# From domain model to Django view: the basic Django web application setup guide

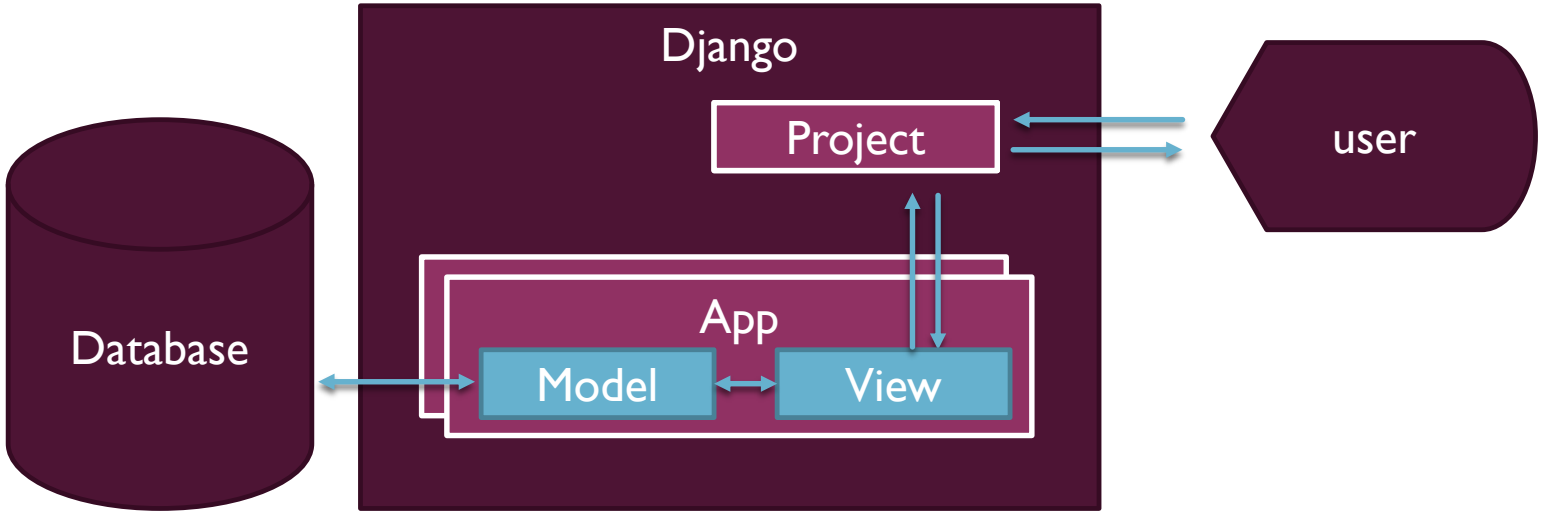
COMP3297 Introduction to Software Engineering 2016-2017

by Kevin Lam (yklam2)

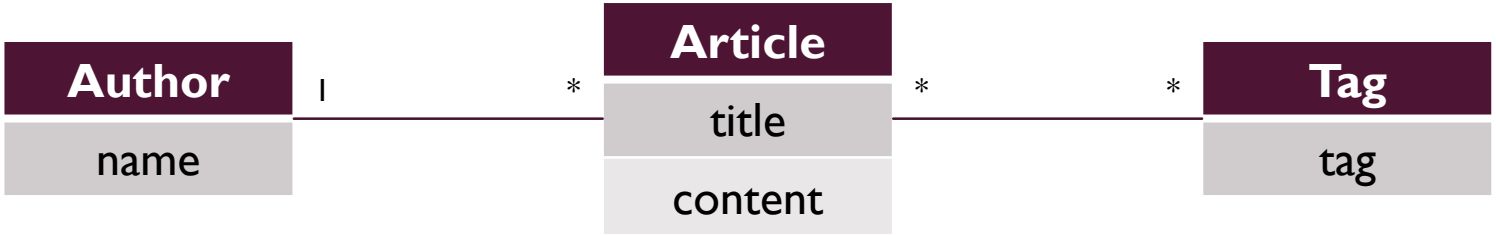
# Objectives

- By this tutorial, you will learn:
  - How to model relationships in database.
  - How to set up Django to run a web application in the development server.
  - How to write a simple web application in Django to perform basic CRUD (create, read, update, delete) operations.

# Django overview (simplified)



# Example model for this tutorial



# Installing Django

- Django runs on Python, so you need to first install Python, then install Django.
- **Python (3.5)**
  - Homepage: https://www.python.org/
  - Installation: https://docs.python.org/3/using/windows.html#installing-python
- **Django (1.10+)**
  - Homepage: https://www.djangoproject.com/
  - Installation: https://www.djangoproject.com/download/
  - Quick start tutorial: https://docs.djangoproject.com/en/1.10/intro/tutorial01/

# Creating a project

- Pick a place to keep the project files, and create a project there using the specific command. (Replace *project_name* with some other name)

```
django-admin startproject project_name
```

```
C:\Users\kevin\Desktop\c3297\tutorial>django-admin startproject tutorial

C:\Users\kevin\Desktop\c3297\tutorial>tree /F
Folder PATH listing for volume Local Disk
Volume serial number is 000000B9 6CB7:9AFA
C:.
└───tutorial
    │   manage.py
    │
    └───tutorial
            settings.py
            urls.py
            wsgi.py
            __init__.py
```

This create a project named "tutorial"

The command created these files and folders

A new folder to house all files

A folder for the "tutorial" project

# Testing project using development server

- Django comes with its own web server for development.
  - No need to setup any web server during development.
  - For development only.
- Do this in the new project folder (created by the previous command)

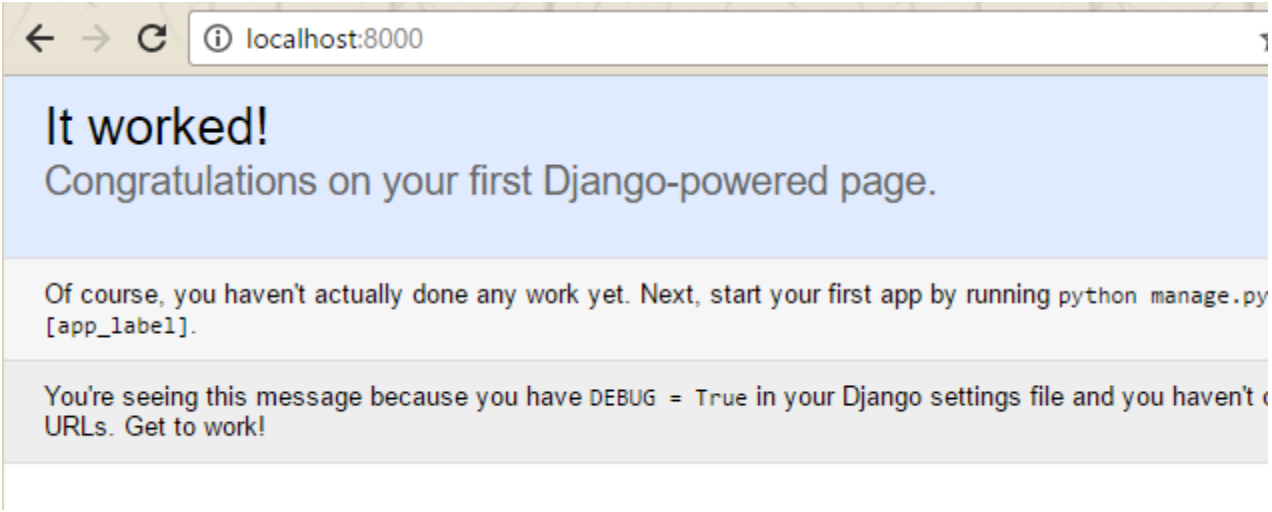
```
python manage.py runserver
```

```
C:\Users\kevin\Desktop\c3297\tutorial\tutorial>python manage.py runserver
Performing system checks...

System check identified no issues (0 silenced).

You have 13 unapplied migration(s). Your project may not work
 auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
October 10, 2016 - 11:38:53
Django version 1.10.2, using settings 'tutorial.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

Keep this in the command prompt during development, no need to restart server.
Press Ctrl-C to end it.

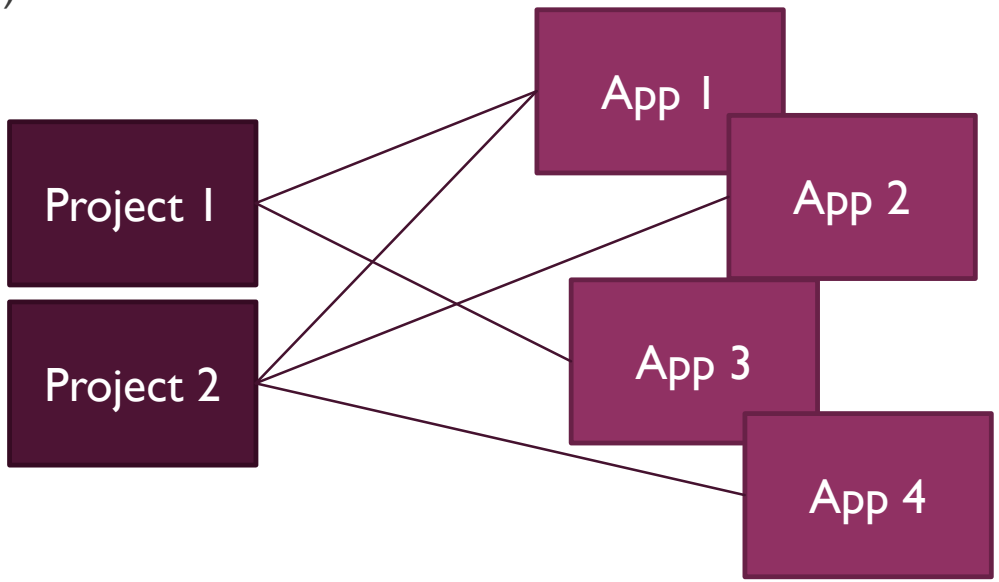
# Testing development server (2)

- Visit **http://localhost:8000** using any browser.



# Django projects and apps

- A **project** in Django is a collection of **apps**, working as a whole to provide features.
  - In most cases the project will be the complete product.
- An app is a web application with its own feature sets (e.g, blog, forum, etc..).
- A project may consist of multiple apps, an app can be used in multiple projects. (like modules)



# Creating apps

- To simplify things, we create the app under the project folder (in the same folder with manage.py). (Stop the development server at the moment)

```
python manage.py startapp app_name
```

```
C:\Users\kevin\Desktop\c3297\tutorial\tutorial>python manage.py startapp articles

C:\Users\kevin\Desktop\c3297\tutorial\tutorial>tree /F
Folder PATH listing for volume Local Disk
Volume serial number is 000000FB 6CB7:9AFA
C:.
    db.sqlite3
    manage.py

    articles
        admin.py
        apps.py
        models.py
        tests.py
        views.py
        __init__.py

        migrations
            __init__.py

    tutorial
```

This create an app named "articles"

The new folder for the new app

The existing project folder

# Adding app to project

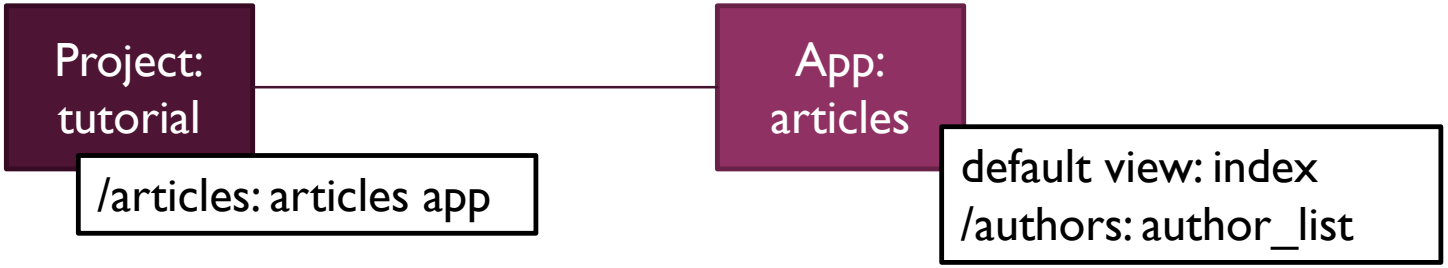
- In **settings.py** (in project folder), add the app in the **INSTALLED_APPS** setting.

```
INSTALLED_APPS = [
    'articles.apps.ArticlesConfig',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

**ArticlesConfig** is generated automatically in **apps.py**

# Django projects and apps

- Apps with UI should define "**Views**" to be used. A view can be seen as a page viewable on a browser.
- We need to setup URLs pointing to these views in a project, so that we can trigger these views on a browser.




http://localhost:8000/articles → index view in articles app
http://localhost:8000/articles/authors → author_list in articles app

# Our first view

- Edit **`articles/views.py`** as follow

```
from django.http import HttpResponse

def index(request):
    return HttpResponse("Hello, this is View: article.index.")
```

# urls.py

```
#articles/urls.py
from django.conf.urls import url

from . import views

urlpatterns = [
    url(r'^$', views.index, name='index'),
]
```

Regular expression: ^$
Match an empty string

index view defined in **views.py**

A name used to specify this URL

```
#tutorials/urls.py
#...

from django.conf.urls import include, url
from django.contrib import admin

urlpatterns = [
    url(r'^articles/', include('articles.urls')),
    url(r'^admin/', admin.site.urls),
]
```

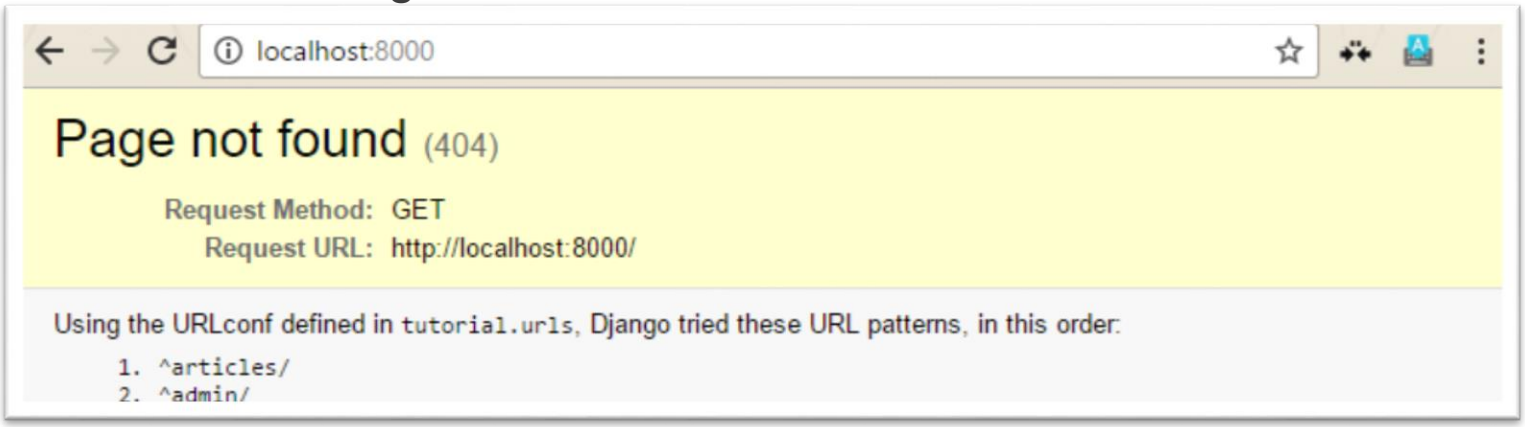
This file already exists with some content

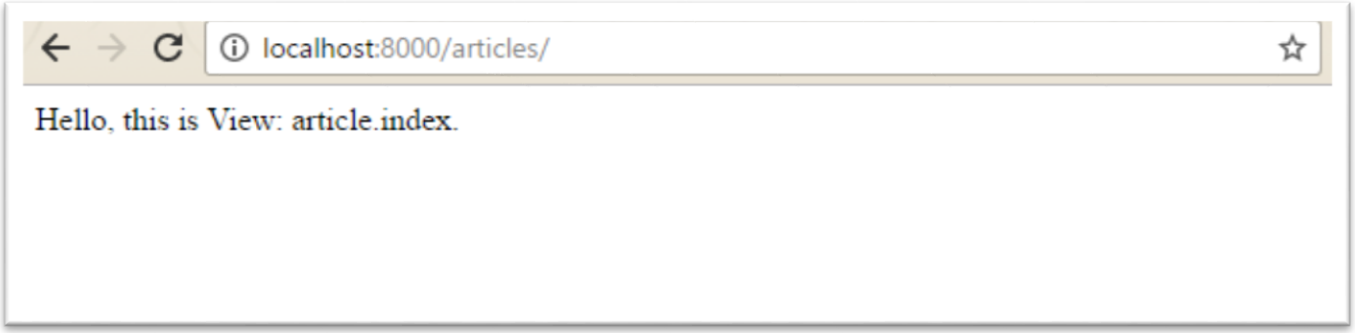
Add these

Paths under **articles/** uses urls defined in articles app.

# Test new view

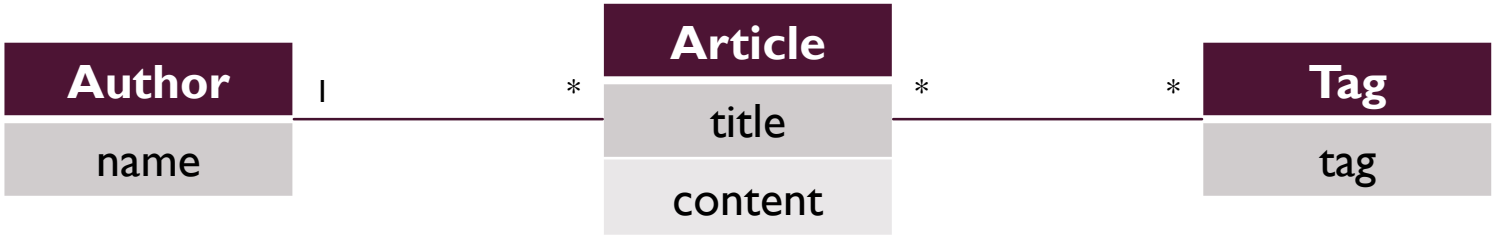
- Restart server again



# Models

- With Django set up, we can now build our **Models** in the app.
- Django can manage the database for us, but first, we need to understand how our models will be translated to database tables in a relational database.



# Relational database


- We uses relational database to store data for our application.
- A relational database stores tables of data.
- How to model relationships in these table?

| name |
|---|
| Alice |
| Bob |
| Carl |
| David |

| title | content |
|---|---|
| title1 | content1 |
| title2 | content2 |
| title3 | content3 |
| title4 | content4 |

# Primary key

- To store the relationships, we need some way to identify a row in the table.
- The value(s) used for identification is assigned as the **primary key** of the table.
- Can we use name/title as primary key?
  - No! primary key must be **unique** and **immutable**! (why?)

| name | article_title |
|---|---|
| Alice | title3 |
| Bob | title1 |
| Carl | title4 |
| David | title2 |

| title | content | author_name |
|---|---|---|
| title1 | content1 | Bob |
| title2 | content2 | David |
| title3 | content3 | Carl |
| title4 | content4 | Alice |

# Using identifier (ID) as primary key

- It is a common practice that an identifier (id) is added to each table, used as the primary key.
  - It is not related to the application data, so immutable is not a problem.
  - Guaranteed to be unique. (how?)
- Common choices of the id field:
  - **Incrementing integer** – simple; can be generated automatically in many DB engine; limit to maximum integer size in the supporting database.
  - **Random string**, e.g., UUID – need to check for uniqueness before usage; need more storage space; not sequential.

# Revisiting our tables


| author_id | name | article_id |
|---|---|---|
| 1 | Alice | 3 |
| 2 | Bob | 1 |
| 3 | Carl | 4 |
| 4 | David | 2 |

| article_id | title | content | author_id |
|---|---|---|---|
| 1 | title1 | content1 | 2 |
| 2 | title2 | content2 | 4 |
| 3 | title3 | content3 | 1 |
| 4 | title4 | content4 | 3 |

We only need one of these columns, as they serve the same purpose.
Which one should we use?
Remember, in a relational database, we keep **one** data per field per record.

# Modelling one-to-many relationship (can also apply to one-to-one)


We call it the foreign key

| author_id | name |
|---|---|
| 1 | Alice |
| 2 | Bob |
| 3 | Carl |
| 4 | David |

| article_id | title | content | author_id |
|---|---|---|---|
| 1 | title1 | content1 | 2 |
| 2 | title2 | content2 | 4 |
| 3 | title3 | content3 | 1 |
| 4 | title4 | content4 | 3 |

We model the relationship at the "many" side.
Or more accurately, we model it at the opposite of the "one" side.
Then, how to model many-to-many relationship?

# Modelling many-to-many relationship


There is no "one" side in many-to-many relationship, we cannot model it directly in a relational database.
Solution is to model the relationship itself as a table.


Note that this is practical implementation choice, which should not affect your domain model design.

# Modelling many-to-many relationship


| article_id | title | content | author_id |
|---|---|---|---|
| 1 | title1 | content1 | 2 |
| 2 | title2 | content2 | 4 |
| 3 | title3 | content3 | 1 |
| 4 | title4 | content4 | 3 |

| article_id | tag_id |
|---|---|
| 1 | 1 |
| 1 | 2 |
| 2 | 2 |
| 3 | 1 |
| 3 | 3 |
| 4 | 3 |

| tag_id | tag |
|---|---|
| 1 | tag1 |
| 2 | tag2 |
| 3 | tag3 |

Extra information may be stored in the relation.
In that case you probably have modeled it in your domain model already.

# Setting up database in Django

- In **settings.py** (in project folder), the default set up uses SQLite as database engine.
  - We will keep this setting.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

- We should also set the **TIME_ZONE** settings, in case we have date/time related operations.

```
# Internationalization
# https://docs.djangoproject.com/en/1.10/topics/i18n/

LANGUAGE_CODE = 'en-us'

TIME_ZONE = 'Asia/Hong_Kong'
```

# Defining Models in Django

- Now, we edit **articles/models.py**

```
from django.db import models

class Author(models.Model):
    name = models.CharField(max_length=200)

class Tag(models.Model):
    tag = models.CharField(max_length=200)

class Article(models.Model):
    author = models.ForeignKey(Author, on_delete=models.CASCADE)
    title = models.CharField(max_length=200)
    content = models.TextField()
    tags = models.ManyToManyField(Tag)
```

Specify foreign key in the "many" side of one-to-many

Specify many-to-many relationship at any side of the relationship

Django will handle the **id** field and relation table for us.

# Activate the models

- With Models set up, we can ask Django to generate the database tables accordingly.
  - This requires the app added to **INSTALLED_APPS** in **settings.py**, which we have previously done.
- This is a two-step process, first generate files according to the changes in Models (in our case, we have a new set of models); then apply the changes

```
python manage.py makemigrations articles
python manage.py migrate
```

```
C:\Users\kevin\Desktop\c3297\tutorial\tutorial>python manage.py makemigrations articles
Migrations for 'articles':
  articles\migrations\0001_initial.py:
    - Create model Article
    - Create model Author
    - Create model Tag
    - Add field author to article

C:\Users\kevin\Desktop\c3297\tutorial\tutorial>python
Operations to perform:
  Apply all migrations: admin, articles, auth, content
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying articles.0001_initial... OK
  Applying contenttypes.0002_remove_content_type_name... OK
```

If it is the first time "migrate" is executed, Django will also create the necessary tables for other modules.

# What's in the database?

- You may try to examine the database using any SQLite clients.

| | | |
|---|---|---|
| Tables (15) | | |
| articles_article | | CREATE TABLE "articles_article" ("id" integer NOT NULL PRIMARY |
| id | integer | `id` integer NOT NULL PRIMARY KEY AUTOINCREMENT |
| title | varchar(200) | `title` varchar(200) NOT NULL |
| content | text | `content` text NOT NULL |
| author_id | integer | `author_id` integer NOT NULL |
| articles_article_tags | | CREATE TABLE "articles_article_tags" ("id" integer NOT NULL PRIM |
| id | integer | `id` integer NOT NULL PRIMARY KEY AUTOINCREMENT |
| article_id | integer | `article_id` integer NOT NULL |
| tag_id | integer | `tag_id` integer NOT NULL |
| articles_author | | CREATE TABLE "articles_author" ("id" integer NOT NULL PRIMARY |
| id | integer | `id` integer NOT NULL PRIMARY KEY AUTOINCREMENT |
| name | varchar(200) | `name` varchar(200) NOT NULL |
| articles_tag | | CREATE TABLE "articles_tag" ("id" integer NOT NULL PRIMARY KE |
| id | integer | `id` integer NOT NULL PRIMARY KEY AUTOINCREMENT |
| tag | varchar(200) | `tag` varchar(200) NOT NULL |

# Activating Django admin interface

- We can use the Django **admin interface** to manage data in the database.
- The admin interface is perfect for site managers to manage data in the database, **you should always build a separate UI for site visitor**.
- To activate admin interface, first we create a super user.

```
python manage.py createsuperuser
```

```
C:\Users\kevin\Desktop\c3297\tutorial\tutorial>python manage.py createsuperuser
Username (leave blank to use 'kevin'): admin
Email address: admin@example.com
Password:
Password (again):
Superuser created successfully.
```

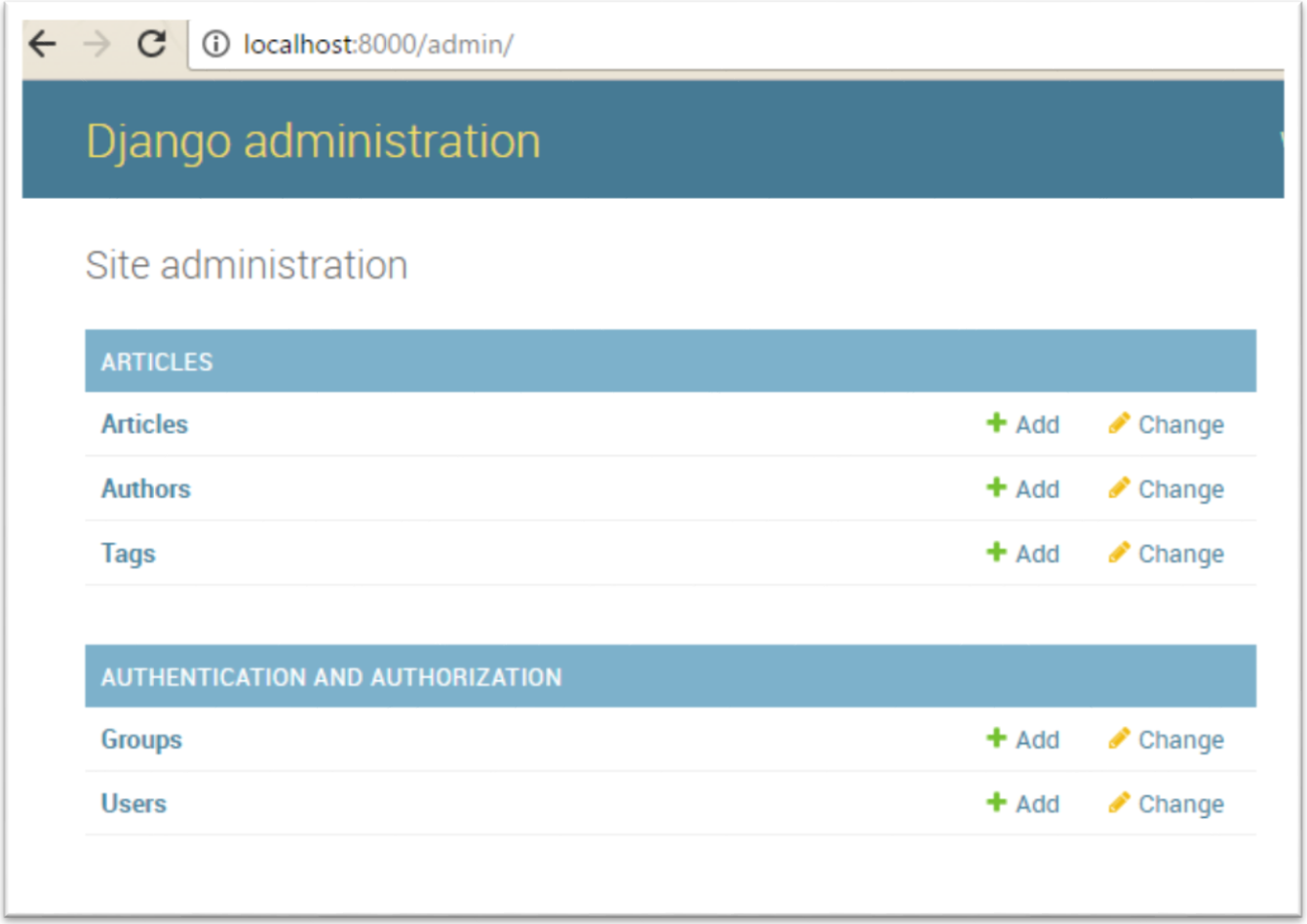
- Then we can access the **/admin** URL.

# Enable app(s) to admin interface

- To see the Models in our apps in the admin interface, we need to modify **admin.py** in the app folder.

```
from django.contrib import admin

from .models import Author
from .models import Article
from .models import Tag

admin.site.register(Author)
admin.site.register(Article)
admin.site.register(Tag)
```

# Admin interface



# Default string representation of models

- We can specify how a model is presented by default.
- If you do not define it, you will not be able to identify different records in the admin interface.

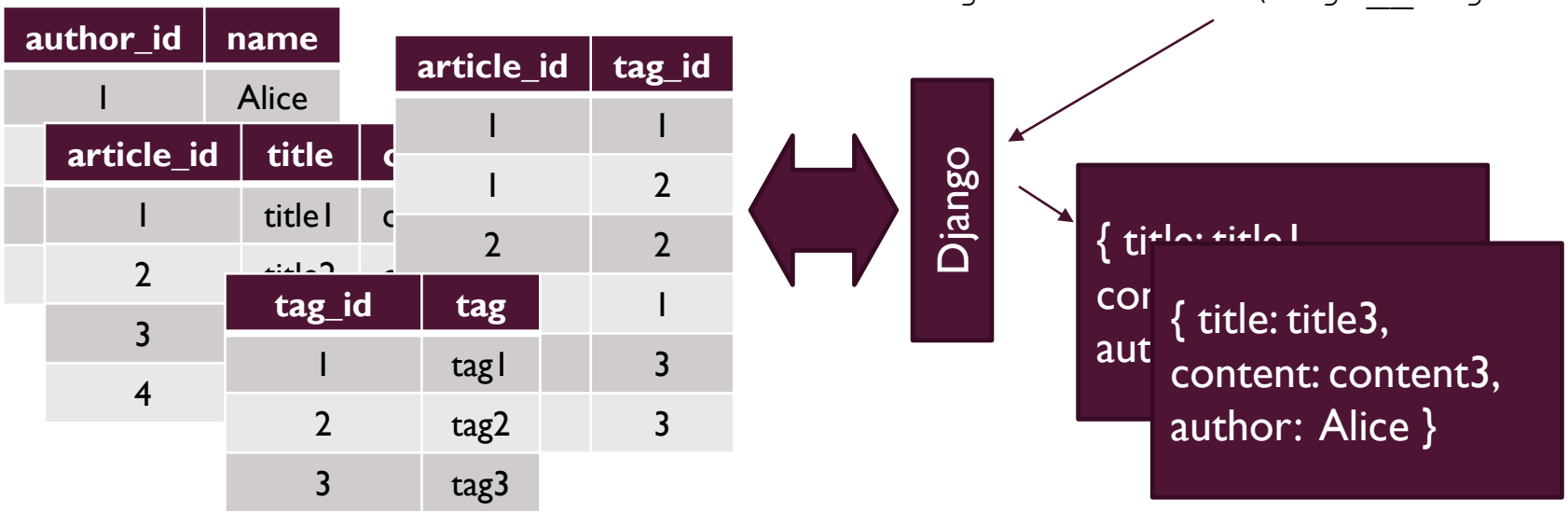

- In **articles/models.py**

```
class Author(models.Model):
    name = models.CharField(max_length=200)
    def __str__(self):
        return self.name
```

# O/R mapping in Django

- With Models set up, we can set up our view to display the data in the database.
- **Object-relation mapping** in Django will return to you the data in the database as objects (The query set) according to your query.

```
Article.objects.filter(tags__tag='tag1')
```



# Understanding query set

- Django provides the Query set as the interface for querying objects from databases.
- A query set is created from the Model class.

```
Article.objects.all()
```

Query all articles

- Basic operations include: filter, exclude, get

```
Article.objects.filter(title="title1")
```

Query all articles with title equals "title1"

```
Article.objects.exclude(title="title1")
```

Query all articles except those with title equals "title1"

```
Article.objects.get(pk=1)
```

Get one article with primary key equals 1

# More on query set

- Query set support chaining

```
Article.objects.all().filter(title="title1").exclude(title="title2")
```

- It is possible to query on relations

```
Article.objects.filter(tags__tag="tag1")
```

- Apart from simple "equal" operator, we can use other operations by specifying it in the name.

```
Article.objects.filter(title__contains="title")
```

```
Article.objects.filter(id_lte=4)
```

# How to test out query set?

- We can use **Django shell** to test out query sets.
- In command prompt:

```
python manage.py shell
```

- Before you can try any query set, you need to import the models.

```
from articles.models import Article, Author, Tag
```

```
C:\Users\kevin\Desktop\c3297\tutorial\tutorial>python manage.py shell
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 25 2016, 22:18:55) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> from articles.models import Article, Author, Tag
>>> Article.objects.all()
<QuerySet [<Article: Hello World by Kevin Lam>, <Article: The second article by Kevin Lam>]>
>>> Article.objects.filter(tags__tag__contains="software")
<QuerySet [<Article: Hello World by Kevin Lam>, <Article: The second article by Kevin Lam>]>
>>> Article.objects.filter(id__lte=4)
<QuerySet [<Article: Hello World by Kevin Lam>, <Article: The second article by Kevin Lam>]>
```

# Other operations

- Create new object and save it to database.

```
au = Author(name="Alice B. Caesar")
au.save()
```

**save()** is also used to update an object

- Save Foreign Key to database

```
au = Author.objects.filter(name="Alice B. Caesar")[0]
ar = Article(title="new post",content="test",author=au)
ar.save()
```

- Save many-to-many relationship

```
t = Tag(tag="test")
t.save()
ar.tags.add(t)
```



More on URLs: **https://docs.djangoproject.com/en/1.10/topics/db/queries/**

# Re-visiting our view


- We can modify our index view to show all articles.

Add this to import Article from model

Query all articles

```
from .models import Article

def index(request):
    all_articles = Article.objects.all()
    list = []
    for a in all_articles:
        html = '<p>{title} by <b>{author_name}</b></p>'
        list.append(html.format(title=a.title, \
                                author_name=a.author.name))
    output = '<hr>'.join(list)
    return HttpResponse(output)
```

# Another view

```
#views.py
def view(request, article_id):
    a = Article.objects.get(pk=article_id)
    tags = [t.tag for t in a.tags.all()]

    html = """<h1>{title}</h1>
              <p><b>{author_name}</b><br>{content}</p>
              <p>Tag: {tag}</p>"""

    output = html.format(title = a.title, \
                         content=a.content, \
                         author_name=a.author.name, \
                         tag=tags)

    return HttpResponse(output)
```

This view takes one extra parameter, **article_id**

Get many-to-many relationships

# Setting up links to view

```
#urls.py
urlpatterns = [
    url(r'^$', views.index, name='index'),
    url(r'^(?P<article_id>\d+)$', views.view, name='view_article'),
]
```

Named as **article_id**
Need to match with view definition

Pattern is **\d+**
(digits only)

Handled by view()
in **views.py**

```
#views.py
def view(request, article_id):
    ...
```




More on URLs: https://docs.djangoproject.com/en/1.10/topics/http/urls/#naming-url-patterns

# What's next?

- Go through Django tutorial yourselves. (up to tutorial 4 will be good enough for simple web application)
  - https://docs.djangoproject.com/en/1.10/intro/tutorial01/
- Learn how to use template.
- Learn how to use generic view.
- …
- Finished your project.