

# Week 05 – ORM and Sequelize

Dr Kiki Adhinugraha

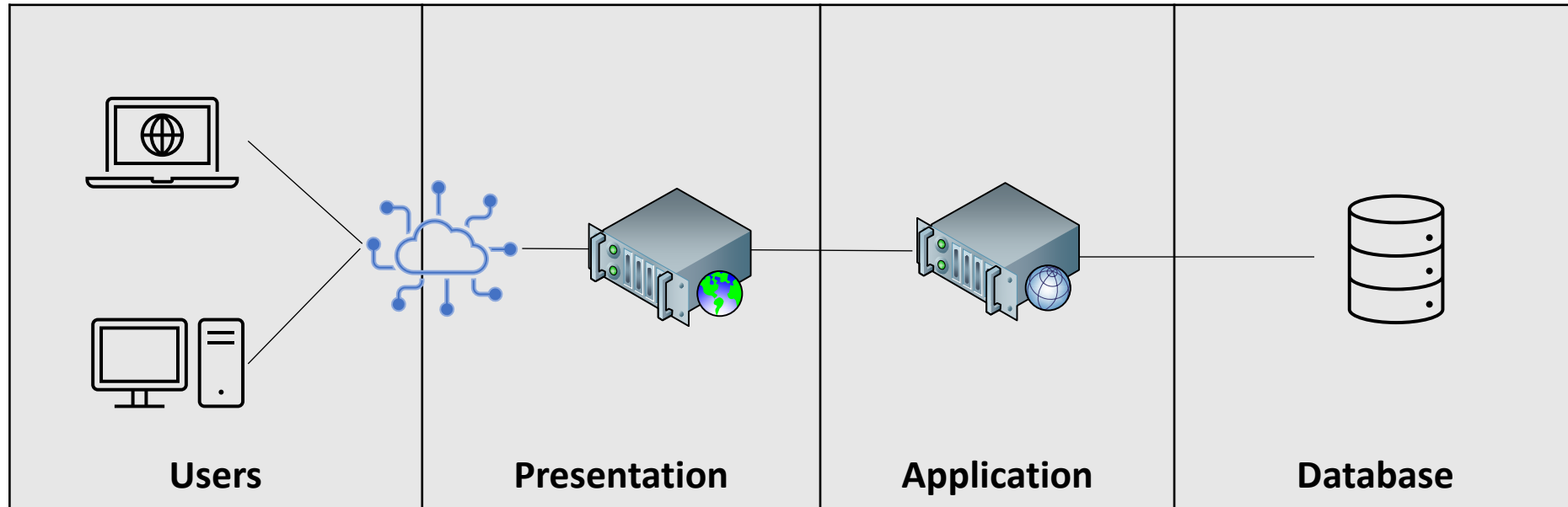
# Topic

- Database and Web Application
- 3-Tier Architecture
- Object Relational Mapper
- Sequelize

# Database and Web Application

- Database is a critical part in web application due to the following reasons
  - Data Storage  
A database is a central repository for storing data
  - Scalability
  - Performance Improvement
  - Security Improvement

# 3-tier Architecture



# 3-Tier Architecture example: PHP & MySQL

- PHP – MySQL is one of the most famous 3-Tier architecture in early 2000
- MySQL is an open-source relational database management system (RDBMS). It is the most popular database system used with PHP
- PHP is a server-side scripting language for web development, introduced in 1995
- Although popularity is decreasing, many websites are still running with PHP engine

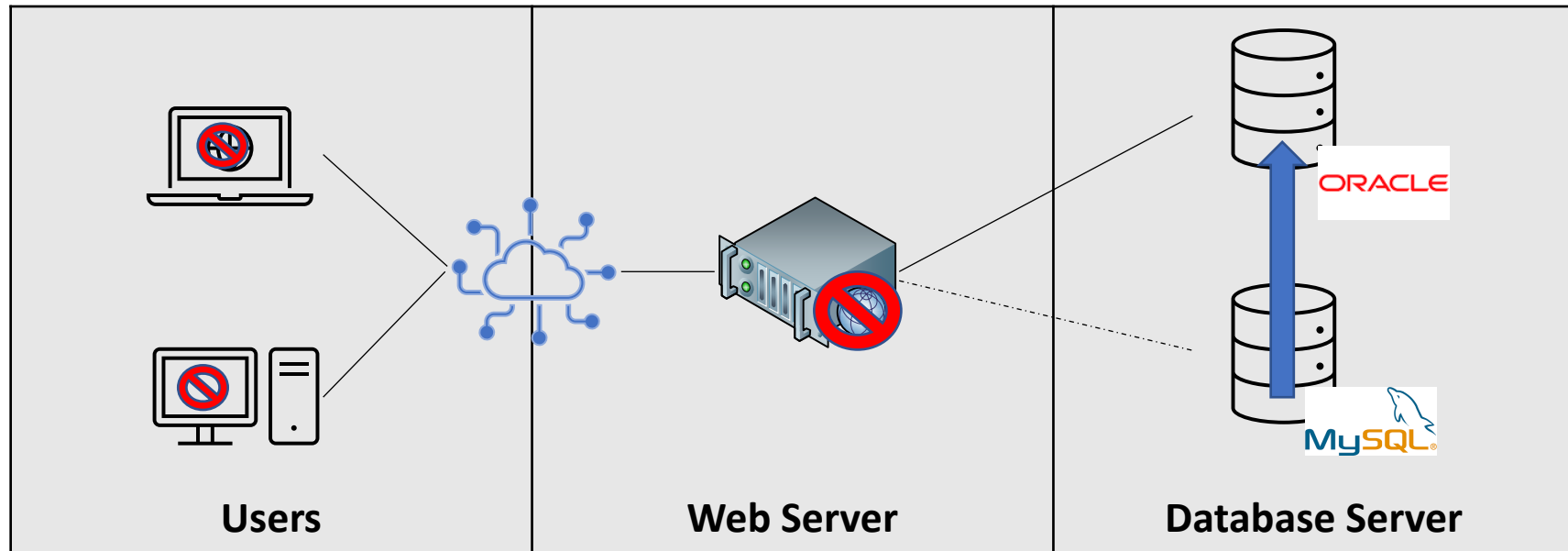


# Problems with 3-Tier architecture

- Although SQL is a general script to work with the RDBMS, every RDBMS have their own “special script” that won’t work in other RDBMS
- In general, all RDBMS have 3 common data types: String, Number, and Date. Every RDBMS have their own name and way how to work with these data types.
- The logic/script to work with the data is locked to a particular database engine
  - Connection settings
  - Method to call the query
  - Result set handle

# Problems with 3-Tier architecture

- **Database migration** is the most fearsome task to do
- In most cases, database migration requires rewriting most of the codes of the program

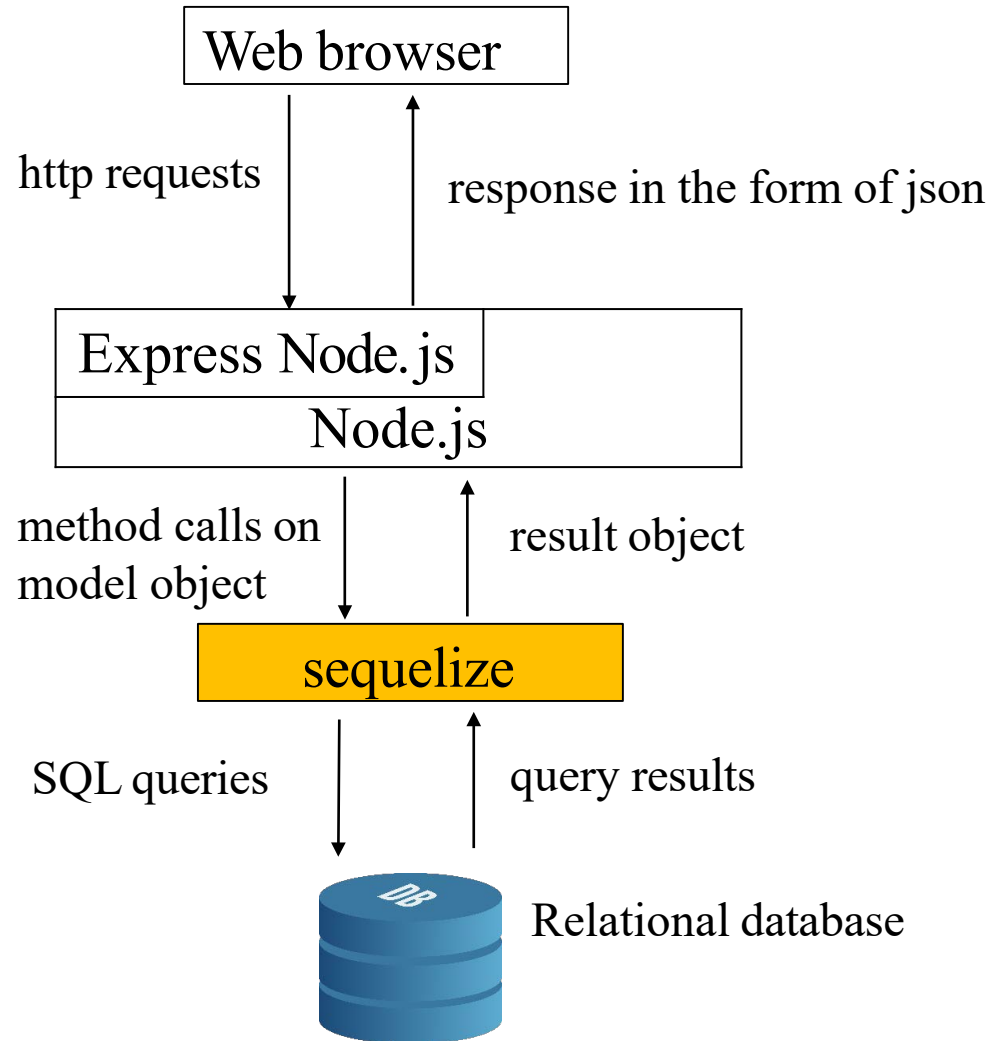


# Object Relational Mapping

- Object Relational Mapping (ORM) is a technique used in creating a "bridge" between the application that needs the data and the data source, in most cases, relational databases.
- ORM is the middleman between the application and database.
- By using ORM:
  - The application only knows that the data come from ORM. The Application use ORM as the data source
  - The RDBMS only knows that the data must be sent to or retrieved from ORM
- Sequelize is one of ORM Tools that are commonly used in web development



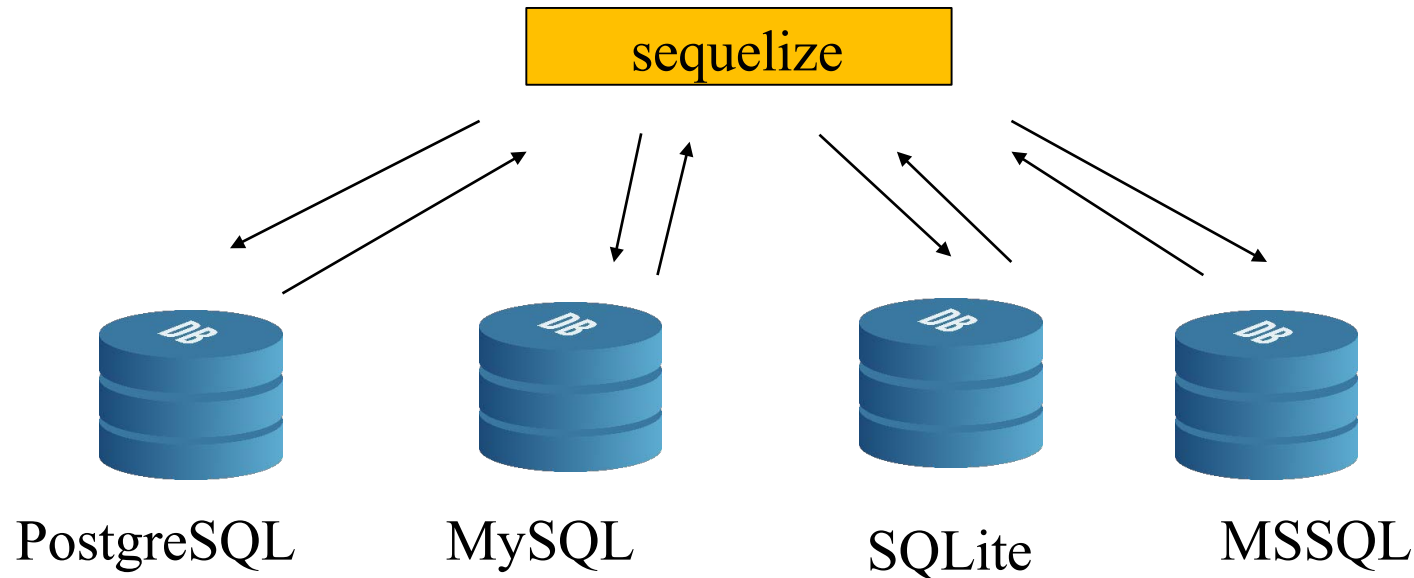
# Sequelize



- There are different options that express node.js can use for storing data
  - noSQL stores, like Mongo DB
  - Directly write SQL queries to store into a relational database
  - Use an object relational mapping (ORM) like sequelize to map methods to SQL queries.
- In this subject we are taking the last option because
  - ORM allows you to code more naturally in an object oriented language like Javascript
  - Relational databases are better than noSQL stores in most cases.

# Why use Sequelize?

- Sequelize supports many different dialects of SQL.
- Do not need to learn different variations for each SQL dialect
  - E.g. different dialects have different ways of encoding dates, but Sequelize abstracts over this allowing you to use the `Sequelize.DATE` data type.



# Why use Object Relational Mapping?

## MySQL

```
const query = "SELECT 'id', 'title', 'FROM  
'articles' WHERE 'articles', 'id' = ?";  
  
connection.query(query, 5, (error, article) =>  
{ console.log(article);  
});  
  
content', 'submittedAt' \
```

## Sequelize

```
Article.findById(5).then(article => {  
  console.log(article.dataValues);  
});
```

### articles

id

title

body

submittedAt

# How to connect to the Database in Sequelize?

```
const Sequelize = require("sequelize");
const dbConfig = {
  host: "hostname"
  port: 3306,
  // here is where you specify the kind of data you will use
  dialect: "mysql"
}
const connection = new Sequelize("db name", "my_username", "my_password", dbConfig);
```

- The above connects to a database
  - with name: "db name"
  - with username: "my\_username"
  - with password: "my\_password"
  - with configuration options specified by object: dbConfig

# Defining a model

- A model corresponds to a table in your database.
- Here is how to define a model in Sequelize and then how to save it into the database
- ```
const Article = connection.define("article",  
  { title: Sequelize.STRING,  
  
    // we use TEXT for type of content because STRING only  
    // allows up to 256 characters  
    content: Sequelize.TEXT  
  
  });  
  
connection.sync();
```
- The `define` method is used to create the new model.
- The `connection.sync()` function generates the SQL command to create the corresponding table in the MySQL database.

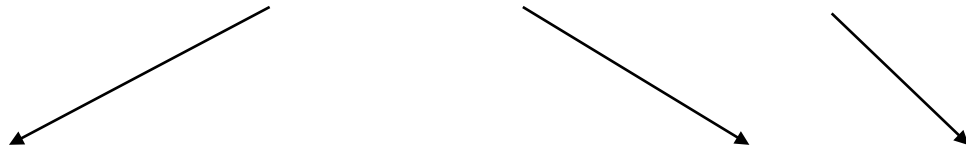
# Defining models

```
const Article = connection.define("article", {  
  title: Sequelize.STRING,  
  content: Sequelize.TEXT  
});
```

The above code will generate a table called article in the database with the following columns:

These columns are automatically generated by sequelize

article



| id | title | content | createdAt | updatedAt |
|----|-------|---------|-----------|-----------|
|    |       |         |           |           |

# Defining models

```
const Article = connection.define("article", { title:  
  Sequelize.STRING,  
  content: Sequelize.TEXT  
});
```

Or you can expand the definition of the content attribute like the following.

```
const Article = connection.define("article", {  
  title: Sequelize.STRING, content: {  
    type: Sequelize.TEXT,  
    // adds a property to the content attribute which specifies a default value  
    // which is used when the user does not specify a value for content. defaultValue:  
    'Coming soon...'  
  }  
});
```

# Defining models


```
const Article = connection.define("article", {
  isbn: {
    type: Sequelize.STRING,
    // This makes the isbn attribute the primary key and the id attribute will
    // no be automatically generated.
    primaryKey: true
  }
  title: {
    type: Sequelize.STRING,
    // property that makes sure that the title attribute is unique.
    unique: true,
    // property that ensure the title attribute value must be specified
    // so you must specify a value for the title attribute
    allowNull: false
  }
  content: {
    type: Sequelize.TEXT,
    defaultValue: 'Coming soon...'
  }
});
```



# What is the difference?

```
const Article = connection.define("article", {  
  title: Sequelize.STRING,  
  content: Sequelize.TEXT  
});
```

Never use this in production systems  
since you will lose all your data!



Next should we use:

`connection.sync();`



or

`connection.sync({force : true})`



If the table already exists in the  
database it will not alter the table.  
If the table does not exist it will create  
its

This will first drop the table if it  
already exists meaning it will **wipe  
everything (DANGER!!)**. Then it will  
create the new table.

# Inserting a new record into the database

```
Article.create({  
  title: 'War and Peace',  
  content: 'A book about fighting and then making up.'  
});
```

- The create method is used to create new records to insert into the database.
- The SQL code below is automatically generated by the above method call

```
INSERT INTO `articles` (`id`,`title`,`content`,`createdAt`,`updatedAt`)  
VALUES (DEFAULT, 'War and Peace ', 'A book about fighting and then  
making up.' , '2019-09-01 09:12:01', '2019-09-02 01:52:01');
```

# This will not work!

- What does this mean?
- Lets take a look at the following code:

```
const Article = connection.define("article", { title:  
    Sequelize.STRING,  
    content: Sequelize.TEXT  
});  
connection.sync({force : true})  
// Inserts a new record into the Article model/table Article.create({  
    title: 'War and Peace',  
    content: 'A book about fighting and then making up.'  
});
```

- The above code looks like it will first create the model and then insert a record into it.
- The reason is javascript is **asynchronous**. Meaning the different lines run concurrently.
  - **Meaning the record insertion can occur before the model has been created!**

# Making it not Asynchronous

- Use the **then** method!

```
const Article = connection.define("article", { title:
  Sequelize.STRING,
  content: Sequelize.TEXT
});
connection.sync({force : true}).then( function() {
  // Inserts a new record into the Article model/table Article.create({
    title: 'War and Peace',
    content: 'A book about fighting and then making up.'
  });
});
```

- Using the **then** method will ensure the code inside the **then** method is run only after the sync method has been completed.
- This ensures that the model/table has been created before data is inserted into it.

# What if we want to wait for many things to finish?

```
Article.create({
  title: 'War and Peace',
  content: 'A book about fighting and then making up.'
}).then( function() { Article.create({
  title: 'Sequelize for dummies',
  content: 'Writing lots of cool javascript code that get turned into SQL.'
});
}).then (function() { Article.create({
  title: 'I like tomatoes',
  content: 'The story about the adventures of a tomato lover.'
});
}).then (function() {
  // The code you want to run after the above commands have all finished.
  ...
})
```

- This is not nice! So many **then** method calls. Can we do it in a nicer way?
  - Yes, we can use **promise**

# Using promises to wait for multiple commands to complete

```
const Promise = Sequelize.Promise;
const a1 = Article.create({
  title: 'War and Peace',
  content: 'A book about fighting and then making up.'
});
const a2 = Article.create({
  title: 'Sequelize for dummies',
  content: 'Writing lots of cool javascript code that get turned into SQL.'
});
const a3 = Article.create({
  title: 'I like tomatoes',
  content: 'The story about the adventures of a tomato lover.'
});
Promise.all([a1, a2, a3]).then(function() {
  // The code you want to run after the above commands have all finished.
  ...
})
```

- The `Promise.all([ ])` method takes an array of items and makes sure they have been computed before moving on to the `then` method.

# Validation

```
const Article = connection.define("article", {
  isbn: {
    type: Sequelize.STRING,
    primaryKey: true
  }
  title: {
    type: Sequelize.STRING,
    unique: true,
    allowNull: false,
    // Ensures that the length of the title is between 10 and 150 characters.
    validate: {
      len: [10, 150]
    }
  }
  content: {
    type: Sequelize.TEXT,
    defaultValue: 'Coming soon...'
  }
});
```

- You can find many other validation rules from the following web page:  
<http://docs.sequelizejs.com/en/latest/docs/models-definition/#validations>

# Querying (findById)

Lets write a query that outputs the information for the Article with id of 2


```
Article.findById(2)
.then(article => {
  console.log('# Article with id=2');
  console.log(article.dataValues);
  console.log();
})
```

Query written in sequelize



```
SELECT `id`, `title`, `content`,
`createdAt`, `updatedAt` FROM
`articles` AS `articles` WHERE
`articles`.`id` = 2;
```

SQL query automatically  
generated by sequelize



## Output:

```
# Article with id=2
{ id: 2,
  title: 'Sequelize for dummies',
  content: 'Writing lots of cool javascript code that get turned into SQL.',
  createdAt: 2016-07-12T01:52:01.000Z,
  updatedAt: 2016-07-12T01:52:01.000Z }
```



# Querying (findAll)

- The following query outputs the information for all articles

```
Article.findAll()  
.then(articles => {  
  console.log('# All articles');  
  articles.forEach(article => {  
    console.log(article.dataValues);  
  });  
  console.log();  
})
```

```
SELECT `id`, `title`, `content`,  
`createdAt`, `updatedAt` FROM  
`articles` AS `articles`
```

## Output:

```
# All articles  
{ id: 1,  
  title: 'War and Peace',  
  content: 'A book about fighting and then making up.',  
  createdAt: 2016-07-12T01:52:01.000Z,  
  updatedAt: 2016-07-12T01:52:01.000Z }  
{ id: 2,  
  title: 'Sequelize for dummies',  
  content: 'Writing lots of cool javascript code that get turned into SQL.',  
  createdAt: 2016-07-12T01:52:01.000Z,  
  updatedAt: 2016-07-12T01:52:01.000Z }
```

# Querying (Where Clause)

- This will find all articles whose id is between 1 and 2.

```
Article.findAll({  
  where: {  
    id: { $between: [1, 2] }  
  }  
})
```

```
SELECT `id`, `title`, `content`,  
`createdAt`, `updatedAt` FROM  
`articles` AS `articles` WHERE  
`articles`.`id` BETWEEN 1 AND 2;
```

- This will find all articles whose id is greater than 3.

```
Article.findAll({  
  where: {  
    id: { $gt: 3 }  
  }  
})
```

```
SELECT `id`, `title`, `content`,  
`createdAt`, `updatedAt` FROM  
`articles` AS `articles` WHERE  
`articles`.`id` > 3;
```


# Deleting records

- The code below deletes all articles with id greater than 3

```
Article.findAll({  
  where: {  
    id: { $gt: 3 }  
  }  
}))  
.then(articles => {  
  articles.forEach(article => {  
    article.destroy();  
  });  
})
```

```
SELECT `id`, `title`, `content`,  
`createdAt`, `updatedAt` FROM  
`articles` AS `articles` WHERE  
`articles`.`id` > 3;
```

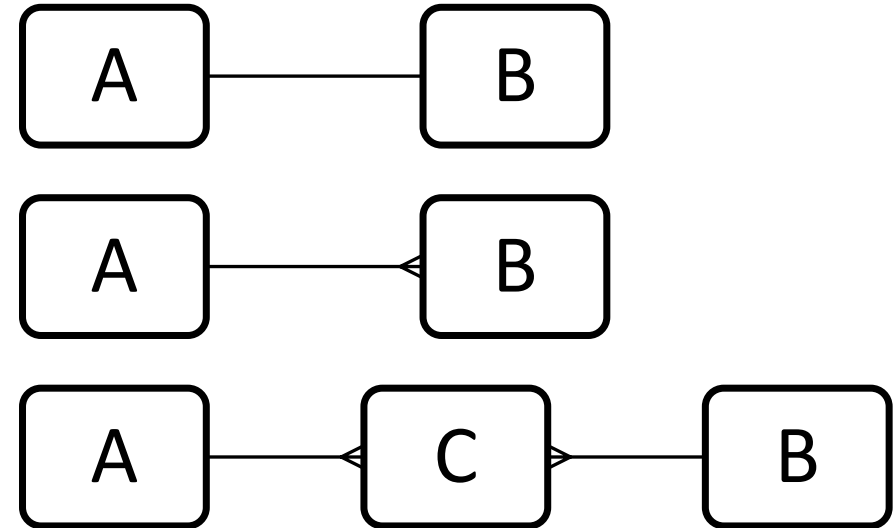
```
DELETE FROM `articles`  
WHERE `id` = 4 LIMIT 1  
DELETE FROM `articles`  
WHERE `id` = 5 LIMIT 1
```



In the example there are 5 articles hence the deletion of articles 4 and 5

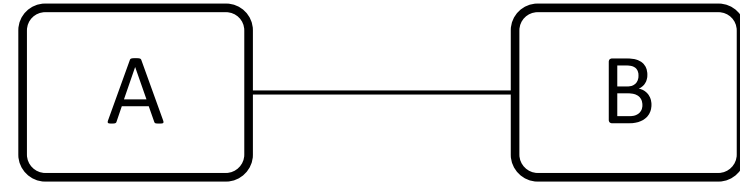
# Associations / Cardinality

- One-to-One
- One-to-Many
- Many-to-Many
  - Will have C as the junction table



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

# One-to-One association

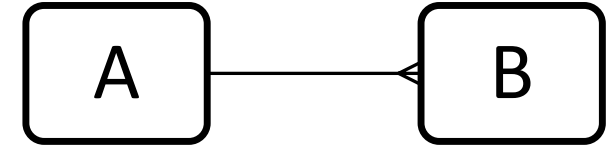


```
A.hasOne(B, { /* options */ });
```

```
A.belongsTo(B, { /* options */ });
```

- The `A.hasOne(B)` association means that a One-To-One relationship exists between A and B, with the foreign key being defined in the target model (B).
- The `A.belongsTo(B)` association means that a One-To-One relationship exists between A and B, with the foreign key being defined in the source model (A).
- These calls will cause Sequelize to automatically add foreign keys to the appropriate models (unless they are already present).
- Note: `belongsTo` can be seen as a one-to-many relationship from B's perspective. 1 B can have many As

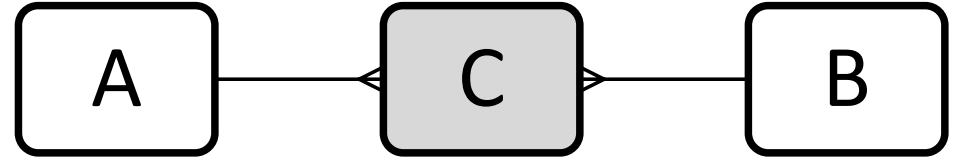
# One-to-Many



```
A.hasMany(B, { /* options */ });
```

- The A.hasMany(B) association means that a One-To-Many relationship exists between A and B, with the foreign key being defined in the target model (B).
- This call will cause Sequelize to automatically add foreign keys to the appropriate models (unless they are already present).
- Note: This condition can also be seen as one-to-one from B perspective. 1 B belongs to 1 A

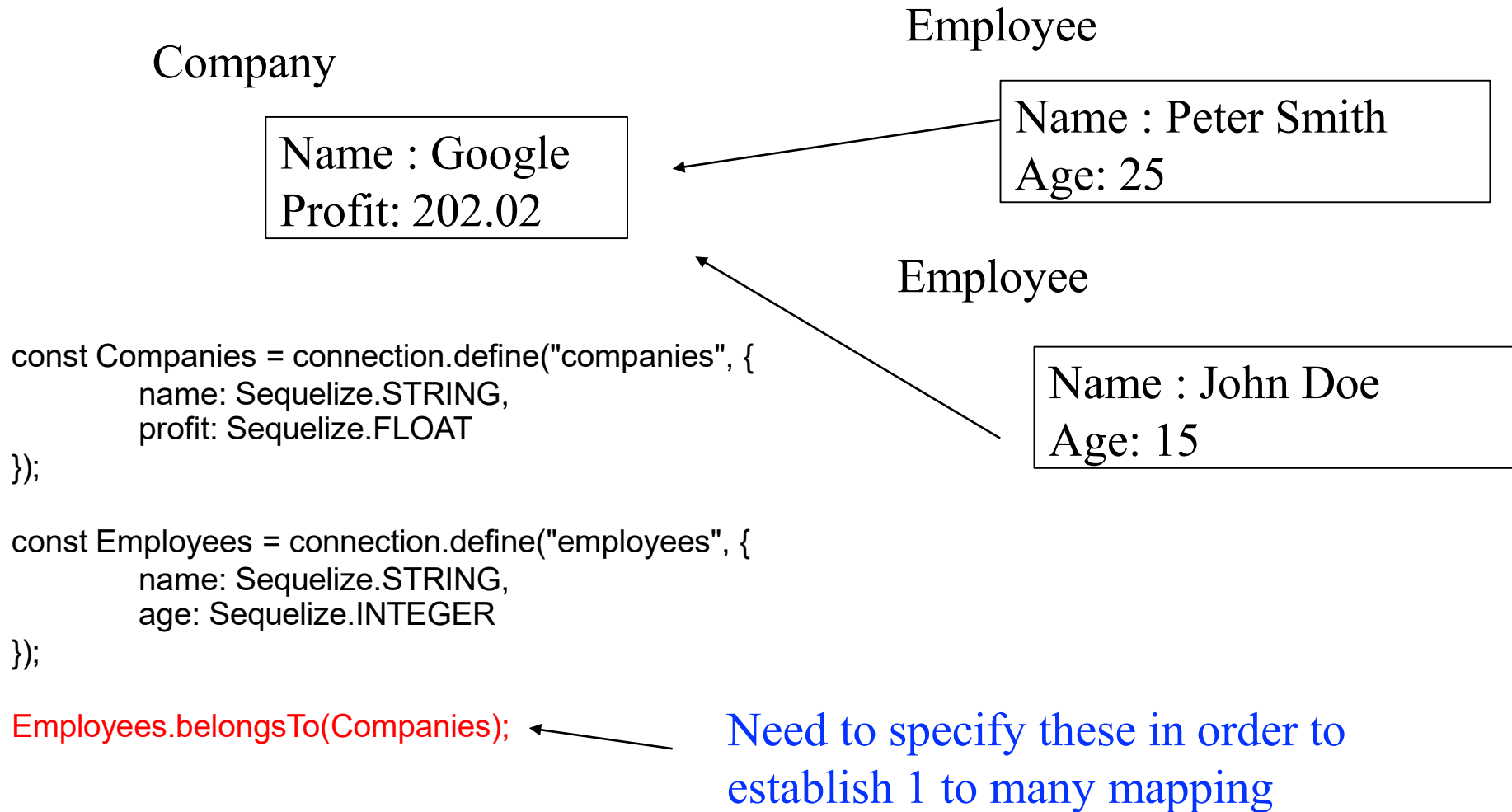
# Many-to-Many



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

- The `A.belongsToMany(B, { through: 'C' })` association means that a Many-To-Many relationship exists between A and B, using table C as junction table, which will have the foreign keys (ald and bld, for example).
- Sequelize will automatically create this model C (unless it already exists) and define the appropriate foreign keys on it.

# Associations (1 to many)






# Associations (1 to many)

```
const Employees = connection.define("employees", {  
  name: Sequelize.STRING,  
  age: Sequelize.INTEGER  
});  
Employees.belongsTo(Companies);
```

The system automatically generates a foreign key on the **employees** table which can be used to join with the **companies** table.

Foreign key automatically generated by Sequelize



| id | name | age | createdAt | updatedAt | companyID |
|----|------|-----|-----------|-----------|-----------|
|    |      |     |           |           |           |

# Inserting Data with Associations

- The code below inserts a company and its employees together. Notice the records are linked the `companyId` foreign key inside the `employee` table.

```
→ const c1 = Companies.create({  
  name: 'Apple',  
  profit: 20202.1  
}).then(c => {  
  const e1 = Employees.create({  
    name: 'Peter Smith',  
    age: 20,  
    // links the employee to the company.  
    companyId: c.id  
  });  
  const e2 = Employees.create({  
    name: 'Peter Senior',  
    age: 10,  
    companyId: c.id  
  });  
  ...  
});
```

```
INSERT INTO `companies`  
(`id`,`name`,`profit`,`createdAt`,`updatedAt`  
) VALUES  
(DEFAULT,'Apple',20202.1,'2016-07-12  
02:30:32','2016-07-12 02:30:32');
```

```
INSERT INTO `employees`  
(`id`,`name`,`age`,`createdAt`,`updatedAt`,`  
companyId`) VALUES (DEFAULT,'Peter  
Smith',20,'2016-07-12 02:30:32','2016-07-12  
02:30:32',1);
```

```
INSERT INTO `employees`  
(`id`,`name`,`age`,`createdAt`,`updatedAt`,`  
companyId`) VALUES (DEFAULT,'Peter  
Senior',10,'2016-07-12 02:30:33','2016-07-  
12 02:30:33',1);
```

# Querying (Joins)

- Finds any employee and then prints out the company he or she works in

Notice this **include** part which is what tells the system a join is needed.

↓

```
Employees.findOne({ include:  
  [Companies] }).then(employee => {  
  console.log(employee.dataValues.name  
    + ' works at ' +  
    employee.company.dataValues.name);  
})
```

Crazy complex SQL code  
automatically generated by sequelize

↗

```
SELECT `employees`.`id`,  
  `employees`.`name`, `employees`.`age`,  
  `employees`.`createdAt`,  
  `employees`.`updatedAt`,  
  `employees`.`companyId`,  
  `company`.`id` AS `company.id`,  
  `company`.`name` AS `company.name`,  
  `company`.`profit` AS `company.profit`,  
  `company`.`createdAt` AS  
  `company.createdAt`,  
  `company`.`updatedAt` AS  
  `company.updatedAt` FROM  
  `employees` AS `employees` LEFT  
  OUTER JOIN `companies` AS  
  `company` ON `employees`.`companyId`  
  = `company`.`id` LIMIT 1;
```

# Creating and linking

- This code creates a new company and makes the employee with id 1 as one of its employees

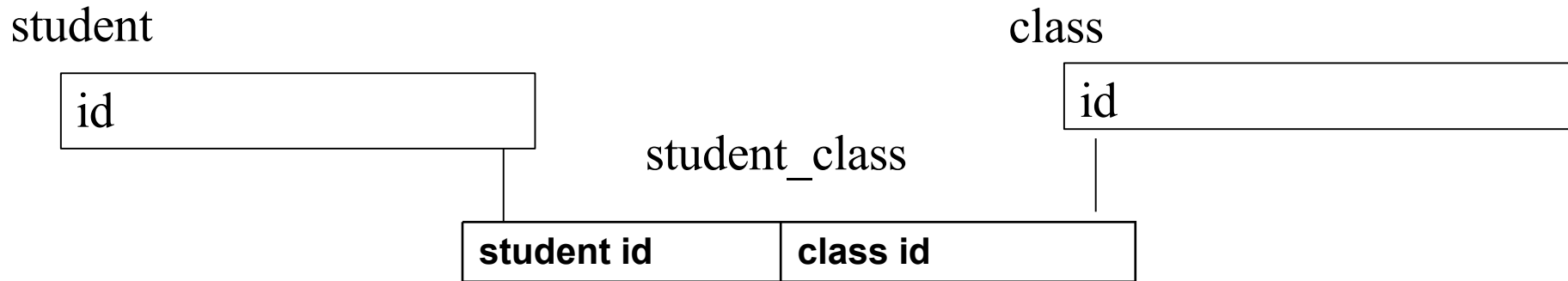
```
Companies.create({  
  name: 'Dell',  
  profit: 10.0  
}))  
.then(c => {  
  return Employees.findById(1).then(e =>  
    e.update({companyId: c.id}));  
})
```

**INSERT** INTO `companies`  
(`id`,`name`,`profit`,`createdAt`,`updatedAt`)  
VALUES (DEFAULT,'Dell',10,'2016-07-12 02:34:39','2016-07-12 02:34:39');

**SELECT** `id`,`name`,`age`,`createdAt`,`updatedAt`,`companyId` FROM  
`employees` AS `employees` WHERE  
`employees`.`id` = 1;

**UPDATE** `employees` SET  
`companyId`=3,`updatedAt`='2016-07-12 02:34:39' WHERE `id` = 1

# Associations (many to many)



- In order to model many to many relationships in a relational database, a join table needs to be added (in this case it is the **student\_class** table).
- Sequelize automatically generates this table using the **through** property when specifying the association see below.

```
const Students = db.define('students', {  
  name: Sequelize.STRING,  
});
```

```
const Classes = db.define('classes', {  
  code: Sequelize.STRING,  
});
```

- `Classes.belongsToMany(Students, { through: 'student_class' });`
- `Students.belongsToMany(Classes, { through: 'student_class' });`

# Creating and Linking up

```
return Promise.all([
  Classes.create({ code: 'CSE3CWA' }),
  Classes.create({ code: 'CSE5006' }),
  Students.create({ name: 'Steve Jobs' }),
  Students.create({ name: 'Mark Zuckerberg' })
]).then(() => {
  return Classes.findAll().then(allClasses => {
    return Students.findAll().then(allStudents => {
      return Promise.all([
        allClasses[0].setStudents(allStudents),
        allClasses[1].setStudents(allStudents)
      ]);
    })
  })
})
```

Use Promise.all to ensure all the records are created before then are linked up below

Enrolls all the students into the first class

Enrolls all student to the second class

# Example query for many to many association

- The query below prints out all students that belong to class with id of 1.

```
Classes.findById(1, {include : [Students]}).then(c => {  
  console.log("The students enrolled in " + c.dataValues.code + " are:");  
  c.students.forEach(s => console.log(s.dataValues.name));  
})
```

Nice sequelize code

Crazy SQL query!!

```
SELECT `classes`.`id`, `classes`.`code`, `classes`.`createdAt`, `classes`.`updatedAt`, `students`.`id`  
AS `students.id`, `students`.`name` AS `students.name`, `students`.`createdAt` AS  
`students.createdAt`, `students`.`updatedAt` AS `students.updatedAt`,  
`students.student_class`.`createdAt` AS `students.student_class.createdAt`,  
`students.student_class`.`updatedAt` AS `students.student_class.updatedAt`,  
`students.student_class`.`classId` AS `students.student_class.classId`,  
`students.student_class`.`studentId` AS `students.student_class.studentId` FROM `classes` AS  
`classes` LEFT OUTER JOIN (`student_class` AS `students.student_class` INNER JOIN `students`  
AS `students` ON `students`.`id` = `students.student_class`.`studentId`) ON `classes`.`id` =  
`students.student_class`.`classId` WHERE `classes`.`id` = 1;
```

## Output

The students enrolled in CSE2WDC are:  
Steve Jobs  
Mark Zuckerberg

# Database Migration

- Suppose you have database table below:

| Firstname | Lastname | Age | Gender |
|-----------|----------|-----|--------|
| Zhen      | He       | 80  | Male   |
| Steve     | Jobs     | 60  | Male   |
| Sandra    | Smith    | 30  | Female |
| Roger     | Federer  | 33  | Male   |

- You want to add a column to the table like below:

| Firstname | Lastname | Age | Gender | Salary |
|-----------|----------|-----|--------|--------|
| Zhen      | He       | 80  | Male   |        |
| Steve     | Jobs     | 60  | Male   |        |
| Sandra    | Smith    | 30  | Female |        |
| Roger     | Federer  | 33  | Male   |        |

- If you do not use database migration then you would have to manually move the data in order to add the extra column.
- Sequelize has a migration tool which helps you automatically perform the migration.



# Database Migration

- In order to use database migration in sequelize we use the following command to generate code to make the tables and also the corresponding models

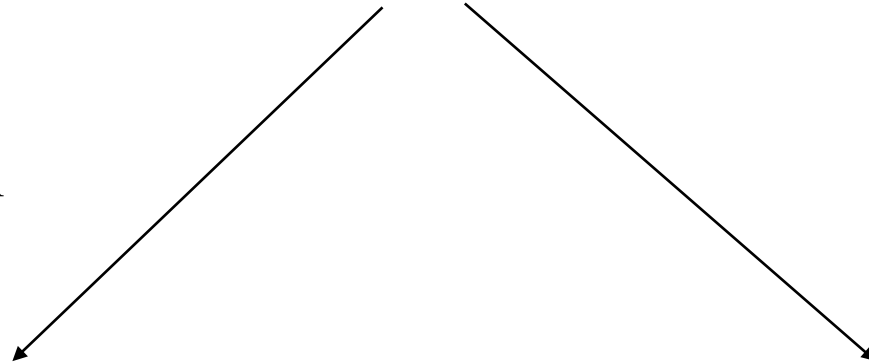
Creating the table and model in one command

```
sequelize model:create --name Post --attributes title:string, content:text
```

DB migration

Database Table

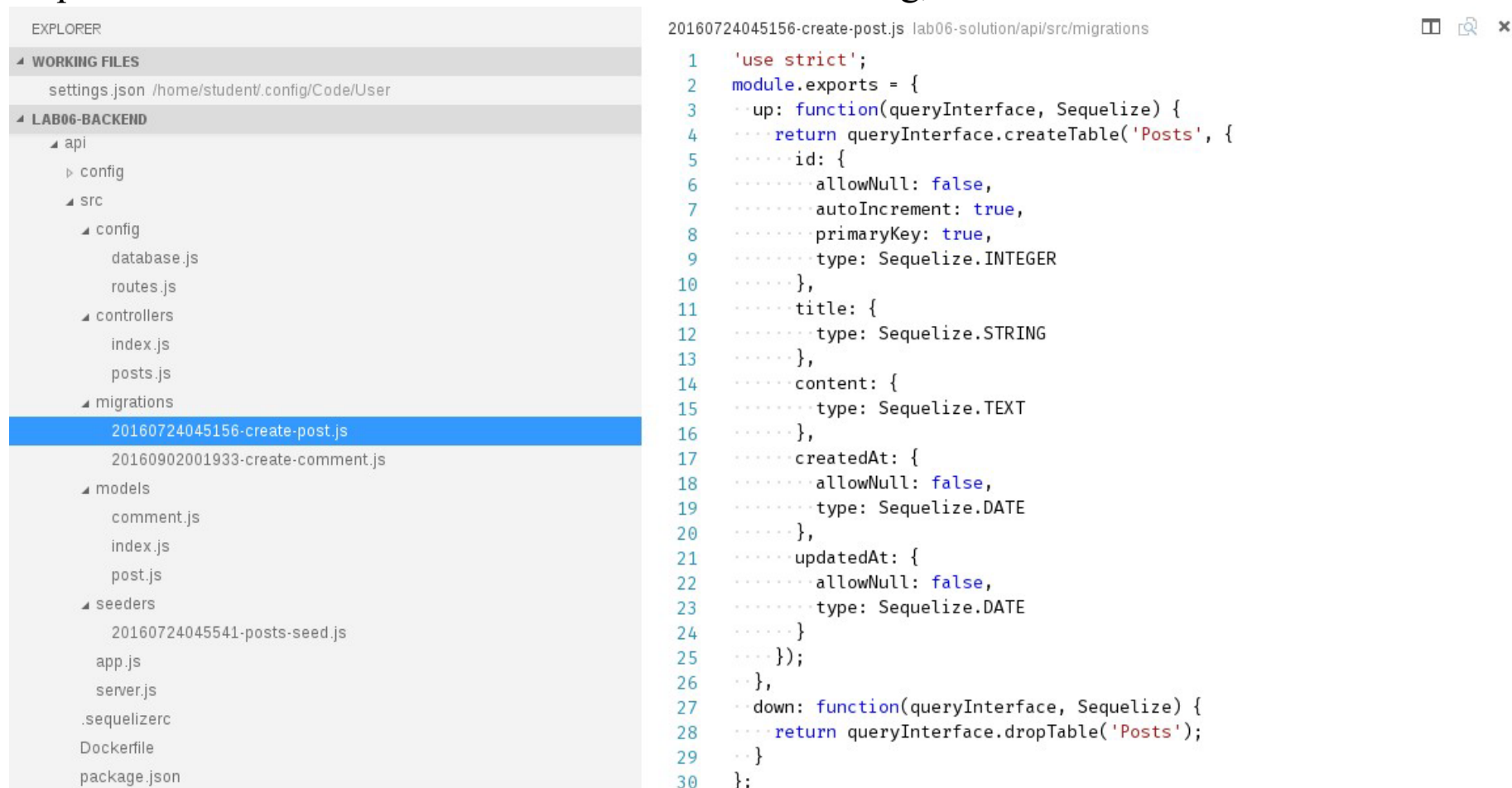
Sequelize Model



# Database Migration : Create DB table

As mentioned before this command creates the following sequelize function call (createTable) which is used to create the database table Posts.

sequelize model:create --name Post --attributes title:string, content:text



```
1  'use strict';
2  module.exports = {
3    up: function(queryInterface, Sequelize) {
4      return queryInterface.createTable('Posts', {
5        id: {
6          allowNull: false,
7          autoIncrement: true,
8          primaryKey: true,
9          type: Sequelize.INTEGER
10       },
11       title: {
12         type: Sequelize.STRING
13       },
14       content: {
15         type: Sequelize.TEXT
16       },
17       createdAt: {
18         allowNull: false,
19         type: Sequelize.DATE
20       },
21       updatedAt: {
22         allowNull: false,
23         type: Sequelize.DATE
24       }
25     });
26  },
27  down: function(queryInterface, Sequelize) {
28    return queryInterface.dropTable('Posts');
29  }
30  };
```

# Database Migration : Foreign key

- If we want to link two tables using database migration, we need to add the foreign key manually.
- For example in our lab we want to create a table for comments which are linked to posts. Hence we need to manually put in a field in the comments table that acts as the foreign key to the posts table. Here is the command to do it.
- `sequelize model:create --name Comment --attributes name: string, content: text, postID:integer`

# Database Migration : Add Column

- The code below shows an example of how to add a column to an existing table.

```
queryInterface.addColumn(  
  'nameOfAnExistingTable',  
  'nameOfTheNewAttribute',  
  Sequelize.STRING  
)
```

# Conclusion

- Sequelize ORM makes querying and updating relational databases easier.