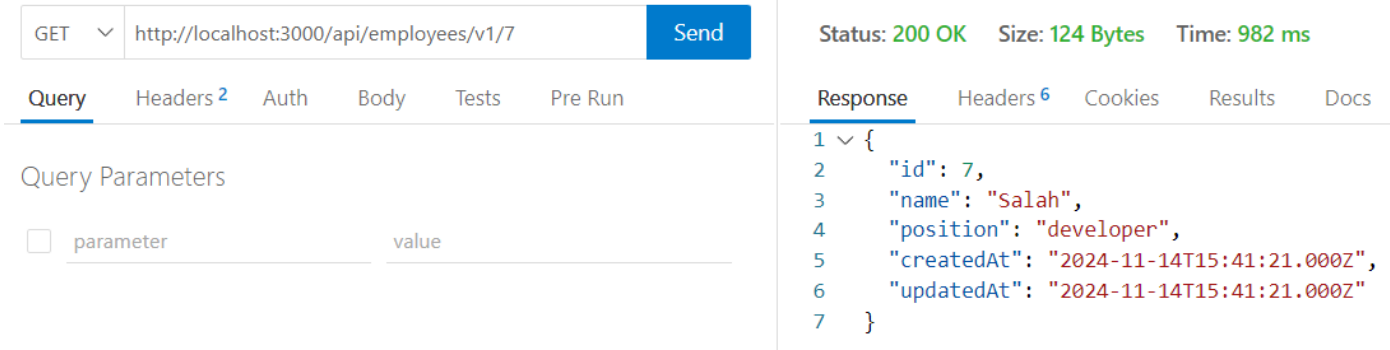
Change the method to a GET request



The result is a JSON array

Search for employees by id

```
// Search by id
router.get('/employees/v1/:id', async (request, response) => {
  try {
     const employee = await Employee.findOne({
       where: {
         id: request.params.id
       }
     });
     response.status(200).json(employee);
   } catch (error) {
    response.json(error);
  }
})
```



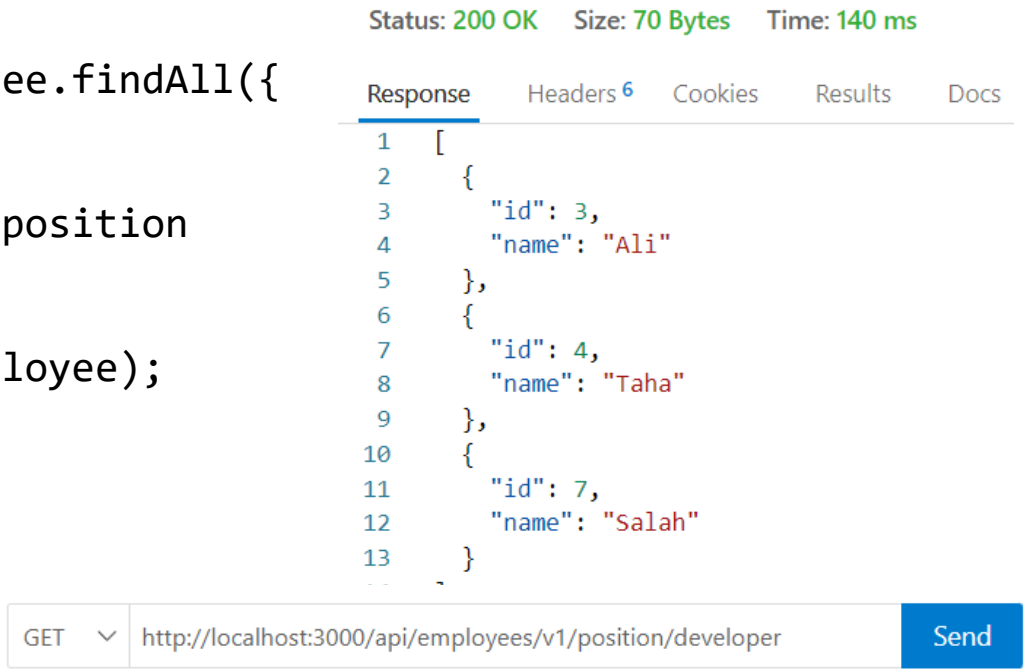
Search for employees by criteria

```
// Search by position
router.get(‘/employees/v1/position/:position', async (request, response) =>
{
  try {
    const employee = await Employee.findAll({
      where: {
        position: request.params.position
      }
    });
    response.status(200).json(employee);
  } catch (error) {
    response.json(error);
  }
})
```

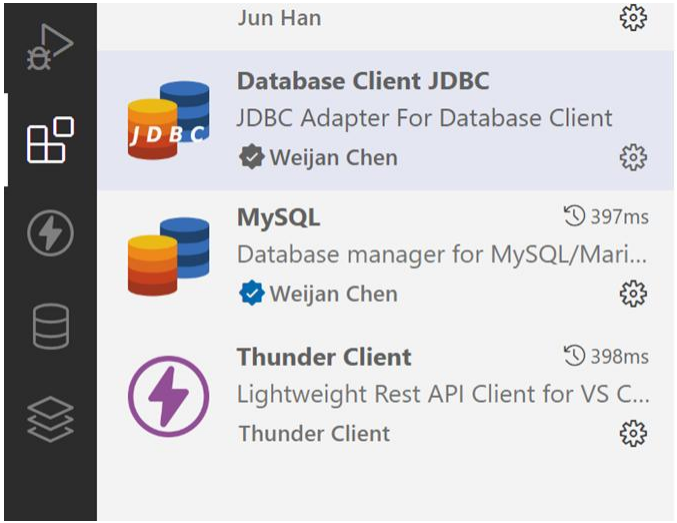
Use projections on the result

Doing a projection means selecting the properties that you want to get from the database (For example, some properties should stay secret).

```
router.get('/employees/v1/position/:position', async (request, response) =>
{
  try {
    const employee = await Employee.findAll({
      attributes: ['id', 'name'],
      where: {
        position: request.params.position
      }
    });
    response.status(200).json(employee);
  } catch (error) {
    response.json(error);
  }
})
```



Install following Extensions



Cloud Database

aiven
CONSOLE

Fundamentals of Web Development

Third Edition by Randy Connolly and Ricardo Hoar

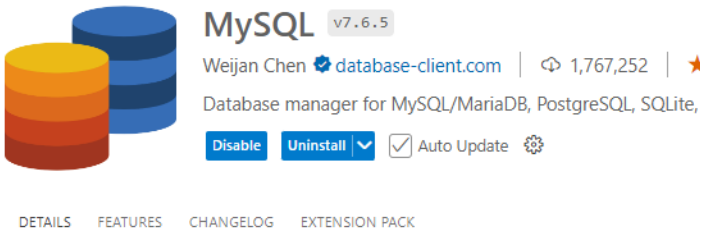
Chapter 7

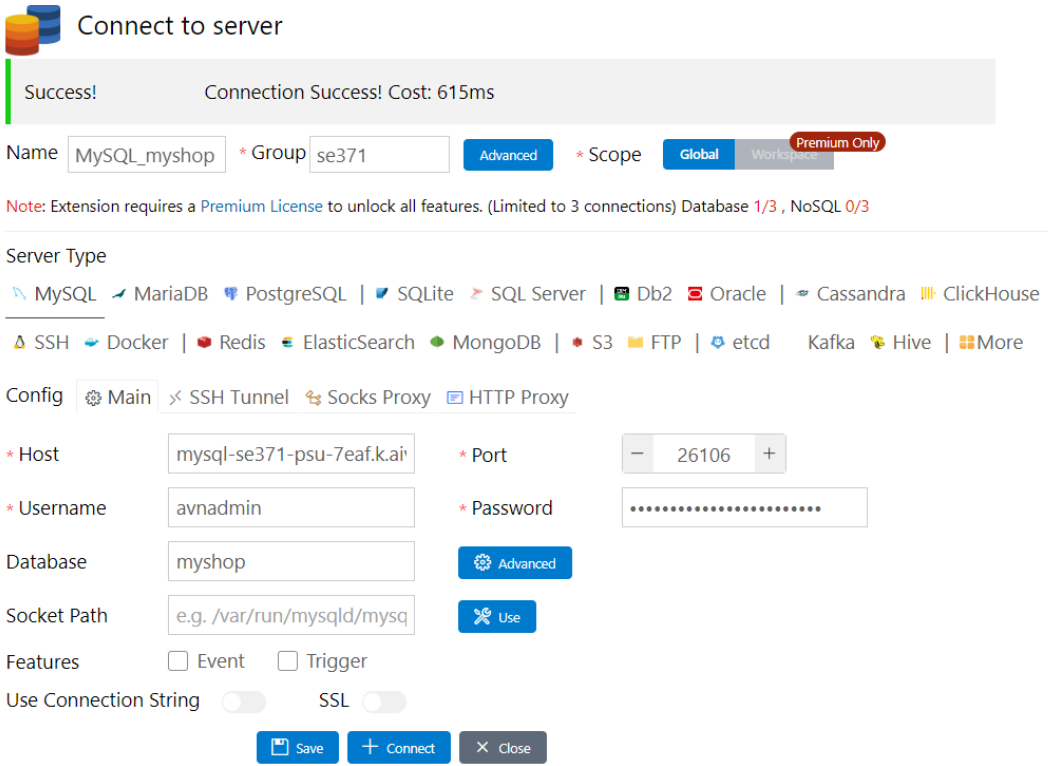
Working with Databases

Part 2: More on queries

Connecting to your server from VSCode

Install this VSCode extension then create a database connection as shown in this figure.
To generate mock data execute “generate mock data”.

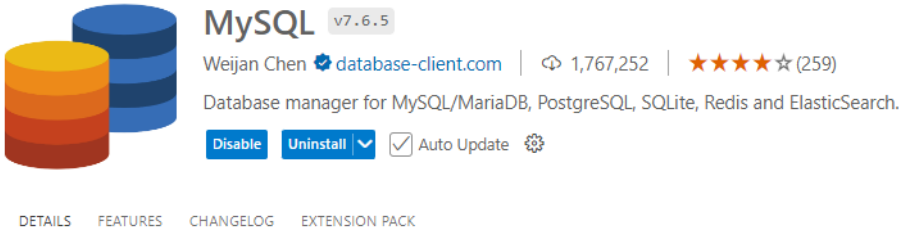




Generating Mock data

```
{
    "table": "Employees",
    "mockStartIndex": "auto",
    "mockCount": 100,
    "mock": {
        "id": {
            "type": "int",
            "value": "#mockIndex"
        },
        "name": {
            "type": "varchar(255)",
            "value": "@name"
        },
        "position": {
            "type": "varchar(255)",
            "value": "@pick(['Manager', 'Developer', 'Tester', 'Designer'])"
        },
```

Install this VSCode extension then execute “generate mock data”



Generating Mock data (2)

```
        "city": {
            "type": "varchar(255)",
            "value": "@pick(['Riyadh', 'Jeddah', 'Jizan', 'Dammam'])"
        },
        "age": {
            "type": "int",
            "value": "@integer(20,65)"
        },
        "createdAt": {
            "type": "datetime",
            "value": "@datetime()"
        },
        "updatedAt": {
            "type": "datetime",
            "value": "@datetime()"
        }
    }
}
```

Search results | Export | Cost: 134ms | 1 2 | Total 101

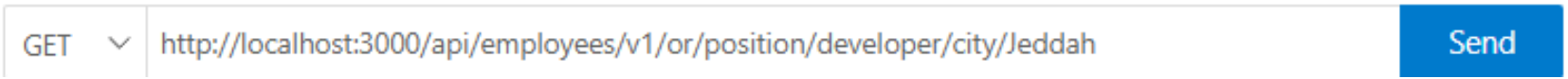
id int	name varchar(255)	position varchar(255)	city varchar(255)	age int	createdAt datetime	updatedAt datetime
Filter	Filter	Filter	Filter	Filter	Filter	Filter
6	Ali	developer	Jeddah	32	2024-11-14 18:05:0	2024-11-14 18:05:0
7	Shirley Martin	Tester	Dammam	30	1996-09-10 18:03:0	1987-12-31 14:21:4
8	Anthony Thomas	Developer	Jizan	33	2001-10-30 07:50:0	2024-03-24 18:04:4
9	Edward White	Tester	Dammam	44	1972-09-06 21:02:1	1987-10-17 18:48:2
10	Margaret Thomp	Designer	Jeddah	22	2000-12-05 03:59:5	2002-08-04 07:55:3
11	Sharon Clark	Designer	Riyadh	49	1974-02-02 22:42:4	2014-11-14 16:26:1
12	Linda Allen	Manager	Tabuk	48	2007-05-13 22:32:4	1986-03-15 02:06:2
13	Jessica Anderson	Designer	Riyadh	35	1980-06-15 03:19:4	2023-09-15 09:31:0
14	Charles Miller	Developer	Dammam	47	2014-05-20 15:11:1	1973-09-23 10:13:0
15	Michael Rodrigue	Developer	Jeddah	59	1975-08-11 18:07:3	2010-01-02 16:33:1
16	Mary Robinson	Marketing	Tabuk	46	2013-11-23 18:00:3	2010-07-22 03:34:0
17	Brenda Wilson	Developer	Tabuk	56	2010-09-12 08:15:2	1978-07-26 11:58:3
18	John Rodriguez	Developer	Dammam	23	1973-09-17 00:10:1	2019-10-06 09:09:3
19	Sandra Clark	Developer	Jeddah	34	1980-03-24 10:12:0	2024-01-28 08:56:0
20	Linda Garcia	Developer	Jeddah	22	1992-10-29 18:44:3	1978-02-01 14:48:2
21	Larry Gonzalez	Tester	Jizan	33	2003-12-12 06:16:1	1976-08-17 19:17:1

More on WHERE clause: AND

```
// Search by position and city
const { Op } = require('sequelize’); // Import Op obj, it defines operations

router.get('/employees/v1/position/:position/city/:city', async (request, response) =>
{
  try {
   const employee = await Employee.findAll({
     where: {
      [Op.and]: [{ city: request.params.city }, { position: request.params.position }]
     }
   });
   response.status(200).json(employee);
  } catch (error) {     response.json(error);   }
})
```

More on WHERE clause: OR

```
// Search by position or city
router.get('/employees/v1/or/position/:position/city/:city', async (request, response)
=> {
  try {
   const employee = await Employee.findAll({
     where: {
      [Op.or]: [{ city: request.params.city }, { position: request.params.position }]
     }
   });
   response.status(200).json(employee);
  } catch (error) {     response.json(error);   }
})
```



More on WHERE clause: Delete all

```
// Delete all employees from a given city
router.delete('/employees/v1/city/:city', async (request, response) => {
  try {
    // delete record in a given city
    const employee = await Employee.destroy({
      where: {
        city: request.params.city
      }
    });

    // send response
    response.status(200).json(employee);
  } catch (error) {
    response.json(error);
  }
})
```

More on WHERE clause: Not Equal

```
// Find all employees from outside a given city
router.get('/employees/v1/not/city/:city', async (request, response) => {
  try {
    const employee = await Employee.findAll({
      where: {
        city: {
          [Op.ne]: request.params.city       // ne stands for Not Equal
        }
      }
    });

    // send response
    response.status(200).json(employee);
  } catch (error) {
    response.json(error);
  }
})
```

More on WHERE clause: gt, lt

```
// Find all employees with age less than a number
router.get('/employees/v1/lt/age/:age', async (request, response) => {
  try {
    const employee = await Employee.findAll({
      where: {
        age: {
          [Op.lt]: request.params.age
        }
      }
    });

    // send response
    response.status(200).json(employee);
  } catch (error) {
    response.json(error);
  }
})
```

Insert data in a table at creation

We can insert some data into a table using a bulk create. We need to re-execute these operation whenever our tables are deleted and recreated (altered):

```
db.sequelize.sync({ alter: true })
  .then(async () => {
    Country.count()       // counts the number of records
      .then(async (count) => {
        if( !count ) {    // if table empty
          await Country.bulkCreate([    // add these records
            { name: "KSA" },
            { name: "Oman" },
            { name: "Egypt" }
          ]);
        }
      });
  });
```

Work with model associations

```
const Employee = db.sequelize.define('Employee', {
...
});

// Let's add a Country table, every employee will be associated with 1 country
const Country = db.sequelize.define('Country', {
  name: {
    type: DataTypes.STRING,
    unique: true
  }
}, {
  timestamps: false  // no need to store timestamps, not critical data (static)
});

Country.hasMany(Employee);
// Employee will contain the foreign key to a country, it will be called "CountryId"
Employee.belongsTo(Country);
// Employee objects now have a getter .getCountry()
```

Get an employee and his associated country

```
// Search by id and get the country name using the one-to-many association
router.get(‘/employees/v1/:id', async (request, response) => {
  try {
    const employee = await Employee.findOne({
      where: {
        id: request.params.id
      }
    });
    const country = await employee.getCountry();   // enriched API

    response.status(200).json({employee, country});
  } catch (error) {
    response.json(error);
  }
})
```

Data validation and constraints

Validations are checks performed in the Sequelize level, in pure JavaScript. If a validation fails, no SQL query will be sent to the database at all. On the other hand, **constraints** are rules defined at SQL level. If a constraint check fails, an error will be thrown by the database.

```
const User = sequelize.define('user', {
  username: {
    type: DataTypes.STRING,
    allowNull: false,         // not null constraint
    unique: true,             // Unique constraint
  },
  hashedPassword: {
    type: DataTypes.STRING(64),
    validate: {
      is: /^[0-9a-f]{64}$/i,        // Validation enforced by Sequelize
    },
  },
});
```

Sequelize validators

```
is: /^[a-z]+$/i, // matches this RegExp
not: /^[a-z]+$/i, // does not match this RegExp
isEmail: true, // checks for email format (foo@bar.com)
isUrl: true, // checks for url format (https://foo.com)
isIP: true, // checks for IPv4 (129.89.23.1) or IPv6 format
isAlpha: true, // will only allow letters
isAlphanumeric: true, // will only allow alphanumeric characters, so "_abc" will fail
isNumeric: true, // will only allow numbers
isInt: true, // checks for valid integers
isFloat: true, // checks for valid floating point numbers
isDecimal: true, // checks for any numbers
isLowercase: true, // checks for lowercase
isUppercase: true, // checks for uppercase
notNull: true, // won't allow null
isNull: true, // only allows null
notEmpty: true, // don't allow empty strings
```

Sequelize validators (2)

```
equals: 'specific value', // only allow a specific value
contains: 'foo', // force specific substrings
notIn: [['foo', 'bar']], // check the value is not one of these
isIn: [['foo', 'bar']], // check the value is one of these
notContains: 'bar', // don't allow specific substrings
len: [2,10], // only allow values with length between 2 and 10
isDate: true, // only allow date strings
isAfter: "2011-11-05", // only allow date strings after a specific date
isBefore: "2011-11-05", // only allow date strings before a specific date
max: 23, // only allow values <= 23
min: 23, // only allow values >= 23
```

Sequelize validators: example

```
const User = sequelize.define('user', {
  …
  email: {
    type: DataTypes.STRING,
    validate: {
      isEmail: true
    }
  },
});

// Then in the controller:

const user = User.build({ username: 'Salah', email: 'salah' });

if(user.validate()) {
   try {
      await user.save();
```

Class Task

- Create account in aiven.io
- Create Database se371db
- Install extensions: MySQL, Thunder Client
- Connect to database
- Try get, post, del, put

END