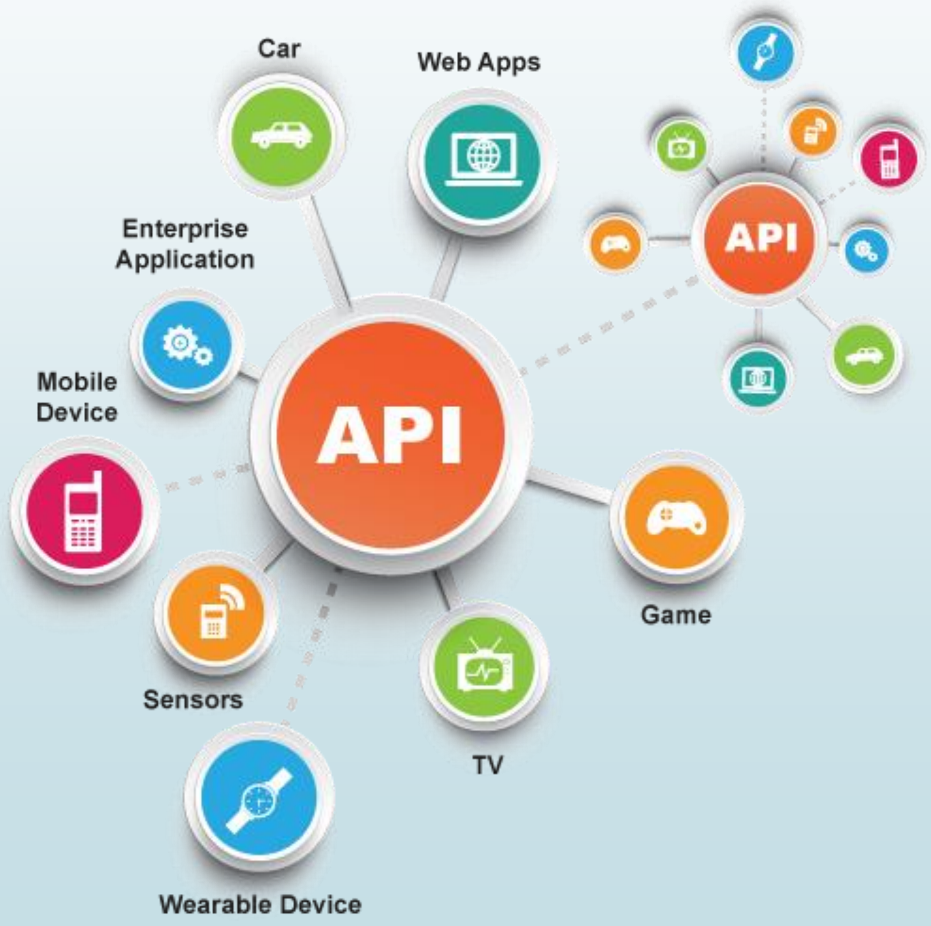
Desenvolvimento de Serviços e APIs

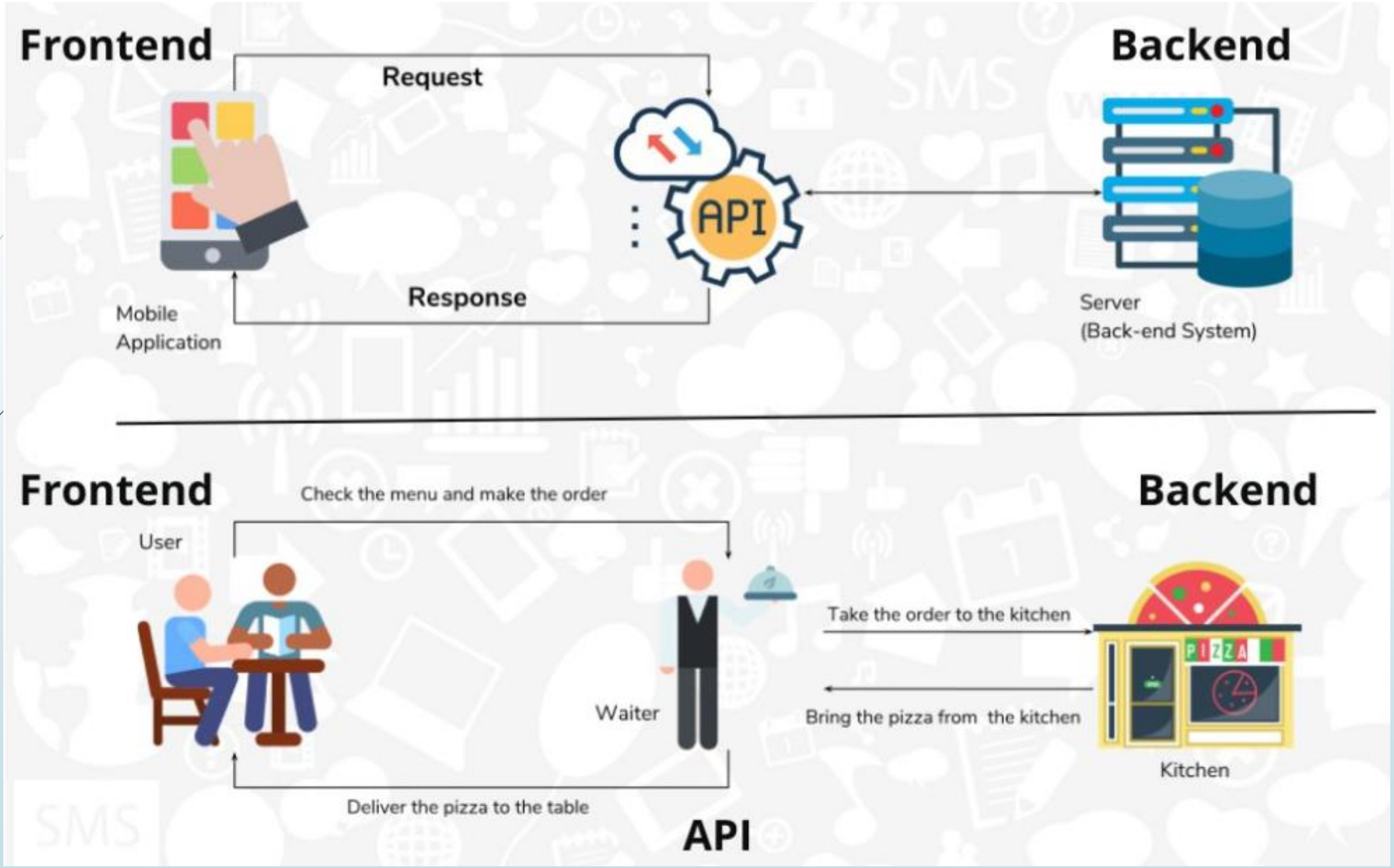
Faculdade Senac Pelotas

Escola de Tecnologia da Informação

Prof. Edécio Fernando Iepsen

Uma API é um conjunto de normas que possibilita a comunicação ou integração entre softwares/plataformas a partir de uma série de padrões e protocolos.




Frontend
Request
Backend
API
Mobile Application
Response
Server (Back-end System)
Frontend
Check the menu and make the order
Backend
User
Take the order to the kitchen
Waiter
Bring the pizza from the kitchen
PIZZA
Kitchen
Deliver the pizza to the table
API

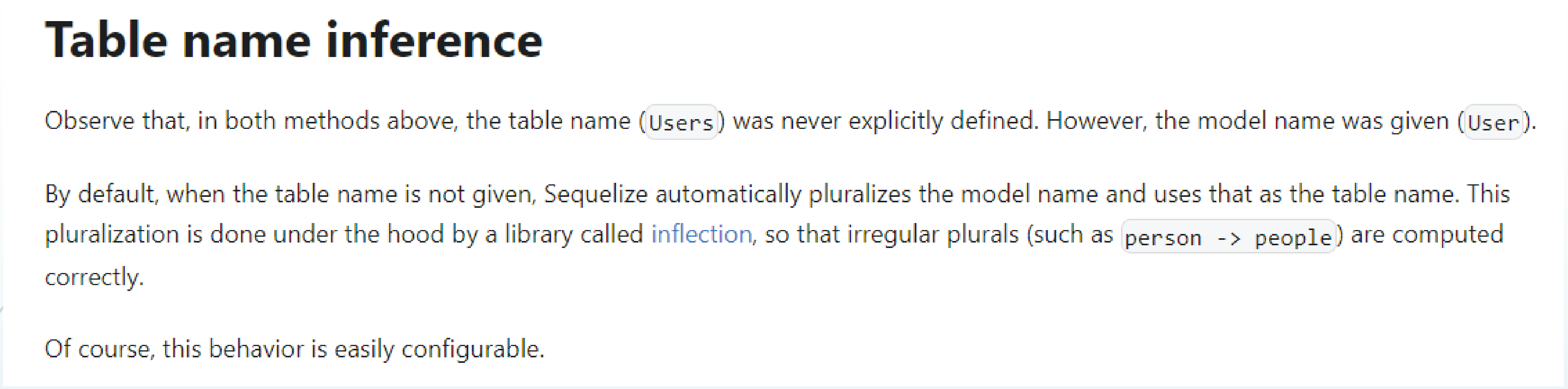
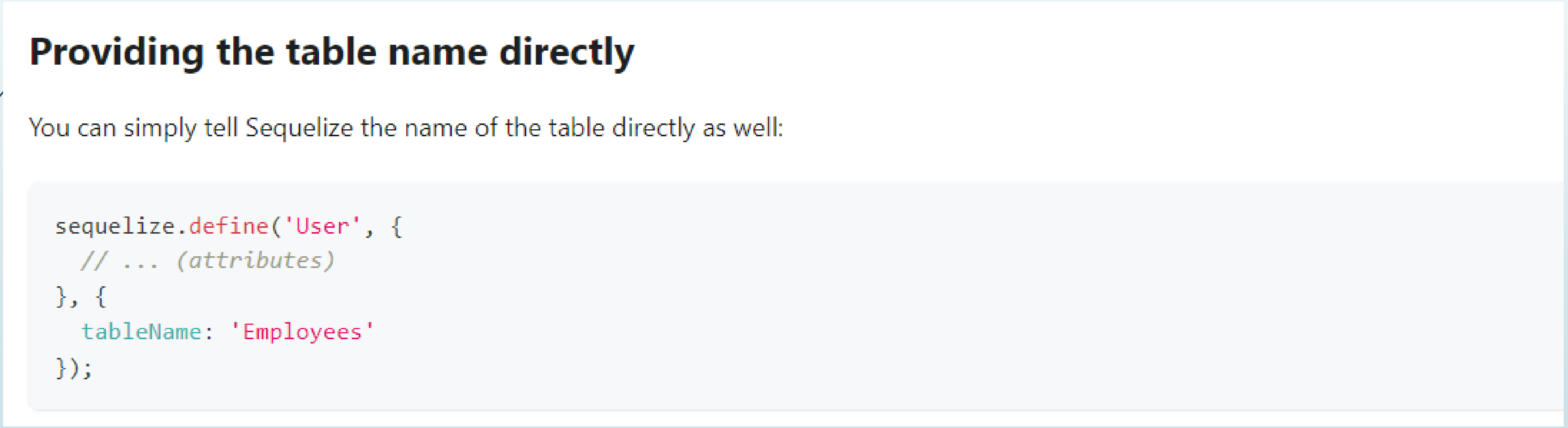


Table name inference

Observe that, in both methods above, the table name (`Users`) was never explicitly defined. However, the model name was given (`User`).

By default, when the table name is not given, Sequelize automatically pluralizes the model name and uses that as the table name. This pluralization is done under the hood by a library called inflection, so that irregular plurals (such as `person -> people`) are computed correctly.

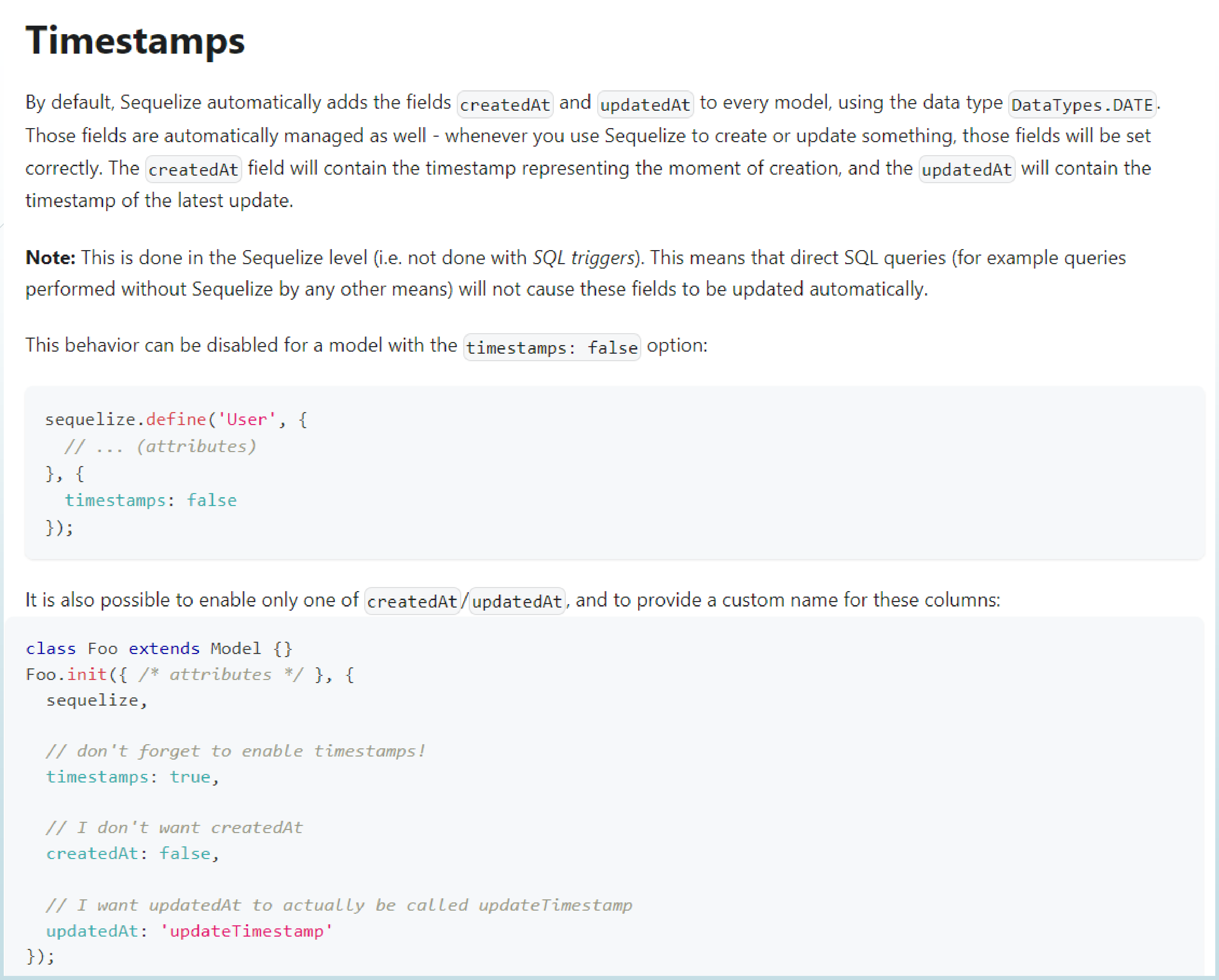
Of course, this behavior is easily configurable.



Providing the table name directly

You can simply tell Sequelize the name of the table directly as well:

```
sequelize.define('User', {
  // ... (attributes)
}, {
  tableName: 'Employees'
});
```



Timestamps

By default, Sequelize automatically adds the fields `createdAt` and `updatedAt` to every model, using the data type `DataTypes.DATE`. Those fields are automatically managed as well - whenever you use Sequelize to create or update something, those fields will be set correctly. The `createdAt` field will contain the timestamp representing the moment of creation, and the `updatedAt` will contain the timestamp of the latest update.

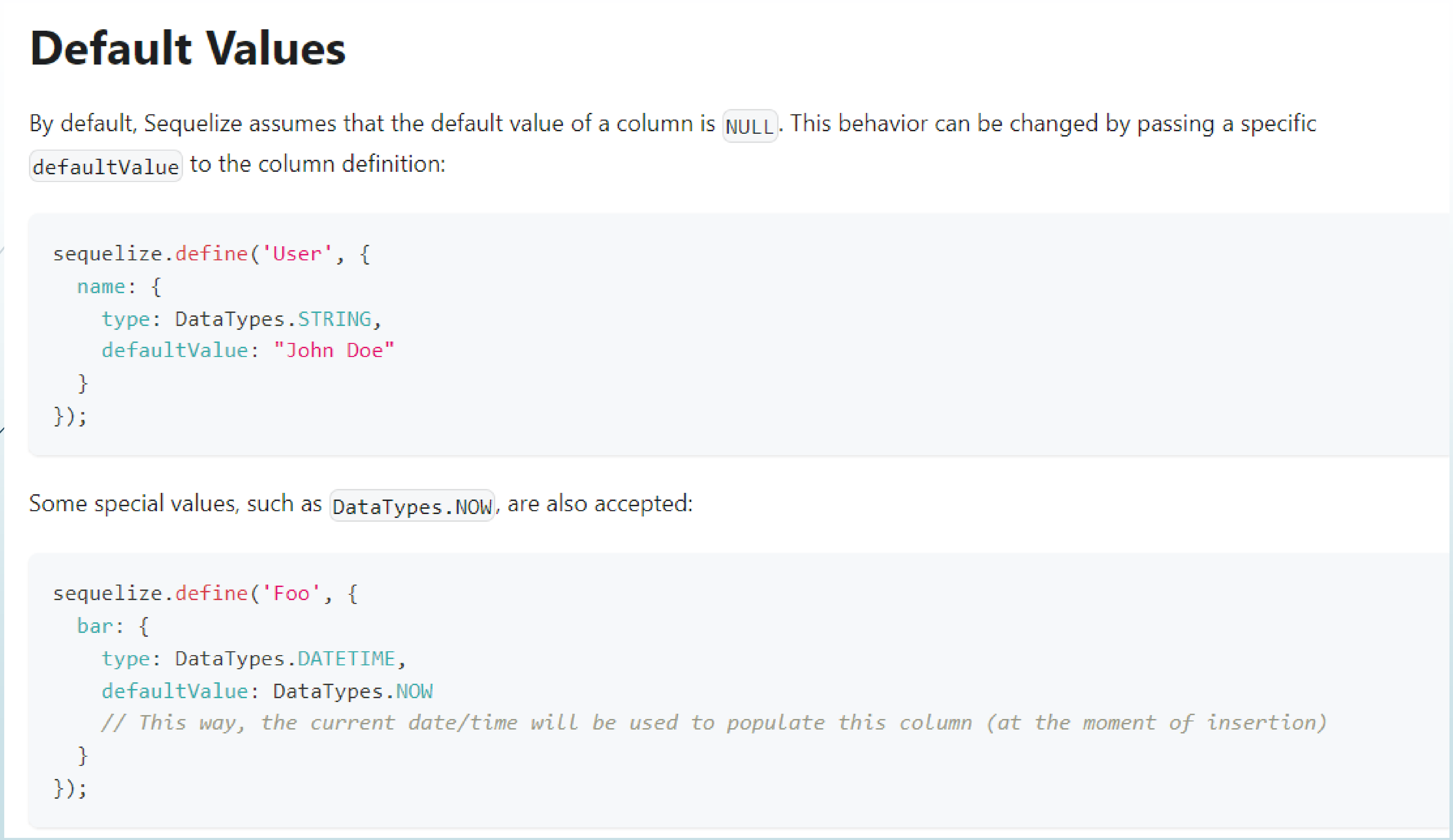
Note: This is done in the Sequelize level (i.e. not done with *SQL triggers*). This means that direct SQL queries (for example queries performed without Sequelize by any other means) will not cause these fields to be updated automatically.

This behavior can be disabled for a model with the `timestamps: false` option:

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

It is also possible to enable only one of `createdAt`/`updatedAt`, and to provide a custom name for these columns:

```
class Foo extends Model {}
Foo.init({ /* attributes */ }, {
  sequelize,

  // don't forget to enable timestamps!
  timestamps: true,

  // I don't want createdAt
  createdAt: false,

  // I want updatedAt to actually be called updateTimestamp
  updatedAt: 'updateTimestamp'
});
```



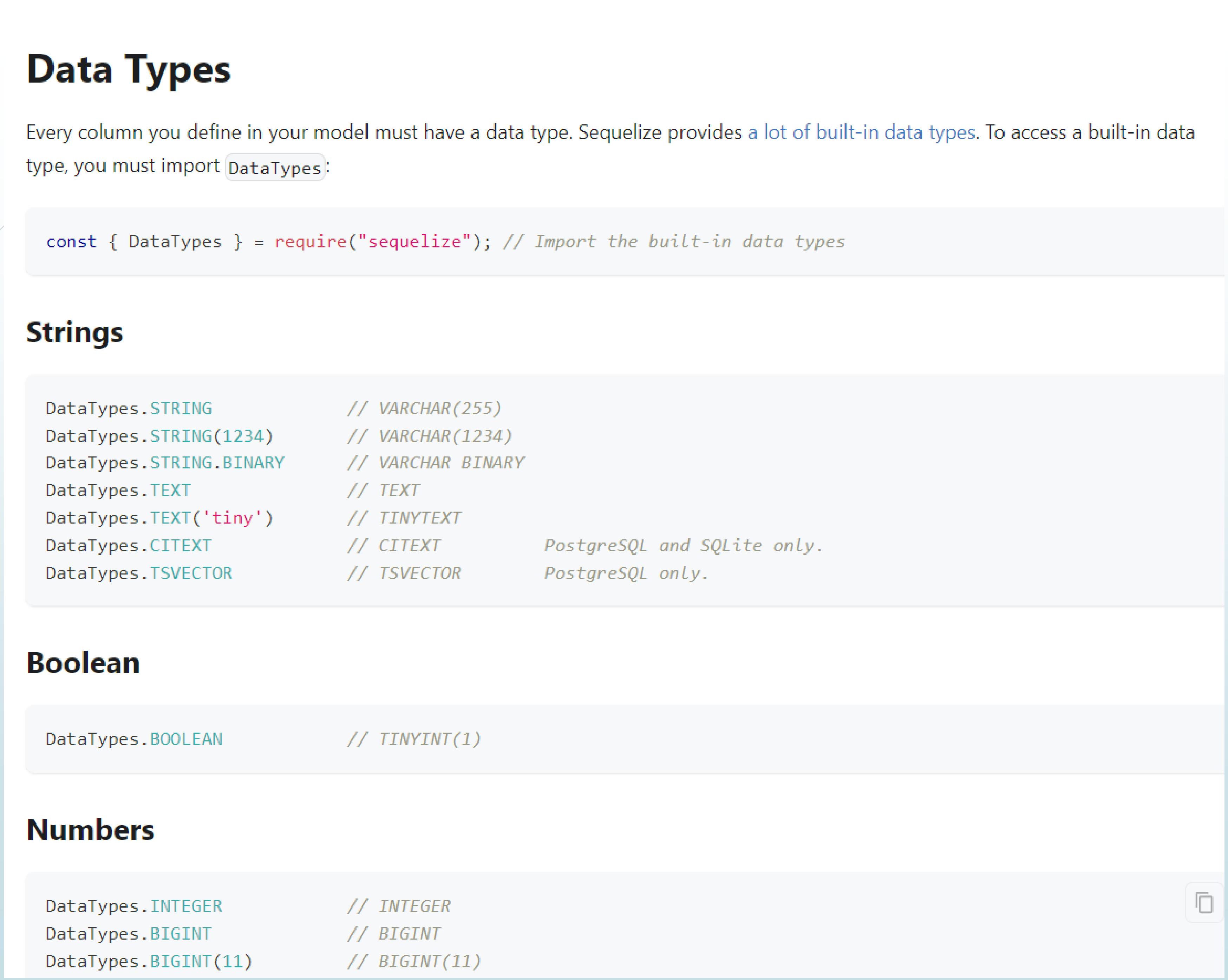
Default Values

By default, Sequelize assumes that the default value of a column is `NULL`. This behavior can be changed by passing a specific `defaultValue` to the column definition:

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

Some special values, such as `DataTypes.NOW`, are also accepted:

```
sequelize.define('Foo', {
  bar: {
    type: DataTypes.DATETIME,
    defaultValue: DataTypes.NOW
    // This way, the current date/time will be used to populate this column (at the moment of insertion)
  }
});
```



Data Types

Every column you define in your model must have a data type. Sequelize provides a lot of built-in data types. To access a built-in data type, you must import `DataTypes`:

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

Strings

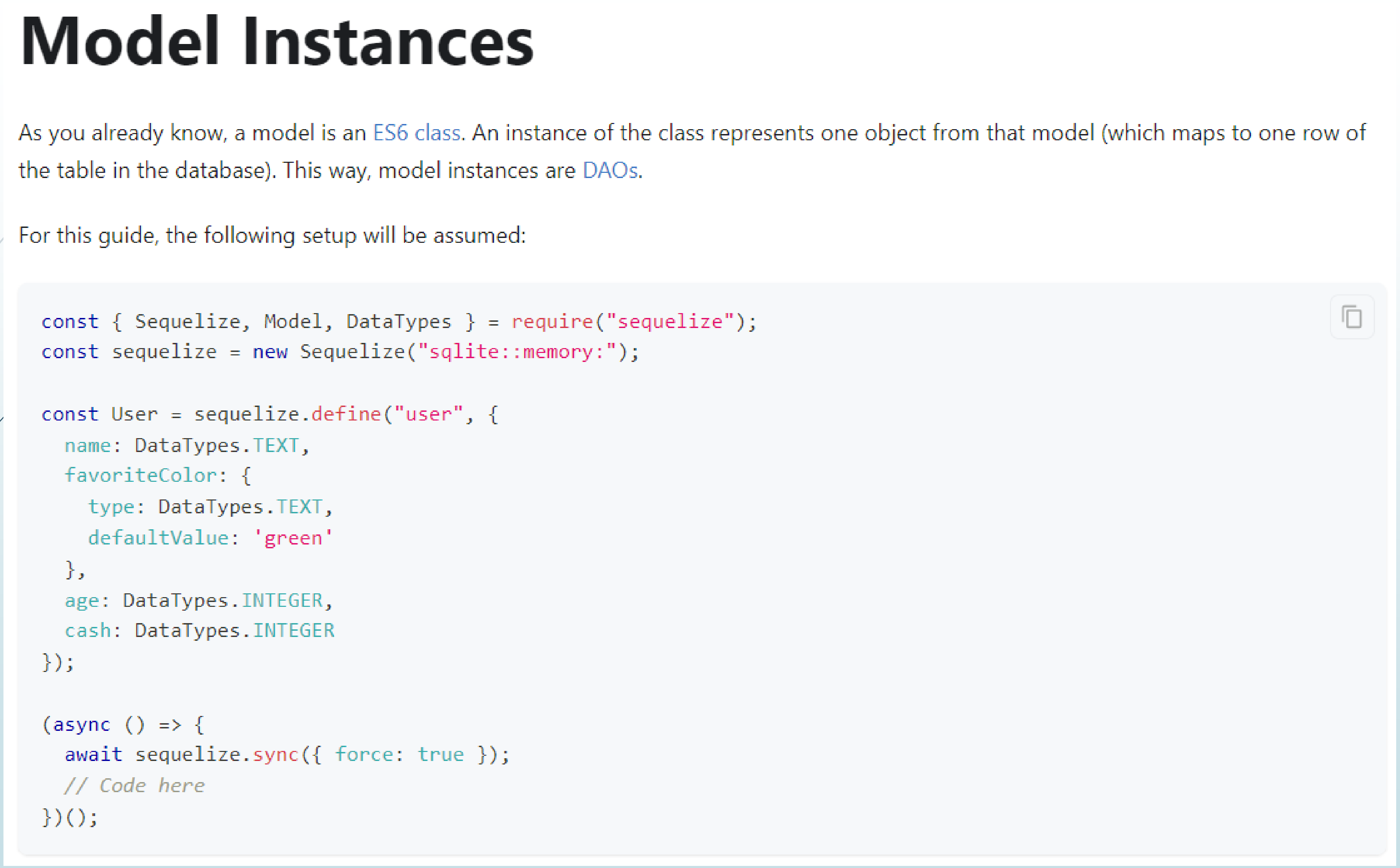
```
DataTypes.STRING             // VARCHAR(255)
DataTypes.STRING(1234)       // VARCHAR(1234)
DataTypes.STRING.BINARY      // VARCHAR BINARY
DataTypes.TEXT               // TEXT
DataTypes.TEXT('tiny')       // TINYTEXT
DataTypes.CITEXT             // CITEXT          PostgreSQL and SQLite only.
DataTypes.TSVECTOR           // TSVECTOR        PostgreSQL only.
```

Boolean

```
DataTypes.BOOLEAN            // TINYINT(1)
```

Numbers

```
DataTypes.INTEGER            // INTEGER
DataTypes.BIGINT             // BIGINT
DataTypes.BIGINT(11)         // BIGINT(11)
```

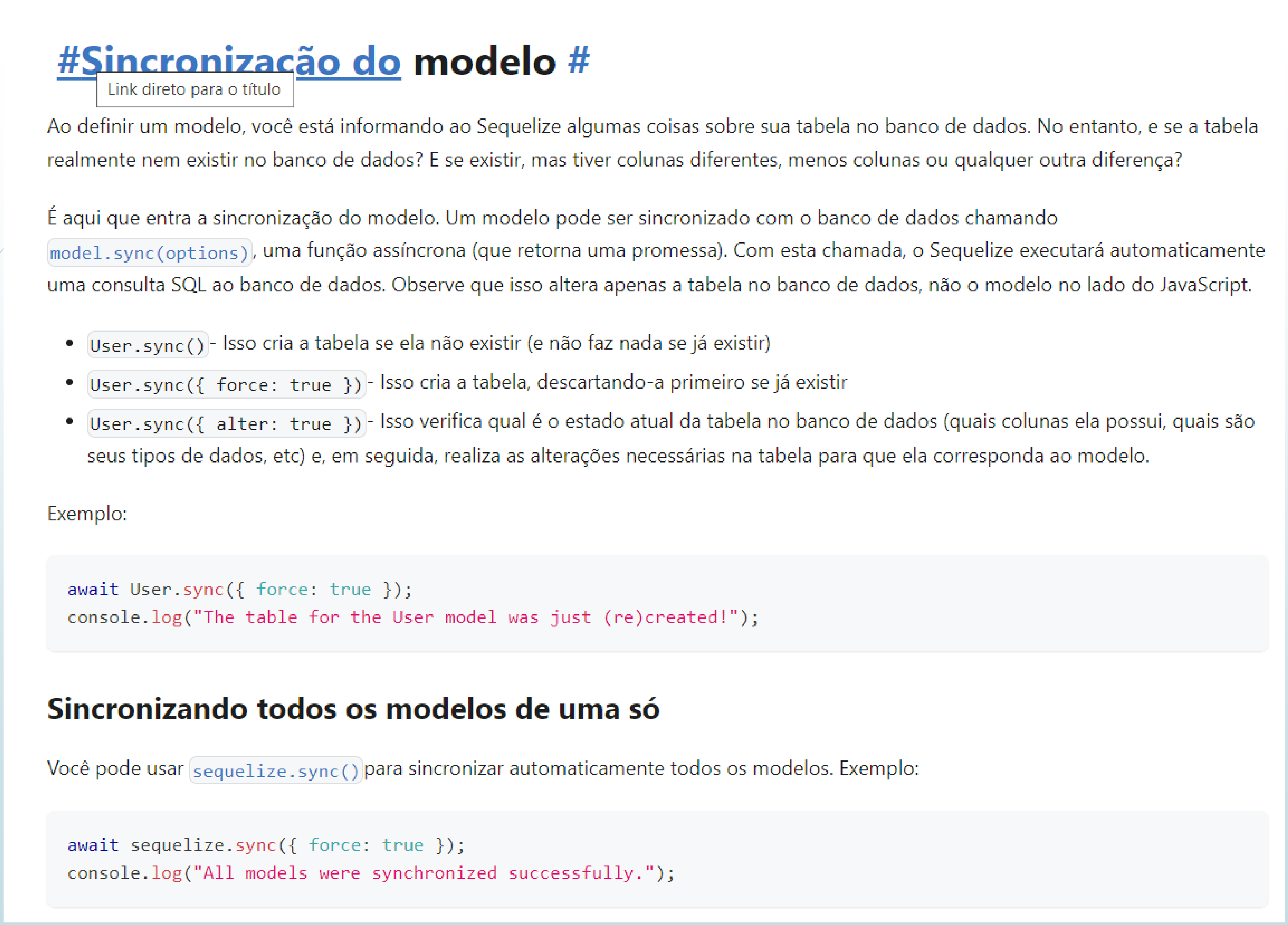


Model Instances

As you already know, a model is an ES6 class. An instance of the class represents one object from that model (which maps to one row of the table in the database). This way, model instances are DAOs.

For this guide, the following setup will be assumed:

```
const { Sequelize, Model, DataTypes } = require("sequelize");
const sequelize = new Sequelize("sqlite::memory:");

const User = sequelize.define("user", {
  name: DataTypes.TEXT,
  favoriteColor: {
    type: DataTypes.TEXT,
    defaultValue: 'green'
  },
  age: DataTypes.INTEGER,
  cash: DataTypes.INTEGER
});

(async () => {
  await sequelize.sync({ force: true });
  // Code here
})();
```



#Sincronização do modelo

Ao definir um modelo, você está informando ao Sequelize algumas coisas sobre sua tabela no banco de dados. No entanto, e se a tabela realmente nem existir no banco de dados? E se existir, mas tiver colunas diferentes, menos colunas ou qualquer outra diferença?

É aqui que entra a sincronização do modelo. Um modelo pode ser sincronizado com o banco de dados chamando `model.sync(options)`, uma função assíncrona (que retorna uma promessa). Com esta chamada, o Sequelize executará automaticamente uma consulta SQL ao banco de dados. Observe que isso altera apenas a tabela no banco de dados, não o modelo no lado do JavaScript.

- `User.sync()`- Isso cria a tabela se ela não existir (e não faz nada se já existir)
- `User.sync({ force: true })`- Isso cria a tabela, descartando-a primeiro se já existir
- `User.sync({ alter: true })`- Isso verifica qual é o estado atual da tabela no banco de dados (quais colunas ela possui, quais são seus tipos de dados, etc) e, em seguida, realiza as alterações necessárias na tabela para que ela corresponda ao modelo.

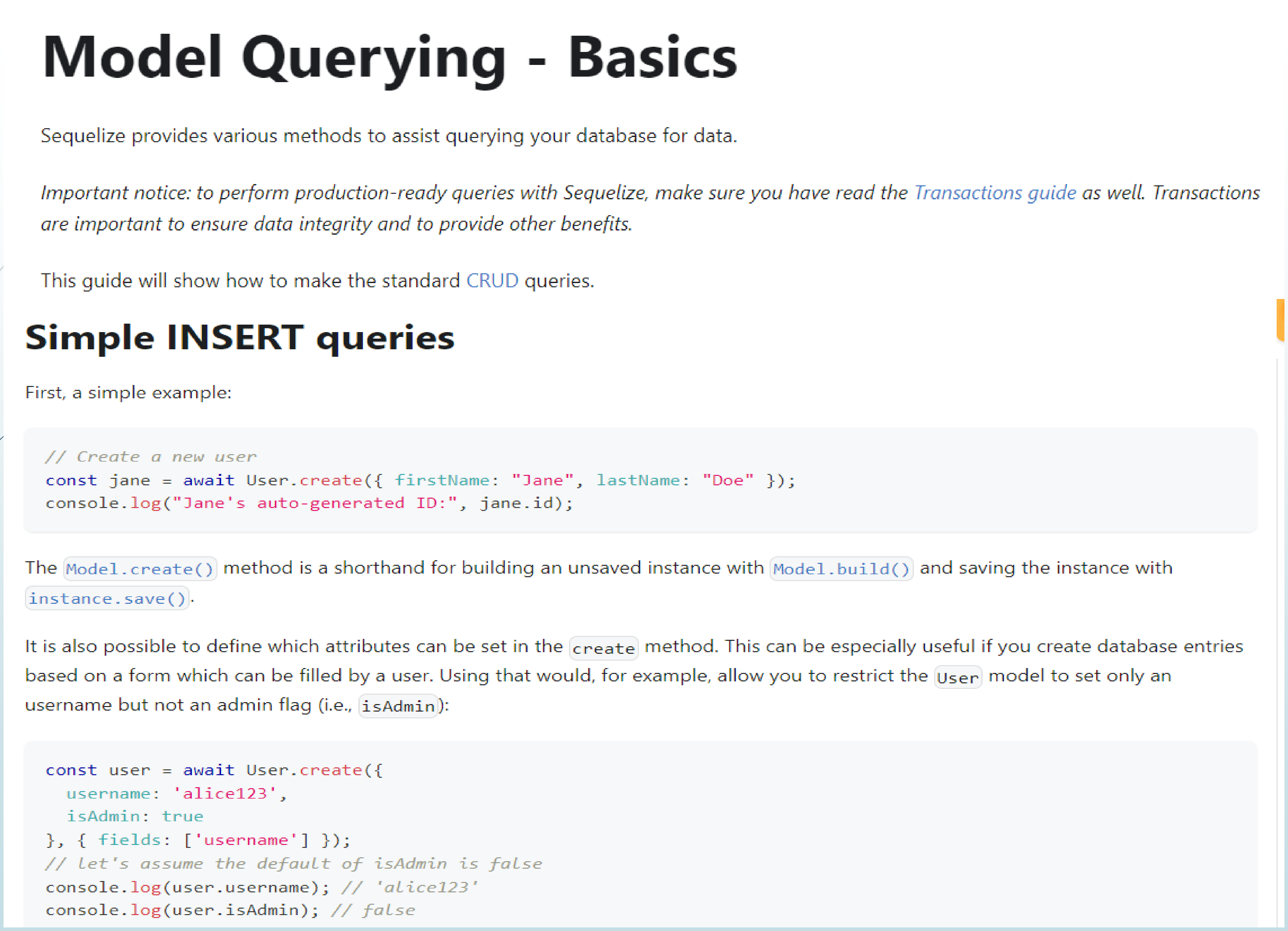
Exemplo:

```
await User.sync({ force: true });
console.log("The table for the User model was just (re)created!");
```

Sincronizando todos os modelos de uma só

Você pode usar `sequelize.sync()`para sincronizar automaticamente todos os modelos. Exemplo:

```
await sequelize.sync({ force: true });
console.log("All models were synchronized successfully.");
```



Model Querying - Basics

Sequelize provides various methods to assist querying your database for data.

Important notice: to perform production-ready queries with Sequelize, make sure you have read the Transactions guide as well. Transactions are important to ensure data integrity and to provide other benefits.

This guide will show how to make the standard CRUD queries.

Simple INSERT queries

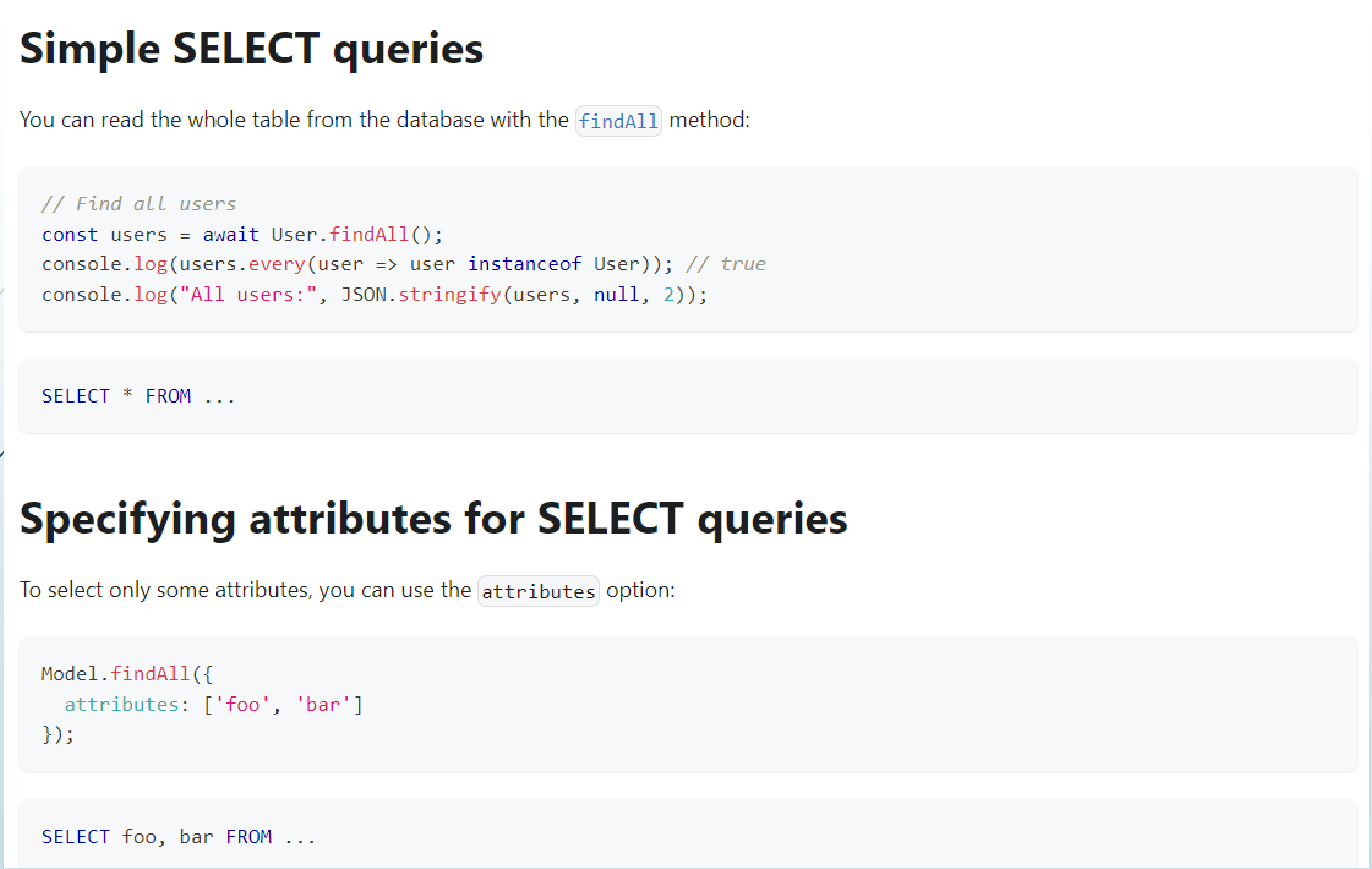
First, a simple example:

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

The `Model.create()` method is a shorthand for building an unsaved instance with `Model.build()` and saving the instance with `instance.save()`.

It is also possible to define which attributes can be set in the `create` method. This can be especially useful if you create database entries based on a form which can be filled by a user. Using that would, for example, allow you to restrict the `User` model to set only an username but not an admin flag (i.e., `isAdmin`):

```
const user = await User.create({
  username: 'alice123',
  isAdmin: true
}, { fields: ['username'] });
// let's assume the default of isAdmin is false
console.log(user.username); // 'alice123'
console.log(user.isAdmin); // false
```



Simple SELECT queries

You can read the whole table from the database with the `findAll` method:

```
// Find all users
const users = await User.findAll();
console.log(users.every(user => user instanceof User)); // true
console.log("All users:", JSON.stringify(users, null, 2));
```

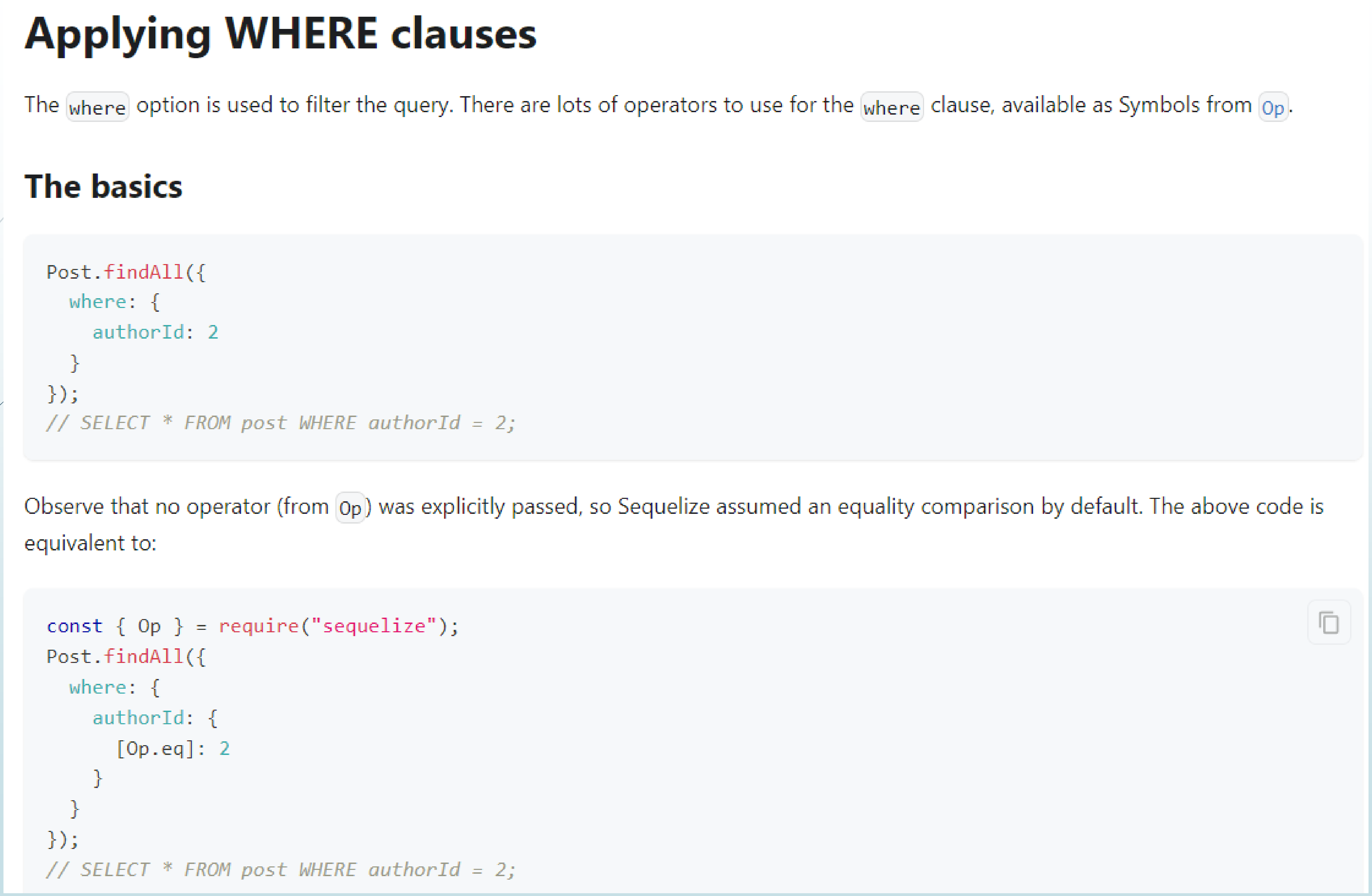
```
SELECT * FROM ...
```

Specifying attributes for SELECT queries

To select only some attributes, you can use the `attributes` option:

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

```
SELECT foo, bar FROM ...
```



Applying WHERE clauses

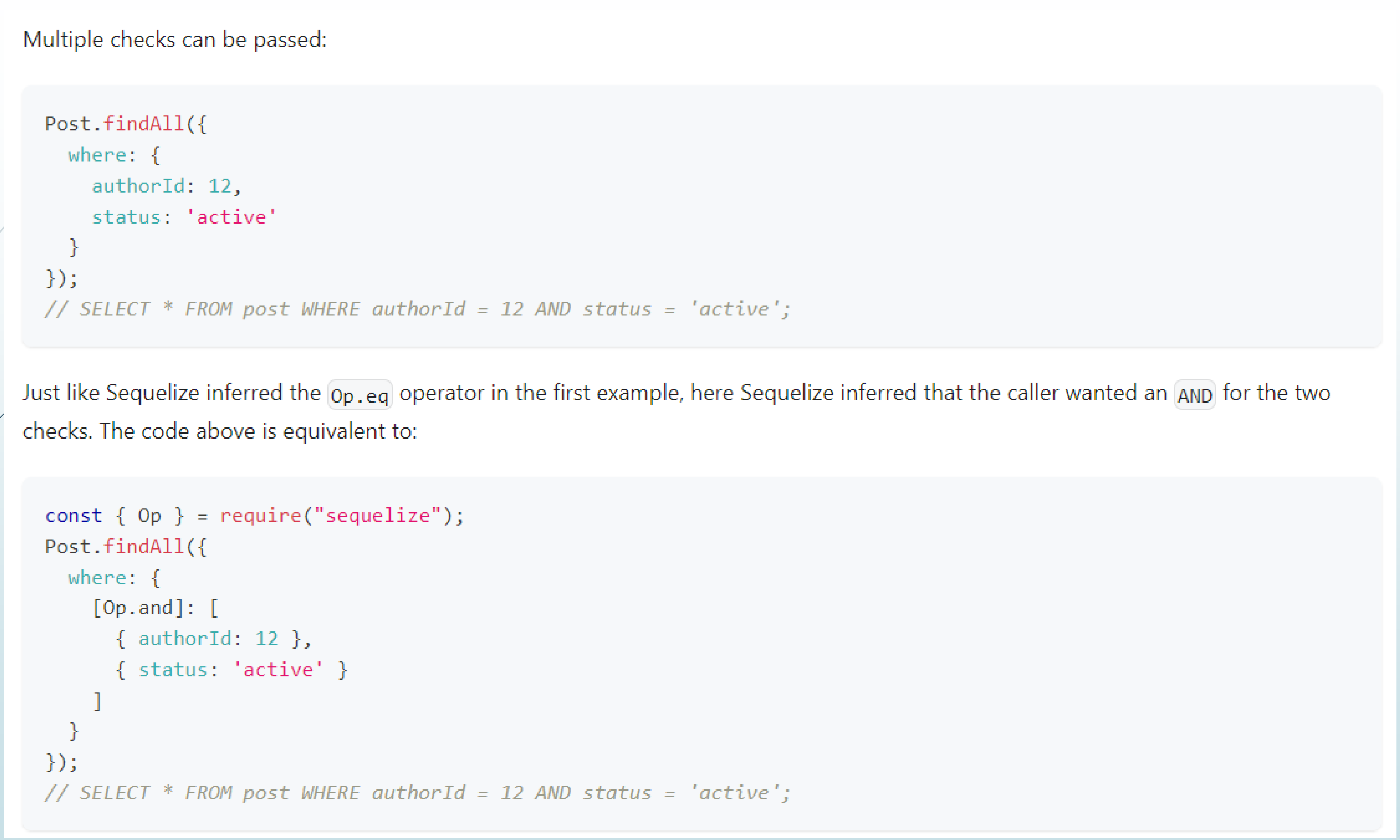
The `where` option is used to filter the query. There are lots of operators to use for the `where` clause, available as Symbols from `Op`.

The basics

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

Observe that no operator (from `Op`) was explicitly passed, so Sequelize assumed an equality comparison by default. The above code is equivalent to:

```
const { Op } = require("sequelize");
Post.findAll({
  where: {
    authorId: {
      [Op.eq]: 2
    }
  }
});
// SELECT * FROM post WHERE authorId = 2;
```

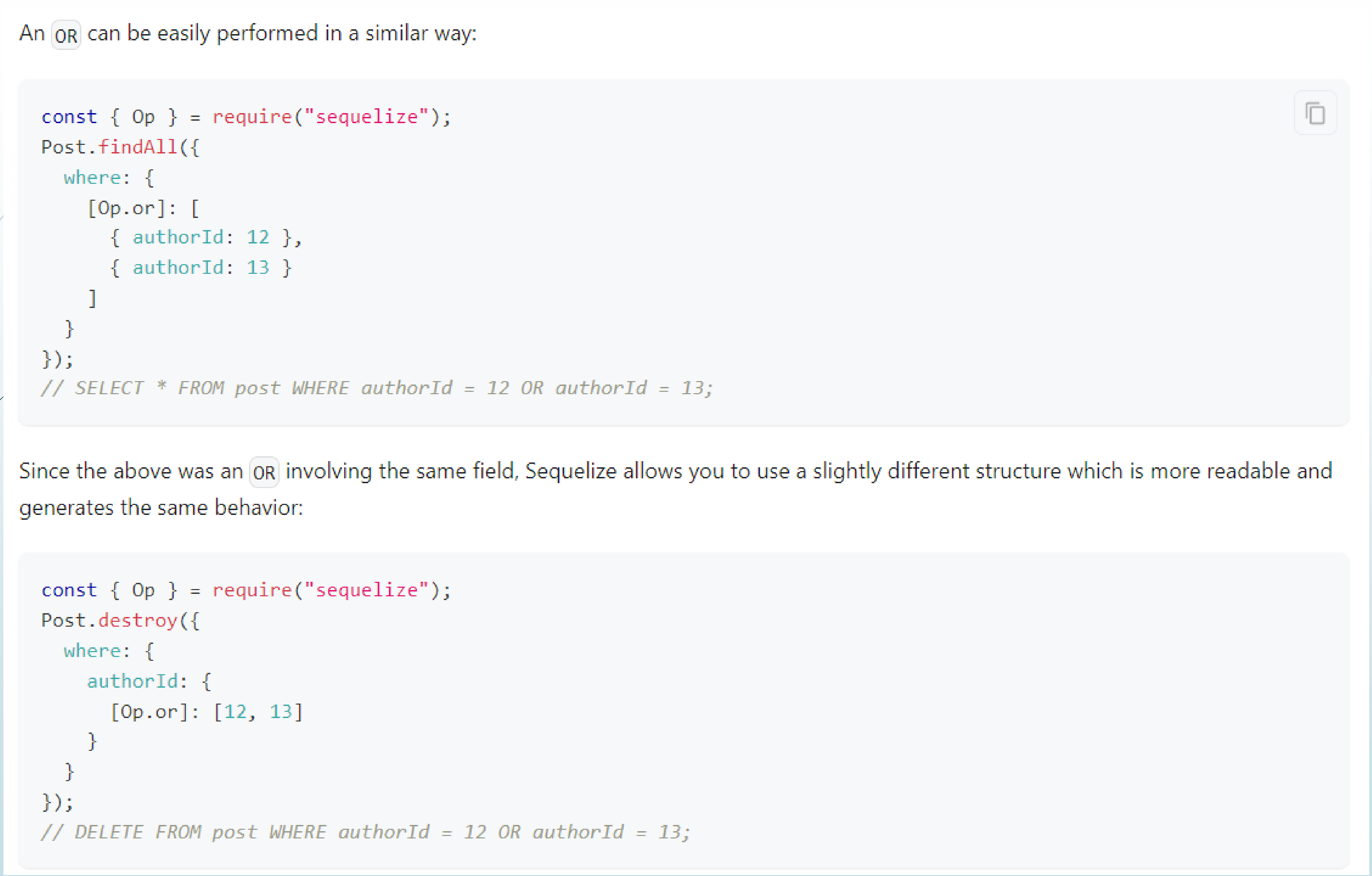


Multiple checks can be passed:

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

Just like Sequelize inferred the `Op.eq` operator in the first example, here Sequelize inferred that the caller wanted an `AND` for the two checks. The code above is equivalent to:

```
const { Op } = require("sequelize");
Post.findAll({
  where: {
    [Op.and]: [
      { authorId: 12 },
      { status: 'active' }
    ]
  }
});
// SELECT * FROM post WHERE authorId = 12 AND status = 'active';
```

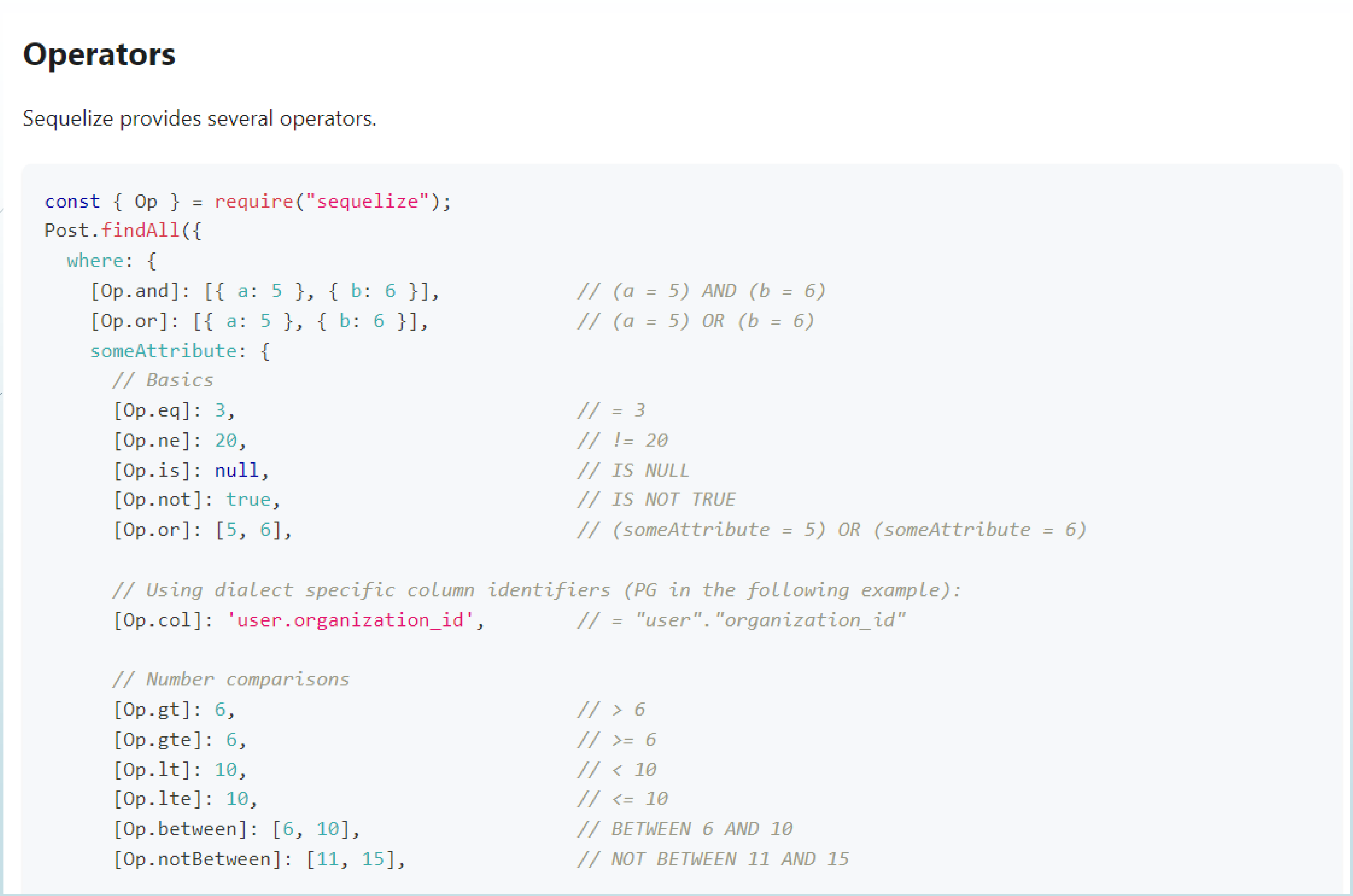


An `OR` can be easily performed in a similar way:

```
const { Op } = require("sequelize");
Post.findAll({
  where: {
    [Op.or]: [
      { authorId: 12 },
      { authorId: 13 }
    ]
  }
});
// SELECT * FROM post WHERE authorId = 12 OR authorId = 13;
```

Since the above was an `OR` involving the same field, Sequelize allows you to use a slightly different structure which is more readable and generates the same behavior:

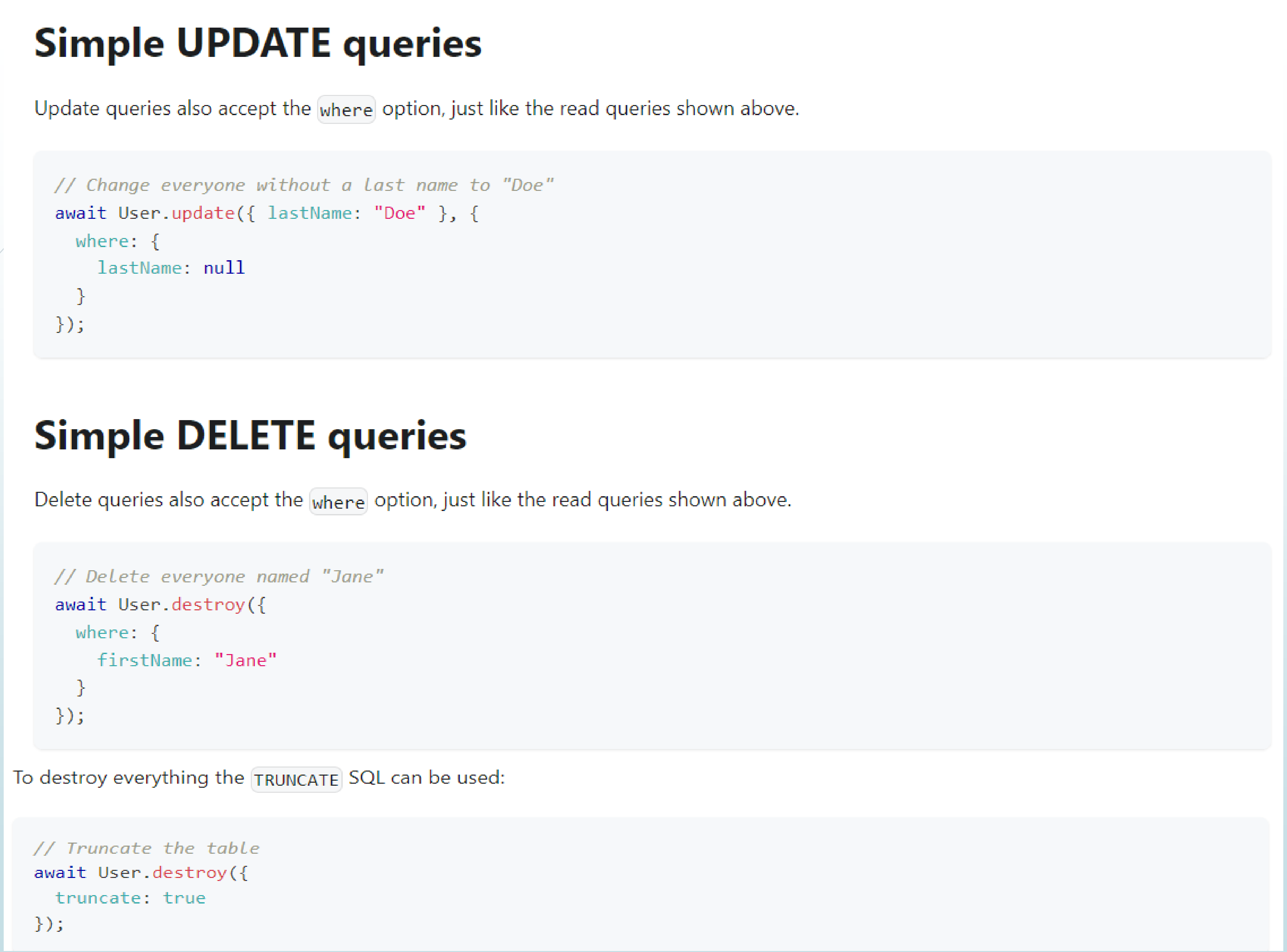
```
const { Op } = require("sequelize");
Post.destroy({
  where: {
    authorId: {
      [Op.or]: [12, 13]
    }
  }
});
// DELETE FROM post WHERE authorId = 12 OR authorId = 13;
```



Operators

Sequelize provides several operators.

```
const { Op } = require("sequelize");
Post.findAll({
  where: {
    [Op.and]: [{ a: 5 }, { b: 6 }],            // (a = 5) AND (b = 6)
    [Op.or]: [{ a: 5 }, { b: 6 }],             // (a = 5) OR (b = 6)
    someAttribute: {
      // Basics
      [Op.eq]: 3,                              // = 3
      [Op.ne]: 20,                             // != 20
      [Op.is]: null,                           // IS NULL
      [Op.not]: true,                          // IS NOT TRUE
      [Op.or]: [5, 6],                         // (someAttribute = 5) OR (someAttribute = 6)

      // Using dialect specific column identifiers (PG in the following example):
      [Op.col]: 'user.organization_id',        // = "user"."organization_id"

      // Number comparisons
      [Op.gt]: 6,                              // > 6
      [Op.gte]: 6,                             // >= 6
      [Op.lt]: 10,                             // < 10
      [Op.lte]: 10,                            // <= 10
      [Op.between]: [6, 10],                   // BETWEEN 6 AND 10
      [Op.notBetween]: [11, 15],               // NOT BETWEEN 11 AND 15
```



Simple UPDATE queries

Update queries also accept the `where` option, just like the read queries shown above.

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

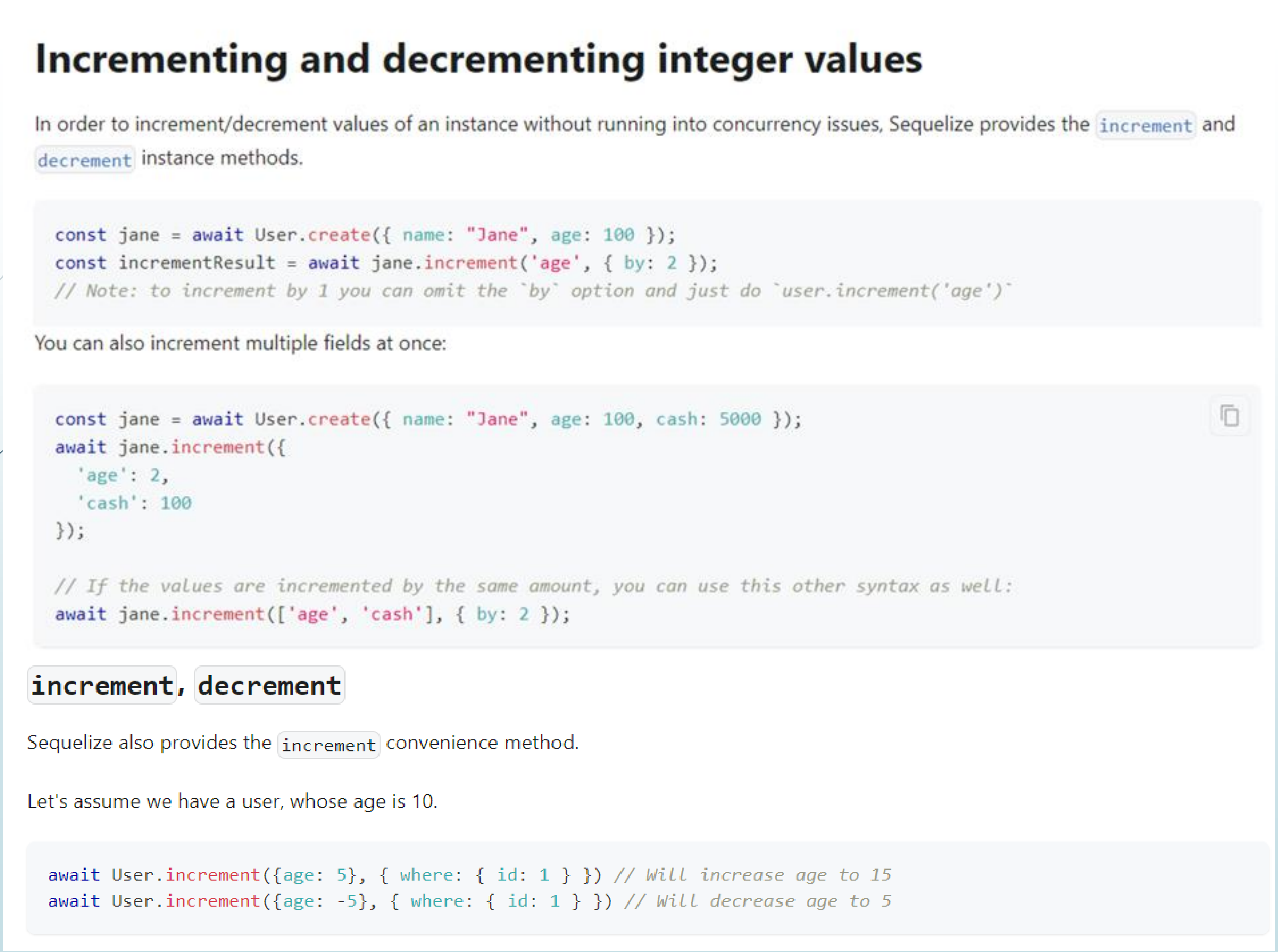
Simple DELETE queries

Delete queries also accept the `where` option, just like the read queries shown above.

```
// Delete everyone named "Jane"
await User.destroy({
  where: {
    firstName: "Jane"
  }
});
```

To destroy everything the `TRUNCATE` SQL can be used:

```
// Truncate the table
await User.destroy({
  truncate: true
});
```



Incrementing and decrementing integer values

In order to increment/decrement values of an instance without running into concurrency issues, Sequelize provides the `increment` and `decrement` instance methods.

```
const jane = await User.create({ name: "Jane", age: 100 });
const incrementResult = await jane.increment('age', { by: 2 });
// Note: to increment by 1 you can omit the `by` option and just do `user.increment('age')`
```

You can also increment multiple fields at once:

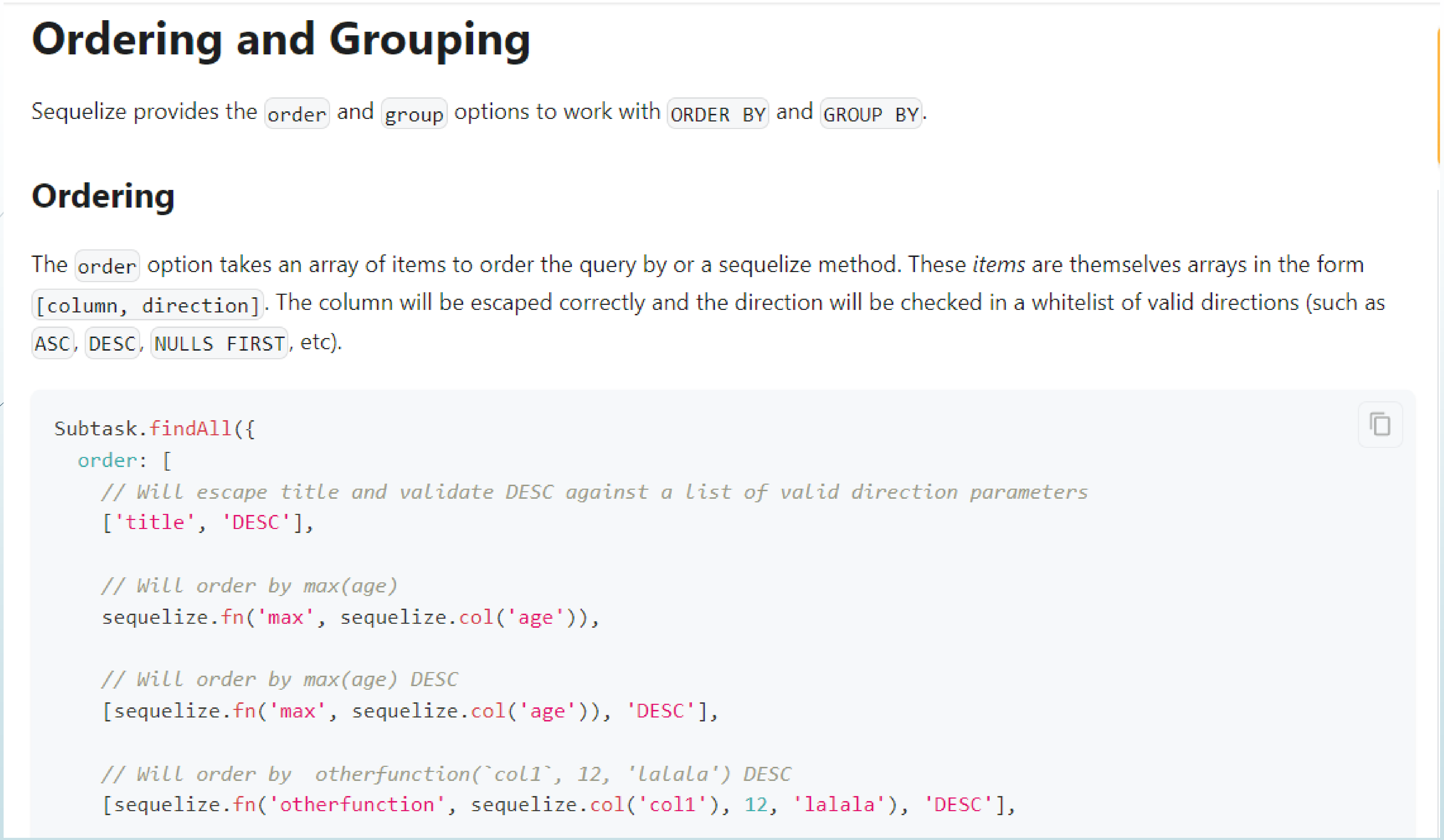
```
const jane = await User.create({ name: "Jane", age: 100, cash: 5000 });
await jane.increment({
  'age': 2,
  'cash': 100
});

// If the values are incremented by the same amount, you can use this other syntax as well:
await jane.increment(['age', 'cash'], { by: 2 });
```

`increment`, `decrement`

Sequelize also provides the `increment` convenience method.

Let's assume we have a user, whose age is 10.

```
await User.increment({age: 5}, { where: { id: 1 } }) // Will increase age to 15
await User.increment({age: -5}, { where: { id: 1 } }) // Will decrease age to 5
```



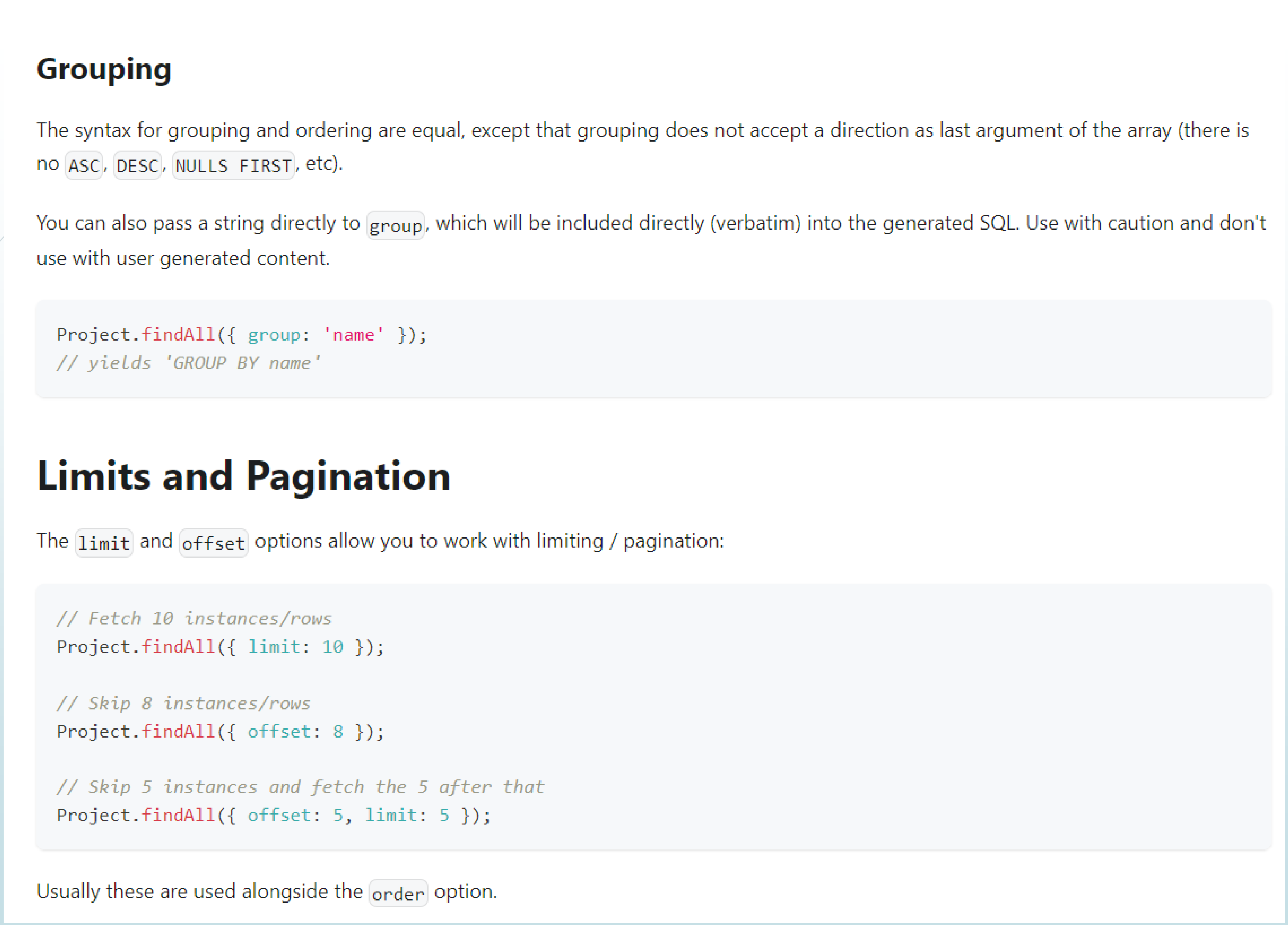
Ordering and Grouping

Sequelize provides the `order` and `group` options to work with `ORDER BY` and `GROUP BY`.

Ordering

The `order` option takes an array of items to order the query by or a sequelize method. These *items* are themselves arrays in the form `[column, direction]`. The column will be escaped correctly and the direction will be checked in a whitelist of valid directions (such as `ASC`, `DESC`, `NULLS FIRST`, etc).

```
Subtask.findAll({
  order: [
    // Will escape title and validate DESC against a list of valid direction parameters
    ['title', 'DESC'],

    // Will order by max(age)
    sequelize.fn('max', sequelize.col('age')),

    // Will order by max(age) DESC
    [sequelize.fn('max', sequelize.col('age')), 'DESC'],

    // Will order by  otherfunction(`col1`, 12, 'lalala') DESC
    [sequelize.fn('otherfunction', sequelize.col('col1'), 12, 'lalala'), 'DESC'],
```



Grouping

The syntax for grouping and ordering are equal, except that grouping does not accept a direction as last argument of the array (there is no `ASC`, `DESC`, `NULLS FIRST`, etc).

You can also pass a string directly to `group`, which will be included directly (verbatim) into the generated SQL. Use with caution and don't use with user generated content.

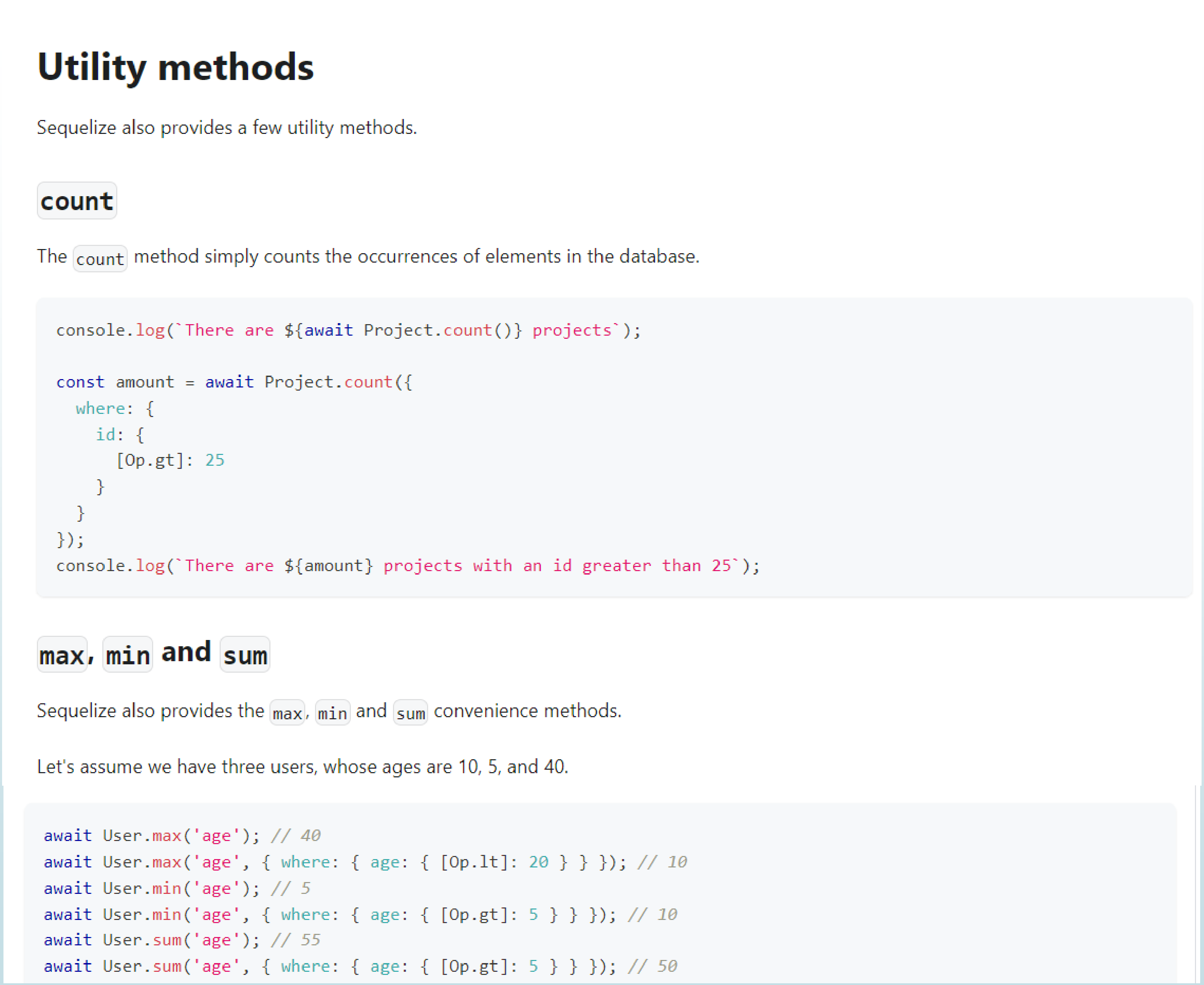
```
Project.findAll({ group: 'name' });
// yields 'GROUP BY name'
```

Limits and Pagination

The `limit` and `offset` options allow you to work with limiting / pagination:

```
// Fetch 10 instances/rows
Project.findAll({ limit: 10 });

// Skip 8 instances/rows
Project.findAll({ offset: 8 });

// Skip 5 instances and fetch the 5 after that
Project.findAll({ offset: 5, limit: 5 });
```

Usually these are used alongside the `order` option.



Utility methods

Sequelize also provides a few utility methods.

`count`

The `count` method simply counts the occurrences of elements in the database.

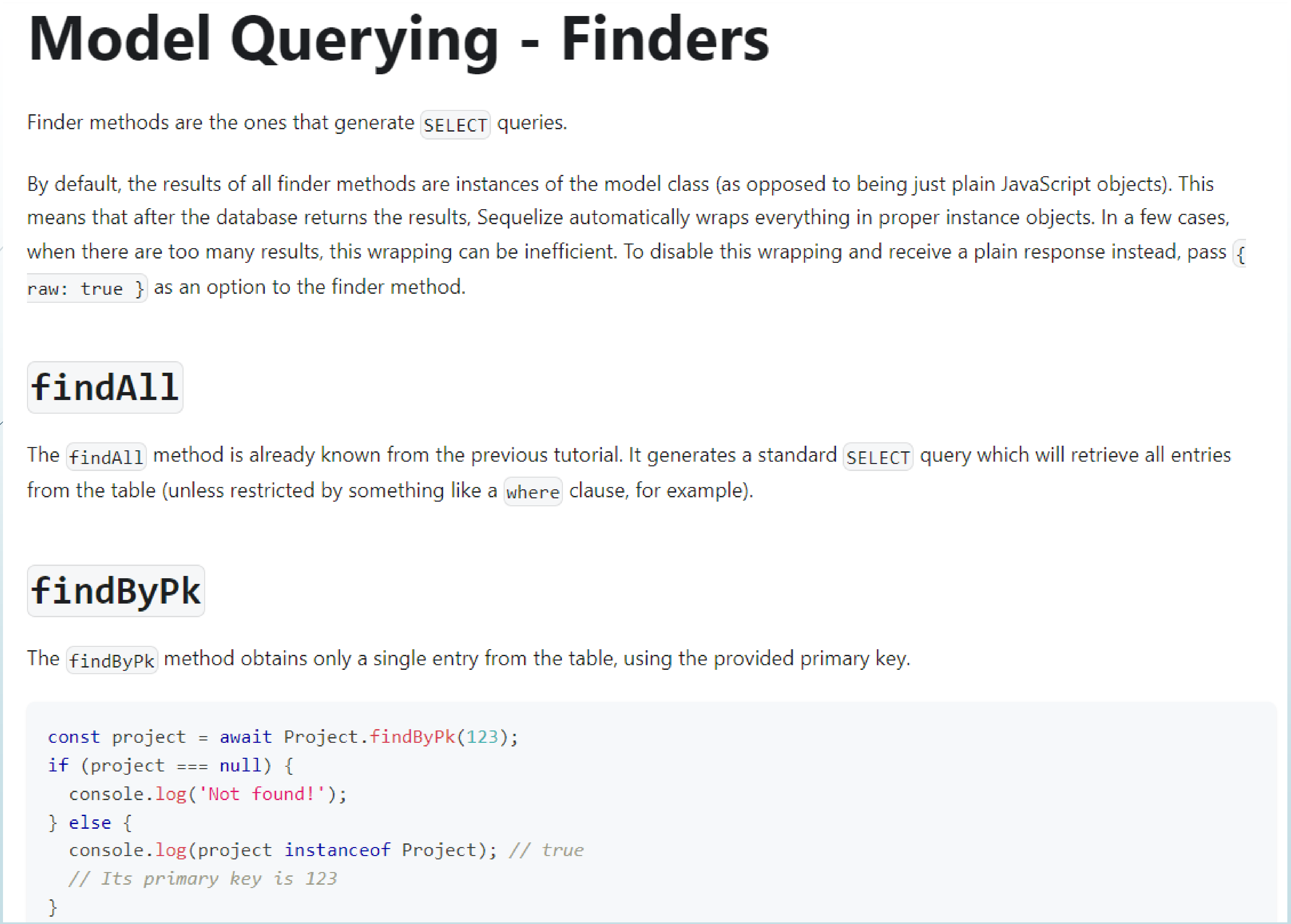
```
console.log(`There are ${await Project.count()} projects`);

const amount = await Project.count({
  where: {
    id: {
      [Op.gt]: 25
    }
  }
});
console.log(`There are ${amount} projects with an id greater than 25`);
```

`max`, `min` and `sum`

Sequelize also provides the `max`, `min` and `sum` convenience methods.

Let's assume we have three users, whose ages are 10, 5, and 40.

```
await User.max('age'); // 40
await User.max('age', { where: { age: { [Op.lt]: 20 } } }); // 10
await User.min('age'); // 5
await User.min('age', { where: { age: { [Op.gt]: 5 } } }); // 10
await User.sum('age'); // 55
await User.sum('age', { where: { age: { [Op.gt]: 5 } } }); // 50
```



Model Querying - Finders

Finder methods are the ones that generate `SELECT` queries.

By default, the results of all finder methods are instances of the model class (as opposed to being just plain JavaScript objects). This means that after the database returns the results, Sequelize automatically wraps everything in proper instance objects. In a few cases, when there are too many results, this wrapping can be inefficient. To disable this wrapping and receive a plain response instead, pass `{ raw: true }` as an option to the finder method.

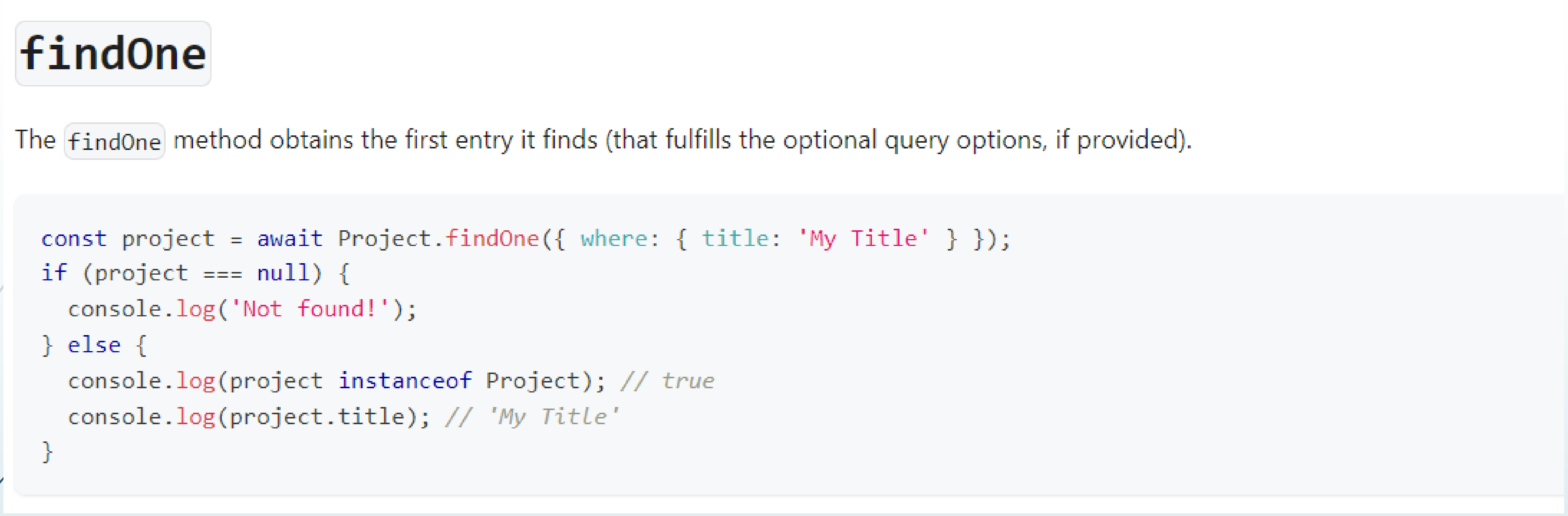
`findAll`

The `findAll` method is already known from the previous tutorial. It generates a standard `SELECT` query which will retrieve all entries from the table (unless restricted by something like a `where` clause, for example).

`findByPk`

The `findByPk` method obtains only a single entry from the table, using the provided primary key.

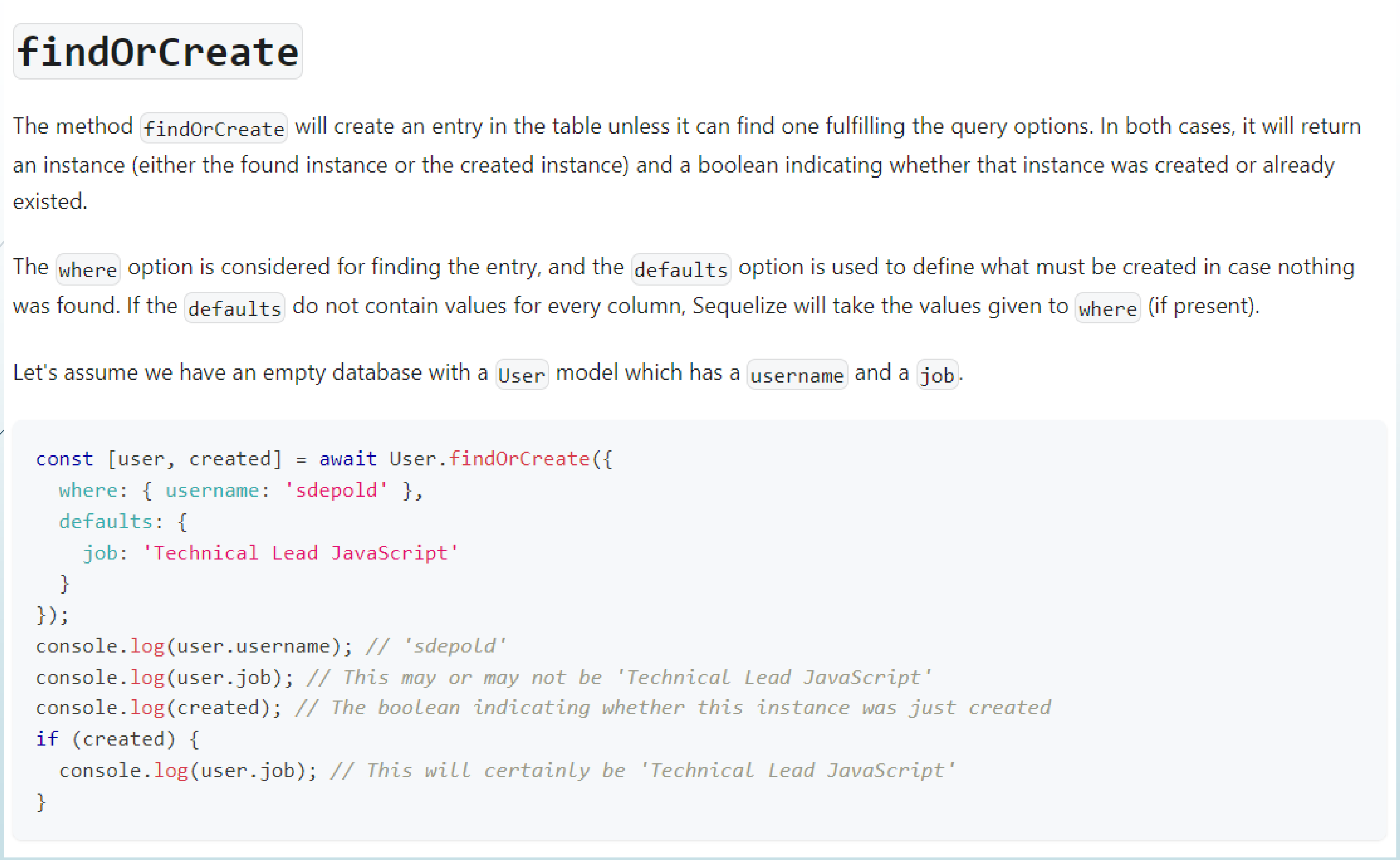
```
const project = await Project.findByPk(123);
if (project === null) {
  console.log('Not found!');
} else {
  console.log(project instanceof Project); // true
  // Its primary key is 123
}
```



`findOne`

The `findOne` method obtains the first entry it finds (that fulfills the optional query options, if provided).

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



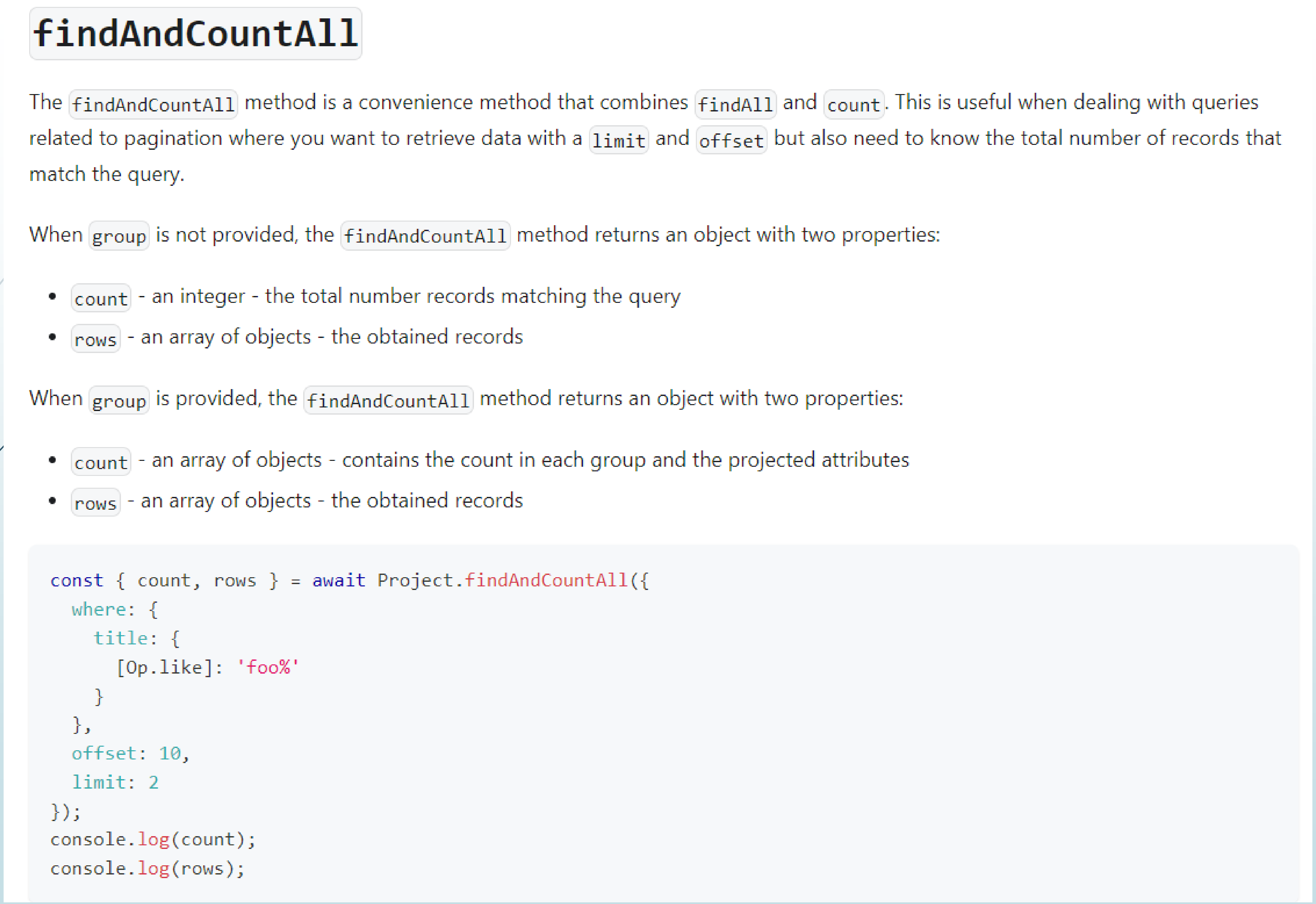
`findOrCreate`

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

The `where` option is considered for finding the entry, and the `defaults` option is used to define what must be created in case nothing was found. If the `defaults` do not contain values for every column, Sequelize will take the values given to `where` (if present).

Let's assume we have an empty database with a `User` model which has a `username` and a `job`.

```
const [user, created] = await User.findOrCreate({
  where: { username: 'sdepold' },
  defaults: {
    job: 'Technical Lead JavaScript'
  }
});
console.log(user.username); // 'sdepold'
console.log(user.job); // This may or may not be 'Technical Lead JavaScript'
console.log(created); // The boolean indicating whether this instance was just created
if (created) {
  console.log(user.job); // This will certainly be 'Technical Lead JavaScript'
}
```



`findAndCountAll`

The `findAndCountAll` method is a convenience method that combines `findAll` and `count`. This is useful when dealing with queries related to pagination where you want to retrieve data with a `limit` and `offset` but also need to know the total number of records that match the query.

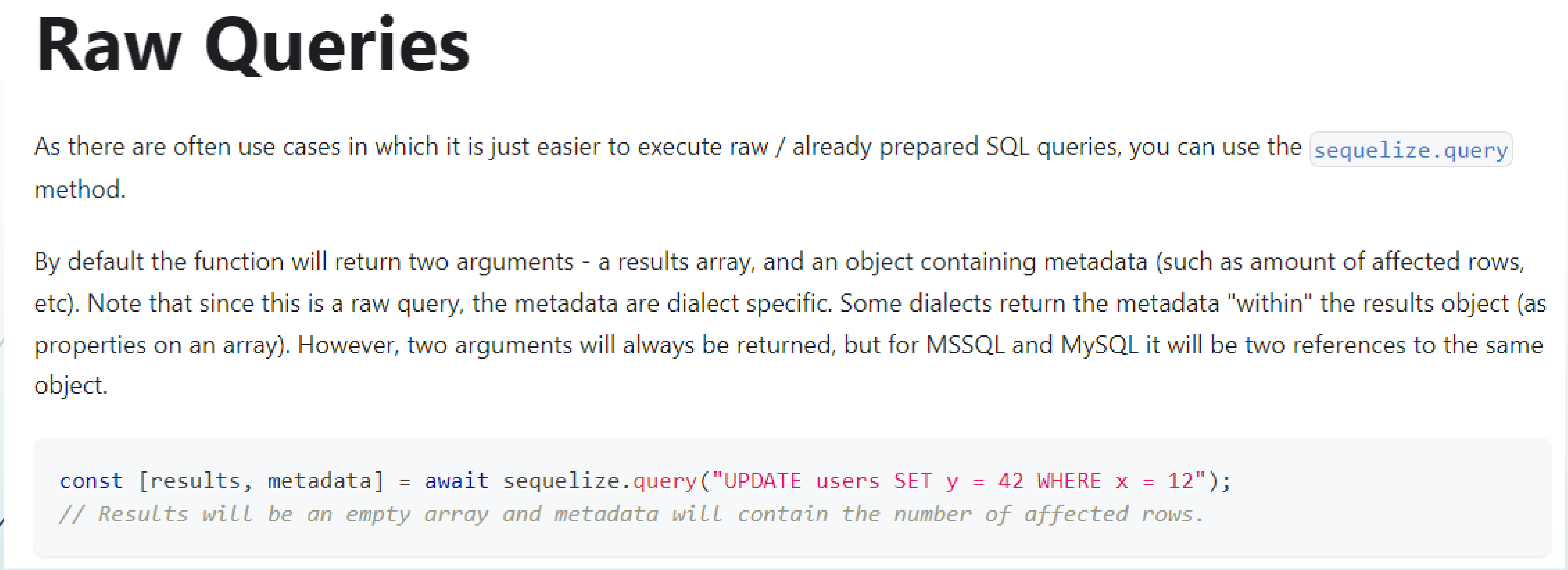
When `group` is not provided, the `findAndCountAll` method returns an object with two properties:

- `count` - an integer - the total number records matching the query
- `rows` - an array of objects - the obtained records

When `group` is provided, the `findAndCountAll` method returns an object with two properties:

- `count` - an array of objects - contains the count in each group and the projected attributes
- `rows` - an array of objects - the obtained records

```
const { count, rows } = await Project.findAndCountAll({
  where: {
    title: {
      [Op.like]: 'foo%'
    }
  },
  offset: 10,
  limit: 2
});
console.log(count);
console.log(rows);
```



Raw Queries

As there are often use cases in which it is just easier to execute raw / already prepared SQL queries, you can use the `sequelize.query` method.

By default the function will return two arguments - a results array, and an object containing metadata (such as amount of affected rows, etc). Note that since this is a raw query, the metadata are dialect specific. Some dialects return the metadata "within" the results object (as properties on an array). However, two arguments will always be returned, but for MSSQL and MySQL it will be two references to the same object.

```
const [results, metadata] = await sequelize.query("UPDATE users SET y = 42 WHERE x = 12");
// Results will be an empty array and metadata will contain the number of affected rows.
```

Exercícios

- Criar um novo projeto, com a model Jogador, contendo os atributos nome, clube, posição, idade e salário. Ajustar para que o nome da tabela no banco fique Jogadores.
- Criar os métodos para realizar o CRUD básico dos dados.
- Criar os seguintes métodos de filtro:
  - Que o nome contenha a string passada como parâmetro
  - Que a idade esteja dentro do intervalo de parâmetros passados
  - Que a string passada como parâmetro conste no nome do jogador, do clube ou na posição
- Criar um método que retorne os seguintes dados estatísticos:
  - Número de jogadores cadastrados
  - Soma dos salários
  - Média de idade
- Criar um método que quando chamado, aumente a idade de todos os jogadores em 1 ano.
- Criar um método que retorne nome e salário dos jogadores em ordem decrescente de salário.
- Alterar a estrutura da model (e da tabela) acrescentando o atributo nacionalidade – que deve conter o valor default 'Brasileira"
- Testar todos os métodos a partir do Insomnia.