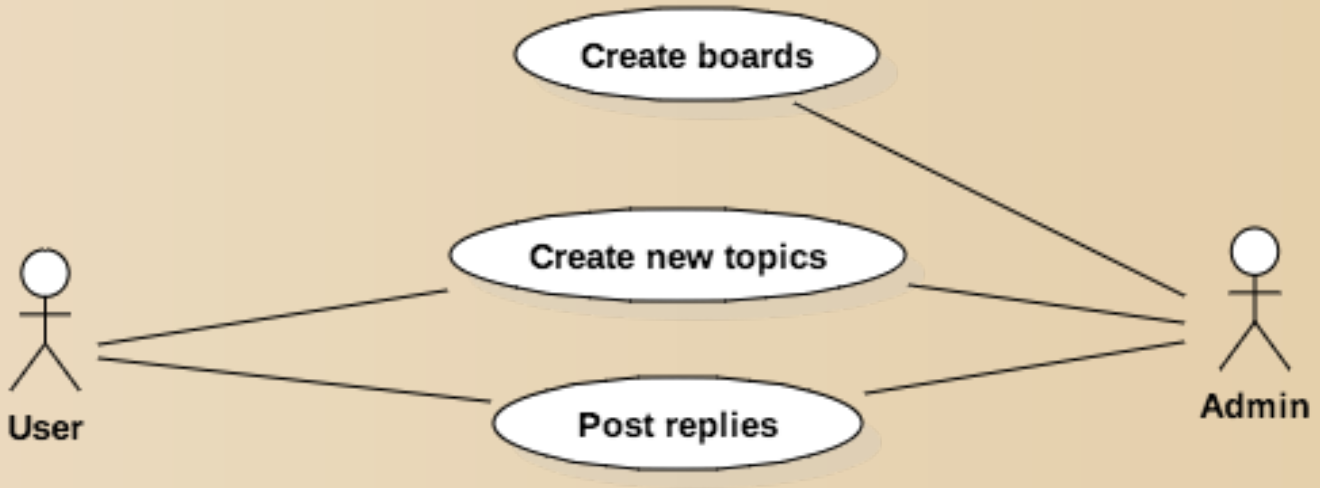
Django Blog

Part Two

Mission Statement

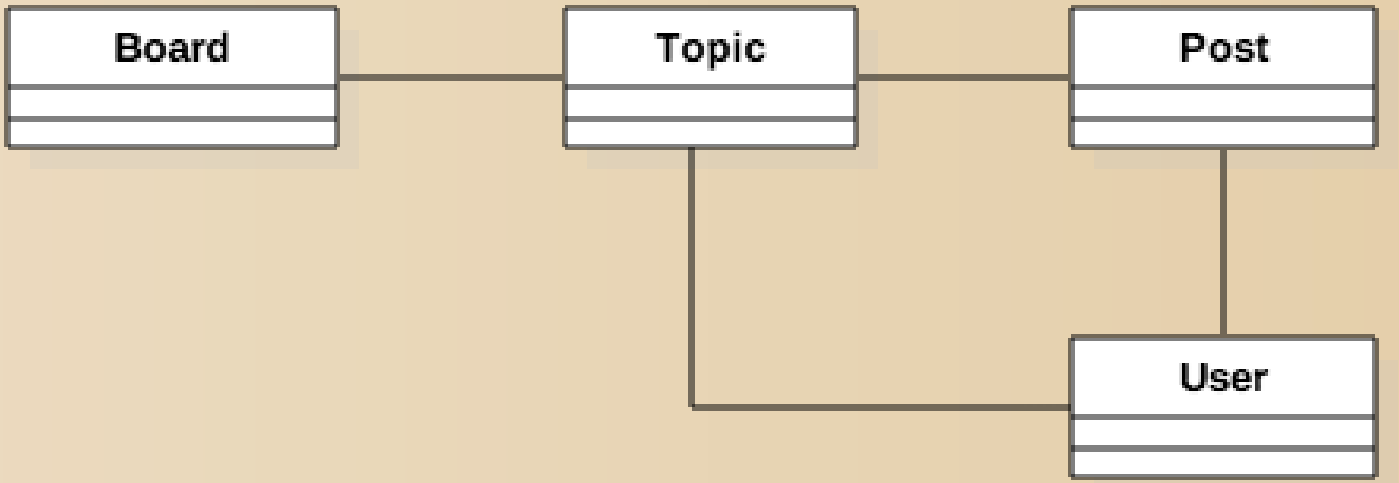
Our project is a discussion board (a forum). The whole idea is to maintain several boards, which will behave like categories. Then, inside a specific board, a user can start a new discussion by creating a new topic. In this topic, other users can engage in the discussion posting replies.

We will need to find a way to differentiate a regular user from an admin user because only the admins are supposed to create new boards. Below, an overview of our main use cases and the role of each type of user:



Class Model

To be able to implement the use cases described in the previous section, we will need to implement at least the following models: Board, Topic, Post, and User.



Class Model

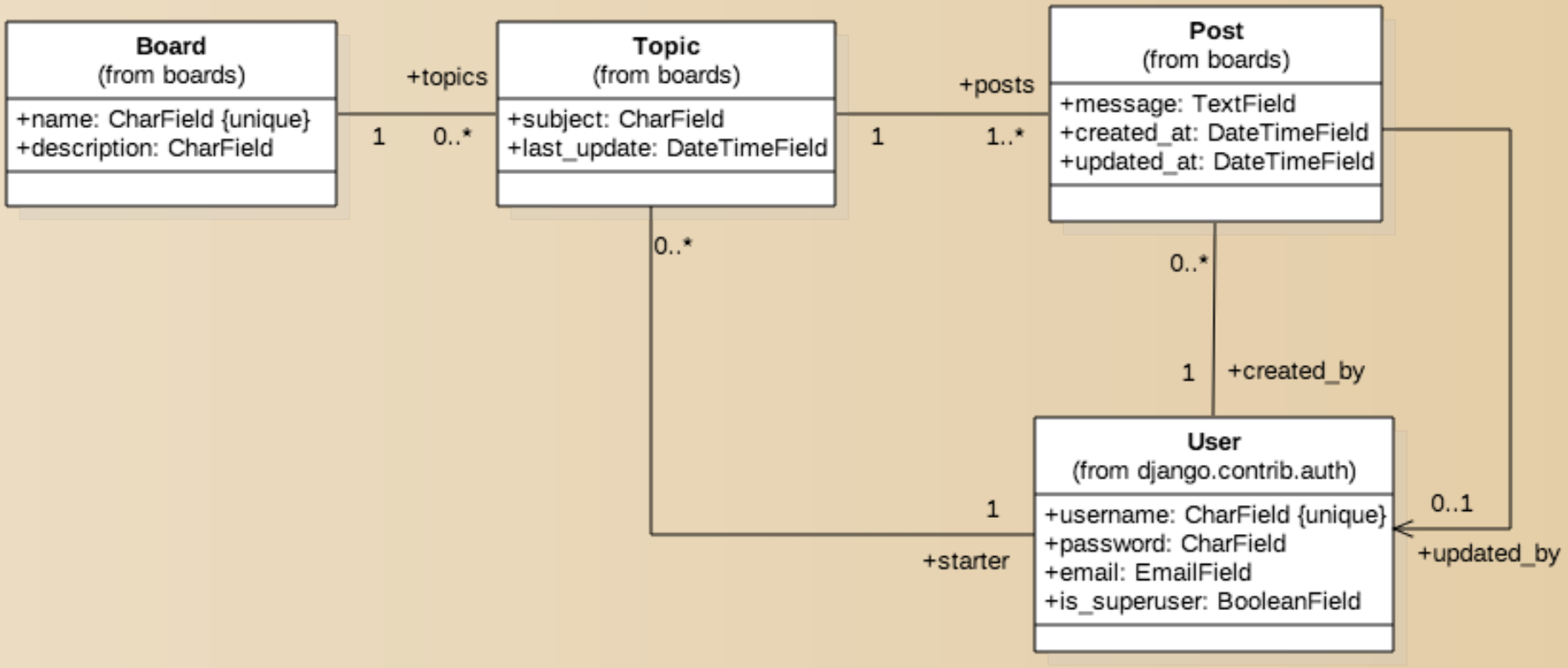
It's important to take the time to think about how do models will relate to each other. What the solid lines are telling us is that, in a Topic, we will need to have a field to identify which Board it belongs to. Similarly, the Post will need a field to represent which Topic it belongs so that we can list in the discussions only Posts created within a specific Topic. Finally, we will need fields in both the Topic to know who started the discussion and in the Post so we can identify who is posting the reply.

We could also have an association with the Board and the User model, so we could identify who created a given Board. But this information is not relevant for the application. There are other ways to track this information, you will see later on.

Now that we have the basic class representation, we have to think what kind of information each of those models will carry. This sort of thing can get complicated very easily. So try to focus on the important bits. The information that you need to start the development. Later on, we can improve the model using migrations, which you will see in great detail in the next tutorial.

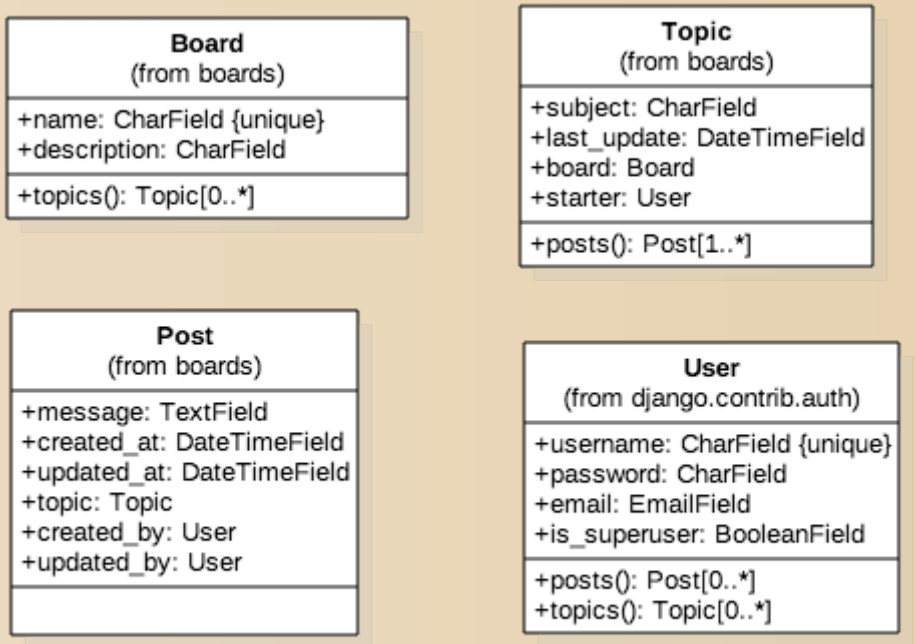
But for now, this would be a basic representation of our models' fields

Simple Django Blog



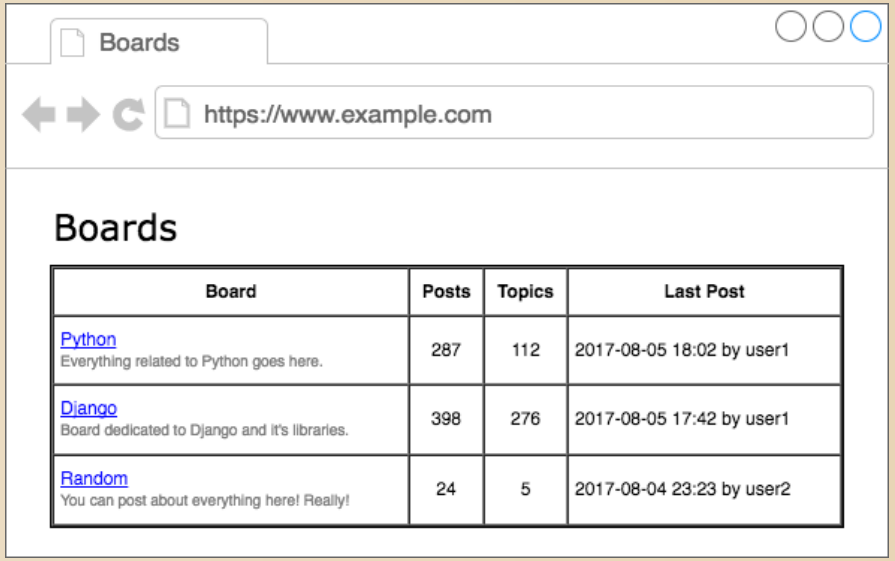
Class Model

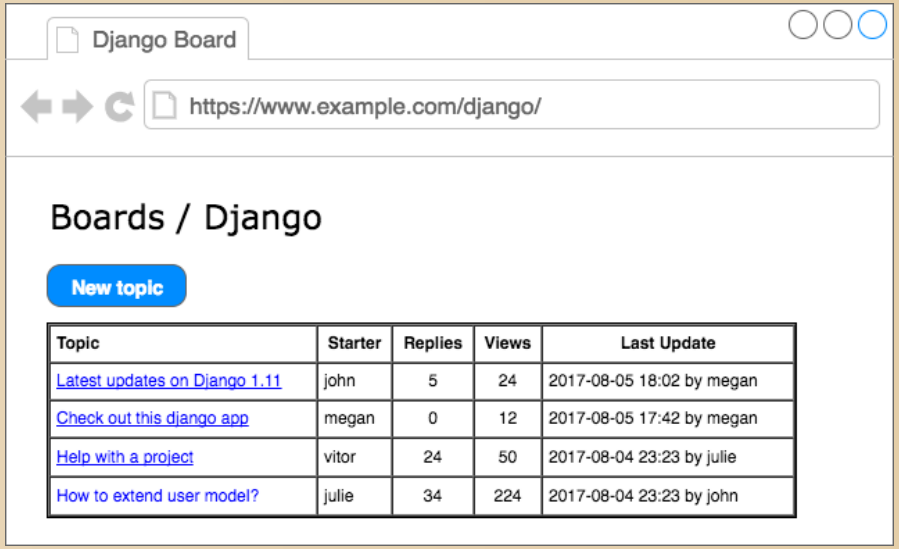
Another way to draw the class diagram is emphasizing the fields rather than in the relationship between the models:

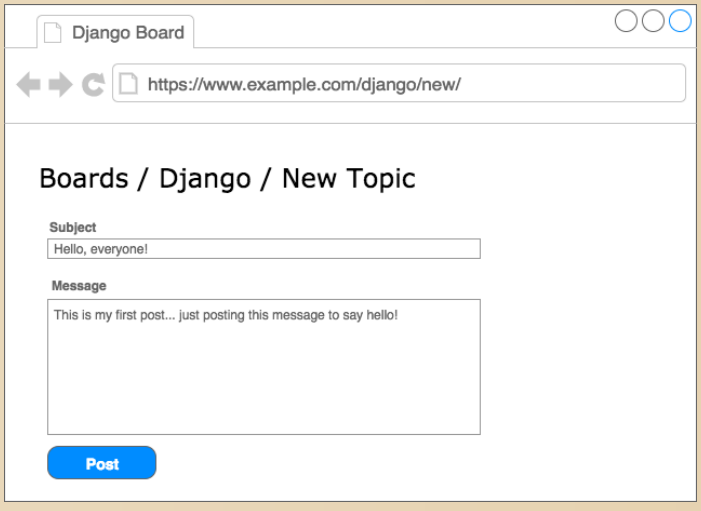


Class Model

So, what do we want to get?







Model Implementation

Models are basically a representation of your application's database layout. What we are going to do in this section is create the Django representation of the classes we modeled in the previous section: Board, Topic, and Post. The User model is already defined inside a built-in app named auth, which is listed in our INSTALLED_APPS configuration under the namespace django.contrib.auth.
We are going to implement the model inside the **boards/models.py** file

```
from django.db import models
from django.contrib.auth.models import User


class Board(models.Model):
    name = models.CharField(max_length=30, unique=True)
    description = models.CharField(max_length=100)


class Topic(models.Model):
    subject = models.CharField(max_length=255)
    last_updated = models.DateTimeField(auto_now_add=True)
    board = models.ForeignKey(Board, related_name='topics')
    starter = models.ForeignKey(User, related_name='topics')


class Post(models.Model):
    message = models.TextField(max_length=4000)
    topic = models.ForeignKey(Topic, related_name='posts')
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(null=True)
    created_by = models.ForeignKey(User, related_name='posts')
    updated_by = models.ForeignKey(User, null=True, related_name='+')
```

Model Implementation

```
from django.db import models
from django.contrib.auth.models import User


class Board(models.Model):
    name = models.CharField(max_length=30, unique=True)
    description = models.CharField(max_length=100)


class Topic(models.Model):
    subject = models.CharField(max_length=255)
    last_updated = models.DateTimeField(auto_now_add=True)
    board = models.ForeignKey(Board, related_name='topics')
    starter = models.ForeignKey(User, related_name='topics')


class Post(models.Model):
    message = models.TextField(max_length=4000)
    topic = models.ForeignKey(Topic, related_name='posts')
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(null=True)
    created_by = models.ForeignKey(User, related_name='posts')
    updated_by = models.ForeignKey(User, null=True, related_name='+')
```

Model Implementation

All models are subclass of the django.db.models.Model class. Each class will be transformed into database tables. Each field is represented by instances of django.db.models.Field subclasses (built-in Django core) and will be translated into database columns.

The fields CharField, DateTimeField, etc., are all subclasses of django.db.models.Field and they come included in the Django core – ready to be used.

Here we are only using CharField, TextField, DateTimeField, and ForeignKey fields to define our models. But Django offers a wide range of options to represent different types of data, such as IntegerField, BooleanField, DecimalField, and many others. We will refer to them as we need.

Some fields have required arguments, such as the CharField. We should always set a max_length. This information will be used to create the database column. Django needs to know how big the database column needs to be. The max_length parameter will also be used by the Django Forms API, to validate user input. More on that later.

In the Board model definition, more specifically in the name field, we are also setting the parameter unique=True, as the name suggests, it will enforce the uniqueness of the field at the database level.

In the Post model, the created_at field has an optional parameter, the auto_now_add set to True. This will instruct Django to set the current date and time when a Post object is created.

One way to create a relationship between the models is by using the ForeignKey field. It will create a link between the models and create a proper relationship at the database level. The ForeignKey field expects a positional parameter with the reference to the model it will relate to.

For example, in the Topic model, the board field is a ForeignKey to the Board model. It is telling Django that a Topic instance relates to only one Board instance. The related_name parameter will be used to create a reverse relationship where the Board instances will have access a list of Topic instances that belong to it.

Django automatically creates this reverse relationship – the related_name is optional. But if we don't set a name for it, Django will generate it with the name: (class_name)_set. For example, in the Board model, the Topic instances would be available under the topic_set property. Instead, we simply renamed it to topics, to make it feel more natural.

In the Post model, the updated_by field sets the related_name='+'. This instructs Django that we don't need this reverse relationship, so it will ignore it.

Model Implementation

The comparison between the class diagram and the source code to generate the models with Django. The green lines represent how we are handling the reverse relationships.

```python
from django.db import models
from django.contrib.auth.models import User
```

```python
class Board(models.Model):
    name = models.CharField(max_length=30, unique=True)
    description = models.CharField(max_length=100)
```

Board
(from boards)

+name: CharField {unique}
+description: CharField

+topics(): Topic[0..*]

User
(from django.contrib.auth)

+username: CharField {unique}
+password: CharField
+email: EmailField
+is_superuser: BooleanField

+posts(): Post[0..*]
+topics(): Topic[0..*]

```python
class Topic(models.Model):
    subject = models.CharField(max_length=255)
    last_updated = models.DateTimeField()
    board = models.ForeignKey(Board, related_name='topics')
    starter = models.ForeignKey(User, related_name='topics')
```

Topic
(from boards)

+subject: CharField
+last_update: DateTimeField
+board: Board
+starter: User

+posts(): Post[0..*]

```python
class Post(models.Model):
    message = models.TextField(max_length=4000)
    topic = models.ForeignKey(Topic, related_name='posts')
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(null=True)
    created_by = models.ForeignKey(User, related_name='posts')
    updated_by = models.ForeignKey(User, related_name='+')
```

Post
(from boards)

+message: TextField
+created_at: DateTimeField
+updated_at: DateTimeField
+topic: Topic
+created_by: User
+updated_by: User

Migrating the Models

The next step is to tell Django to create the database so we can start using it.

Open the Terminal , activate the virtual environment, go to the folder where the manage.py file is, and run the commands below:

python manage.py makemigrations

Here Django created a file named 0001_initial.py inside the boards/migrations directory. It represents the current state of our application's models. In the next step, Django will use this file to create the tables and columns.

Migrating the Models

Migration files are translated into SQL statements. If you are familiar with SQL, you can run the following command to inspect the SQL instructions that will be executed in the database:

python manage.py sqlmigrate boards 0001

If you're not familiar with SQL, don't worry. We won't be working directly with SQL in this tutorial series. All the work will be done using just the Django ORM, which is an abstraction layer that communicates with the database.

The next step now is to apply the migration we generated to the database:

python manage.py migrate

That's it! Our database is ready for use!.

The Models API

We can start a Python shell with our project loaded using the manage.py utility:

python manage.py shell

```
python manage.py shell
```

```
Python 3.6.2 (default, Jul 17 2017, 23:14:31)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>>
```

The Models API

We begin with importing the Board class:

```
from boards.models import Board
```

To create a new board object, we can do the following:

```
board = Board(name='Django', description='This is a board about Django.')
```

To persist this object in the database, we have to call the save method:

```
board.save()
```

The save method is used both to create and update objects. Here Django created a new object because the Board instance had no id. After saving it for the first time, Django will set the id automatically:

The Models API

The `save` method is used both to *create* and *update* objects. Here Django created a new object because the **Board** instance had no **id**. After saving it for the first time, Django will set the id automatically:

```
board.id
1
```

You can access the rest of the fields as Python attributes:

```
board.name
'Django'
```

```
board.description
'This is a board about Django.'
```

To update a value we could do:

```
board.description = 'Django discussion board.'
board.save()
```

The Models API

We can create as many boards as we want...

```
board = Board.objects.create(name='Python', description='General discussion about Python
```

```
board.id
2
```

```
board.name
'Python'
```

The Models API

We will now edit the **models.py** file inside the boards app and see the result:

```
class Board(models.Model):
    name = models.CharField(max_length=30, unique=True)
    description = models.CharField(max_length=100)

    def __str__(self):
        return self.name
```

```
python manage.py shell
```

```
from boards.models import Board

Board.objects.all()
<QuerySet [<Board: Django>, <Board: Python>]>
```

The Models API

We can treat this QuerySet like a list. Let's say we wanted to iterate over it and print the description of each board::

```
boards_list = Board.objects.all()
for board in boards_list:
    print(board.description)
```

```
Django discussion board.
General discussion about Python.
```

The Models API

It's possible to use the model Manager to query the database and return a single object. For that we use the get method:

```
django_board = Board.objects.get(id=1)

django_board.name
'Django'
```

```
board = Board.objects.get(id=3)

boards.models.DoesNotExist: Board matching query does not exist.
```

```
Board.objects.get(name='Django')
<Board: Django>
```

```
Board.objects.get(name='django')
boards.models.DoesNotExist: Board matching query does not exist.
```

Summary of Model's Operations

Operation	Code sample
Create an object without saving	`board = Board()`
Save an object (create or update)	`board.save()`
Create and save an object in the database	`Board.objects.create(name='...', description='...')`
List all objects	`Board.objects.all()`
Get a single object, identified by a field	`Board.objects.get(id=1)`

The End of Part Two