Orator Documentation

Release 0.6.4

Sébastien Eustace

Sep 01, 2017

Contents

1 Installation **3**

2 Basic Usage **5**
- 2.1 Configuration . . . 5
- 2.2 Read / Write connections . . . 6
- 2.3 Running queries . . . 6
- 2.4 Database transactions . . . 7
- 2.5 Accessing connections . . . 7
- 2.6 Query logging . . . 8

3 Query Builder **11**
- 3.1 Introduction . . . 11
- 3.2 Selects . . . 11
- 3.3 Joins . . . 13
- 3.4 Advanced where . . . 14
- 3.5 Aggregates . . . 14
- 3.6 Raw expressions . . . 15
- 3.7 Inserts . . . 15
- 3.8 Updates . . . 16
- 3.9 Deletes . . . 16
- 3.10 Unions . . . 17
- 3.11 Pessimistic locking . . . 17

4 ORM **19**
- 4.1 Introduction . . . 19
- 4.2 Basic Usage . . . 19
- 4.3 Mass assignment . . . 21
- 4.4 Insert, update and delete . . . 22
- 4.5 Soft deleting . . . 24
- 4.6 Relationships . . . 24
- 4.7 Querying relations . . . 30
- 4.8 Eager loading . . . 32
- 4.9 Inserting related models . . . 33
- 4.10 Touching parent timestamps . . . 35
- 4.11 Working with pivot table . . . 36
- 4.12 Timestamps . . . 36
- 4.13 Query Scopes . . . 37

4.14 Global Scopes . . . 38
4.15 Accessors & mutators . . . 39
4.16 Date mutators . . . 40
4.17 Attributes casting . . . 41
4.18 Model events . . . 41
4.19 Model observers . . . 42
4.20 Converting to dictionaries / JSON . . . 42

5 Pagination **45**
5.1 Paginate database records . . . 45
5.2 Displaying Results . . . 46
5.3 Converting Results To JSON . . . 47
5.4 Setting a custom current page resolver . . . 47

6 Schema Builder **49**
6.1 Introduction . . . 49
6.2 Creating and dropping tables . . . 49
6.3 Adding columns . . . 50
6.4 Changing columns . . . 51
6.5 Renaming columns . . . 51
6.6 Dropping columns . . . 52
6.7 Checking existence . . . 52
6.8 Adding indexes . . . 52
6.9 Dropping indexes . . . 53
6.10 Foreign keys . . . 53
6.11 Dropping timestamps and soft deletes . . . 54

7 Migrations **55**
7.1 Creating Migrations . . . 56
7.2 Running Migrations . . . 57
7.3 Rolling back migrations . . . 58
7.4 Getting migrations status . . . 58

8 Seeding **59**
8.1 Introduction . . . 59
8.2 Writing Seeders . . . 59
8.3 Running Seeders . . . 61

9 Testing **63**
9.1 Model Factories . . . 63

10 Collections **67**
10.1 Introduction . . . 67
10.2 Available Methods . . . 67
10.3 Methods Listing . . . 69

11 Extensions **85**
11.1 Cache . . . 85

The Orator ORM provides a simple yet beautiful ActiveRecord implementation.

It is inspired by the database part of the Laravel framework, but modified to be more pythonic.

- A simple but powerful *ORM*
- A database agnostic *Schema Builder*
- A low level *Query Builder* to avoid the overhead of the ORM
- *Migrations*
- Support for **PostgreSQL**, **MySQL** and **SQLite** out of the box

THe Orator ORM supports python versions **2.7+** and **3.2+**

CHAPTER 1

Installation

You can install Orator in 2 different ways:

- The easier and more straightforward is to use pip

```
pip install orator
```

- Install from source using the official repository (https://github.com/sdispater/orator)

Note: The different dbapi packages are not part of the package dependencies, so you must install them in order to connect to corresponding databases:

- PostgreSQL: `psycopg2`
- MySQL: `PyMySQL` or `mysqlclient`
- SQLite: The `sqlite3` module is bundled with Python by default

CHAPTER 2

Basic Usage

Configuration

All you need to get you started is the configuration describing your database connections and passing it to a `DatabaseManager` instance.

```
from orator import DatabaseManager

config = {
    'mysql': {
        'driver': 'mysql',
        'host': 'localhost',
        'database': 'database',
        'user': 'root',
        'password': '',
        'prefix': ''
    }
}

db = DatabaseManager(config)
```

If you have multiple databases configured you can specify which one is the default:

```
config = {
    'default': 'mysql',
    'mysql': {
        'driver': 'mysql',
        'host': 'localhost',
        'database': 'database',
        'user': 'root',
        'password': '',
        'prefix': ''
    }
}
```

Read / Write connections

Sometimes you may wish to use one database connection for SELECT statements, and another for INSERT, UPDATE, and DELETE statements. Orator makes this easy, and the proper connections will always be used whether you use raw queries, the query builder or the actual ORM

Here is an example of how read / write connections should be configured:

```
config = {
    'mysql': {
        'read': {
            'host': '192.168.1.1'
        },
        'write': {
            'host': '192.168.1.2'
        },
        'driver': 'mysql',
        'database': 'database',
        'username': 'root',
        'password': '',
        'prefix': ''
    }
}
```

Note that two keys have been added to the configuration dictionary: `read` and `write`. Both of these keys have dictionary values containing a single key: `host`. The rest of the database options for the `read` and `write` connections will be merged from the main `mysql` dictionary. So, you only need to place items in the `read` and `write` dictionaries if you wish to override the values in the main dictionary. So, in this case, `192.168.1.1` will be used as the "read" connection, while `192.168.1.2` will be used as the "write" connection. The database credentials, prefix, character set, and all other options in the main `mysql` dictionary will be shared across both connections.

Running queries

Once you have configured your database connection, you can run queries.

Running a select query

```
results = db.select('select * from users where id = ?', [1])
```

The `select` method will always return a list of results.

Running an insert statement

```
db.insert('insert into users (id, name) values (?, ?)', [1, 'John'])
```

Running an update statement

```
db.update('update users set votes = 100 where name = ?', ['John'])
```

Running a delete statement

```
db.delete('delete from users')
```

Note: The `update` and `delete` statements return the number of rows affected by the operation.

Running a general statement

```
db.statement('drop table users')
```

Database transactions

To run a set of operations within a database transaction, you can use the `transaction` method which is a context manager:

```
with db.transaction():
    db.table('users').update({votes: 1})
    db.table('posts').delete()
```

Note: Any exception thrown within a transaction block will cause the transaction to be rolled back automatically.

Sometimes you may need to start a transaction yourself:

```
db.begin_transaction()
```

You can rollback a transaction with the `rollback` method:

```
db.rollback()
```

You can also commit a transaction via the `commit` method:

```
db.commit()
```

Warning: By default, all underlying DBAPI connections are set to be in autocommit mode meaning that you don't need to explicitly commit after each operation.

Accessing connections

When using multiple connections, you can access them via the `connection()` method:

```
users = db.connection('foo').table('users').get()
```

You also can access the raw, underlying dbapi connection instance:

```
db.connection().get_connection()
```

Sometimes, you may need to reconnect to a given database:

```
db.reconnect('foo')
```

If you need to disconnect from the given database, use the `disconnect` method:

```
db.disconnect('foo')
```

Query logging

Orator can log all queries that are executed. By default, this is turned off to avoid unnecessary overhead, but if you want to activate it you can either add a `log_queries` key to the config dictionary:

```
config = {
    'mysql': {
        'driver': 'mysql',
        'host': 'localhost',
        'database': 'database',
        'username': 'root',
        'password': '',
        'prefix': '',
        'log_queries': True
    }
}
```

or activate it later on:

```
db.connection().enable_query_log()
```

Now, the logger `orator.connection.queries` will be logging queries at **debug** level:

```
Executed SELECT COUNT(*) AS aggregate FROM "users" in 1.18ms

Executed INSERT INTO "users" ("email", "name", "updated_at") VALUES ('foo@bar.com',
↪'foo', '2015-04-01T22:59:25.810216'::timestamp) RETURNING "id" in 3.6ms
```

Note: These log messages above are those logged for **MySQL** and **PostgreSQL** connections which support displaying full request sent to the database. For **SQLite** connections, the format is as follows:

```
Executed ('SELECT COUNT(*) AS aggregate FROM "users"', []) in 0.12ms
```

Customizing log messages

Each log record sent by the logger comes with the `query` and `elapsed_time` keywords so that you can customize the log message:

```
import logging

logger = logging.getLogger('orator.connection.queries')
```

```
logger.setLevel(logging.DEBUG)

formatter = logging.Formatter(
    'It took %(elapsed_time)sms to execute the query %(query)s'
)

handler = logging.StreamHandler()
handler.setFormatter(formatter)

logger.addHandler(handler)
```

CHAPTER 3

Query Builder

Introduction

The database query builder provides a fluent interface to create and run database queries. It can be used to perform most database operations in your application, and works on all supported database systems.

Note: Since Orator uses DBAPI packages under the hood, there is no need to clean parameters passed as bindings.

Note: The underlying DBAPI connections are automatically configured to return dictionaries rather than the default tuple representation.

Selects

Retrieving all row from a table

```
users = db.table('users').get()

for user in users:
    print(user['name'])
```

Chunking results from a table

```
for users in db.table('users').chunk(100):
    for user in users:
        # ...
```

Retrieving a single row from a table

```
user = db.table('users').where('name', 'John').first()
print(user['name'])
```

Retrieving a single column from a row

```
user = db.table('users').where('name', 'John').pluck('name')
```

Retrieving a list of column values

```
roles = db.table('roles').lists('title')
```

This method will return a list of role titles. It can return a dictionary if you pass an extra key parameter.

```
roles = db.table('roles').lists('title', 'name')
```

Specifying a select clause

```
users = db.table('users').select('name', 'email').get()

users = db.table('users').distinct().get()

users = db.table('users').select('name as user_name').get()
```

Adding a select clause to an existing query

```
query = db.table('users').select('name')

users = query.add_select('age').get()
```

Using where operators

```
users = db.table('users').where('age', '>', 25).get()
```

Or statements

```
users = db.table('users').where('age', '>', 25).or_where('name', 'John').get()
```

Using Where Between

```
users = db.table('users').where_between('age', [25, 35]).get()
```

Using Where Not Between

```
users = db.table('users').where_not_between('age', [25, 35]).get()
```

Using Where In

```
users = db.table('users').where_in('id', [1, 2, 3]).get()

users = db.table('users').where_not_in('id', [1, 2, 3]).get()
```

Using Where Null to find records with null values

```
users = db.table('users').where_null('updated_at').get()
```

Order by, group by and having

```
query = db.table('users').order_by('name', 'desc')
query = query.group_by('count')
query = query.having('count', '>', 100)

users = query.get()
```

Offset and limit

```
users = db.table('users').skip(10).take(5).get()

users = db.table('users').offset(10).limit(5).get()
```

Joins

The query builder can also be used to write join statements.

Basic join statement

```
db.table('users') \
    .join('contacts', 'users.id', '=', 'contacts.user_id') \
    .join('orders', 'users.id', '=', 'orders.user_id') \
    .select('users.id', 'contacts.phone', 'orders.price') \
    .get()
```

Left join statement

```
db.table('users').left_join('posts', 'users.id', '=', 'posts.user_id').get()
```

You can also specify more advance join clauses:

```
clause = JoinClause('contacts').on('users.id', '=', 'contacts.user_id').or_on(...)

db.table('users').join(clause).get()
```

If you would like to use a "where" style clause on your joins, you may use the where and orWhere methods on a join. Instead of comparing two columns, these methods will compare the column against a value:

```
clause = JoinClause('contacts').on('users.id', '=', 'contacts.user_id').
↪where('contacts.user_id', '>', 5)

db.table('users').join(clause).get()
```

Advanced where

Sometimes you may need to create more advanced where clauses such as "where exists" or nested parameter groupings. It is pretty easy to do with the Orator query builder

Parameter grouping

```
db.table('users') \
    .where('name', '=', 'John') \
    .or_where(
        db.query().where('votes', '>', 100).where('title', '!=', 'admin')
    ).get()
```

The query above will produce the following SQL:

```
SELECT * FROM users WHERE name = 'John' OR (votes > 100 AND title != 'Admin')
```

Exists statement

```
db.table('users').where_exists(
    db.table('orders').select(db.raw(1)).where_raw('order.user_id = users.id')
)
```

The query above will produce the following SQL:

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

Aggregates

The query builder also provides a variety of aggregate methods, ' such as `count`, `max`, `min`, `avg`, and `sum`.

```
users = db.table('users').count()

price = db.table('orders').max('price')

price = db.table('orders').min('price')

price = db.table('orders').avg('price')

total = db.table('users').sum('votes')
```

Raw expressions

Sometimes you may need to use a raw expression in a query. These expressions will be injected into the query as strings, so be careful not to create any SQL injection points! To create a raw expression, you may use the `raw()` method:

```
db.table('users') \
    .select(db.raw('count(*) as user_count, status')) \
    .where('status', '!=', 1) \
    .group_by('status') \
    .get()
```

Inserts

Insert records into a table

```
db.table('users').insert(email='foo@bar.com', votes=0)

db.table('users').insert({
    'email': 'foo@bar.com',
    'votes': 0
})
```

Note: It is important to note that there is two notations available. The reason is quite simple: the dictionary notation, though a little less practical, is here to handle columns names which cannot be passed as keywords arguments.

Inserting records into a table with an auto-incrementing ID

If the table has an auto-incrementing id, use `insert_get_id` to insert a record and retrieve the id:

```
id = db.table('users').insert_get_id({
    'email': 'foo@bar.com',
    'votes': 0
})
```

Inserting multiple record into a table

```
db.table('users').insert([
    {'email': 'foo@bar.com', 'votes': 0},
    {'email': 'bar@baz.com', 'votes': 0}
])
```

Updates

Updating records

```
db.table('users').where('id', 1).update(votes=1)

db.table('users').where('id', 1).update({'votes': 1})
```

Note: Like the `insert` statement, there is two notations available. The reason is quite simple: the dictionary notation, though a little less practical, is here to handle columns names which cannot be passed as keywords arguments.

Incrementing or decrementing the value of a column

```
db.table('users').increment('votes')  # Increment the value by 1

db.table('users').increment('votes', 5)  # Increment the value by 5

db.table('users').decrement('votes')  # Decrement the value by 1

db.table('users').decrement('votes', 5)  # Decrement the value by 5
```

You can also specify additional columns to update:

```
db.table('users').increment('votes', 1, name='John')
```

Deletes

Deleting records

```
db.table('users').where('age', '<', 25).delete()
```

Delete all records

```
db.table('users').delete()
```

Truncate

```
db.table('users').truncate()
```

Unions

The query builder provides a quick and easy way to "union" two queries:

```
first = db.table('users').where_null('first_name')

users = db.table('users').where_null('last_name').union(first).get()
```

The `union_all` method is also available.

Pessimistic locking

The query builder includes a few functions to help you do "pessimistic locking" on your SELECT statements.

To run the SELECT statement with a "shared lock", you may use the `shared_lock` method on a query:

```
db.table('users').where('votes', '>', 100).shared_lock().get()
```

To "lock for update" on a SELECT statement, you may use the `lock_for_update` method on a query:

```
db.table('users').where('votes', '>', 100).lock_for_update().get()
```

CHAPTER 4

ORM

Introduction

The ORM provides a simple ActiveRecord implementation for working with your databases. Each database table has a corresponding Model which is used to interact with that table.

Before getting started, be sure to have configured a `DatabaseManager` as seen in the *Basic Usage* section.

```
from orator import DatabaseManager

config = {
    'mysql': {
        'driver': 'mysql',
        'host': 'localhost',
        'database': 'database',
        'username': 'root',
        'password': '',
        'prefix': ''
    }
}

db = DatabaseManager(config)
```

Basic Usage

To actually get started, you need to tell the ORM to use the configured `DatabaseManager` for all models inheriting from the `Model` class:

```
from orator import Model

Model.set_connection_resolver(db)
```

And that's pretty much it. You can now define your models.

Defining a Model

```
class User(Model):
    pass
```

Note that we did not tell the ORM which table to use for the `User` model. The plural "snake case" name of the class name will be used as the table name unless another name is explicitly specified. In this case, the ORM will assume the `User` model stores records in the `users` table. You can specify a custom table by defining a `__table__` property on your model:

```
class User(Model):

    __table__ = 'my_users'
```

Note: The ORM will also assume that each table has a primary key column named `id`. You can define a `__primary_key__` property to override this convention. Likewise, you can define a `__connection__` property to override the name of the database connection that should be used when using the model.

Once a model is defined, you are ready to start retrieving and creating records in your table. Note that you will need to place `updated_at` and `created_at` columns on your table by default. If you do not wish to have these columns automatically maintained, set the `__timestamps__` property on your model to `False`.

Retrieving all models

```
users = User.all()
```

Retrieving a record by primary key

```
user = User.find(1)

print(user.name)
```

Note: All methods available on the *Query Builder* are also available when querying models.

Retrieving a Model by primary key or raise an exception

Sometimes it may be useful to throw an exception if a model is not found. You can use the `find_or_fail` method for that, which will raise a `ModelNotFound` exception.

```
model = User.find_or_fail(1)

model = User.where('votes', '>', 100).first_or_fail()
```

Querying using models

```
users = User.where('votes', '>', 100).take(10).get()

for user in users:
    print(user.name)
```

Aggregates

You can also use the query builder aggregate functions:

```
count = User.where('votes', '>', 100).count()
```

If you feel limited by the builder's fluent interface, you can use the `where_raw` method:

```
users = User.where_raw('age > ? and votes = 100', [25]).get()
```

Chunking Results

If you need to process a lot of records, you can use the `chunk` method to avoid consuming a lot of RAM:

```
for users in User.chunk(100):
    for user in users:
        # ...
```

Specifying the query connection

You can specify which database connection to use when querying a model by using the `on` method:

```
user = User.on('connection-name').find(1)
```

If you are using *Read / Write connections*, you can force the query to use the "write" connection with the following method:

```
user = User.on_write_connection().find(1)
```

Mass assignment

When creating a new model, you pass attributes to the model constructor. These attributes are then assigned to the model via mass-assignment. Though convenient, this can be a serious security concern when passing user input into a model, since the user is then free to modify **any** and **all** of the model's attributes. For this reason, all models protect against mass-assignment by default.

To get started, set the `__fillable__` or `__guarded__` properties on your model.

Defining fillable attributes on a model

The `__fillable__` property specifies which attributes can be mass-assigned.

```
class User(Model):

    __fillable__ = ['first_name', 'last_name', 'email']
```

Defining guarded attributes on a model

The `__guarded__` is the inverse and acts as "blacklist".

```
class User(Model):

    __guarded__ = ['id', 'password']
```

Warning: When using `__guarded__`, you should still never pass any user input directly since any attribute that is not guarded can be mass-assigned.

You can also block **all** attributes from mass-assignment:

```
__guarded__ = ['*']
```

Insert, update and delete

Saving a new model

To create a new record in the database, simply create a new model instance and call the `save` method.

```
user = User()

user.name = 'John'

user.save()
```

Note: Your models will probably have auto-incrementing primary keys. However, if you wish to maintain your own primary keys, set the `__autoincrementing__` property to `False`.

You can also use the `create` method to save a model in a single line, but you will need to specify either the `__fillable__` or `__guarded__` property on the model since all models are protected against mass-assigment by default.

After saving or creating a new model with auto-incrementing IDs, you can retrieve the ID by accessing the object's `id` attribute:

```
inserted_id = user.id
```

Using the create method

```
# Create a new user in the database
user = User.create(name='John')

# Retrieve the user by attributes, or create it if it does not exist
user = User.first_or_create(name='John')

# Retrieve the user by attributes, or instantiate it if it does not exist
user = User.first_or_new(name='John')
```

Updating a retrieved model

```
user = User.find(1)

user.name = 'Foo'

user.save()
```

You can also run updates as queries against a set of models:

```
affected_rows = User.where('votes', '>', 100).update(status=2)
```

Saving a model and relationships

Sometimes you may wish to save not only a model, but also all of its relationships. To do so, you can use the `push` method:

```
user.push()
```

Deleting an existing model

To delete a model, simply call the `delete` model:

```
user = User.find(1)

user.delete()
```

Deleting an existing model by key

```
User.destroy(1)

User.destroy(1, 2, 3)
```

You can also run a delete query on a set of models:

```
affected_rows = User.where('votes', '>' 100).delete()
```

Updating only the model's timestamps

If you want to only update the timestamps on a model, you can use the `touch` method:

```
user.touch()
```

Soft deleting

When soft deleting a model, it is not actually removed from your database. Instead, a `deleted_at` timestamp is set on the record. To enable soft deletes for a model, make it inherit from the `SoftDeletes` mixin:

```
from orator import Model, SoftDeletes


class User(Model, SoftDeletes):

    __dates__ = ['deleted_at']
```

To add a `deleted_at` column to your table, you may use the `soft_deletes` method from a migration (see *Schema Builder*):

```
table.soft_deletes()
```

Now, when you call the `delete` method on the model, the `deleted_at` column will be set to the current timestamp. When querying a model that uses soft deletes, the "deleted" models will not be included in query results.

Forcing soft deleted models into results

To force soft deleted models to appear in a result set, use the `with_trashed` method on the query:

```
users = User.with_trashed().where('account_id', 1).get()
```

The `with_trashed` method may be used on a defined relationship:

```
user.posts().with_trashed().get()
```

Relationships

Changed in version 0.7.1: As of version 0.7.1, the decorator notation is the only one supported.

The previous notation (via properties) is now deprecated and is no longer supported. It will be removed in the next major version.

Orator makes managing and working with relationships easy. It supports many types of relationships:

- *One To One*
- *One To Many*
- *Many To Many*
- *Has Many Through*
- *Polymorphic relations*

- *Many To Many polymorphic relations*

One To One

Defining a One To One relationship

A one-to-one relationship is a very basic relation. For instance, a `User` model might have a `Phone`. We can define this relation with the ORM:

```
from orator.orm import has_one


class User(Model):

    @has_one
    def phone(self):
        return Phone
```

The return value of the relation is the class of the related model. Once the relationship is defined, we can retrieve it using *Dynamic properties*:

```
phone = User.find(1).phone
```

The SQL performed by this statement will be as follow:

```
SELECT * FROM users WHERE id = 1

SELECT * FROM phones WHERE user_id = 1
```

The Orator ORM assumes the foreign key of the relationship based on the model name. In this case, `Phone` model is assumed to use a `user_id` foreign key. If you want to override this convention, you can pass a first argument to the `has_one` decorator. Furthermore, you may pass a second argument to the decorator to specify which local column should be used for the association:

```
@has_one('foreign_key')
def phone(self):
    return Phone

@has_one('foreign_key', 'local_key')
def phone(self):
    return Phone
```

Defining the inverse of the relation

To define the inverse of the relationship on the `Phone` model, you can use the `belongs_to` decorator:

```
from orator.orm import belongs_to


class Phone(Model):

    @belongs_to
    def user(self):
        return User
```

In the example above, the Orator ORM will look for a `user_id` column on the `phones` table. You can define a different foreign key column, you can pass it as the first argument of the `belongs_to` decorator:

```
@belongs_to('local_key')
def user(self):
    return User
```

Additionally, you can pass a third parameter which specifies the name of the associated column on the parent table:

```
@belongs_to('local_key', 'parent_key')
def user(self):
    return User
```

One To Many

An example of a one-to-many relation is a blog post that has many comments:

```
from orator.orm import has_many


class Post(Model):

    @has_many
    def comments(self):
        return Comment
```

Now you can access the post's comments via *Dynamic properties*:

```
comments = Post.find(1).comments
```

Again, you may override the conventional foreign key by passing a first argument to the `has_many` decorator. And, like the `has_one` relation, the local column may also be specified:

```
@has_many('foreign_key')
def comments(self):
    return Comment

@has_many('foreign_key', 'local_key')
def comments(self):
    return Comment
```

Defining the inverse of the relation:

To define the inverse of the relationship on the `Comment` model, we use the `belongs_to` method:

```
from orator.orm import belongs_to

class Comment(Model):

    @belongs_to
    def post(self):
        return Post
```

Many To Many

Many-to-many relations are a more complicated relationship type. An example of such a relationship is a user with many roles, where the roles are also shared by other users. For example, many users may have the role of "Admin". Three database tables are needed for this relationship: `users`, `roles`, and `roles_users`. The `roles_users` table is derived from the alphabetical order of the related table names, and should have the `user_id` and `role_id` columns.

We can define a many-to-many relation using the `belongs_to_many` decorator:

```
from orator.orm import belongs_to_many


class User(Model):

    @belongs_to_many
    def roles(self):
        return Role
```

Now, we can retrieve the roles through the `User` model:

```
roles = User.find(1).roles
```

If you want to use an unconventional table name for your pivot table, you can pass it as the first argument to the `belongs_to_many` method:

```
@belongs_to_many('user_role')
def roles(self):
    return Role
```

You can also override the conventional associated keys:

```
@belongs_to_many('user_role', 'user_id', 'foo_id')
def roles(self):
    return Role
```

Of course, you also can define the inverse of the relationship on the `Role` model:

```
class Role(Model):

    @belongs_to_many
    def users(self):
        return User
```

Has Many Through

The "has many through" relation provides a convenient short-cut for accessing distant relations via an intermediate relation. For example, a `Country` model might have many `Post` through a `User` model. The tables for this relationship would look like this:

```
countries
    id - integer
    name - string

users:
    id - integer
```

```
    country_id - integer
    name - string

posts:
    id - integer
    user_id - integer
    title - string
```

Even though the `posts` table does not contain a `country_id` column, the `has_many_through` relation will allow access to a country's posts via `country.posts`:

```
from orator.orm import has_many_through


class Country(Model):

    @has_many_through(User)
    def posts(self):
        return Post
```

If you want to manually specify the keys of the relationship, you can pass them as the second and third arguments to the decorator:

```
@has_many_through(User, 'country_id', 'user_id')
def posts(self):
    return Post
```

Polymorphic relations

New in version 0.3.

Polymorphic relations allow a model to belong to more than one other model, on a single association. For example, you might have a `Photo` model that belongs to either a `Staff` model or an `Order` model.

```
from orator.orm import morph_to, morph_many


class Photo(Model):

    @morph_to
    def imageable(self):
        return

class Staff(Model):

    @morph_many('imageable')
    def photos(self):
        return Photo

class Order(Model):

    @morph_many('imageable')
    def photos(self):
        return Photo
```

Retrieving a polymorphic relation

Now, we can retrieve the photos for either a staff member or an order:

```
staff = Staff.find(1)

for photo in staff.photos:
    # ...
```

Retrieving the owner of a polymorphic relation

You can also, and this is where polymorphic relations shine, access the staff or order model from the `Photo` model:

```
photo = Photo.find(1)

imageable = photo.imageable
```

The `imageable` relation on the `Photo` model will return either a `Staff` or `Order` instance, depending on which type of model owns the photo.

Polymorphic relation table structure

To help understand how this works, let's explore the database structure for a polymorphic relation:

```
staff
    id - integer
    name - string

orders
    id - integer
    price - integer

photos
    id - integer
    path - string
    imageable_id - integer
    imageable_type - string
```

The key fields to notice here are the `imageable_id` and `imageable_type` on the `photos` table. The ID will contain the ID value of, in this example, the owning staff or order, while the type will contain the class name of the owning model. This is what allows the ORM to determine which type of owning model to return when accessing the `imageable` relation.

Note: When accessing or loading the relation, Orator will retrieve the related class using the `imageable_type` column value.

By default it will assume this value is the table name of the related model, so in this example `staff` or `orders`. If you want to override this convention, just add the `__morph_name__` attribute to the related class:

```
class Order(Model):

    __morph_name__ = 'order'
```

Many To Many polymorphic relations

New in version 0.3.

Polymorphic Many To Many Relation Table Structure

In addition to traditional polymorphic relations, you can also specify many-to-many polymorphic relations. For example, a blog `Post` and `Video` model could share a polymorphic relation to a `Tag` model. First, let's examine the table structure:

```
posts
    id - integer
    name - string

videos
    id - integer
    name - string

tags
    id - integer
    name - string

taggables
    tag_id - integer
    taggable_id - integer
    taggable_type - string
```

The `Post` and `Video` model will both have a `morph_to_many` relationship via a `tags` method:

```
class Post(Model):

    @morph_to_many('taggable')
    def tags(self):
        return Tag
```

The `Tag` model can define a method for each of its relationships:

```
class Tag(Model):

    @morphed_by_many('taggable')
    def posts(self):
        return Post

    @morphed_by_many('taggable')
    def videos(self):
        return Video
```

Querying relations

Querying relations when selection

When accessing the records for a model, you may wish to limit the results based on the exeistence of a relationship. For example, you may wish to retrieve all blog posts that have at least one comment. To actually do so, you can use the `has` method:

```
posts = Post.has('comments').get()
```

This would execute the following SQL query:

```
SELECT * FROM posts
WHERE (
    SELECT COUNT(*) FROM comments
    WHERE comments.post_id = posts.id
) >= 1
```

You can also specify an operator and a count:

```
posts = Post.has('comments', '>', 3).get()
```

This would execute:

```
SELECT * FROM posts
WHERE (
    SELECT COUNT(*) FROM comments
    WHERE comments.post_id = posts.id
) > 3
```

Nested `has` statements can also be constructed using "dot" notation:

```
posts = Post.has('comments.votes').get()
```

And the corresponding SQL query:

```
SELECT * FROM posts
WHERE (
    SELECT COUNT(*) FROM comments
    WHERE comments.post_id = posts.id
    AND (
        SELECT COUNT(*) FROM votes
        WHERE votes.comment_id = comments.id
    ) >= 1
) >= 1
```

If you need even more power, you can use the `where_has` and `or_where_has` methods to put "where" conditions on your has queries:

```
posts = Post.where_has(
    'comments',
    lambda q: q.where('content', 'like', 'foo%')
).get()
```

Dynamic properties

The Orator ORM allows you to access your relations via dynamic properties. It will automatically load the relationship for you. It will then be accessible via a dynamic property by the same name as the relation. For example, with the following model `Post`:

```
class Phone(Model):

    @belongs_to
    def user(self):
```

```
        return User

phone = Phone.find(1)
```

You can then print the user's email like this:

```
print(phone.user.email)
```

Now, for one-to-many relationships:

```
class Post(Model):

    @has_many
    def comments(self):
        return Comment

post = Post.find(1)
```

You can then access the post's comments like this:

```
comments = post.comments
```

If you need to add further constraints to which comments are retrieved, you may call the `comments` method and continue chaining conditions:

```
comments = post.comments().where('title', 'foo').first()
```

Note: Relationships that return many results will return an instance of the `Collection` class.

Eager loading

Eager loading exists to alleviate the N + 1 query problem. For example, consider a `Book` that is related to an `Author`:

```
class Book(Model):

    @belongs_to
    def author(self):
        return Author
```

Now, consider the following code:

```
for book in Book.all():
    print(book.author.name)
```

This loop will execute 1 query to retrieve all the books on the table, then another query for each book to retrieve the author. So, if we have 25 books, this loop will run 26 queries.

To drastically reduce the number of queries you can use eager loading. The relationships that should be eager loaded can be specified via the `with_` method.

```
for book in Book.with_('author').get():
    print(book.author.name)
```

In this loop, only two queries will be executed:

```
SELECT * FROM books

SELECT * FROM authors WHERE id IN (1, 2, 3, 4, 5, ...)
```

You can eager load multiple relationships at one time:

```
books = Book.with_('author', 'publisher').get()
```

You can even eager load nested relationships:

```
books = Book.with_('author.contacts').get()
```

In this example, the `author` relationship will be eager loaded as well as the author's `contacts` relation.

Eager load constraints

Sometimes you may wish to eager load a relationship but also specify a condition for the eager load. Here's an example:

```
users = User.with_({
    'posts': Post.query().where('title', 'like', '%first%')
}).get()
```

In this example, we're eager loading the user's posts only if the post's title contains the word "first".

When passing a query as a constraint, only the where clause is supported, if you want to be more specific you can use a callback:

```
users = User.with_({
    'posts': lambda q: q.where('title', 'like', '%first%').order_by('created_at',␣
↪'desc')
})
```

Lazy eager loading

It is also possible to eagerly load related models directly from an already existing model collection. This may be useful when dynamically deciding whether to load related models or not, or in combination with caching.

```
books = Book.all()

books.load('author', 'publisher')
```

You can also pass conditions:

```
books.load({
   'author': Author.query().where('name', 'like', '%foo%')
})
```

Inserting related models

You will often need to insert new related models, like inserting a new comment for a post. Instead of manually setting the `post_id` foreign key, you can insert the new comment from its parent `Post` model directly:

```
comment = Comment(message='A new comment')

post = Post.find(1)

comment = post.comments().save(comment)
```

If you need to save multiple comments:

```
comments = [
    Comment(message='Comment 1'),
    Comment(message='Comment 2'),
    Comment(message='Comment 3')
]

post = Post.find(1)

post.comments().save_many(comments)
```

Associating models (Belongs To)

When updatings a `belongs_to` relationship, you can use the associate method:

```
account = Account.find(1)

user.account().associate(account)

user.save()
```

Inserting related models (Many to Many)

You can also insert related models when working with many-to-many relationship. For example, with `User` and `Roles` models:

Attaching many to many models

```
user = User.find(1)
role = Roles.find(3)

user.roles().attach(role)

# or
user.roles().attach(3)
```

You can also pass a dictionary of attributes that should be stored on the pivot table for the relation:

```
user.roles().attach(3, {'expires': expires})
```

The opposite of `attach` is `detach`:

```
user.roles().detach(3)
```

Both `attach` and `detach` also take list of IDs as input:

```
user = User.find(1)

user.roles().detach([1, 2, 3])

user.roles().attach([{1: {'attribute1': 'value1'}}, 2, 3])
```

Using sync to attach many to many models

You can also use the `sync` method to attach related models. The `sync` method accepts a list of IDs to place on the pivot table. After this operation, only the IDs in the list will be on the pivot table:

```
user.roles().sync([1, 2, 3])
```

Adding pivot data when syncing

You can also associate other pivot table values with the given IDs:

```
user.roles().sync([{1: {'expires': True}}])
```

Sometimes you might want to create a new related model and attach it in a single command. For that, you can use the save method:

```
role = Role(name='Editor')

User.find(1).roles().save(role)
```

You can also pass attributes to place on the pivot table:

```
User.find(1).roles().save(role, {'expires': True})
```

Touching parent timestamps

When a model `belongs_to` another model, like a `Comment` belonging to a `Post`, it is often helpful to update the parent's timestamp when the chil model is updated. For instance, when a `Comment` model is updated, you may want to automatically touch the `updated_at` timestamp of the owning `Post`. For this to actually happen, you just have to add a `__touches__` property containing the names of the relationships:

```
class Comment(Model):

    __touches__ = ['posts']

    @belongs_to
    def post(self):
        return Post
```

Now, when you update a `Comment`, the owning `Post` will have its `updated_at` column updated.

Working with pivot table

Working with many-to-many reationships requires the presence of an intermediate table. Orator makes it easy to interact with this table. Let's take the `User` and `Roles` models and see how you can access the `pivot` table:

```
user = User.find(1)

for role in user.roles:
    print(role.pivot.created_at)
```

Note that each retrieved `Role` model is automatically assigned a `pivot` attribute. This attribute contains e model instance representing the intermediate table, and can be used as any other model.

By default, only the keys will be present on the `pivot` object. If your pivot table contains extra attributes, you must specify them when defining the relationship:

```
class User(Model):

    @belongs_to_many(with_pivot=['foo', 'bar'])
    def roles(self):
        return Role
```

Now the `foo` and `bar` attributes will be accessible on the `pivot` object for the `Role` model.

If you want your pivot table to have automatically maintained `created_at` and `updated_at` timestamps, use the `with_timestamps` keyword argument on the relationship definition:

```
class User(Model):

    @belongs_to_many(with_timestamps=True)
    def roles(self):
        return Role
```

Deleting records on a pivot table

To delete all records on the pivot table for a model, you can use the `detach` method:

```
User.find(1).roles().detach()
```

Updating a record on the pivot table

Sometimes you may need to update your pivot table, but not detach it. If you wish to update your pivot table in place you may use `update_existing_pivot` method like so:

```
User.find(1).roles().update_existing_pivot(role_id, attributes)
```

Timestamps

By default, the ORM will maintain the `created_at` and `updated_at` columns on your database table automatically. Simply add these `timestamp` columns to your table. If you do not wish for the ORM to maintain these columns, just add the `__timestamps__` property:

```
class User(Model):

    __timestamps__ = False
```

Providing a custom timestamp format

If you whish to customize the format of your timestamps (the default is the ISO Format) that will be returned when using the `serialize` or the `to_json` methods, you can override the `get_date_format` method:

```
class User(Model):

    def get_date_format(self):
        return 'DD-MM-YY'
```

Query Scopes

Defining a query scope

Scopes allow you to easily re-use query logic in your models. To define a scope, simply prefix a model method with `scope`:

```
class User(Model):

    def scope_popular(self, query):
        return query.where('votes', '>', 100)

    def scope_women(self, query):
        return query.where_gender('W')
```

Using a query scope

```
users = User.popular().women().order_by('created_at').get()
```

Dynamic scopes

Sometimes you may wish to define a scope that accepts parameters. Just add your parameters to your scope function:

```
class User(Model):

    def scope_of_type(self, query, type):
        return query.where_type(type)
```

Then pass the parameter into the scope call:

```
users = User.of_type('member').get()
```

Global Scopes

Sometimes you may wish to define a scope that applies to all queries performed on a model. In essence, this is how Orator's own "soft delete" feature works. Global scopes are defined using a combination of mixins and an implementation of the `Scope` class.

First, let's define a mixin. For this example, we'll use the `SoftDeletes` that ships with Orator:

```
from orator import SoftDeletingScope


class SoftDeletes(object):

    @classmethod
    def boot_soft_deletes(cls, model_class):
        """
        Boot the soft deleting mixin for a model.
        """
        model_class.add_global_scope(SoftDeletingScope())
```

If an Orator model inherits from a mixin that has a method matching the `boot_name_of_trait` naming convention, that mixin method will be called when the Orator model is booted, giving you an opportunity to register a global scope, or do anything else you want. A scope must be an instance of the `Scope` class, which specifies two methods: `apply` and `remove`.

The apply method receives an `Builder` query builder object and the `Model` it's applied to, and is responsible for adding any additional `where` clauses that the scope wishes to add. The `remove` method also receives a `Builder` object and `Model` and is responsible for reversing the action taken by `apply`. In other words, `remove` should remove the `where` clause (or any other clause) that was added. So, for our `SoftDeletingScope`, it would look something like this:

```
from orator import Scope


class SoftDeletingScope(Scope):

    def apply(self, builder, model):
        """
        Apply the scope to a given query builder.

        :param builder: The query builder
        :type builder: orator.orm.builder.Builder

        :param model: The model
        :type model: orator.orm.Model
        """
        builder.where_null(model.get_qualified_deleted_at_column())

    def remove(self, builder, model):
        """
        Remove the scope from a given query builder.

        :param builder: The query builder
        :type builder: orator.orm.builder.Builder

        :param model: The model
        :type model: orator.orm.Model
        """
```

```
        column = model.get_qualified_deleted_at_column()

        query = builder.get_query()

        wheres = []
        for where in query.wheres:
            # If the where clause is a soft delete date constraint,
            # we will remove it from the query and reset the keys
            # on the wheres. This allows the developer to include
            # deleted model in a relationship result set that is lazy loaded.
            if not self._is_soft_delete_constraint(where, column):
                wheres.append(where)

        query.wheres = wheres
```

Accessors & mutators

Orator provides a convenient way to transform your model attributes when getting or setting them.

Defining an accessor

Simply use the `accessor` decorator on your model to declare an accessor:

```
from orator.orm import Model, accessor


class User(Model):

    @accessor
    def first_name(self):
        first_name = self.get_raw_attribute('first_name')

        return first_name[0].upper() + first_name[1:]
```

In the example above, the `first_name` column has an accessor.

Note: The name of the decorated function **must** match the name of the column being accessed.

Defining a mutator

Mutators are declared in a similar fashion:

```
from orator.orm import Model, mutator


class User(Model):

    @mutator
    def first_name(self, value):
        self.set_raw_attribute(value.lower())
```

Note: If the column being mutated already has an accessor, you can use it has a mutator:

```
from orator.orm import Model, accessor


class User(Model):

    @accessor
    def first_name(self):
        first_name = self.get_raw_attribute('first_name')

        return first_name[0].upper() + first_name[1:]

    @first_name.mutator
    def set_first_name(self, value):
        self.set_raw_attribute(value.lower())
```

The inverse is also possible:

```
from orator.orm import Model, mutator


class User(Model):

    @mutator
    def first_name(self, value):
        self.set_raw_attribute(value.lower())

    @first_name.accessor
    def get_first_name(self):
        first_name = self.get_raw_attribute('first_name')

        return first_name[0].upper() + first_name[1:]
```

Date mutators

By default, the ORM will convert the `created_at` and `updated_at` columns to instances of Arrow, which eases date and datetime manipulation while behaving pretty much like the native Python date and datetime.

You can customize which fields are automatically mutated, by either adding them with the `__dates__` property or by completely overriding the `get_dates` method:

```
class User(Model):

    __dates__ = ['synchronized_at']
```

```
class User(Model):

    def get_dates(self):
        return ['created_at']
```

When a column is considered a date, you can set its value to a UNIX timestamp, a date string `YYYY-MM-DD`, a datetime string, a native `date` or `datetime` and of course an `Arrow` instance.

To completely disable date mutations, simply return an empty list from the `get_dates` method.

```
class User(Model):

    def get_dates(self):
        return []
```

Attributes casting

If you have some attributes that you want to always convert to another data-type, you may add the attribute to the `__casts__` property of your model. Otherwise, you will have to define a mutator for each of the attributes, which can be time consuming. Here is an example of using the `__casts__` property:

```
__casts__ = {
    'is_admin': 'bool'
}
```

Now the `is_admin` attribute will always be cast to a boolean when you access it, even if the underlying value is stored in the database as an integer. Other supported cast types are: `int`, `float`, `str`, `bool`, `dict`, `list`.

The `dict` cast is particularly useful for working with columns that are stored as serialized JSON. For example, if your database has a TEXT type field that contains serialized JSON, adding the `dict` cast to that attribute will automatically deserialize the attribute to a dictionary when you access it on your model:

```
__casts__ = {
    'options': 'dict'
}
```

Now, when you utilize the model:

```
user = User.find(1)

# options is a dict
options = user.options

# options is automatically serialized back to JSON
user.options = {'foo': 'bar'}
```

Model events

Orator models fire several events, allowing you to hook into various points in the model's lifecycle using the following methods: `creating`, `created`, `updating`, `updated`, `saving`, `saved`, `deleting`, `deleted`, `restoring`, `restored`.

Whenever a new item is saved for the first time, the `creating` and `created` events will fire. If an item is not new and the `save` method is called, the `updating` / `updated` events will fire. In both cases, the `saving` / `saved` events will fire.

Cancelling save operations via events

If `False` is returned from the `creating`, `updating`, `saving`, or `deleting` events, the action will be cancelled:

```
User.creating(lambda user: user.is_valid())
```

Model observers

To consolidate the handling of model events, you can register a model observer. An observer class can have methods that correspond to the various model events. For example, `creating`, `updating`, `saving` methods can be on an observer, in addition to any other model event name.

So, for example, a model observer might look like this:

```
class UserObserver(object):

    def saving(user):
        # ...

    def saved(user):
        # ...
```

You can register an observer instance using the `observe` method:

```
User.observe(UserObserver())
```

Converting to dictionaries / JSON

Converting a model to a dictionary

When building JSON APIs, you may often need to convert your models and relationships to dictionaries or JSON. So, Orator includes methods for doing so. To convert a model and its loaded relationship to a dictionary, you may use the `serialize` method:

```
user = User.with_('roles').first()

return user.serialize()
```

Note that entire collections of models can also be converted to dictionaries:

```
return User.all().serialize()
```

Converting a model to JSON

To convert a model to JSON, you can use the `to_json` method!

```
return User.find(1).to_json()
```

Hiding attributes from dictionary or JSON conversion

Sometimes you may wish to limit the attributes that are included in you model's dictionary or JSON form, such as passwords. To do so, add a `__hidden__` property definition to you model:

```
class User(model):

    __hidden__ = ['password']
```

Alternatively, you may use the __visible__ property to define a whitelist:

```
__visible__ = ['first_name', 'last_name']
```

Appendable attributes

Occasionally, you may need to add dictionary attributes that do not have a corresponding column in your database. To do so, simply define an accessor for the value:

```
class User(Model):

    @accessor
    def is_admin(self):
        return self.get_raw_attribute('admin') == 'yes'
```

Once you have created the accessor, just add the value to the __appends__ property on the model:

```
class User(Model):

    __append__ = ['is_admin']

    @accessor
    def is_admin(self):
        return self.get_raw_attribute('admin') == 'yes'
```

Once the attribute has been added to the __appends__ list, it will be included in both the model's dictionary and JSON forms. Attributes in the __appends__ list respect the __visible__ and __hidden__ configuration on the model.

CHAPTER 5

Pagination

Paginate database records

There are several ways to paginate items. The simplest is by using the `paginate` method on the *Query Builder* or an *ORM* query. The `paginate` method provided by Orator automatically takes care of setting the proper limit and offset based on the current page. By default, the current page needs to be specified. However, as we'll see it later, you can create a custom current page resolver.

Paginating Query Builder results

First, let's take a look at calling the `paginate` method on a query generated by the query builder:

```
users = db.table('users').paginate(15, 2)
```

In this example, the first argument passed to paginate is the number of items we would like displayed "per page" and the second is the current page we want to display. So, in this case, we want to retrieve `15` items on page `2`.

Warning: Currently, pagination operations that use a `group_by` statement cannot be executed efficiently by Orator. If you need to use a `group_by` with a paginated result set, it is recommended that you query the database and create a paginator manually.

Paginating models

You can also paginate *ORM* queries. In this example, we will paginate the `User` model with `15` items per page for the second page. As you can see, the syntax is nearly identical to paginating query builder results:

```
all_users = User.paginate(15, 2)
```

Of course, you can call paginate after setting other constraints on the query:

```
some_users = User.where('votes', '>', 100).paginate(15, 2)
```

Simple pagination

If you only need "Next" and "Previous" pages in your pagination, you have the option of using the `simple_paginate` method to perform a more efficient query.

```
some_users = User.where('votes', '>', 100).simple_paginate(15, 2)
```

Creating a Paginator manually

Sometimes you may wish to create a pagination instance manually, passing it a list of items. You can do so by creating either a `Paginator` or a `LengthAwarePaginator` instance, depending on your needs.

The `Paginator` class does not need to know the total number of items in the result set; however, because of this, the class does not have methods to retrieve the index of the last page. The `LengthAwarePaginator` accepts almost the same arguments as the `Paginator`, except that it does require a count of the total number of items in the result set.

In other words, the `Paginator` corresponds to the `simple_paginate` method on the query builder and the ORM, while the `LengthAwarePaginator` corresponds to the `paginate` method.

Displaying Results

When you call the `paginate` or `simple_paginate` methods on a query builder or ORM query, you will receive a paginator instance. When calling the `paginate` method, you will receive an instance of `LengthAwarePaginator`. When calling the `simple_paginate` method, you will receive an instance of `Paginator`. These objects provide several methods and attributes that describe the result set. In addition to these helpers methods, the paginator instances are iterators and may be looped as a list.

```
for user in some_users:
    print(user.name)
```

Additional attributes and helper methods

You may also access additional pagination information via the following methods and attributes on paginator instances:

Method or attribute	Description
`results.count()`	Returns the number of results on the current page
`results.current_page`	The current page of the paginator
`results.has_more_pages()`	Returns `True` if there is more pages else `False`
`results.last_page`	The number of the last page (Not available when using `simple_paginate`)
`results.next_page`	The number of the next page if it exists else `None`
`results.per_page`	The number of results per page
`results.previous_page`	The number of the previous page if it exists else `None`
`results.total`	The total number of results (Not available when using `simple_paginate`

Converting Results To JSON

The Orator paginator result classes expose the `to_json` method, so it's very easy to convert your pagination results to JSON.

Be default, it will return the JSON formatted underlying `Collection`:

```
some_users = User.where('votes', '>', 100).paginate(15, 2)

print(some_users.to_json())
```

Setting a custom current page resolver

Sometimes, you may want to compute the current page based on external parameters. For that purpose, you can set a custom current page resolver on the `Paginator` class. In this example, the current page will be determined by a `?page` query string parameter:

```
from orator import Paginator
from flask import request


def current_page_resolver():
    return int(request.args.get('page', 1))

Paginator.current_page_resolver(current_page_resolver)
```

So from now on, It is no longer necessary to specify the current page:

```
some_users = User.where('votes', '>', 100).paginate(15)
```

CHAPTER 6

Schema Builder

Introduction

The `Schema` class provides a database agnostic way of manipulating tables.

Before getting started, be sure to have configured a `DatabaseManager` as seen in the *Basic Usage* section.

```
from orator import DatabaseManager, Schema

config = {
    'mysql': {
        'driver': 'mysql',
        'host': 'localhost',
        'database': 'database',
        'username': 'root',
        'password': '',
        'prefix': ''
    }
}

db = DatabaseManager(config)
schema = Schema(db)
```

Creating and dropping tables

To create a new database table, the `create` method is used:

```
with schema.create('users') as table:
    table.increments('id')
```

The `table` variable is a `Blueprint` instance which can be used to define the new table.

To rename an existing database table, the `rename` method can be used:

```
schema.rename('from', 'to')
```

To specify which connection the schema operation should take place on, use the `connection` method:

```
with schema.connection('foo').create('users') as table:
    table.increments('id')
```

To drop a table, you can use the `drop` or `drop_if_exists` methods:

```
schema.drop('users')

schema.drop_if_exists('users')
```

Adding columns

To update an existing table, you can use the `table` method:

```
with schema.table('users') as table:
    table.string('email')
```

The table builder contains a variety of column types that you may use when building your tables:

Command	Description
`table.big_increments('id')`	Incrementing ID using a "big integer" equivalent
`table.big_integer('votes')`	BIGINT equivalent to the table
`table.binary('data')`	BLOB equivalent to the table
`table.boolean('confirmed')`	BOOLEAN equivalent to the table
`table.char('name', 4)`	CHAR equivalent with a length
`table.date('created_on')`	DATE equivalent to the table
`table.datetime('created_at')`	DATETIME equivalent to the table
`table.decimal('amount', 5, 2)`	DECIMAL equivalent to the table with a precision and scale
`table.double('column', 15, 8)`	DOUBLE equivalent to the table with precision, 15 digits in total and 8 afte
`table.enum('choices', ['foo', 'bar'])`	ENUM equivalent to the table
`table.float('amount')`	FLOAT equivalent to the table
`table.increments('id')`	Incrementing ID to the table (primary key)
`table.integer('votes')`	INTEGER equivalent to the table
`table.json('options')`	JSON equivalent to the table
`table.long_text('description')`	LONGTEXT equivalent to the table
`table.medium_integer('votes')`	MEDIUMINT equivalent to the table
`table.medium_text('description')`	MEDIUMTEXT equivalent to the table
`table.morphs('taggable')`	Adds INTEGER `taggable_id` and STRING `taggable_type`
`table.nullable_timestamps()`	Same as `timestamps()`, except allows NULLs
`table.small_integer('votes')`	SMALLINT equivalent to the table
`table.soft_deletes()`	Adds **deleted_at** column for soft deletes
`table.string('email')`	VARCHAR equivalent column
`table.string('votes', 100)`	VARCHAR equivalent with a length
`table.text('description')`	TEXT equivalent to the table
`table.time('sunrise')`	TIME equivalent to the table
`table.timestamp('added_at')`	TIMESTAMP equivalent to the table
`table.timestamps()`	Adds **created_at** and **updated_at** columns

Contin

Table 6.1 – continued from previous page

Command	Description
`.nullable()`	Designate that the column allows NULL values
`.default(value)`	Declare a default value for a column
`.unsigned()`	Set INTEGER to UNSIGNED

Changing columns

Sometimes you may need to modify an existing column. For example, you may wish to increase the size of a string column. To do so, you can use the `change` method. For example, let's increase the size of the `name` column from 25 to 50:

```
with schema.table('users') as table:
    table.string('name', 50).change()
```

You could also modify the column to be nullable:

```
with schema.table('user') as table:
    table.string('name', 50).nullable().change()
```

Warning: The column change feature, while tested, is still considered in **beta** stage. Please report any encountered issue or bug on the Github project

Renaming columns

To rename a column, you can use use the `rename_column` method on the Schema builder:

```
with schema.table('users') as table:
    table.rename('from', 'to')
```

Warning: Prior to **MySQL 5.6.6**, foreign keys are **NOT** automatically updated when renaming columns. Therefore, you will need to **drop** the foreign key constraint, **rename** the column and **recreate** the constraint to avoid an error.

```
with schema.table('posts') as table:
    table.drop_foreign('posts_user_id_foreign')
    table.rename('user_id', 'author_id')
    table.foreign('author_id').references('id').on('users')
```

In future versions, Orator **might** handle this automatically.

Warning: The rename column feature, while tested, is still considered in **beta** stage (especially for SQLite). Please report any encountered issue or bug on the Github project

Dropping columns

To drop a column, you can use use the `drop_column` method on the Schema builder:

Dropping a column from a database table

```
with schema.table('users') as table:
    table.drop_column
```

Dropping multiple columns from a database table

```
with schema.table('users') as table:
    table.drop_column('votes', 'avatar', 'location')
```

Checking existence

You can easily check for the existence of a table or column using the `has_table` and `has_column` methods:

Checking for existence of a table

```
if schema.has_table('users'):
    # ...
```

Checking for existence of a column:

```
if schema.has_column('users', 'email'):
    # ...
```

Adding indexes

The schema builder supports several types of indexes. There are two ways to add them. First, you may fluently define them on a column definition:

```
table.string('email').unique()
```

Or, you may choose to add the indexes on separate lines. Below is a list of all available index types:

Command	Description
`table.primary('id')`	Adds a primary key
`table.primary(['first', 'last'])`	Adds composite keys
`table.unique('email')`	Adds a unique index
`table.index('state')`	Adds a basic index

Dropping indexes

To drop an index you must specify the index's name. Orator assigns a reasonable name to the indexes by default. Simply concatenate the table name, the names of the column in the index, and the index type. Here are some examples:

Command	Description
`table.drop_primary('user_id_primary')`	Drops a primary key from the "users" table
`table.drop_unique('user_email_unique')`	Drops a unique index from the "users" table
`table.drop_index('geo_state_index')`	Drops a basic index from the "geo" table

Foreign keys

Orator also provides support for adding foreign key constraints to your tables:

```
table.integer('user_id').unsigned()
table.foreign('user_id').references('id').on('users')
```

In this example, we are stating that the `user_id` column references the `id` column on the `users` table. Make sure to create the foreign key column first!

You may also specify options for the "on delete" and "on update" actions of the constraint:

```
table.foreign('user_id')\
    .references('id').on('users')\
    .on_delete('cascade')
```

To drop a foreign key, you may use the `drop_foreign` method. A similar naming convention is used for foreign keys as is used for other indexes:

```
table.drop_foreign('posts_user_id_foreign')
```

Note: When creating a foreign key that references an incrementing integer, remember to always make the foreign key column `unsigned`.

Note: Changed in version 0.6.3.

By default, SQLite will not honor the `ON DELETE` and `ON UPDATE` statements. Orator takes care of the problem by executing the following SQL query:

```
PRAGMA foreign_keys = ON
```

If you do not want this behavior, just set the configuration parameter `foreign_keys` to `False`:

```
config = {
    'sqlite': {
        'driver': 'sqlite',
        'database': ':memory:',
        'foreign_keys': False
    }
}
```

Dropping timestamps and soft deletes

To drop the `timestamps`, `nullable_timestamps` or `soft_deletes` column types, you may use the following methods:

Command	Description
`table.drop_timestamps()`	Drops the **created_at** and **deleted_at** columns
`table.drop_soft_deletes()`	Drops the **deleted_at** column

CHAPTER 7

Migrations

Migrations are a type of version control for your database. They allow a team to modify the database schema and stay up to date on the current schema state. Migrations are typically paired with the *Schema Builder* to easily manage your database's schema.

Note: For the migrations to actually work, you need a configuration file describing your databases. It can be a `orator.yml` file or a `orator.py` located where the `orator` command is executed, or any other yaml or python file (these must then be explicitely specified when executing commands with the `--config\-c`) which follow the following requirements:

- yaml files must follow this structure:

```
databases:
    mysql:
        driver: mysql
        host: localhost
        database: database
        username: root
        password: ''
        prefix: ''
```

- python files must follow this structure

```
DATABASES = {
    'mysql': {
        'driver': 'mysql',
        'host': 'localhost',
        'database': 'database',
        'username': 'root',
        'password': '',
        'prefix': ''
    }
}
```

Creating Migrations

To create a migration, you can use the `migrations:make` command on the Orator CLI:

```
orator migrations:make create_users_table
```

This will create a migration file that looks like this:

```
from orator.migrations import Migration


class CreateTableUsers(Migration):

    def up(self):
        """
        Run the migrations.
        """
        pass

    def down(self):
        """
        Revert the migrations.
        """
        pass
```

By default, the migration will be placed in a `migrations` folder relative to where the command has been executed, and will contain a timestamp which allows the framework to determine the order of the migrations.

If you want the migrations to be stored in another folder, use the `--path/-p` option:

```
orator migrations:make create_users_table -p my/path/to/migrations
```

The `--table` and `--create` options can also be used to indicate the name of the table, and whether the migration will be creating a new table:

```
orator migrations:make add_votes_to_users_table --table=users

orator migrations:make create_users_table --table=users --create
```

These commands would respectively create the following migrations:

```
from orator.migrations import Migration


class AddVotesToUsersTable(Migration):

    def up(self):
        """
        Run the migrations.
        """
        with self.schema.table('users') as table:
            pass

    def down(self):
        """
        Revert the migrations.
        """
```

```
        with self.schema.table('users') as table:
            pass
```

```
from orator.migrations import Migration


class CreateTableUsers(Migration):

    def up(self):
        """
        Run the migrations.
        """
        with self.schema.create('users') as table:
            table.increments('id')
            table.timestamps()

    def down(self):
        """
        Revert the migrations.
        """
        self.schema.drop('users')
```

Note: `Migration` instances have a `db` attribute which is an instance of the current `Connection`.

Running Migrations

To run all outstanding migrations, just use the `migrations:run` command:

```
orator migrations:run -c databases.py
```

Note: By default, all migrations are run inside a transaction. If you want queries to be executed directly just set the `transactional` attribute to `False`. You then must explicitely declare the transactions:

```
class CreateTableUsers(Migration):

    transactional = False

    def up(self):
        """
        Run the migrations.
        """
        with self.db.transaction():
            with self.schema.create('users') as table:
                table.increments('id')
                table.timestamps()

    def down(self):
        """
        Revert the migrations.
        """
        with self.db.transaction():
            self.schema.drop('users')
```

Rolling back migrations

Rollback the last migration operation

```
orator migrations:rollback
```

Rollback all migrations

```
orator migrations:reset
```

Getting migrations status

To see the status of the migrations, just use the `migrations:status` command:

```
orator migrations:status
```

This would output something like this:

```
+-----------------------------------------------------+------+
| Migration                                           | Ran? |
+-----------------------------------------------------+------+
| 2015_05_02_04371430559457_create_users_table        | Yes  |
| 2015_05_04_02361430725012_add_votes_to_users_table | No   |
+-----------------------------------------------------+------+
```

CHAPTER 8

Seeding

Introduction

Orator includes a simple method of seeding your database with test data using seed classes. Seed classes may have any name you wish, but probably should follow some sensible convention, such as `UsersTableSeeder`, etc. By default, a `DatabaseSeeder` class is defined for you the first time you create a seed class. From this class, you can use the `call` method to run other seed classes, allowing you to control the seeding order.

Writing Seeders

To generate a seeder, you may issue the `seeders:make` command. All seeders generated by the command will be placed in a `seeders` directory relative to where the command has been executed:

```
orator seeds:make user_table_seeder
```

Note: If you want the seed classes to be stored in another directory, use the `-p/--path` option

A seeder class only contains one method by default: `run`. This method is called when the `db:seed` command is executed. Within the `run` method, you can insert data into your database however you wish. You can use the *Query Builder* to manually insert data or you can use *Model Factories*.

As an example, let's modify the `UsersTableSeeder` class you just created. Let's add a database insert statement to the `run` method:

```
from orator.seeds import Seeder


class UsersTableSeeder(Seeder):

    def run(self):
```

```
        """
        Run the database seeds.
        """

        # Here you could just use random string generators
        # rather than hardcoded values
        self.db.table('users').insert({
            'name': 'john',
            'email': 'john@doe.com'
        })
```

Using Model Factories

Of course, manually specifying the attributes for each model seed is cumbersome. Instead, you can use *Model Factories* to conveniently generate large amounts of database records. First, review the *model factory documentation* to learn how to define your factories. You can use an external factory or use the seeder class `factory` attribute.

For example, let's create 50 users and attach a relationship to each user:

```
from orator.seeds import Seeder
from orator.orm import Factory


factory = Factory()


@factory.define(User)
def users_factory(faker):
    return {
        'name': faker.name(),
        'email': faker.email()
    }


@factory.define(Post)
def posts_factory(faker):
    return {
        'title': faker.name(),
        'content': faker.text()
    }


class UsersTableSeeder(Seeder):

    factory = factory  # This is only needed when using an external factory

    def run(self):
        """
        Run the database seeds.
        """
        self.factory(User, 50).create().each(
            lambda u: u.posts().save(self.factory(Post).make())
        )
```

Or using directly the `factory` attribute without an external factory:

```
class UsersTableSeeder(Seeder):

    def run(self):
        """
        Run the database seeds.
        """
        self.factory.register(User, self.users_factory)
        self.factory.register(Post, self.posts_factory)

        self.factory(User, 50).create().each(
            lambda u: u.posts().save(self.factory(Post).make())
        )

    def users_factory(self, faker):
        return {
            'name': faker.name(),
            'email': faker.email()
        }

    def posts_factory(self, faker):
        return {
            'title': faker.name(),
            'content': faker.text()
        }
```

Calling Additional Seeders

Within the `DatabaseSeeder` class, you can use the `call` method to execute additional seed classes. Using the `call` method allows you to break up your database seeding into multiple files so that no single seeder class becomes overwhelmingly large. Simply pass the seeder class you wish to run:

```
def run(self):
    """
    Run the database seeds.
    """
    self.call(UsersTableSeeder)
    self.call(PostsTableSeeder)
    self.call(CommentsTableSeeder)
```

Running Seeders

Once you have written your seeder classes, you may use the `db:seed` command to seed your database. By default, the `db:seed` command runs the `database_seeder` file, which can be used to call other seed classes. However, you can use the `--seeder` option to specify a specific seeder class to run individually:

```
orator db:seed

orator db:seed --seeder users_table_seeder
```

You can also seed your database using the `migrations:refresh` command, which will also rollback and re-run all of your migrations. This command is useful for completely re-building your database:

```
orator migrations:refresh --seed
```

CHAPTER 9

Testing

Model Factories

When testing, it is common to need to insert a few records into your database before executing your test. Instead of manually specifying the value of each column when you create this test data, Orator allows you to define a default set of attributes for each of your models using "factories":

```
from orator.orm import Factory

factory = Factory()

@factory.define(User)
def users_factory(faker):
    return {
        'name': faker.name(),
        'email': faker.email()
    }
```

Within the function (here `users_factory`), which serves as the factory definition, you can return the default test values of all attributes on the model. The function will receive an instance of the Faker library, which allows you to conveniently generate various kinds of random data for testing.

Multiple Factory Types

Sometimes you may wish to have multiple factories for the same model class. For example, perhaps you would like to have a factory for "Administrator" users in addition to normal users. You can define these factories using the `define_as` method:

```
@factory.define_as(User, 'admin')
def admins_factory(faker):
    return {
        'name': faker.name(),
        'email': faker.email(),
```

```
        'admin': True
    }
```

Instead of duplicating all of the attributes from your base user factory, you can use the `raw` method to retrieve the base attributes. Once you have the attributes, simply supplement them with any additional values you require:

```
@factory.define_as(User, 'admin')
def admins_factory(faker):
    user = factory.raw(User)

    user.update({
        'admin': True
    })

    return user
```

Using Factories In Tests

Once you have defined your factories, you can use them in your tests or database seed files to generate model instances calling the `Factory` instance. So, let's take a look at a few examples of creating models. First, we'll use the `make` method, which creates models but does not save them to the database:

```
def test_database():
    user = factory(User).make()

    # Use model in tests
```

If you would like to override some of the default values of your models, you can pass keyword arguments to the `make` method. Only the specified values will be replaced while the rest of the values remain set to their default values as specified by the factory:

```
def test_database():
    user = factory(User).make(name='John')
```

You can also create a `Collection` of many models or create models of a given type:

```
# Create 3 User instances
users = factory(User, 3).make()

# Create a User "admin" instance
admin = factory(User, 'admin').make()

# Create three User "admin" instances
admins = factory(User, 'admin', 3).make()
```

Persisting Factory Models

The `create` method not only creates the model instances, but also saves them to the database using models' `save` method:

```
def test_database():
    user = factory(User).create()

    # Use model in tests
```

Again, you can override attributes on the model by passing an array to the `create` method:

```
def test_database():
    user = factory(User).create(name='John')
```

Adding Relations To Models

You may even persist multiple models to the database. In this example, we'll even attach a relation to the created models. When using the `create` method to create multiple models, a `Collection` instance is returned, allowing you to use any of the convenient functions provided by the collection, such as `each`:

```
users = factory(User, 3).create()
users.each(lambda u: u.save(factory(Post).make()))
```

CHAPTER 10

Collections

Introduction

The `Collection` class provides a fluent, convenient wrapper for working with list or dictionaries of data.

It's behind every ORM queries that return multiple results. For example, check out the following code:

```
users = User.all()
names = users.map(lambda user: user.name.lower())
names = names.reject(lambda name: len(name) == 0)
```

This returns very users names that are not empty.

Available Methods

For the remainder of this documentation, we'll discuss each method available on the `Collection` class. Remember, all of these methods may be chained for fluently manipulating the underlying list or dict. Furthermore, almost every method returns a new `Collection` instance, allowing you to preserve the original copy of the collection when necessary.

You may select any method from this table to see an example of its usage:

- *all*
- *avg*
- *chunk*
- *collapse*
- *contains*
- *count*
- *diff*

- *each*
- *every*
- *filter*
- *first*
- *flip*
- *forget*
- *for_page*
- *get*
- *implode*
- *is_empty*
- *keys*
- *last*
- *map*
- *merge*
- *only*
- *pluck*
- *pop*
- *prepend*
- *pull*
- *push*
- *put*
- *reduce*
- *reject*
- *reverse*
- *serialize*
- *shift*
- *sort*
- *sum*
- *take*
- *to_json*
- *transform*
- *unique*
- *values*
- *where*
- *without*
- *zip*

Methods Listing

all()

The `all` method simply returns the underlying list or dict represented by the collection:

```
Collection([1, 2, 3]).all()

# [1, 2, 3]
```

avg()

The `avg` method returns the average of all items in the collection:

```
Collection([1, 2, 3, 4, 5]).avg()

# 3
```

If the collection contains nested objects or dictionaries, you must pass a key to use for determining which values to calculate the average:

```
collection = Collection([
    {'name': 'JavaScript: The Good Parts', 'pages': 176},
    {'name': 'JavaScript: The Definitive Guide', 'pages': 1096}
])

collection.avg('pages')

# 636
```

chunk()

The `chunk` method breaks the collection into multiple, smaller collections of a given size:

```
collection = Collection([1, 2, 3, 4, 5, 6, 7])

chunks = collection.chunk(4)

chunks.serialize()

# [[1, 2, 3, 4], [5, 6, 7]]
```

collapse()

The `collapse` method collapses a collection of lists into a flat collection:

```
collection = Collection([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

collapsed = collection.collapse()

collapsed.all()
```

```
# [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

contains()

The `contains` method determines whether the collection contains a given item:

```
collection = Collection(['foo', 'bar'])

collection.contains('foo')

# True

collection = Collection({'foo': 'bar'})

collection.contains('foo')

# True
```

You can also use the `in` keyword:

```
'foo' in collection

# True
```

You can also pass a key / value pair to the `contains` method, which will determine if the given pair exists in the collection:

```
collection = Collection([
    {'name': 'John', 'id': 1},
    {'name': 'Jane', 'id': 2}
])

collection.contains('name', 'Simon')

# False
```

Finally, you may also pass a callback to the `contains` method to perform your own truth test:

```
collection = Collection([1, 2, 3, 4, 5])

collection.contains(lambda item: item > 5)

# False
```

count()

The `count` method returns the total number of items in the collection:

```
collection = Collection([1, 2, 3, 4])

collection.count()

# 4
```

The `len` function can also be used:

```
len(collection)

# 4
```

diff()

The `diff` method compares the collection against another collection, a `list` or a `dict`:

```
collection = Collection([1, 2, 3, 4, 5])

diff = collection.diff([2, 4, 6, 8])

diff.all()

# [1, 3, 5]
```

each()

The `each` method iterates over the items in the collection and passes each item to a given callback:

```
posts.each(lambda post: post.author().save(author))
```

Return `False` from your callback to break out of the loop:

```
posts.each(lambda post: post.author().save(author) if author.name == 'John' else␣
↪False)
```

every()

The `every` method creates a new collection consisting of every n-th element:

```
collection = Collection(['a', 'b', 'c', 'd', 'e', 'f'])

collection.every(4).all()

# ['a', 'e']
```

You can optionally pass the offset as the second argument:

```
collection.every(4, 1).all()

# ['b', 'f']
```

filter()

The `filter` method filters the collection by a given callback, keeping only those items that pass a given truth test:

```
collection = Collection([1, 2, 3, 4])

filtered = collection.filter(lambda item: item > 2)

filtered.all()

# [3, 4]
```

first()

The `first` method returns the first element in the collection that passes a given truth test:

```
collection = Collection([1, 2, 3, 4])

collection.first(lambda item: item > 2)

# 3
```

You can also call the `first` method with no arguments to get the first element in the collection. If the collection is empty, `None` is returned:

```
collection.first()

# 1
```

flip()

The `flip` method swaps the collection's keys with their corresponding values:

```
collection = Collection({'name': 'john', 'votes': 100})

flipped = collection.flip()

flipped.all()

# {'john': 'name', 100: 'votes'}
```

forget()

The `forget` method removes an item from the collection by its key:

```
collection = Collection({'name': 'john', 'votes': 100})

collection.forget('name')

collection.all()

# {'votes': 100}
```

Warning: Unlike most other collection methods, `forget` does not return a new modified collection; it modifies the collection it is called on.

for_page

The `for_page` method returns a new collection containing the items that would be present on a given page number:

```
collection = Collection([1, 2, 3, 4, 5, 6, 7, 8, 9])

chunk = collection.for_page(2, 3)

chunk.all()

# 4, 5, 6
```

The method requires the page number and the number of items to show per page, respectively.

get()

The `get` method returns the item at a given key. If the key does not exist, `None` is returned:

```
collection = Collection({'name': 'john', 'votes': 100})

collection.get('name')

# john

collection = Collection([1, 2, 3])

collection.get(3)

# None
```

You can optionally pass a default value as the second argument:

```
collection = Collection({'name': 'john', 'votes': 100})

collection.get('foo', 'default-value')

# default-value
```

implode()

The `implode` method joins the items in a collection. Its arguments depend on the type of items in the collection.

If the collection contains dictionaries or objects, you must pass the key of the attributes you wish to join, and the "glue" string you wish to place between the values:

```
collection = Collection([
    {'account_id': 1, 'product': 'Desk'},
    {'account_id': 2, 'product': 'Chair'}
])

collection.implode('product', ', ')

# Desk, Chair
```

If the collection contains simple strings, simply pass the "glue" as the only argument to the method:

```
collection = Collection(['foo', 'bar', 'baz'])

collection.implode('-')

# foo-bar-baz
```

is_empty()

The `is_empty` method returns `True` if the collection is empty; otherwise, `False` is returned:

```
Collection([]).is_empty()

# True
```

keys()

The `keys` method returns all of the collection's keys:

```
collection = Collection({
    'account_id': 1,
    'product': 'Desk'
})

keys = collection.keys()

keys.all()

# ['account_id', 'product']
```

last()

The `last` method returns the last element in the collection that passes a given truth test:

```
collection = Collection([1, 2, 3, 4])

last = collection.last(lambda item: item < 3)

# 2
```

You can also call the `last` method with no arguments to get the last element in the collection. If the collection is empty, `None` is returned:

```
collection.last()

# 4
```

map()

The `map` method iterates through the collection and passes each value to the given callback. The callback is free to modify the item and return it, thus forming a new collection of modified items:

```
collection = Collection([1, 2, 3, 4])

multiplied = collection.map(lambda item: item * 2)

multiplied.all()

# [2, 4, 6, 8]
```

Warning: Like most other collection methods, `map` returns a new `Collection` instance; it does not modify the collection it is called on. If you want to transform the original collection, use the *transform* method.

merge()

The merge method merges the given dict or list into the collection:

```
collection = Collection({
    'product_id': 1, 'name': 'Desk'
})

collection.merge({
    'price': 100,
    'discount': False
})

collection.all()

# {
#     'product_id': 1,
#     'name': 'Desk',
#     'price': 100,
#     'discount': False
# }
```

For lists collections, the values will be appended to the end of the collection:

```
collection = Collection(['Desk', 'Chair'])

collection.merge(['Bookcase', 'Door'])

collection.all()

# ['Desk', 'Chair', 'Bookcase', 'Door']
```

Warning: Unlike most other collection methods, `merge` does not return a new modified collection; it modifies the collection it is called on.

only()

The `only` method returns the items in the collection with the specified keys:

```
collection = Collection({
    'product_id': 1,
    'name': 'Desk',
    'price': 100,
    'discount': False
})

filtered = collection.only('product_id', 'name')

filtered.all()

# {'product_id': 1, 'name': 'Desk'}
```

For the inverse of `only`, see the *without* method.

pluck()

The `pluck` method retrieves all of the collection values for a given key:

```
collection = Collection([
    {'product_id': 1, 'product': 'Desk'},
    {'product_id': 2, 'product': 'Chair'}
])

plucked = collection.pluck('product')

plucked.all()

# ['Desk', 'Chair']
```

You can also specify how you wish the resulting collection to be keyed:

```
plucked = collection.pluck('name', 'product_id')

plucked.all()

# {1: 'Desk', 2: 'Chair'}
```

pop()

The `pop` method removes and returns the last item from the collection:

```
collection = Collection([1, 2, 3, 4, 5])

collection.pop()

# 5

collection.all()

# [1, 2, 3, 4]
```

prepend()

The prepend method adds an item to the beginning of the collection:

```
collection = Collection([1, 2, 3, 4])

collection.prepend(0)

collection.all()

# [0, 1, 2, 3, 4]
```

pull()

The pull method removes and returns an item from the collection by its key:

```
collection = Collection({
    'product_id': 1, 'product': 'Desk'
})

collection.pull('product_id')

collection.all()

# {'product': 'Desk'}
```

push()/append()

The push (or append) method appends an item to the end of the collection:

```
collection = Collection([1, 2, 3, 4])

collection.push(5)

collection.all()

# [1, 2, 3, 4, 5]
```

put()

The put method sets the given key and value in the collection:

```
collection = Collection({
    'product_id': 1, 'product': 'Desk'
})

collection.put('price', 100)

collection.all()

# {'product_id': 1, 'product': 'Desk', 'price': 100}
```

Note: It is equivalent to:

```
collection['price'] = 100
```

reduce()

The `reduce` method reduces the collection to a single value, passing the result of each iteration into the subsequent iteration:

```
collection = Collection([1, 2, 3])

collection.reduce(lambda result, item: (result or 0) + item)

# 6
```

The value for `result` on the first iteration is `None`; however, you can specify its initial value by passing a second argument to reduce:

```
collection.reduce(lambda result, item: result + item, 4)

# 10
```

reject()

The `reject` method filters the collection using the given callback. The callback should return `True` for any items it wishes to remove from the resulting collection:

```
collection = Collection([1, 2, 3, 4])

filtered = collection.reject(lambda item: item > 2)

filtered.all()

# [1, 2]
```

For the inverse of `reject`, see the *filter* method.

reverse()

The `reverse` method reverses the order of the collection's items:

```
collection = Collection([1, 2, 3, 4, 5])

reverse = collection.reverse()

reverse.all()

# [5, 4, 3, 2, 1]
```

serialize

The `serialize` method converts the collection into a `dict` or a `list`. If the collection's values are *ORM* models, the models will also be converted to dictionaries:

```
collection = Collection({'name': 'Desk', 'product_id': 1})

collection.serialize()

# {'name': 'Desk', 'product_id': 1}

collection = Collection([User.find(1)])

collection.serialize()

# [{'id': 1, 'name': 'John'}]
```

shift()

The `shift` method removes and returns the first item from the collection:

```
collection = Collection([1, 2, 3, 4, 5])

collection.shift()

# 1

collection.all()

# [2, 3, 4, 5]
```

sort()

The `sort` method sorts the collection:

```
collection = Collection([5, 3, 1, 2, 4])

sorted = collection.sort()

sorted.all()

# [1, 2, 3, 4, 5]
```

sum()

The `sum` method returns the sum of all items in the collection:

```
Collection([1, 2, 3, 4, 5]).sum()

# 15
```

If the collection contains dictionaries or objects, you must pass a key to use for determining which values to sum:

```
collection = Collection([
    {'name': 'JavaScript: The Good Parts', 'pages': 176},
    {'name': 'JavaScript: The Defnitive Guide', 'pages': 1096}
])

collection.sum('pages')

# 1272
```

In addition, you can pass your own callback to determine which values of the collection to sum:

```
collection = Collection([
    {'name': 'Chair', 'colors': ['Black']},
    {'name': 'Desk', 'colors': ['Black', 'Mahogany']},
    {'name': 'Bookcase', 'colors': ['Red', 'Beige', 'Brown']}
])

collection.sum(lambda product: len(product['colors']))

# 6
```

take()

The `take` method returns a new collection with the specified number of items:

```
collection = Collection([0, 1, 2, 3, 4, 5])

chunk = collection.take(3)

chunk.all()

# [0, 1, 2]
```

You can also pass a negative integer to take the specified amount of items from the end of the collection:

```
chunk = collection.chunk(-2)

chunk.all()

# [4, 5]
```

Warning: `serialize` also converts all of its nested objects. If you want to get the underlying items as is, use the *all* method instead.

to_json()

The `to_json` method converts the collection into JSON:

```
collection = Collection({'name': 'Desk', 'price': 200})

collection.to_json()

# '{"name": "Desk", "price": 200}'
```

transform()

The `transform` method iterates over the collection and calls the given callback with each item in the collection. The items in the collection will be replaced by the values returned by the callback:

```
collection = Collection([1, 2, 3, 4, 5])

collection.transform(lambda item: item * 2)

collection.all()

# [2, 4, 6, 8, 10]
```

> **Warning:** Unlike most other collection methods, `transform` modifies the collection itself. If you wish to create a new collection instead, use the *map* method.

unique()

The `unique` method returns all of the unique items in the collection:

```
collection = Collection([1, 1, 2, 2, 3, 4, 2])

unique = collection.unique()

unique.all()

# [1, 2, 3, 4]
```

When dealing with dictionaries or objects, you can specify the key used to determine uniqueness:

```
collection = Collection([
    {'name': 'iPhone 6', 'brand': 'Apple', 'type': 'phone'},
    {'name': 'iPhone 5', 'brand': 'Apple', 'type': 'phone'},
    {'name': 'Apple Watch', 'brand': 'Apple', 'type': 'watch'},
    {'name': 'Galaxy S6', 'brand': 'Samsung', 'type': 'phone'},
    {'name': 'Galaxy Gear', 'brand': 'Samsung', 'type': 'watch'}
])

unique = collection.unique('brand')

unique.all()

# [
#     {'name': 'iPhone 6', 'brand': 'Apple', 'type': 'phone'},
#     {'name': 'Galaxy S6', 'brand': 'Samsung', 'type': 'phone'}
# ]
```

You can also pass your own callback to determine item uniqueness:

```
unique = collection.unique(lambda item: item['brand'] + item['type'])

unique.all()

# [
#     {'name': 'iPhone 6', 'brand': 'Apple', 'type': 'phone'},
```

```
#     {'name': 'Apple Watch', 'brand': 'Apple', 'type': 'watch'},
#     {'name': 'Galaxy S6', 'brand': 'Samsung', 'type': 'phone'},
#     {'name': 'Galaxy Gear', 'brand': 'Samsung', 'type': 'watch'}
# ]
```

values()

The `values` method returns all of the collection's values:

```
collection = Collection({
    'account_id': 1,
    'product': 'Desk'
})

values = collection.values()

values.all()

# [1, 'Desk']
```

where()

The `where` method filters the collection by a given key / value pair:

```
collection = Collection([
    {'name': 'Desk', 'price': 200},
    {'name': 'Chair', 'price': 100},
    {'name': 'Bookcase', 'price': 150},
    {'name': 'Door', 'price': 100},
])

filtered = collection.where('price', 100)

filtered.all()

# [
#     {'name': 'Chair', 'price': 100},
#     {'name': 'Door', 'price': 100}
# ]
```

without()

The `without` method returns all items in the collection except for those with the specified keys:

```
collection = Collection({
    'product_id': 1,
    'name': 'Desk',
    'price': 100,
    'discount': False
})

filtered = collection.without('price', 'discount')
```

```
filtered.all()

# {'product_id': 1, 'name': 'Desk'}
```

For the inverse of `without`, see the *only* method.

zip()

The `zip` method merges together the values of the given list with the values of the collection at the corresponding index:

```
collection = Collection(['Chair', 'Desk'])

zipped = collection.zip([100, 200])

zipped.all()

# [('Chair', 100), ('Desk', 200)]
```

CHAPTER 11

Extensions

This section is a list of all the official Orator extensions that provide additional functionalities not bundled in the core package.

Cache

Installation

```
pip install orator-cache
```

Introduction

The `orator-cache` package provides query results caching to Orator. It uses the Cachy library to ease cache manipulation.

To activate the caching ability you just need to use the provided `DatabaseManager` class instead of the default one and passing it a `Cache` instance:

```
from orator_cache import DatabaseManager, Cache

stores = {
    'stores': {
        'redis': {
            'driver': 'redis',
            'host': 'localhost',
            'port': 6379,
            'db': 0
        },
        'memcached': {
            'driver' 'memcached',
            'servers': [
```

```
                '127.0.0.1:11211'
            ]
        }
    }
}

cache = Cache(stores)

db = DatabaseManager(config, cache=cache)
```

Note: Since the `Cache` class is just a subclass of the Cachy `CacheManager` class. You can refer to the Cachy documentation to configure the underlying stores.

Warning: Even though, the extension provides a way to cache queries, the invalidation of the caches is the responsability of the developer.

Caching queries

You can easily cache the results of a query using the `remember` or `remember_forever` methods:

```
users = db.table('users').remember(10).get()
```

In this example, the results of the query will be cached for ten minutes. While the results are cached, the query will not be run against the database, and the results will be loaded from the default cache driver.

Note: You can also specify which cache driver to use:

```
users = db.table('users').cache_driver('redis').remember(10).get()
```

If you are using a supported cache driver, you can also add tags to the caches:

```
users = db.table('users').cache_tags(['people', 'authors']).remember(10).get()
```