QuickModel Documentation

Release 0.1

Daniel França

Sep 13, 2018

Contents

QuickModel is a simple/easy to setup ORM library for Qt/QML. Tha main goal is to provide a very simple ORM layer on top of the SQLite access.

To achieve those goals we aim the following

- Single file library/Just need to import a single file into your QML project
- Consistent interface inspired by Django ORM

CHAPTER 1

Quick Start

Clone the repository and import the library/quickmodel.js file into your project.

Define your database and the models in a js file:

```
var quickModel = new QuickModel.QMDatabase('testApp', '1.0'); //Define objects
var Artist = quickModel.define('Artist', {  name: quickModel.String('Name', {accept_null:false})
});
var Track = quickModel.define('Track', {  title: quickModel.String('Track Name', {accept_null:false}), artist: quickModel.FK('Artist', {'references': 'Artist'})
})
```

Run your queries on your new database:

```
var artist1 = Artist.create({name: 'Lana del Rey'});
var artist2 = Artist.create({name: 'Rammstein'});
var artist3 = Artist.create({name: 'Arctic Monkeys'});
var artist4 = Artist.create({name: 'Johnny Cash'});
var artist5 = Artist.create({name: 'Johnny Bravo'});
var track = Track.create({title: 'Born to die', artist: artist.id});
var artists_johnny = Artist.filter({name__like: 'Johnny'}).all();
var sorted_artists = Artist.order('name').limit(3).all();
var lana = Artist.filter({name: 'Lana del Rey'}).get();
```

1.1 Documentation

1.1.1 QMDatabase

QMDatabase represents a database connection, it's the first class you need initialize before use the library.

To initialize it as simple as: import "quickmodel.js" as QuickModel

```
var quickModel = new QuickModel.QMDatabase("myApp", '1.0');
```

Where the "myApp" is the name of your app and "1.0" is the version

With the QMDatabase instance you can define your tables/objects:

```
var Book = quickModel.define('Book', {
        title: quickModel.String('Title', {accept_null:false}),
        authorName: quickModel.String('Author Name', {accept_null:false}),
        pages: quickModel.Integer('Pages', {default: 0})
    });
```

1.1.2 QMModel

QMModel represents a model attached to a database table, the model is created when you define a new table/object:

```
var Book = quickModel.define('Book', {
        title: quickModel.String('Title', {accept_null:false}),
        authorName: quickModel.String('Author Name', {accept_null:false}),
        pages: quickModel.Integer('Pages', {default: 0})
    });
```

1.1.3 QMObject

QMObject represents a single instance of an object in the database:

```
//Create a new object in the database and store it on myBook variable
var myBook = Book.create({title: "Haunted", authorName: "Chuck Palahniuk"});
//Retrieve the object from the database
var book = Book.filter(authorName: "Chuck Palahniuk").get();
//Retrieve a collection of objects from the database sorted by title
var books = Book.order('title').all();
```

1.1.4 Defining

As a first step before use the library you must define the classes (database entities) related to your poject.

For this you can simply create a js file, containing the initialization. .import "quickmodel.js" as QuickModel

```
var qmdb; var Author; var Book;
```

```
function init() {
    qmdb = new QuickModel.QMDatabase("MyApp", "1.0");
    Author = qmdb.define("Author", { name: qmdb.String("Name", {accept_null: false}), email: qmdb.String("Email")
    });
    Book = qmdb.define("Book", { author: qmdb.FK("Author", {references: 'Author'}) title: qmdb.String("Title", {accept_null: false}), pages: qmdb.Integer("Pages", {accept_null: false}),
    });
}
```

Then, you must call your initialization function before any attempt to use the data from the models. And you can create your objects:

```
var author1 = Artist.create({name: 'Chuck Palahniuk'}); var author2 = Artist.create({name: 'Isaac Asimov'});
```

1.1.5 Queries

The queries interface is inspired in Django-ORM, so you can expect a familiar interface if you are used to Django.

i.e: If you want to find a specific book, you can query by title

```
var books = Book.filter({title: "Fight Club"}).all()
```

If you want to list all books sorted by title: var books = Book.order('title').all()

You can use special operators for things like "greater than, less than" adding double underscore and the operator as the dict key:

If you want to list all books with the word "World" in the title var books = Book.filter({title__like:'World'}).all()

If you want to list only the books with more than 100 pages var books = Book.filter({pages__gt:100}).all()

All operators: like: The column contains the string startswith: The column starts with the string endswith: The column ends with the string gt: Greater than ge: Greater than or equal lt: Less than le: Less than or equal null: is NULL

If you want to list only the books from a specific author var author = Author.filter({name: "Isaac Asimov"}).get() var books = Book.filter({author:author.id}).all()

If you want to update an item or a list of items you just need to filter first and call update with the new parameters Author.filter({name: "Isac Asimov"}).update({name: "Isaac Asimov"});

The same approach to exclude an item Author.filter({title: 'Foundation'}).remove();

1.1.6 Migrations

CHAPTER 2

Indices and tables

- genindex
- modindex
- search