Instances

Building a non-persistent instance

In order to create instances of defined classes just do as follows. You might recognize the syntax if you coded Ruby in the past. Using the `build` -method will return an unsaved object, which you explicitly have to save.

```
const project = Project.build({
  title: 'my awesome project',
  description: 'woot woot. this will make me a
rich man'
})

const task = Task.build({
  title: 'specify the project idea',
  description: 'bla',
  deadline: new Date()
})
```

Built instances will automatically get default values when they were defined:

```
// first define the model
const Task = sequelize.define('task', {
  title: Sequelize.STRING,
  rating: { type: Sequelize.STRING,
defaultValue: 3 }
})

// now instantiate an object
const task = Task.build({title: 'very important
task'})

task.title  // ==> 'very important task'
task.rating // ==> 3
```

To get it stored in the database, use the `save` -method and catch the events ... if needed:

```
project.save().then(() => {
  // my nice callback stuff
})

task.save().catch(error => {
  // mhhh, wth!
})
```

```
object with chaining:
Task
  .build({ title: 'foo', description: 'bar',
deadline: new Date() })
  .save()
  .then(anotherTask => {
    // you can now access the currently saved
task with the variable anotherTask... nice!
  })
  .catch(error => {
    // Ooops, do some error-handling
  })
```

Creating persistent instances

While an instance created with `.build()` requires an explicit `.save()` call to be stored in the database, `.create()` omits that requirement altogether and automatically stores your instance's data once called.

```
Task.create({ title: 'foo', description: 'bar',
deadline: new Date() }).then(task => {
  // you can now access the newly created task
via the variable task
})
```

It is also possible to define which attributes can be set via the create method. This can be especially very handy 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 a username and an address but not an admin flag:

```
User.create({ username: 'barfooz', isAdmin:
true }, { fields: [ 'username' ] }).then(user
=> {
  // let's assume the default of isAdmin is
false:
  console.log(user.get({
    plain: true
  })) // => { username: 'barfooz', isAdmin:
false }
})
```

Updating / Saving / Persisting an instance

Now lets change some values and save changes to the database... There are two ways to do that:

```
task.save().then(() => {})

// way 2
task.update({
  title: 'a very different title now'
}).then(() => {})
```

It's also possible to define which attributes should be saved when calling `save`, by passing an array of column names. This is useful when you set attributes based on a previously defined object. E.g. if you get the values of an object via a form of a web app. Furthermore this is used internally for `update`. This is how it looks like:

```
task.title = 'foooo'
task.description = 'baaaaaar'
task.save({fields: ['title']}).then(() => {
 // title will now be 'foooo' but description
is the very same as before
})

// The equivalent call using update looks like
this:
task.update({ title: 'foooo', description:
'baaaaaar'}, {fields: ['title']}).then(() => {
 // title will now be 'foooo' but description
is the very same as before
})
```

When you call `save` without changing any attribute, this method will execute nothing;

Destroying / Deleting persistent instances

Once you created an object and got a reference to it, you can delete it from the database. The relevant method is `destroy`:

```
Task.create({ title: 'a task' }).then(task => {
  // now you see me...
  return task.destroy();
}).then(() => {
 // now i'm gone :)
})
```

If the `paranoid` options is true, the object will not be deleted, instead the `deletedAt` column will be set to the current timestamp. To force the deletion, you can pass `force: true` to the destroy call:

Working in bulk (creating, updating and destroying multiple rows at once)

In addition to updating a single instance, you can also create, update, and delete multiple instances at once. The functions you are looking for are called

- `Model.bulkCreate`
- `Model.update`
- `Model.destroy`

Since you are working with multiple models, the callbacks will not return DAO instances. BulkCreate will return an array of model instances/DAOs, they will however, unlike `create`, not have the resulting values of autoIncrement attributes. `update` and `destroy` will return the number of affected rows.

First lets look at bulkCreate

```
User.bulkCreate([
  { username: 'barfooz', isAdmin: true },
  { username: 'foo', isAdmin: true },
  { username: 'bar', isAdmin: false }
]).then(() => { // Notice: There are no
arguments here, as of right now you'll have
to...
  return User.findAll();
}).then(users => {
  console.log(users) // ... in order to get the
array of user objects
})
```

To update several rows at once:

```
Task.bulkCreate([
  {subject: 'programming', status:
'executing'},
  {subject: 'reading', status: 'executing'},
  {subject: 'programming', status: 'finished'}
]).then(() => {
  return Task.update(
    { status: 'inactive' }, /* set attributes'
value */
    { where: { subject: 'programming' }} /*
where criteria */
  );
}).spread((affectedCount, affectedRows) => {
  // .update returns two values in an array,
therefore we use .spread
```

```
true

  // affectedCount will be 2
  return Task.findAll();
}).then(tasks => {
  console.log(tasks) // the 'programming' tasks
will both have a status of 'inactive'
})
```

And delete them:

```
Task.bulkCreate([
  {subject: 'programming', status:
'executing'},
  {subject: 'reading', status: 'executing'},
  {subject: 'programming', status: 'finished'}
]).then(() => {
  return Task.destroy({
    where: {
      subject: 'programming'
    },
    truncate: true /* this will ignore where
and truncate the table instead */
  });
}).then(affectedRows => {
  // affectedRows will be 2
  return Task.findAll();
}).then(tasks => {
  console.log(tasks) // no programming, just
reading :(
})
```

If you are accepting values directly from the user, it might be beneficial to limit the columns that you want to actually insert. `bulkCreate()` accepts an options object as the second parameter. The object can have a `fields` parameter, (an array) to let it know which fields you want to build explicitly

```
User.bulkCreate([
  { username: 'foo' },
  { username: 'bar', admin: true}
], { fields: ['username'] }).then(() => {
  // nope bar, you can't be admin!
})
```

`bulkCreate` was originally made to be a mainstream/fast way of inserting records, however, sometimes you want the luxury of being able to insert multiple rows at once without sacrificing model validations even when you explicitly tell Sequelize which columns to sift through. You can do by adding a `validate: true` property to the options object.

```
const Tasks = sequelize.define('task', {
  name: {
    type: Sequelize.STRING,
```

```
    }
  },
  code: {
    type: Sequelize.STRING,
    validate: {
      len: [3, 10]
    }
  }
})

Tasks.bulkCreate([
  {name: 'foo', code: '123'},
  {code: '1234'},
  {name: 'bar', code: '1'}
], { validate: true }).catch(errors => {
  /* console.log(errors) would look like:
  [
    { record:
    ...
    name: 'SequelizeBulkRecordError',
    message: 'Validation error',
    errors:
      { name: 'SequelizeValidationError',
        message: 'Validation error',
        errors: [Object] } },
    { record:
      ...
      name: 'SequelizeBulkRecordError',
      message: 'Validation error',
      errors:
        { name: 'SequelizeValidationError',
        message: 'Validation error',
        errors: [Object] } }
  ]
  */
})
```

Values of an instance

If you log an instance you will notice, that there is a lot of additional stuff. In order to hide such stuff and reduce it to the very interesting information, you can use the `get` -attribute. Calling it with the option `plain` = true will only return the values of an instance.

```
Person.create({
  name: 'Rambow',
  firstname: 'John'
}).then(john => {
  console.log(john.get({
    plain: true
  }))
})

// result:
```

```
//   id: 1,
//   createdAt: Tue, 01 May 2012 19:12:16 GMT,
//   updatedAt: Tue, 01 May 2012 19:12:16 GMT
// }
```

Hint: You can also transform an instance into JSON by using `JSON.stringify(instance)`. This will basically return the very same as `values`.

Reloading instances

If you need to get your instance in sync, you can use the method `reload`. It will fetch the current data from the database and overwrite the attributes of the model on which the method has been called on.

```
Person.findOne({ where: { name: 'john' }
}).then(person => {
  person.name = 'jane'
  console.log(person.name) // 'jane'

  person.reload().then(() => {
    console.log(person.name) // 'john'
  })
})
```

Incrementing

In order to increment values of an instance without running into concurrency issues, you may use `increment`.

First of all you can define a field and the value you want to add to it.

```
User.findById(1).then(user => {
  return user.increment('my-integer-field',
{by: 2})
}).then(user => {
  // Postgres will return the updated user by
default (unless disabled by setting {
returning: false })
  // In other dialects, you'll want to call
user.reload() to get the updated instance...
})
```

Second, you can define multiple fields and the value you want to add to them.

```
User.findById(1).then(user => {
  return user.increment([ 'my-integer-field',
```

Third, you can define an object containing fields and its increment values.

```
User.findById(1).then(user => {
  return user.increment({
    'my-integer-field':    2,
    'my-very-other-field': 3
  })
}).then(/* ... */)
```

Decrementing

In order to decrement values of an instance without running into concurrency issues, you may use `decrement`.

First of all you can define a field and the value you want to add to it.

```
User.findById(1).then(user => {
  return user.decrement('my-integer-field',
{by: 2})
}).then(user => {
  // Postgres will return the updated user by
default (unless disabled by setting {
returning: false })
  // In other dialects, you'll want to call
user.reload() to get the updated instance...
})
```

Second, you can define multiple fields and the value you want to add to them.

```
User.findById(1).then(user => {
  return user.decrement([ 'my-integer-field',
'my-very-other-field' ], {by: 2})
}).then(/* ... */)
```

Third, you can define an object containing fields and its decrement values.

```
User.findById(1).then(user => {
  return user.decrement({
    'my-integer-field':    2,
    'my-very-other-field': 3
  })
}).then(/* ... */)
```