Learning Objectives

- **Create an app that stores information about movies**
- **Use the OMDb API to fetch information about movies**
- **Conceptualize how the user and site will behave**
- **Increase efficiency by limiting the number of times the API is queried**
- **Identify the benefits of keeping Python logic separate from Django**

Course 4 Project

Course 4 Project

The next few assignments will create a new Django app that uses the Open Movie Database API. As such, we will need to fork a new GitHub repo before creating the app.

Fork the Repo

- Go to the course4_proj repo. This repo contains the starting point for this project.
- Click on the "Fork" button in the top-right corner.
- Click the green "Code" button.
- Copy the SSH information. It should look something like this:

```
git@github.com:<your_github_username>/course4_proj.git
```

In the Terminal

- Clone the repo. Your command should look something like this:

```
git clone git@github.com:<your_github_username>/course4_proj.git
```

- You should see a `course4_proj` directory appear in the file tree.

You are now ready for the next assignment.

Intro

Intro

The Open Movie Database is a free REST web service that can be queried to get information about movies. To follow along with these examples, and complete modules 3 and 4, you'll need an API key. One can be obtained free from https://www.omdbapi.com/apikey.aspx.

Once you have a key, it's passed in the URL as the `apikey` parameter. Free keys are limited to 1,000 requests per day.

Here's some example code that uses Requests to get movie details by title (in this case, `star wars`). It loads the OMDb key from an environment variable.

```
import os

import requests

params = {"apikey": os.environ["DJANGO_OMDB_KEY"], "t": "star
        wars"}
resp = requests.get("https://www.omdbapi.com/", params=params)
# actual URL will be https://www.omdbapi.com/?apikey=
        <key>&t=star+wars

print(resp.json())
```

Here's the output:

```
{'Title': 'Star Wars', 'Year': '1977', 'Rated': 'PG', 'Released': '25 May 1977', 'Runtime': '121 min'
        Adventure, Fantasy', 'Director': 'George Lucas', 'Writer': 'George Lucas', 'Actors': 'Mark Ha
        Carrie Fisher', 'Plot': "Luke Skywalker joins forces with a Jedi Knight, a cocky pilot, a Woo
        save the galaxy from the Empire's world-destroying battle station, while also attempting to r
        from the mysterious Darth Vad", 'Language': 'English', 'Country': 'United States, United King
        Oscars. 63 wins & 29 nominations total', 'Poster': 'https://m.media-
        amazon.com/images/M/MV5BNzVlY2MwMjktM2E4OS00Y2Y3LWE3ZjctYzhkZGM3YzA1ZWM2XkEyXkFqcGdeQXVyNzkwM
        , 'Ratings': [{'Source': 'Internet Movie Database', 'Value': '8.6/10'}, {'Source': 'Rotten To
        '92%'}, {'Source': 'Metacritic', 'Value': '90/100'}], 'Metascore': '90', 'imdbRating': '8.6',
        '1,271,153', 'imdbID': 'tt0076759', 'Type': 'movie', 'DVD': '06 Dec 2005', 'BoxOffice': '$460
        'Production': 'Lucasfilm Ltd.', 'Website': 'N/A', 'Response': 'True'}
```

We can also search the API. Here's code that performs a search for `star wars`:

```
params = {"apikey": os.environ["DJANGO_OMDB_KEY"], "s": "star
        war", "type": "movie"}

resp = requests.get("https://www.omdbapi.com/", params=params)

print(resp.json())
```

And this is the output.

```
{'Search': [{'Title': 'Star Trek: Enterprise - In a Time of War', 'Year': '2014', 'imdbID': 'tt344540
        'https://m.media-amazon.com/images/M/MV5BMTk4NDA4MzUwM15BM15BanBnXkFtZTgwMTg3NjY5MDE@._V1_SX3
        Next Generation - Survive and Suceed: An Empire at War', 'Year': '2013', 'imdbID': 'tt3060318
        'https://m.media-amazon.com/images/M/MV5BMjM5ODY0MDQ2NF5BM15BanBnXkFtZTgwMjQ5NDgwMDE@._V1_SX3
        Star Hand Kid Volume 3 - Time War', 'Year': '1989', 'imdbID': 'tt0410598', 'Type': 'movie', '
        Five Star Heroes: Gods of War", 'Year': '1998', 'imdbID': 'tt0371529', 'Type': 'movie', 'Post
        War', 'Year': '2014', 'imdbID': 'tt4254746', 'Type': 'movie', 'Poster': 'https://m.media-
        amazon.com/images/M/MV5BMzI1MDczMjc4N15BM15BanBnXkFtZTgwNjk1NjU3NTE@._V1_SX300.jpg'}, {'Title
        Cold War: Declassified', 'Year': '2014', 'imdbID': 'tt3445422', 'Type': 'movie', 'Poster': 'N
        Wars Fan Film', 'Year': '2017', 'imdbID': 'tt6314408', 'Type': 'movie', 'Poster': 'https://m.
        amazon.com/images/M/MV5BZjAyMzRhZDYtZTM2NC00NjgyLWE0ODItZGNjYzVlODBmZGYwL2ltYWdlL2ltYWdlXkEyX
        , {'Title': 'Sky Corps - Star Beast War', 'Year': '2019', 'imdbID': 'tt10368144', 'Type': 'mo
        'totalResults': '8', 'Response': 'True'}
```

We can page through results by setting a `page` parameter as well.

Now that we have a basic idea of the data that's being returned from the OMDb API, we now need to plan how we're going to use it in Django, so let's talk about the site we're building to use this data and what considerations need to be made.

The Site & User Behavior

The Site & User Behavior

We're going to build a site that allows users to leave comments on movies. Well, we're going to build some of it. We already saw, when building Blango, how to create a comment system, so we won't build that part. We're mostly going to focus on designing the models based on the data that OMDb returns, and discuss how to keep our data up to date. Let's start by talking about the main use case we foresee.

A user will visit our site in order to comment on a movie, or read previous comments about a movie. When they visit the site, let's assume they will begin by performing a search for that movie. How should this search operate?

If the user is performing a search for a title that is not in our database, then we'll need to first find matching films on OMDb and use that to populate our database. Then we can query our own database and get results. Remember we should keep in mind that searching the local database is generally a lot faster than a remote API, and that it costs us (uses up API quota) every time a request is made to the API. However, we want to make sure our own database is (reasonably) up to date.

We're going to solve this problem by keeping a record of search terms that have been used. We'll only query the API if the search term hasn't been searched for in the past 24 hours. Otherwise, we only search our local database.

Also notice in the API responses, the list response contains only some of the data (`Title`, `Year`, `imdbID`, `Poster` (URL) and `Type`), whereas the detailed response contains a lot more data. At this point, we need to decide if we want to store data that the detailed response contains, or if it's only necessary to store list (summary) data.

In our case, we want to also store the plot and genre(s) of the movie, which means we'll have to retrieve the detailed response too. This will allow us to display them on a movie detail page. In theory, it would also allow searching by these fields. However, those searches would have to go directly to our database. We can't search OMDb by genre or plot, so we'd only be able to enable this once we have a fairly "decent-sized" database that gives reasonable results.

We need to consider how we go from summary results to detail results. In some applications, the detailed response might change over time. For example, if we were going to display the ratings of the movie, you could expect them to vary slightly over time. To get the latest rating values, you'd need to make sure that the movie data was re-fetched frequently to stay up to date, perhaps once a day or once a week. This would mean storing a *last-fetched* date/time and re-fetching after a certain period of time has elapsed since then.

Since we don't expect any of our data to change, we'll just store a flag to indicate if it's the full record or not. If someone tries to view a movie that doesn't have the full record we can go and fetch it in realtime, and expect it not to ever need to be updated.

With all that considered, here's how the flow would work:

- A user performs a search query on our site.
  - If it has been more than 24 hours since we've queried the API for that search term, re-fetch results.
  - Each result returned will be stored in our database with the `is_full_record` flag set to `False`.
- We'll query our local database using the search term.
- A list of results will be displayed.
- When the user visits the *detail* page for a movie, we'll check the `is_full_record` flag.
  - If it's `False`, we'll query the API to get the full data.
- The data that's displayed is fetched from our local database.

We'll go into each step in more detail as we build the site. Next we'll look at modeling the movie data in Django.

Django Models

It was important to see the response that was received from the API so that we know what fields are available and what we're going to store. We're going to need three models: `Movie`, `Genre` and `SearchTerm`. The first two should be obvious, the third will keep a record of search terms that were used so that we know when to re-run the queries.

Try It Out

Try It Out

You can follow along to set up a new Django project and get the models created. The fork you made of the GitHub repo `course4_proj` is the starting point for this new project. It is the equivalent of having run the `django-admin startproject course4_proj` command.

Once it's set up, we'll make the changes to support Django Configurations. Start in the `manage.py` file. Change the line:

```
from django.core.management import execute_from_command_line
```

to:

```
from configurations.management import execute_from_command_line
```

And underneath the line:

```
os.environ.setdefault('DJANGO_SETTINGS_MODULE',
        'course4_proj.settings')
```

Add this line:

```
os.environ.setdefault('DJANGO_CONFIGURATION', 'Dev')
```

Save `manage.py` and head over to `settings.py`. Add the Django Configurations imports:

Open settings.py

```
from configurations import Configuration
from configurations import values
```

Then define the `Dev` class:

```
class Dev(Configuration):
```

The easiest way to get all the settings in the `Dev` class it to select them all and hit **Tab** twice to indent them four spaces, so they become attributes of the `Dev` class.

You should then set up the logging configurations so we can see some debug messages. Add the `LOGGING` setting:

```
    LOGGING = {
        "version": 1,
        "disable_existing_loggers": False,
        "formatters": {
            "verbose": {
                "format": "{levelname} {asctime} {module}
    {process:d} {thread:d} {message}",
                "style": "{",
            },
        },
        "handlers": {
            "console": {
                "class": "logging.StreamHandler",
                "stream": "ext://sys.stdout",
                "formatter": "verbose",
            }
        },
        "root": {
            "handlers": ["console"],
            "level": "DEBUG",
        },
    }
```

We also need to update the settings so Django works with Codio. Start by importing the `os` module.

```
import os
```

Then update `ALLOWED_HOSTS` and add the other variables as well.

```
ALLOWED_HOSTS = values.ListValue(["localhost", "0.0.0.0",
    ".codio.io", os.environ.get('CODIO_HOSTNAME') +
    '-8000.codio.io'])
X_FRAME_OPTIONS = "ALLOW-FROM " +
    os.environ.get("CODIO_HOSTNAME") + "-8000.codio.io"
CSRF_COOKIE_SAMESITE = None
CSRF_TRUSTED_ORIGINS = [os.environ.get("CODIO_HOSTNAME") +
    "-8000.codio.io"]
CSRF_COOKIE_SECURE = True
SESSION_COOKIE_SECURE = True
CSRF_COOKIE_SAMESITE = "None"
SESSION_COOKIE_SAMESITE = "None"
```

Comment out the following lines of code in the `MIDDLEWARE` list.

```
MIDDLEWARE = [
    "debug_toolbar.middleware.DebugToolbarMiddleware",
    "django.middleware.security.SecurityMiddleware",
    "django.contrib.sessions.middleware.SessionMiddleware",
    "django.middleware.common.CommonMiddleware",
#     'django.middleware.csrf.CsrfViewMiddleware',

    "django.contrib.auth.middleware.AuthenticationMiddleware
    ",
    "django.contrib.messages.middleware.MessageMiddleware",
#
    'django.middleware.clickjacking.XFrameOptionsMiddleware',


]
```

Start the `movies` app by running `manage.py startapp movies` in the terminal:

Open the terminal

```
python3 manage.py startapp movies
```

If all your changes went OK, then the `movies` app will be created without any issue. Update `settings.py` by adding `'movies'` to `INSTALLED_APPS` (we'll create it now).

Open settings.py

Now let's look the the models. First `SearchTerm`:

Open models.py

```
class SearchTerm(models.Model):
    class Meta:
        ordering = ["id"]

    term = models.TextField(unique=True)
    last_search = models.DateTimeField(auto_now=True)
```

The term that's searched for, `term`, is `unique`. The `last_search` is the date that the search was last performed. We use this to make sure the search isn't repeated too often.

Next is `Genre`. It doesn't really need any explanation:

```
class Genre(models.Model):
    class Meta:
        ordering = ["name"]

    name = models.TextField(unique=True)
```

Finally there's `Movie`:

```
class Movie(models.Model):
    class Meta:
        ordering = ["title", "year"]

    title = models.TextField()
    year = models.PositiveIntegerField()
    runtime_minutes = models.PositiveIntegerField(null=True)
    imdb_id = models.SlugField(unique=True)
    genres = models.ManyToManyField(Genre,
        related_name="movies")
    plot = models.TextField(null=True, blank=True)
    is_full_record = models.BooleanField(default=False)
```

It has the fields that you would expect, to match what the API is supplying us. We use the `is_full_record` flag to determine if the `Movie` contains only the values in the list response, or if it has been supplemented with the full detail response.

Notice how `imdb_id` (the ID of the movie from the Internet Movie DataBase) has been defined. It's a `unique` `SlugField`. We're actually going to treat this as kind of like a primary key. In the movie detail URL, we'll use this, and then query the database on it. It will also allow us to create a mapping between the record in our local DB and the data provided by OMDb. OMDb in turn uses this ID to map back to IMDb.

Add these three models to `models.py` and save the file. Then, run the `makemigrations` and `migrate` management commands.

Next we're going to briefly discuss client design.

OMDb Client

OMDb Client

When building a REST client that's going to be used with Django, it's tempting to write it so that it's tightly integrated with Django. For example, OMDb requires an API key for each request. We're going to store this in `settings.py`, and it would be tempting to write an OMDb client that automatically retrieves the setting value when it's instantiated. We could also have our client directly write to the Django database.

However, this violates the idea of separation of concerns in software development: we're coupling two unrelated operations together (data fetching and database writing).

When writing our OMDb client, we'll try to stick to these rules:

- Have a single method that's used to make requests, so we have just one place to add authentication and error handling.
- It should not need to know about Django at all, this will allow us to refactor more easily and re-use the code in a non-Django project.
- The transformation from JSON to Python should take place in the client. The consumers of the client should not need to know about the data structure of the response. This means our API can change and only our client code needs to be updated, not different parsing code throughout our codebase.

Now you can follow along as we implement the client and data helper class.

Try It Out

Start by creating a new directory in the root of the Django project (the first `course4_proj`), called `omdb`. Inside it, create an empty file called `__init__.py`, then another file called `client.py`.

Inside `client.py` we need to start with some imports, and set up two global variables. Add this code:

```
import logging

import requests

logger = logging.getLogger(__name__)

OMDB_API_URL = "https://www.omdbapi.com/"
```

Here we're just setting up the logger and defining the API URL which we'll use in the client.

▼ **Define API URL**
Another option could be to define the API URL in the Django settings. You'd probably prefer this if the API you were working with had both dev and production URLs. Since OMDb doesn't and we don't expect it to change, we'll just leave it set here.

Next we are going to define a class that acts as an intermediary/transformer between the JSON dictionary returned from OMDb and raw Python data. It's responsible for:

- Validating and transforming the movie's runtime.
- Converting the movie's year into and `int`.
- Checking if keys are set and raising exceptions if trying to access detail keys on non-detail response.
- Splitting the genre into a list.

You should be able to see the advantage of using this class. We're moving all the transformations from API to Python into a single place. There is a separation between the data that's returned from the API and how we're using said data in Python. If the API response were to change, for example, `Title` changes to `Name`, we could still refer to `title` in our code and just change the key that's being used to fetch it from the data.

Add this class to `client.py`:

```
class OmdbMovie:
    """A simple class to represent movie data coming back from
        OMDb
    and transform to Python types."""

    def __init__(self, data):
        """Data is the raw JSON/dict returned from OMDb"""
        self.data = data

    def check_for_detail_data_key(self, key):
        """Some keys are only in the detail response, raise an
```

```
        exception if the key is not found."""

        if key not in self.data:
            raise AttributeError(
                f"{key} is not in data, please make sure this is
        a detail response."
            )

    @property
    def imdb_id(self):
        return self.data["imdbID"]

    @property
    def title(self):
        return self.data["Title"]

    @property
    def year(self):
        return int(self.data["Year"])

    @property
    def runtime_minutes(self):
        self.check_for_detail_data_key("Runtime")

        rt, units = self.data["Runtime"].split(" ")

        if units != "min":
            raise ValueError(f"Expected units 'min' for runtime.
        Got '{units}")

        return int(rt)

    @property
    def genres(self):
        self.check_for_detail_data_key("Genre")

        return self.data["Genre"].split(", ")

    @property
    def plot(self):
        self.check_for_detail_data_key("Plot")
        return self.data["Plot"]
```

Here's an example of how it would be used:

```
# data is a dictionary from the API
>>> data = {"Title": "My Great Movie", "Year": "1991"}
>>> movie = OmdbMovie(data)
>>> movie.title
'My Great Movie'
>>> movie.year
1991  # notice that this is an integer
>>> movie.plot
# ...
AttributeError: Plot is not in data, please make sure this is a
        detail response.
```

Next, we'll the `OmdbClient`. It has two "public" methods:

- `get_by_imdb_id()`, which takes the IMDB ID of the movie and fetches the full detail response. It then decodes the JSON, and returns an `OmdbMovie` instance created from this data.
- `search()`: This takes a search term and performs a search against the API for the term. It will then iterate over each movie summary, and `yield` a `OmdbMovie` instance for each. It will then fetch the next page of results and `yield` for each of those, and so on.

▼ **Yield**

If you're not familiar with the `yield` keyword, it turns a Python function into a generator. This means the function's "return value" must be iterated across. For example, `results = client.search(term)` doesn't mean results contain a list of results. You would have to do something like `results = [movie for movie in client.search(term)]`. Going more in-depth into generators is beyond the scope of this course, the Python Wiki page on Generators is a good place to start for more information.

Both of these methods call just one method to make the request: the `make_request()` method. Since each API endpoint is on the same URL, it only varies on the paramaters passed. `make_request()` accepts just a single argument: the dictionary of paramaters to pass to the URL. The method will automatically add the API key to the `params` when sending to the API.

Finally the `__init__()` method accepts an `api_key` argument which is stored on `self` and referred to in `make_request()`. Here's the `OmdbClient` in full, you can copy this into `client.py`, making sure it comes after the `OmdbMovie` class definition:

```
class OmdbClient:
    def __init__(self, api_key):
        self.api_key = api_key

    def make_request(self, params):
        """Make a GET request to the API, automatically adding
        the `apikey` to parameters."""
        params["apikey"] = self.api_key

        resp = requests.get(OMDB_API_URL, params=params)
        resp.raise_for_status()
        return resp

    def get_by_imdb_id(self, imdb_id):
        """Get a movie by its IMDB ID"""
        logger.info("Fetching detail for IMDB ID %s", imdb_id)
        resp = self.make_request({"i": imdb_id})
        return OmdbMovie(resp.json())

    def search(self, search):
        """Search for movies by title. This is a generator so
        all results from all pages will be iterated across."""
        page = 1
        seen_results = 0
        total_results = None

        logger.info("Performing a search for '%s'", search)

        while True:
            logger.info("Fetching page %d", page)
            resp = self.make_request({"s": search, "type":
"movie", "page": str(page)})
            resp_body = resp.json()
            if total_results is None:
                total_results = int(resp_body["totalResults"])

            for movie in resp_body["Search"]:
                seen_results += 1
                yield OmdbMovie(movie)

            if seen_results >= total_results:
                break

            page += 1
```

Even though this class is separate from Django, that doesn't stop us from writing some helper code to make it easier to get an `OmdbClient` instance. Let's write a function that will instantiate an `OmdbClient` and pass in the key

in the Django settings. Create a file called `django_client.py` inside the `omdb` directory. Add this content:

Open django_client.py

```
from django.conf import settings

from omdb.client import OmdbClient


def get_client_from_settings():
    """Create an instance of an OmdbClient using the OMDB_KEY
        from the Django settings."""
    return OmdbClient(settings.OMDB_KEY)
```

As you can see, the `get_client_from_settings()` function instantiates an `OmdbClient` from the `OMDB_KEY` in the Django settings. But for it to work, we need to add that setting. Head back to `settings.py`. You have two choices. The more "correct" method would be to make use of Django Configuration's `values.SecretValue` class. For example:

```
    OMDB_KEY = values.SecretValue()
```

This prevents `manage.py` from running unless the `DJANGO_OMDB_KEY` value is present as an environment variable. So, you would either need to export this variable before running `manage.py`:

```
$ export DJANGO_OMDB_KEY=abc123
$ python manage.py [command...]
```

Or prepend it to the command:

```
$ DJANGO_OMDB_KEY=abc123 python manage.py [command...]
```

Since you're only using this project and key for your own personal development, you might find it tedious having to do this. Therefore, you can choose to hard code the API key directly into the `settings.py` file. **Important**, use *your* OMDb key in place of `"abc123"`.

Open settings.py

```
    OMDB_KEY = "abc123"
```

As we saw in Course One, this is kind of like having the `SECRET_KEY` in `settings.py`. It's fine for development, but when going into production you would want to make sure it was stored securely and not in your codebase.

Helper Functions

Helper Functions

Rather than integrate `OmdbClient` directly into our Django views, we'll write three helper functions that contain the logic. This will allow to add some management commands without repeating code. The helper functions will be:

- `get_or_create_genres()`: Accepts a list of genre names (strings) and `yields` a `Genre` object for each of them.
- `fill_movie_details()`: Accepts a `Movie` object and then queries the OMDb API to fill in the missing data.
- `search_and_save()`: Accepts a search term (string). Checks if the search has been performed in the past 24 hours, and returns if so. Otherwise, it performs the search against the OMDb API and creates `Movie` objects for each result.

Try It Out

Now you will implement these helper functions. Start by creating the file `omdb_integration.py`, inside the `movies` app directory. You'll first need to add some imports and set up the logger:

```
import logging
import re
from datetime import timedelta

from django.utils.timezone import now

from movies.models import Genre, SearchTerm, Movie
from omdb.django_client import get_client_from_settings

logger = logging.getLogger(__name__)
```

Next, implement the `get_or_create_genres()` function:

```
def get_or_create_genres(genre_names):
    for genre_name in genre_names:
        genre, created =
        Genre.objects.get_or_create(name=genre_name)
        yield genre
```

Then the `fill_movie_details()` function:

```
def fill_movie_details(movie):
    """
    Fetch a movie's full details from OMDb. Then, save it to the
        DB. If the movie already has a `full_record` this does
    nothing, so it's safe to call with any `Movie`.
    """
    if movie.is_full_record:
        logger.warning(
            "'%s' is already a full record.",
            movie.title,
        )
        return
    omdb_client = get_client_from_settings()
    movie_details = omdb_client.get_by_imdb_id(movie.imdb_id)
    movie.title = movie_details.title
    movie.year = movie_details.year
    movie.plot = movie_details.plot
    movie.runtime_minutes = movie_details.runtime_minutes
    movie.genres.clear()
    for genre in get_or_create_genres(movie_details.genres):
        movie.genres.add(genre)
    movie.is_full_record = True
    movie.save()
```

Notice that if the movie already has all the data (`is_full_record` is `True`) then we do nothing. Otherwise we instantiate an `OmdbClient`, fetch the details, then update and save the `Movie`.

Finally the `search_and_save()` function:

```
def search_and_save(search):
    """
    Perform a search for search_term against the API, but only
        if it hasn't been searched in the past 24 hours. Save
    each result to the local DB as a partial record.
    """
    # Replace multiple spaces with single spaces, and lowercase
        the search
    normalized_search_term = re.sub(r"\s+", " ", search.lower())

    search_term, created =
        SearchTerm.objects.get_or_create(term=normalized_search_term)



    if not created and (search_term.last_search > now() -
        timedelta(days=1)):
        # Don't search as it has been searched recently
        logger.warning(
            "Search for '%s' was performed in the past 24 hours
        so not searching again.",
            normalized_search_term,
        )
        return


    omdb_client = get_client_from_settings()


    for omdb_movie in omdb_client.search(search):
        logger.info("Saving movie: '%s' / '%s'",
        omdb_movie.title, omdb_movie.imdb_id)
        movie, created = Movie.objects.get_or_create(
            imdb_id=omdb_movie.imdb_id,
            defaults={
                "title": omdb_movie.title,
                "year": omdb_movie.year,
            },
        )


        if created:
            logger.info("Movie created: '%s'", movie.title)


    search_term.save()
```

Notice that the search term is normalized (multiple spaces removed, and then lowercased) before checking if the search has been done before. If the search has not been made, then an `OmdbClient` is instantiated and a search performed, with each result being saved to the local database.

Now in order to test these we'll create some management commands. This will allow testing without having to write views and templates yet. In the `movies` app directory, create a directory called `management` with an empty

`__init__.py` file inside it. Then inside the `management` directory, create another directory called `commands`, also with an empty `__init__.py` file inside.

The fist management command we'll create is called `movie_search`, which will allow searching and storing movies. It will make use of the `search_and_save()` function.

Create a file inside the `commands` directory called `movie_search.py`. Copy and paste this content:

Open movie_search.py

```
from django.core.management.base import BaseCommand

from movies.omdb_integration import search_and_save


class Command(BaseCommand):
    help = "Search OMDb and populates the database with results"

    def add_arguments(self, parser):
        parser.add_argument("search", nargs="+")

    def handle(self, *args, **options):
        search = " ".join(options["search"])
        search_and_save(search)
```

To test this out and perform your first search, execute the `movie_search` command with `manage.py`:

Open the terminal

```
python3 manage.py movie_search django unchained
```

You should see output similar to this:

```
DEBUG 2021-09-20 23:36:09,123 selector_events 97768 4502050240
        Using selector: KqueueSelector
INFO 2021-09-20 23:36:09,138 client 97768 4502050240 Performing
        a search for 'django unchained'
INFO 2021-09-20 23:36:09,138 client 97768 4502050240 Fetching
        page 1
DEBUG 2021-09-20 23:36:09,176 connectionpool 97768 4502050240
        Starting new HTTPS connection (1): www.omdbapi.com:443
DEBUG 2021-09-20 23:36:10,484 connectionpool 97768 4502050240
        https://www.omdbapi.com:443 "GET /?
        s=django+unchained&type=movie&page=1&apikey=key
        HTTP/1.1" 200 None
INFO 2021-09-20 23:36:10,488 omdb_integration 97768 4502050240
        Saving movie: 'Django Unchained' / 'tt1853728'
INFO 2021-09-20 23:36:10,493 omdb_integration 97768 4502050240
        Movie created: 'Django Unchained'

...
```

You'll probably see a lot more output, and sometimes the search can take a while to complete if there are a lot of results. But now, you will have `Movie` objects in your database. You can verify this by starting a Django shell and querying the database.

You can also test running the same search more than once, and verify that the search is not performed again within 24 hours.

```
python3 manage.py movie_search django unchained
```

The output should inform the user that a search was done in the past 24 hours,

```
DEBUG 2021-09-21 02:30:11,215 selector_events 98566 4474152384
        Using selector: KqueueSelector
WARNING 2021-09-21 02:30:11,229 omdb_integration 98566
        4474152384 Search for 'django unchained' was performed
        in the past 24 hours so not searching again.
```

The final command we'll create is the one to fetch the full data for a movie, given its IMDB ID. This will use the `fill_movie_details() function`. Create a file called `movie_fill.py` inside the `commands` directory.

Open movie_fill.py

Enter this content:

```
import logging

from django.core.management.base import BaseCommand

from movies.models import Movie
from movies.omdb_integration import fill_movie_details

logger = logging.getLogger(__name__)


class Command(BaseCommand):
    help = "Search OMDb and populates the database with results"

    def add_arguments(self, parser):
        parser.add_argument("imdb_id", nargs=1)

    def handle(self, *args, **options):
        try:
            movie = Movie.objects.get(imdb_id=options["imdb_id"]
        [0])
        except Movie.DoesNotExist:
            logger.error("Movie with IMDB ID '%s' was not
        found", options["imdb_id"][0])
            return

        fill_movie_details(movie)
```

To use this command, you need to provide an IMDb ID at the command line, like this:

```
python3 manage.py movie_fill tt1853728
```

Your output should look something like this:

```
DEBUG 2021-09-21 02:27:21,083 selector_events 98528 4615587264
        Using selector: KqueueSelector
INFO 2021-09-21 02:27:21,091 client 98528 4615587264 Fetching
        detail for IMDB ID tt1853728
DEBUG 2021-09-21 02:27:21,131 connectionpool 98528 4615587264
        Starting new HTTPS connection (1): www.omdbapi.com:443
DEBUG 2021-09-21 02:27:21,891 connectionpool 98528 4615587264
        https://www.omdbapi.com:443 "GET /?
        i=tt1853728&apikey=key HTTP/1.1" 200 None
```

Try running the command more than once with the same IMDb ID.

```
python3 manage.py movie_fill tt1853728
```

You'll get a warning that the particular movie is already a full record.

```
DEBUG 2021-09-21 02:27:43,192 selector_events 98544 4607014336
        Using selector: KqueueSelector
WARNING 2021-09-21 02:27:43,204 omdb_integration 98544
        4607014336 'Django Unchained' is already a full record.
```

Wrap-Up

Wrap-Up

We've implemented the models, client, and functions to tie them together. At this point it would be trivial to write the views and templates. We're not going to do that as they should be straightforward:

- A *search* view will accept a search query, then call `search_and_save()` with it. Remember that we can call `search_and_save()` repeatedly as it has the logic for checking if the period is elapsed before re-running the search. Then after the database has been populated, we can search it with the search term to find matching movies.
- A *detail* view that takes the IMDb ID as part of the URL. It will fetch the `Movie` object from the database using this ID, returning a 404 if one is not found. Then, it will call `fill_movie_details()` with the `Movie` object. We can safely call this multiple times as the data will not be re-populated if `is_full_record` is already true. A template can be rendered containing the movie data.

▼ **Results**

We should note that we're assuming our movie search will return the same results as OMDb's. For example, if OMDb uses a full-text search but we use a simple `contains` query then the records might not match. This is something that you would need to test with trial and error to make sure the results were consistent.

The purpose of this section is not to be a tutorial on integrating OMDb with Django. Rather, an overview of the steps/thought process of designing an API integration with Django.

The important takeaway points are:

- API keys can be fetched from Django settings. Consider using a `SecretValue` field to retrieve them from the environment, similar to the `SECRET_KEY` setting.
- Your REST client should have a single method that's called to make requests. This means you have only one place to implement authentication and HTTP error handling, rather than littering your code with them.
- REST clients should not be directly tied to Django. Allow your REST client to be instantiated without knowing about any Django settings.
- The client should also not be responsible for inserting any data into the

Django database.

- It is OK to create a helper class or function that will instantiate the REST client from the Django settings.
- Don't return the response body directly from your client (the decoded JSON response dictionary, for example). Instead, use a transformation function or class to turn it into a standard format. This means if the API starts returning different data you only have one place to update the parser.
- Check what data comes back for a list or detail response. Your models may need to keep track if they are a full response or not. This could be just a flag, or a datetime if the model data needs to be refreshed periodically.
- Consider some protection against users causing repeated API calls for the same data.
- (This point is not REST API specific:) Having helper functions outside of your view code means you can use the functionality in management commands as well, without having to write duplicate code.

Finally it's important to mention that you need to follow the licensing rules of the API. The terms of use may not allow you to make a local copy of your data. Fortunately for us the data in OMDb is licensed under the Creative Commons Attribution-NonCommercial 4.0 International License. This means we are allowed to copy, redistribute, remix, transform, etc, the data, as long as the source is attributed and we use it in a non-commercial way. If we were building a UI for our site, we'd make sure to have attribution and follow these terms.

In the next section, we're going to look at interacting with an API using a third-party library instead of our own client.

Pushing to GitHub

Pushing to GitHub

Before continuing, you must push your work to GitHub. In the terminal:

- Commit your changes:

```
git add .
git commit -m "Finish omdb api"
```

- Push to GitHub:

```
git push
```