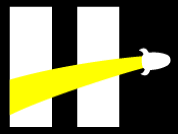
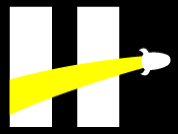

HENRY

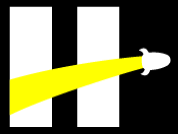


Parte 1

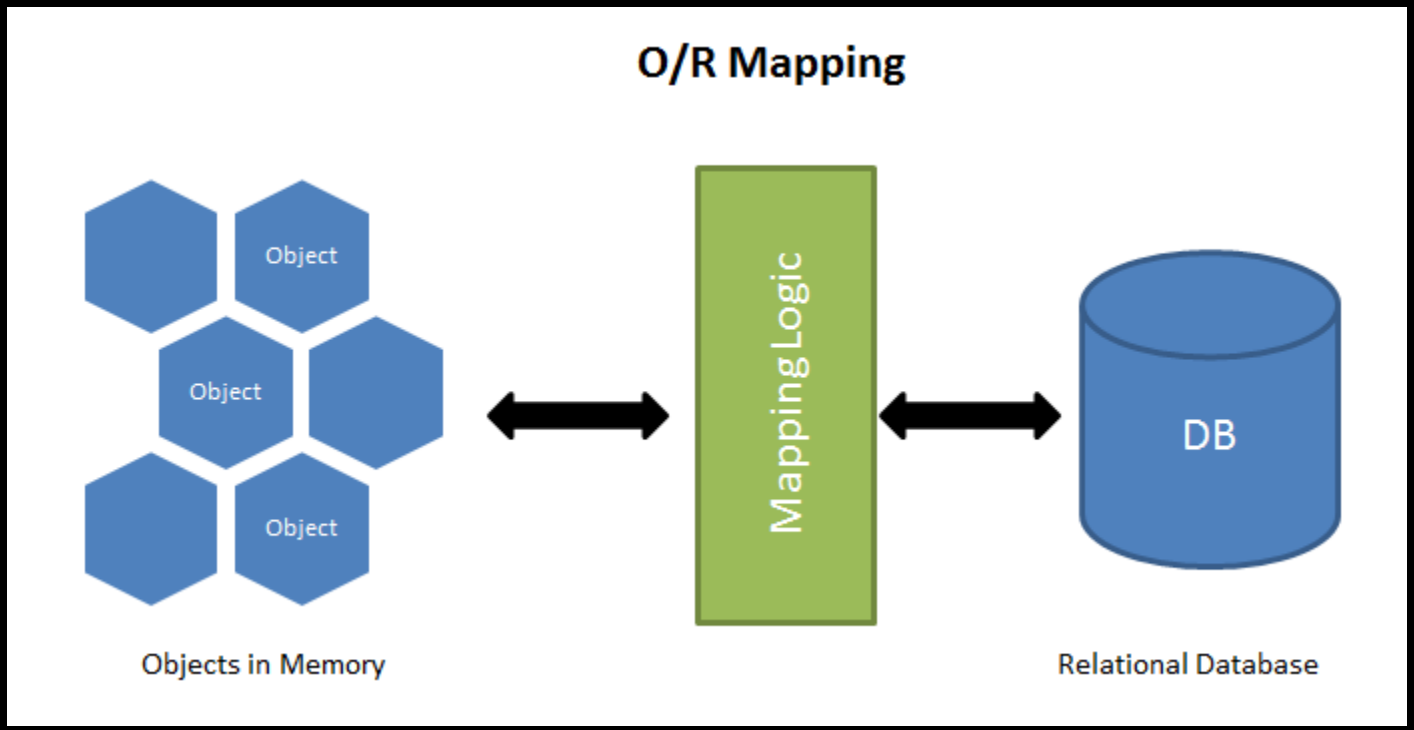


ORMs

Object-Relational Mapping



ORMs

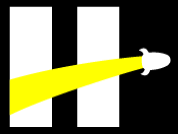

O/R Mapping
Object
Object
Object
Mapping Logic
DB
Objects in Memory
Relational Database

Sequelize

"**Promise-based** Node.js ORM for Postgres, MySQL, MariaDB, SQLite and Microsoft SQL Server"

La mayoría de los métodos son asíncronos por lo que devuelven promesas

Node v10+



Sequelize

Installing

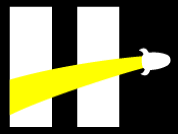
```
npm install --save sequelize
npm install --save pg pg-hstore # Postgres
```

Connecting

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

// Opción 1: Connection URI
const sequelize = new Sequelize('postgres://user:pass@example.com:5432/dbname')

// Opción 2: Parámetros separados
const sequelize = new Sequelize('database', 'username', 'password', {
  host: 'localhost',
  dialect: /* one of 'mysql' | 'mariadb' | 'postgres' | 'mssql' */
});
```



Data Types

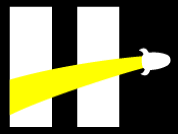
```
// TEXTO
DataTypes.STRING                    // VARCHAR(255)
DataTypes.STRING(1234)              // VARCHAR(1234)
DataTypes.TEXT                      // TEXT

// NUMEROS
DataTypes.INTEGER                   // INTEGER
DataTypes.FLOAT                     // FLOAT

// FECHAS
DataTypes.DATE                      // TIMESTAMP WITH TIME ZONE
DataTypes.DATEONLY                  // DATE without time

// OTROS
DataTypes.ENUM('foo', 'bar') // An ENUM with allowed values 'foo' and 'bar'
```

Listado Completo



Model

Definition

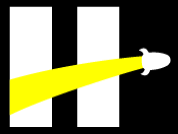
Abstracción que representa una tabla de nuestra base de datos

```sequelize
const User = sequelize.define('User', {
  firstName: {
    type: DataTypes.STRING
  },
  lastName: {
    type: DataTypes.STRING
  }
});
```

```sequelize
class User extends Model {}

User.init({
  firstName: {
    type: DataTypes.STRING
  },
  lastName: {
    type: DataTypes.STRING
  }
}, {
  sequelize, // Connection instance
  modelName: 'User' // Model name
});
```

User === sequelize.models.User



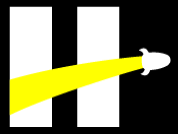
Model

Synchronization

Definir el modelo arma el esqueleto pero no aplica los cambios en la base de datos, para eso debemos sincronizar los modelos

- **Model.sync()**: crea la tabla si no existe o no hace nada si ya existe
- **Model.sync({force: true})**: elimina (drop) la tabla y luego la vuelve a crear
- **Model.sync({alter: true})**: aplica los cambios necesarios a la tabla actual para que coincida con el modelo

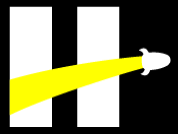
```
1 // Un modelo a la vez
2 await Model.sync({ force: true });
3
4 // Todos los modelos juntos
5 await sequelize.sync({ force: true });
```



Sequelize

Logging

```
1  const sequelize = new Sequelize('postgres://user:pass@example.com:5432/dbname', {
2    // Opciones:
3
4    // Default
5    logging: console.log,
6
7    // Muestra información adicional más allá de la Query SQL
8    logging: (...msg) => console.log(msg),
9
10   // Deshabilita el logging
11   logging: false,
12 });
```



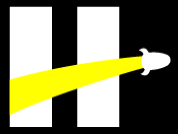
Model

Automatic Timestamps

Se agregan automáticamente los campos createdAt y updatedAt que a su vez se actualizan sólos al crear o modificar una instancia del modelo

```
sequelize.define('User', {
  // ... (attributes)
}, {
  timestamps: false
});
```

```
sequelize.define('User', {
  // ... (attributes)
}, {
  timestamps: true,
  createdAt: false,
  updatedAt: 'actualizado'
});
```



Model

Column Options

```
1  class Foo extends Model {}
2  Foo.init({
3    flag: { type: DataTypes.BOOLEAN, allowNull: false, defaultValue: true },
4    myDate: { type: DataTypes.DATE, defaultValue: DataTypes.NOW },
5    someUnique: { type: DataTypes.STRING, unique: true },
6
7    // Composite unique key.
8    uniqueOne: { type: DataTypes.STRING,  unique: 'compositeIndex' },
9    uniqueTwo: { type: DataTypes.INTEGER, unique: 'compositeIndex' },
10
11
12   identifier: { type: DataTypes.STRING, primaryKey: true },
13   incrementMe: { type: DataTypes.INTEGER, autoIncrement: true },
14
15 });
```

Instances

Representa un objeto con la estructura definida en el modelo que va a mapearse con una fila de la tabla asociada

```
1 const jane = User.build({ name: "Jane" });
2 await jane.save();
```

```
1 const jane = await User.create({ name: "Jane" });
```

Instances

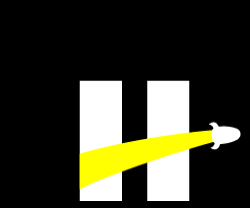
Modifications

```
1 const jane = await User.create({ name: "Jane" });
2 jane.name = "Ada";
3 await jane.save();
```

```
1 const jane = await User.create({ name: "Jane" });
2 await jane.destroy();
```

El método save esta optimizado para sólo actualizar los campos que efectivamente fueron modificados. Si se invoca sin ningún cambio ni siquiera ejecutará una query

```
1 const jane = await User.create({ name: "Jane" });
2 console.log(jane); // Mal! (En principio para lo que queremos)
3 console.log(jane.toJSON()); // Bien!
```



Queries

SELECT

```js
// SELECT * FROM ...
const instancias = await Model.findAll();

// SELECT foo, bar FROM ...
Model.findAll({
  attributes: ['foo', 'bar']
});

// SELECT foo, bar as baz FROM ...
Model.findAll({
  attributes: ['foo', ['bar', 'baz']]
});

// Exclude some attribute
Model.findAll({
  attributes: { exclude: ['baz'] }
});
```

```js
const instances = Model.findAll({
  where: {
    clothe: 'orange'
    status: 'good'
  }
});
```

Queries

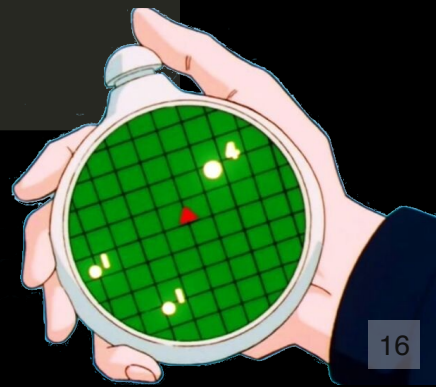
Finders

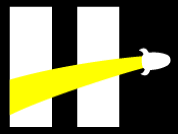
```
1 const instance = await Model.findByPk(4); // null si no lo encuentra
```

```
1 const instance = await Model.findOne({
2  where: { name: 'Goku' }
3 }); // null si no lo encuentra
```

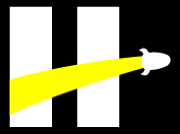
```
1 const [instance, created] = await Model.findOrCreate({
2  where: { name: 'Goku' },
3  defaults: {
4    gender: 'M',
5    race: 'Saiyan'
6  }
7 });
```

findOrCreate busca si existe un registro según las condiciones de búsqueda y si no encuentra ninguno procede a crear uno nuevo. Luego retorna la instancia creada o encontrada y un booleano indicando cual de los dos caminos tomó





Parte 2



Queries
Operators

```
1  const { Op } = require("sequelize");
2  Model.findAll({
3    where: {
4      [Op.and]: [{ a: 5 }, { b: 6 }],           // (a = 5) AND (b = 6)
5      [Op.or]: [{ a: 5 }, { b: 6 }],            // (a = 5) OR (b = 6)
6      someAttribute: {
7        // Basics
8        [Op.eq]: 3,                             // = 3
9        [Op.ne]: 20,                            // != 20
10       [Op.is]: null,                          // IS NULL
11       [Op.not]: true,                         // IS NOT TRUE
12
13       // Number comparisons
14       [Op.gt]: 6,                             // > 6
15       [Op.lt]: 10,                            // < 10
16       [Op.between]: [6, 10],                  // BETWEEN 6 AND 10
17       [Op.notBetween]: [11, 15],              // NOT BETWEEN 11 AND 15
18
19       // Other operators
20       [Op.in]: [1, 2],                        // IN [1, 2]
21       [Op.notIn]: [1, 2],                     // NOT IN [1, 2]
22
23                 ...
24     }
25   }
26 });
```

Listado Completo

Queries

UPDATE

```
// UPDATE table
// SET transformation = 'SS1'
// WHERE name = 'Goku'

await User.update({ transformation: 'SS1' }, {
  where: {
    name: 'Goku'
  }
});
```

Actualizará todas las instancias que coincidan con la claúsula where indicada. Si no se coloca ninguna condición actualizará todos los registros

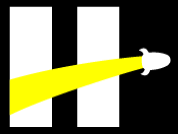
Queries

DELETE

```
1  // DELETE FROM table
2  // WHERE race = 'Android'
3
4  await Model.destroy({
5    where: {
6      race: 'Android'
7    }
8  });
9
10 // Truncate
11 await Model.destroy({
12   truncate: true
13 });
```

Borrará todas las instancias que coincidan con la claúsula where indicada. Si no se coloca ninguna condición borrará todos los registros

¿TRUNCATE = DESTROY? Truncate no acepta condiciones y elimina todos los registros de una mientras que destroy va revisando registro a registro



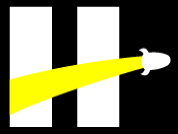
Getters

```
1 const instance = sequelize.define('model', {
2   attribute: {
3     type: DataTypes.STRING,
4     get() {
5       const rawValue = this.getDataValue(attribute);
6       return rawValue ? rawValue.toUpperCase() : null;
7     }
8   }
9 });
```

Se llama de forma automática cuando se intenta acceder al atributo

El valor NO se va a modificar en la base de datos

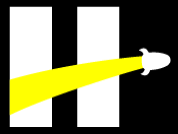
this.attribute generaría un loop infinito (Si accedemos a otro attribuito que no sea donde estamos definiendo el getter funcionaria ok el this)



Setters

```
1 const User = sequelize.define('user', {
2   password: {
3     type: DataTypes.STRING,
4     set(value) {
5       this.setDataValue('password', hash(this.username + value));
6     }
7   }
8 });
```

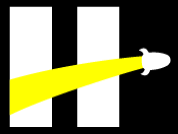
Se llama de forma automática previo al almacenamiento de los datos en la base de datos



Virtual Fields

```
1  const User = sequelize.define('user', {
2    firstName: DataTypes.TEXT,
3    lastName: DataTypes.TEXT,
4    fullName: {
5      type: DataTypes.VIRTUAL,
6      get() {
7        return `${this.firstName} ${this.lastName}`;
8      },
9      set(value) {
10       throw new Error('Do not try to set the `fullName` value!');
11     }
12   }
13 });
```

El valor NO se va a guardar en la base de datos

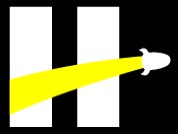


Validators

```
sequelize.define('foo', {
  bar: {
    type: DataTypes.STRING,
    validate: {
      is: /^[a-z]+$/i,
      isEmail: true,
      isUrl: true,
      isAlpha: true,
      isAlphanumeric: true,
      isNumeric: true,
      isLowercase: true,
      notNull: true,
      notEmpty: true,
      equals: 'specific value',
      contains: 'foo',
      isIn: [['foo', 'bar']],
      notContains: 'bar',
      len: [2,10],
      isAfter: "2011-11-05",
      max: 23,

      // Custom validators:
      isEven(value) {
        if (parseInt(value) % 2 !== 0) {
          throw new Error('Only even values are allowed!');
        }
      }
    }
  }
});
```

Listado Completo



Associations

One-To-One

```
1 Foo.hasOne(Bar);
2 Bar.belongsTo(Foo);
```

1:1

One-To-Many

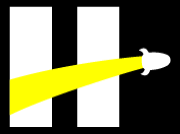
```
1 Team.hasMany(Player);
2 Player.belongsTo(Team);
```

1:N

Many-To-Many

```
1 Movie.belongsToMany(Actor, { through: 'ActorMovies' });
2 Actor.belongsToMany(Movie, { through: 'ActorMovies' });
```

N:N



Mixins

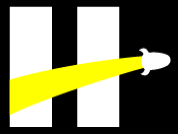
Foo.hasOne(Bar)

Foo.belongsTo(Bar)

```
1 fooInstance.getBar()
2 fooInstance.setBar()
3 fooInstance.createBar()
```

Foo.hasMany(Bar)

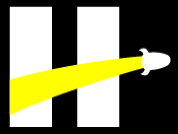
```
 1 fooInstance.getBars()
 2 fooInstance.countBars()
 3 fooInstance.hasBar()
 4 fooInstance.hasBars()
 5 fooInstance.setBars()
 6 fooInstance.addBar()
 7 fooInstance.addBars()
 8 fooInstance.removeBar()
 9 fooInstance.removeBars()
10 fooInstance.createBar()
```



Mixins

Foo.belongsToMany(Bar, {through: Baz})

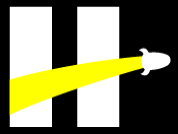
```
1  fooInstance.getBars()
2  fooInstance.countBars()
3  fooInstance.hasBar()
4  fooInstance.hasBars()
5  fooInstance.setBars()
6  fooInstance.addBar()
7  fooInstance.addBars()
8  fooInstance.removeBar()
9  fooInstance.removeBars()
10 fooInstance.createBar()
```



Fetching Associations

Lazy Loading

```
1  const awesomeCaptain = await Captain.findOne({
2    where: {
3      name: "Jack Sparrow"
4    }
5  });
6
7  console.log('Name:', awesomeCaptain.name);
8  console.log('Skill Level:', awesomeCaptain.skillLevel);
9
10 const hisShip = await awesomeCaptain.getShip();
11
12 console.log('Ship Name:', hisShip.name);
13 console.log('Amount of Sails:', hisShip.amountOfSails);
```

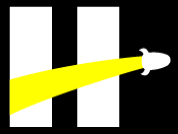


Fetching Associations

Eager Loading

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

console.log('Name:', awesomeCaptain.name);
console.log('Skill Level:', awesomeCaptain.skillLevel);
console.log('Ship Name:', awesomeCaptain.ship.name);
console.log('Amount of Sails:', awesomeCaptain.ship.amountOfSails);
```



Hooks

Lifecycle Events

```
(1)
  beforeValidate(instance, options)

[... validation happens ...]

(2)
  afterValidate(instance, options)
  validationFailed(instance, options, error)
(3)
  beforeCreate(instance, options)
  beforeDestroy(instance, options)
  beforeUpdate(instance, options)

[... creation/update/destruction happens ...]

(4)
  afterCreate(instance, options)
  afterDestroy(instance, options)
  afterUpdate(instance, options)
```

```
User.beforeCreate((user, options) => {
  const hashedPassword = hashPassword(user.password);
  user.password = hashedPassword;
});
```