INTO THE BOX EDITION

Session

qb, migrations & seeders

LED BY

Gavin Pickin

Ortus Solutions

Adobe

SPEAKER AT ITB2023

Gavin Pickin

- Software Consultant for Ortus
- Work with ColdBox, CommandBox, ContentBox APIs and VueJS every day!
- Working with Coldfusion since 1999 - V4
- Love learning and sharing the lessons learned
- From New Zealand, live in Bakersfield, Ca
- Loving wife, lots of kids, and countless critters

What is QB?

qb is a fluent query builder for CFML.

Written by Eric Peterson
It is heavily inspired by Eloquent from Laravel.

What is QB?

Using qb, you can:

- Quickly scaffold simple queries
- Make complex, out-of-order queries possible
- Abstract away differences between database engines
- It is fluent, chainable

Why use QB?

- No more string concatenation
- Worry about what you want to do, not how to do it.
- Reduce Typos with Builder format
- Bridge the differences between DB engines
- Make some of the bad DB engine syntax much more developer friendly
- OO patterns and helpers with your SQL.
- Keep the related statements for your query close in the code, and let QB create the order of your sql statement for you.

Querying a Table

QueryBuilder

```
query.from( "users" );
```

MySQL

```
SELECT * FROM `users`
```

Querying a Table with Selects

QueryBuilder

```
query
  .select( [ "fname AS firstName", "age" ] )
  .from( "users" );
```

mysql

```
SELECT `fname` AS `firstName`, `age` FROM
`users`
```

Querying with Joins

QueryBuilder

```
query
    .from( "users" )
    .join( "posts", "users.id", "=",
"posts.author_id" );
```

mysql

```
SELECT *
FROM `users`
JOIN `posts`
  ON `users`.`id` = `posts`.`author_id`
```

Query Where Clauses

QueryBuilder

```
query.from( "users" )
     .where( "active", "=", 1 );
```

MySQL

```
SELECT *
FROM `users`
WHERE `active` = ?
```

Query Where Clauses

QueryBuilder

```
query.from( "users" )
     .whereBetween( "id", 1, 2 );
```

MySQL

```
SELECT *
FROM `users`
WHERE `id` BETWEEN ? AND ?
```

Query Where Clauses

QueryBuilder

```
query.from( "orders" )
    .whereExists( function( q ) {
        q.select( q.raw( 1 ) )
            .from( "products" )
            .whereColumn( "products.id", "orders.id" );
    } );
```

MySQL

```
SELECT *
FROM `orders`
WHERE EXISTS (
    SELECT 1
    FROM `products`
    WHERE `products`.`id` = `orders`.`id`
)
```

Query Where Clauses

QueryBuilder

```
query.from( "users" )
     .whereLike( "username", "J%" );
```

MySQL

```
SELECT *
FROM `users`
WHERE `username` LIKE ?
```

Query Where Clauses

QueryBuilder

```
query.from( "orders" )
     .whereIn( "id", [ 1, 4, 66 ] );
```

MySQL

```
SELECT *
FROM `orders`
WHERE `id` IN (?, ?, ?)
```

Order by

QueryBuilder

```
query.from( "users" )
     .orderBy( "email" );
```

MySQL

```
SELECT *
FROM `users`
ORDER BY `email` ASC
```

Order by

QueryBuilder

```
query.from( "users" )
     .orderBy( "email" )
     .orderBy( "username", "desc" );
```

MySQL

```
SELECT *
FROM `users`
ORDER BY
   `email` ASC,
   `username` DESC
```

Order by

QueryBuilder

```
query.from( "users" )
    .orderBy( "email|asc,username", "desc" );
```

MySQL

```
SELECT *
FROM `users`
ORDER BY
  `email` ASC,
  `username` DESC
```

Order by

QueryBuilder

```
query.from( "users" )
    .orderBy( "email|asc,username", "desc" );
```

MySQL

```
SELECT *
FROM `users`
ORDER BY
  `email` ASC,
  `username` DESC
```

Group By

QueryBuilder

```
query.from( "users" )
     .groupBy( "country" )
     .groupBy( "city" );
```

MySQL

```
SELECT *
FROM `users`
GROUP BY `country`, `city`
```

Group By & Having

QueryBuilder

```
query.from( "users" )
     .groupBy( "email" )
     .having( "email", ">", 1 );
```

MySQL

```
SELECT *
FROM `users`
GROUP BY `email`
HAVING `email` > ?
```

Limit

QueryBuilder

```
query.from( "users" )
     .limit( 5 );
```

MySQL

```
SELECT *
FROM `users`
LIMIT 5
```

Offset

QueryBuilder

```
query.from( "users" )
     .offset( 25 );
```

MySQL

```
SELECT *
FROM `users`
OFFSET 25
```

For page

QueryBuilder

```
query.from( "users" )
     .forPage( 3, 15 );
```

MySQL

```
SELECT *
FROM `users`
LIMIT 15
OFFSET 30
```

When Conditionals

QueryBuilder

```
qb.from( "users" )
    .where( "active", 1 )
    .when( len( url.q ), function( q ) {
        q.where( "username", "LIKE", q & "%" )
            .orWhere( "email", "LIKE", q & "%" );
    } );
```

MySQL

```
SELECT *
FROM "users"
WHERE "active" = ?
    AND (
        "username" = ?
        OR "email" = ?
    )
```

Query Params

QueryBuilder

```
query.from( "users" )
    .where( "id", "=",
    { value = 18, cfsqltype = "CF_SQL_VARCHAR" } );
```

MySQL

```
SELECT *
FROM `users`
WHERE `id` = ?
```

Terminating the Query Builder

get() - The get method is the most common method used for retrieving results. It executes using the configured QueryBuilder and returns the results.

.first() - If you just need to retrieve a single row from the database table, you may use the first method. This method will return a single record (a Struct by default). If no row is found an empty Struct will be returned by default.

Terminating the Query Builder

.firstOrFail() - Returns the first matching row for the configured query, just like first. If no records are found, it throws an EntityNotFound exception.

.values(“firstName”) - If you don't even need an entire row, you may extract a single value from each record using the values method. The values method will return the column of your choosing as a simple array.

Terminating the Query Builder

.value(“firstName”) - This method is similar to values except it only returns a single, simple value. Where values calls get under the hood, this method calls first.

.paginate(page, maxRows, options) - Generates a pagination struct along with the results of the executed query. It does this by calling both count and forPage.

Aggregates

.exists()
.count()
.max(column, options)
.min(column)
.sum(column)
.columnList()

Inserts

.insert()
.insertIgnore()
.insertUsing()
.returning()

Updates

.update()
.updateOrInsert()
.upsert()

Deletes

.delete()

Datasources

In Application.cfc you can specify your default datasource which will be used by qb.

If you want to retrieve data from other datasources you can specify this in all retrieval functions by using the extra options parameter such as:

```
query.from( "users" )
    .get( options = { datasource:
"MyOtherDatasourceName" } );
```

qb Return Format

returnFormat refers to the transformation your executed query makes (if any) before being returned to you.

You can choose one of four formats:

- "array"
- "query"
- "none"
- A custom function

Globally setting return format


```
//config/Coldbox.cfc

moduleSettings = {
    "qb": {
        "returnFormat": "array"
    }
};
```

Schema builder

The syntax is expressive and fluent, making it easy to understand what is being executed

The syntax is database-agnostic. Specific quirks are isolated in a Grammar file, making it easy to migrate between engines.

Schema builder


```
schema.create( "users", function( t ) {
    t.increments( "id" );
    t.string( "email" );
    t.string( "password" );
} );
```

Debugging qb

toSQL() - Returns the SQL that would be executed for the current query.

toSQL(showBindings=true) - Returns the SQL that would be executed for the current query… with ? replaced with

```
{ value=”thevalue”, cfsqltype=”CF_SQL_VARCHAR” }
```

toSQL(showBindings=inline) - Returns the SQL that would be executed for the current query… with ? replaced with the actual value. This makes it easy to copy and paste into your Sql Client

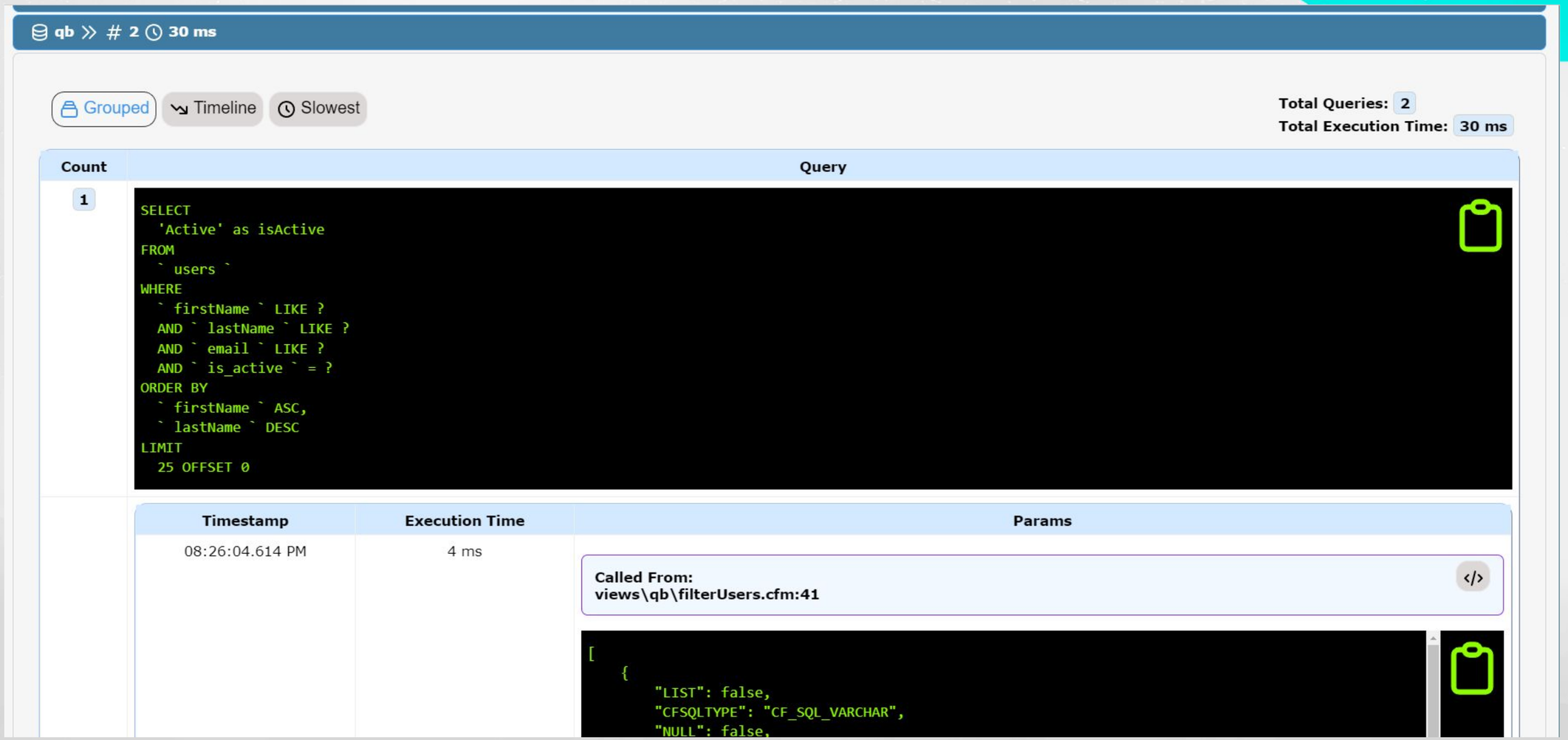
Debugging qb

NEW FEATURE

toSQL(showBindings=inline) - Returns the SQL that would be executed for the current query… with ? replaced with the actual value. This makes it easy to copy and paste into your Sql Client

So new - the docs aren’t even updated yet :)

CBDebugger



CBDebugger has some useful information when working with QB.

migrations

What are Migrations?

Database migrations, also known as schema migrations, database schema migrations, or simply migrations.

They are controlled sets of changes developed to modify the structure of the objects within a relational database.

What are Migrations?

Migrations help transition database schemas from their current state to a new desired state, whether that involves adding tables and columns, removing elements, splitting fields, or changing types and constraints.

These can be checked into and managed by normal version control software to track changes and share among team members.

What are Migrations?

cfMigrations is a module that provides schema migrations for ColdBox applications.

It allows developers to easily migrate, rollback, and refresh their database schema

Either directly from the application or via the CLI CommandBox Module.

What are Migrations?

INTO THE BOX EDITION

A migration file is a component with two methods up and down.

The function up should define how to apply the migration.

The function down should define how to undo the change down in up.

Why use Migrations?

Migrations are artifacts or files that can be shared, applied to multiple database systems, and stored in version control.

The database schema and the application's assumptions about that structure can evolve in tandem… and most importantly they are

DEPLOYED TOGETHER!!!

Using CFMigrations

To install the Coldbox module within your application, simply run:

```
box install cfmigrations
```

To install the CommandBox CLI module run:

```
box install commandbox-migrations
```

CFMigrations Configuration

The module is configured by default with a single migration service that interact with your database, optionally using qb. Multiple migration services with different managers may also be configured. The default manager for the cfmigrations is QBMigrationManager, but you may use others, such as those included with the cbmongodb and cbelasticsearch modules or roll your own.

https://cfmigrations.ortusbooks.com/overview/configuration

CFMigrations Configuration

For QBMigrationManager migrations (which is the default), the up and down functions are passed an instance of SchemaBuilder@qb and QueryBuilder@qb as arguments.

Schema Builder Example

INTO THE BOX EDITION

```
component {

    function up( schema, qb ) {
       schema.create( "users", function( t ) {
            t.increments( "id" );
            t.string( "email" );
            t.string( "password" );
        } );
    }

    function down( schema, qb ) {
        schema.drop( "users" );
    }

}
```

Updating database content in a migration file

Sometimes you want to do multiple content updates or inserts in a migration.

In this case you can use the QueryBuilder for the updates. When doing your second update you have to reset the Querybuilder object by using the newQuery method.

Updating database content in a migration file


```
component {

    function up( SchemaBuilder schema, QueryBuilder query ) {
    query.from('users')
        .where( "username", "superuser")
        .update( {"hassuperpowers" = true} )
    query.newQuery().from('users')
        .where('username','RandomUser')
        .update( {"hassuperpowers" = false} )
    }

    function down( SchemaBuilder schema, QueryBuilder query ) {
        ......
    }

}
```

Running Migrations

CommandBox CLI Migrations

```
migrate up
```

Application Migrations

```
getInstance( "MigrationService@cfmigrations"
).runAllMigrations( "up" )
```

seeders

What are Seeders?

A seeder is a special class used to generate and insert sample data (seeds) in a database.

This is an important feature in development environments, since it allows you to recreate the application with a fresh database, using sample values that you'd otherwise have to manually insert each time the database is recreated.

Why use Seeders?

Seeders are a great way to generate and insert dummy data into a database for development or testing purposes.

Seeders in CFMigrations

A seeder is a cfc file with a single required method - run.

For the QBMigrationManager, it is passed a QueryBuilder instance and a MockData instance, useful for creating fake data to insert into your database.

Example Seeders


```
component {
    function run( qb, mockdata ) {
        qb.table( "users" ).insert(
            mockdata.mock(
                $num = 25,
                "firstName": "firstName",
                "lastName": "lastName",
                "email": "email",
                "password": "string-secure"
            )
        );
    }
}
```

Why can't I use seeders in Production?

Seeders are only for generated dummy data

If you want to add data into your database for production, you should add them to your actual migrations.

Time for some code

INTO THE BOX EDITION

Cool Tools

SQL to QB - By Andrew Davis - might be out of date
https://cooltools.blusol.io/#/sql2qb

INTO THE BOX EDITION

SCAN ME

THANK YOU

Thanks to our sponsors

Lucid Outsourcing Solutions
Intelligent Transparent

aws

Tera Tech
The ColdFusion Experts

Ortus Solutions

Adobe

WEB SERVICES

foundeo inc.
Fixinator • FuseGuard • HackMyCF • Consulting

COMPUTER KNOW HOW