Node ORM2 Documentation

Release 2.0.0-alpha6

Joe Simpson

Nov 10, 2017

Contents

Node ORM is an Object Relationship Manager for Node.js.

Basically I help you to work with your database using an object orientated approach.

I currently support MySQL, SQLite and Progress

Find out more on Github

CHAPTER 1

Documentation

Contents:

1.1 Getting Started

To get started is very simple.

1.1.1 Adding ORM to your app

First of all you need to add node-orm as a dependency to your node application.

This involves using the `package.json` file of which you can find a cheatsheat here

Add to your node.js application to use @kennydude's version:

```
"dependencies" : {
        "orm" : "https://github.com/kennydude/node-orm2/tarball/master"
}
```

or to use the master version:

```
"dependencies" : {
        "orm" : ">=2.0.0-alpha6"
}
```

Then run `npm install` to install orm to the local `node_modules` directory automatically.

1.1.2 Using ORM in your app

Simply require it:

```
var orm = require("orm");
```

And then connect to your database:

```
orm.connect("mysql://test:test@localhost/text", function(err, db){
        // db is now available to use! ^__^
});
```

And that is about it. For more on the connections string see *ORM Class*.

At this point you will declare your models like so:

```
orm.connect("mysql://test:test@localhost/text", function(err, db){
        // db is now available to use! ^__^
        var Person = db.define('person', {
        name      : String
});
});
```

1.2 Querying

Querying is an important area of interacting with a database.

Note: The following only applies if you are using @kennydude's version

To query the database you use the `model.find()` method from the *Model Class*

A simple = operation can be done like so:

```
model.find( { "field" : "value" })
```

1.2.1 Operators

To use more than a straightforward = or `in` operator you use a slightly different syntax like so:

```
model.find({
        field : { "operator" : "value" }
});
```

Those operators are:

Operator	Values to use operator
Equals	=
Less Than	<, "less than"
Less Than or Equal to	<=, "less than or equal to"
More than	>, "more than"
More than or equal to	>=, "more than or equal to"

If you want to use `IN` you present a list of values which is done like so:

```
mode.find({
        field : [ "item 1", "item 2" ]
})
```

1.3 Validation

Validation is an important part of managing a database. By putting it in the ORM it makes it very difficult to get around when coding. This is great for security and making your code simpler!

To use validation methods when you create your model, add an options parameter containing `validations` like so:

```
    var Person = db.define('person', {
    name      : String,
    surname   : String,
    age       : Number,
}, {
    validations: {
        age: orm.validators.rangeNumber(18, undefined, 'under-age')
    }
});
```

That's it!

To see the whole range of built in validation checks see *Built-in Validators*

1.3.1 Handling Errors

You must take care to handle errors in validating your content. At the moment, once an error has been reached no more will be checked.

When saving, the callback will contain and `error` parameter. If it is not null then it will be an error validating or an error saving to the database.

A validation error will contain

- message - The message defined by the validation to show. When defining these do not use full language texts, as this is bad for translation.
- field - The field that failed validation
- value - The value that failed validation

1.3.2 Multiple Validation Checks

You can easily use multiple validation checks using a list of checks like so:

```
    validations: {
    age: [
            orm.validators.rangeNumber(18, undefined, 'under-age'),
            orm.validators.rangeNumber(18, undefined, 'under-age'),
    ]
}
```

1.3.3 Custom Validation Checks

It is quite simple to write your own validation checks.

Simply write a function which takes these parameters in this order:

- *value* - The value to test

- *next* - The function to call when you're done
- *data* - The rest of the data we are validating
- *Model* - The model itself
- *prop* - The name of the property we are validating.

You will generally only need *value* and *next*.

If you call `next()` without any parameters it means the validation was successful. If not, send a string of the message you want to return.

1.4 Relationships

Relationships can be quickly defined using `model.hasOne()` and `model.hasMany()` methods in the *Model Class*.

Note: <relationship> is the name of the relationship you pass to the relationship creation methods.

1.4.1 One-to-one relationship

When defining this it assumes the current model can only have 1 of the other model.

It provides a number of methods such as `instance.get<relationship>()` on the current model. If you have enabled autoFetch then it will appear as a property on the model (bear in mind this may take slightly longer to return).

Also (@kennydude's version only!) you can reverse-lookup one-to-one relationships very quickly using `model.findBy<relationship>( otherModel, ... )` which makes it very useful.

If you have a social network, you may want to find all of 1 user's messages, so you could do `messages.findByUser( myuser, function(err, results){ ... } )` which would work well.

1.4.2 Many-to-many relationships

Similar to the above, except works for more than one item.

However, it introduces some more methods than the above which are as follows:

`instance.`**`set<relationship>`**(*items*, *function(err){ .. }*)

Arguments

- **`items`** (*`list[instance]`*) – replacement list
- **`callback`** – Called when function is finished and returns the error if there was one

This replaces the whole list of items that the current instance is related to.

Warning: This is not recommended if you can avoid it

`instance.`**`remove<relationship>`**(*function(err){ .. }*)

Arguments

- **`callback`** – Called when the list is deleted

Removes the entire list

`instance.`**`remove<relationship>`**(*items..., function(err){ .. }*)

Arguments

- **`items`** (*`instance...`*) – Items to delete (separated arguments)
- **`callback`** – Called when the items are destroyed

Deletes specified items from the list

`instance.`**`add<relationship>`**(*item..., extra, function(err){ .. }*)

Arguments

- **`items`** (*`instance...`*) – Items to add (separated arguments)
- **`extra`** (*`object`*) – Optional object of items to add extra to the link table
- **`callback`** – Called when the items are added

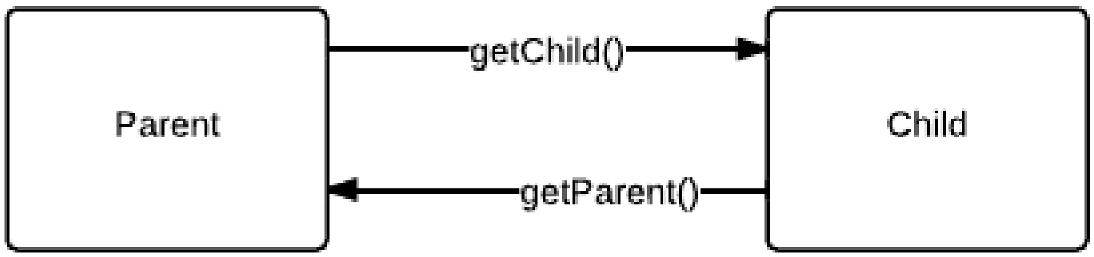
Adds the specified items to the list

1.4.3 Reverse Relationships

If you pass the reverse option in your model, it allows the child models to get the associated parent model.

To put it into more sense here is a diagram where we have said:

```
parent.hasOne("child", child, {
        "reverse" : "parent"
});
```



1.5 Reference

This is the section which contains all of the class reference for Node ORM

1.5.1 Instance Class

See *Model Class* for how to create these. Do not do this manually.

You can add custom methods to this using the `methods` option in the `options` parameter of `orm.define()` *See More...*

instance.save(callback)

Save the instance to the database. This is required if you have not turned on `autoSave`.

Callback is sent `error` and `instance` as parameters.

Emits an `save` event once completed with `error` and `instance` as parameters. This is useful if you need to notify something like socket.io of new data without requiring to reference it everywhere in your code.

instance.saved

If the instance is saved or not

instance.remove(callback)

Remove the instance from the database.

Emits and `remove` event once completed with `error` and `instance` as parameters.

instance.isInstance()

Is this instance an instance? Returns true

instance.on(*event*, *callback*)

Adds an event handler to an event that is fired.

1.5.2 Model Class

Please note: If you are using custom-built tables, you must have an id column!

model.get(*id*, *options*, *callback*)

id is the id of the object you are wishing to get

options is an optional object of options which uses the same flags as the `orm.define()` function in *ORM Class*.

callback is a function which takes two parameters which are `error` and `instance`. Instance is an *Instance Class*.

model.find(...)

This function takes the following parameters

- *conditions*, Object: The conditions you wish to search by
- *options*, object: The options of the query, Optional;
- *limit*, Integer: The maximum number of records to return per-query
- *ordering*, list: Ordering. This will generally contain 2 elements. The first is the field to sort by and the second is the method. This is `A` for Ascending order and `Z` (default) descending order. Optional
- *callback*, function: Called when the data is available. Takes two parameters: `error` and `items` (items is an array of *Instance Class*)

Conditions is an object which is quite limited at the moment. You can include a value to compare with (= comparator) or a list (`IN` comparator). *See More*

Options can contain more including (all of which are optional):

- __merge which joins two tables together and contains two values of `from` and `to` which each contain a `table` and `field` value
- offset: Integer, the number to start loading data from
- limit: Integer, the number to limit to per query.
- only: List, a list of properties that you wish to contain if you want to restrict them

model.clear()

Clears the table in the database.

Warning: THIS WILL DESTROY DATA! BE CAREFUL!

model.hasOne(*type*, *another_model*, *opts*)

Relates this item to another.

- *type*, String: What relationship does the current item have to *another_model*
- *another_model*, Model: Model to relate to. Optional, if ommited it defaults to itself.
- *opts*, Object: Options to apply to the relationship. Optional

The options available to customize are: * *reverse* If you add this you must provide it with a value to add to the other model to get it's parent. For more information see ../relationships

For example:

```
var Person = db.define('person', {
    name : String
});
var Animal = db.define('animal', {
    name : String
});
Animal.hasOne("owner", Person);
```

With this example it assumes `animal` has a field called `owner_id` which is an Integer.

If you have enabled `autoFetch`, then instances will have a *type* property with the other model instance. Otherwise you will have a function with the name of get*type* (although *type* is capitalized for CammelCase typing).

Note: If you are using @kennydude's version the following applies:

This will also attach a reverse-lookup function to your model with the name of model.findBy *type* (*other_model*, *extra*, *callback*)

Where *extra* is optional, and *callback* takes 2 arguments, `error`` and `item` (a *Instance Class*)

model.hasMany(*type*, *extra*, *another_model*)

Relates this model to another in a many-to-many fashion.

- *type*, String: What relationship does the current item have to *another_model*
- *extra*, Object: Extra attributes on the intermediate table you want to include. Optional
- *another_model*, Model: Model to relate to. Optional, if ommited it defaults to itself.

For example:

```
var Person = db.define('person', {
    name : String
});
Person.hasMany("friends", {
    rate : Number
});

Person.get(123, function (err, John) {
    John.getFriends(function (err, friends) {
        // assumes rate is another column on table person_friends
        // you can access it by going to friends[N].extra.rate
    });
});
```

You require an intermediate table with relationshipType_id and anotherModelName_id fields at least. The table is called thisModelName_type. For the above you would have a table called person_friends with the fields friend_id and person_id.

Note: If you are using @kennydude's version the following applies:

This will also attach a reverse-lookup function to your model with the name of model.findBy *type* (*other_model*, *extra*, *callback*)

Where *extra* is optional, and *callback* takes 2 arguments, `error`` and `items` (array of *Instance Class*)

model.createSQL()

Returns a string with the `CREATE TABLE` syntax that should be used for the current database.

Note: If the current database is non-relational this may not return anything ("tables" are created on-demand)

1.5.3 ORM Class

To access this class you will typically do this:

```
var orm = require("orm");
```

static orm.connect(*connection_string*, *callback*)

Connects to a database. *connection_string* is a URI.

Currently supported schemas/drivers are:

- sqlite://*path_to_database*
- mysql://username:password@host/database
- postgres://username:password@host/database

Returns a *ORM* instance via *callback* once connected

orm.define(*name*, *properties*, *options*)

Define a *Model Class*. Models represent the tables in your database.

name will be the name of your table

properties will be the fields in your table expressed in an object fashion like so:

```
{
        name : String,
        number_of_warnings : Number,
        data_applied : Date
}
```

Data type accepted for *properties* are:

- String
- Number
- Boolean
- ['value', 'x'] - Enum type
- Buffer (aka Binary)
- Object - JSON Encoded

options is an object of the following

- cache: Boolean; Option to flag if you want to store caches of objects
- autoSave: Boolean; Option to flag if you want the model to auto-save when you change properties. Default is false
- autoFetch: Boolean; Option to flag if you want to automatically fetch any associated objects. Default is false
- autoFetchLimit: Integer; How far to go with fetching associated objects. Required if you use autoFetch
- methods, Object; An object of methods you want to attach to the *Instance Class* items.
- validations, Object; An object of validators you wish to attach *See more*.

orm.close()

Closes the database connection. There is no open function, you have to reconnect globally.

orm.validators

Built-in validators. See *Built-in Validators* for more information.

1.5.4 Built-in Validators

For information on how to use validation or these functions see *Validation*.

validators.rangeNumber(*min*, *max*, *message*)

Checks if the value is a number between *min* and *max*. If not *message* is returned.

validators.rangeLength(*min*, *max*, *message*)

Checks if the value is a string between *min* and *max* length.

validators.insideList(*values*, *message*)

Check if a value is contained in the *values* list. If not returns *message*.

For example you could pass a list of months of the year. This would mean there would be no way to create your own month.

validators.outsideList(*values*, *message*)

Check if a value is **not** contained on the *values* list.

This is useful for creating a blacklist of values, however do not use it for large instances.

validators.equalToProperty(*property*, *message*)

Checks if the value you are validating is the same as the value in *property*.

For example take this block of data:

```
{ "password" : "blah", "retype_password" : "xyz" }
```

And you validated `retype_password` with the property as `password` it would fail validation because `blah` != `xyz`

validators.notEmptyString(*message*)

Checks if the value is empty or not

validators.unique(*message*)

Checks if the value is unique in the database.

This will slow down validation! You should handle errors returned by the database instead

validators.patterns.match(*regex*, *message*)

Checks if the value matches the Regex *regex*

validators.patterns.hexString(*message*)

Checks if the value is a hexadecimal string. (0-9 and A-F)

validators.patterns.email(*message*)

Checks if the value is a valid email address

validators.patterns.ipv4(*message*)

Checks if the value is a valid IP v4 address.

CHAPTER 2

Indices and tables

- genindex
- search

Index