Sequelize CRUD Actions Cheat Sheet

For the examples outlined in this guide, we'll be using this User model:

```js
module.exports = function(sequelize, DataTypes) {
  var User = sequelize.define("User", {
    email: DataTypes.STRING,
    password: DataTypes.STRING,
    age: DataTypes.INTEGER,
    name: DataTypes.STRING
  });
  return User;
};
```

If this is your first time using Sequelize, refer to the Sequelize Quick Start Guide for assistance with getting a project set up with Sequelize.

Otherwise proceed to the next page.

Part One: CREATE

In this example we create a new User in our database.

```
1    var db = require("./models");
2
3    db.User.create({
4        email: "tom@myspace.com",
5        password: "superinsecurepassword123",
6        age: 46,
7        name: "Tom Anderson"
8    }).then(function(dbUser) {
9        console.log(dbUser);
10   });
```

- On **line 1** we require our models.
- On **line 3** we call the Sequelize model "create" method. All Sequelize models have access to this method.
  - create takes in at least one argument: An object containing key value pairs describing the new record we want to create.
- On **line 8** we chain a promise with the result
  - "then" is a function that is called after our User has been created.
  - "then" takes in a callback function as an argument, with our newly created User (dbUser) as the argument for the callback function.
- On **line 9** we print dbUser to the console.

Here is what some of the console output looks like for the CREATE example

```
{ id: 2,
  email: 'tom@myspace.com',
  password: 'superinsecurepassword123',
  age: 46,
  name: 'Tom Anderson',
  updatedAt: Fri Nov 25 2016 12:07:44 GMT-0500 (EST),
  createdAt: Fri Nov 25 2016 12:07:44 GMT-0500 (EST) }
```

The newly created User has all of the properties we've described along with an `id`, `updatedAt`, and `createdAt` columns which Sequelize gives us automatically. The entire object logged to the console is a bit larger than this because it contains other information about our model such as which properties we're updating and information about the model's configuration. Try this yourself and explore the output.

NOTE: Even though the object returned from the query is much larger, the snippet above is the only data sent back to the client if we were to `res.json(dbUser)`. If you need to explicitly filter out just the data for the row's values, you can do so by accessing dbUser.dataValues.

Part Two: READ

There are a few options for reading data from our database with Sequelize. We will go over the two most commonly used and probably most important to understand.

findOne

```
1    var db = require("./models");
2
3    db.User.findOne({ where: { id: 1} }).then(function(dbUser) {
4      console.log(dbUser);
5    });
```

findOne takes in a "where" object as an argument. The key of this object is the word "where" and it's value is another object with keys and values describing the User we want to find.

We want to find one user where the id is equal to one. We can be as specific as we want here. We can also do the following:

```
1    var db = require("./models");
2
3    db.User.findOne({
4      where: {
5        id: 1,
6        name: "Tom Anderson",
7        age: 46
8      }
9    }).then(function(dbUser) {
10      console.log(dbUser);
11    });
```

This would find one User where the id = 1, name = "Tom Anderson", and age = 46.

findOne will only return a single record. If multiple match our "where", it will only return the first result.

findAll

```
1    var db = require("./models");
2
3    db.User.findAll().then(function(dbUser) {
4      console.log(dbUser);
5    });
```

findAll returns an array of all of the records found. In this case we're returning all of our Users. You may optionally pass in a "where" object here if you only want to return specific Users who match the query.

Part Three: UPDATE

```
1    var db = require("./models");
2
3    var newTom = {
4      age: 25,
5      email: "mark@facebook.com"
6    };
7
8    db.User.update(newTom, {
9      where: {
10       id: 1
11     }
12   }).then(function(dbUser) {
13     console.log(dbUser);
14   });
```

In this example we're updating a User. The update method takes in two arguments:

1. An object with key value pairs we want to update. In this case, we want to update the age and email of a User to the specified values.

2. A "where" object describing which User or Users we want to update.

In this case, the result of the update method is a little different than in the first few examples. Here the value of dbUser is an array with a single value, the number of records updated.

Part Four: DELETE

```
1   var db = require("./models");
2
3   db.User.destroy({
4     where: {
5       id: 1
6     }
7   }).then(function(dbUser) {
8     console.log(dbUser);
9   });
```

To delete a record from our database, we'll use the Sequelize destroy method. The destroy method takes in a "where" object as an argument. We use this to specify which record or records we want to delete. The result of this method (dbUser), is the number of records affected.