Social distancing from your system's dependencies in a healthy way

EuroPython
23 July 2020

Olga Matoula (@olgamatoula)

TechAtBloomberg.com

Bloomberg Engineering

HELLO!

I am Olga Matoula

Software Engineer @Bloomberg

@olgamatoula

🇬🇷 ☀️ 💡 💃 👩‍🍳 👩‍💻

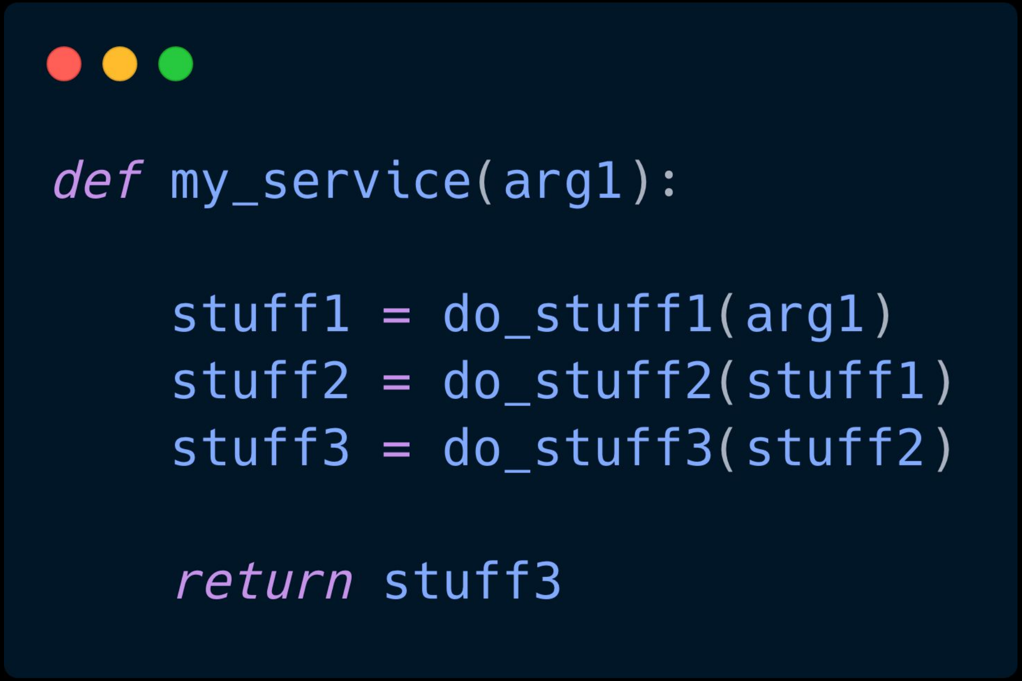
This is the story of a Python service ~~🐍~~

written by C++ developers

This is the story of a Python service ~~🐍~~

written by C++ developers

who envisioned something like this

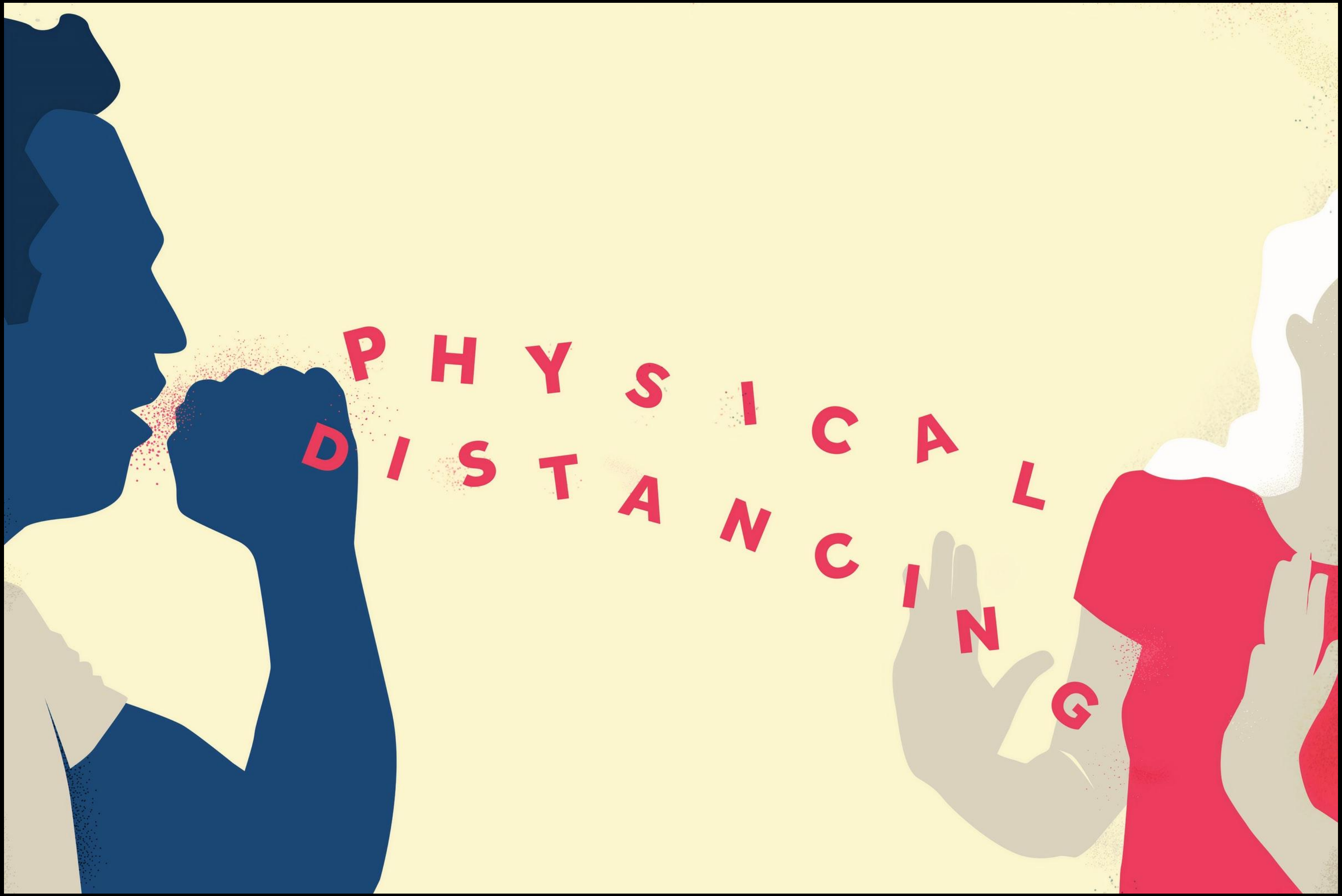


```python
def my_service(arg1):

    stuff1 = do_stuff1(arg1)
    stuff2 = do_stuff2(stuff1)
    stuff3 = do_stuff3(stuff2)

    return stuff3
```

Is this ready to go to production?

Marc - Product Manager

Jenkins was 😡 furious

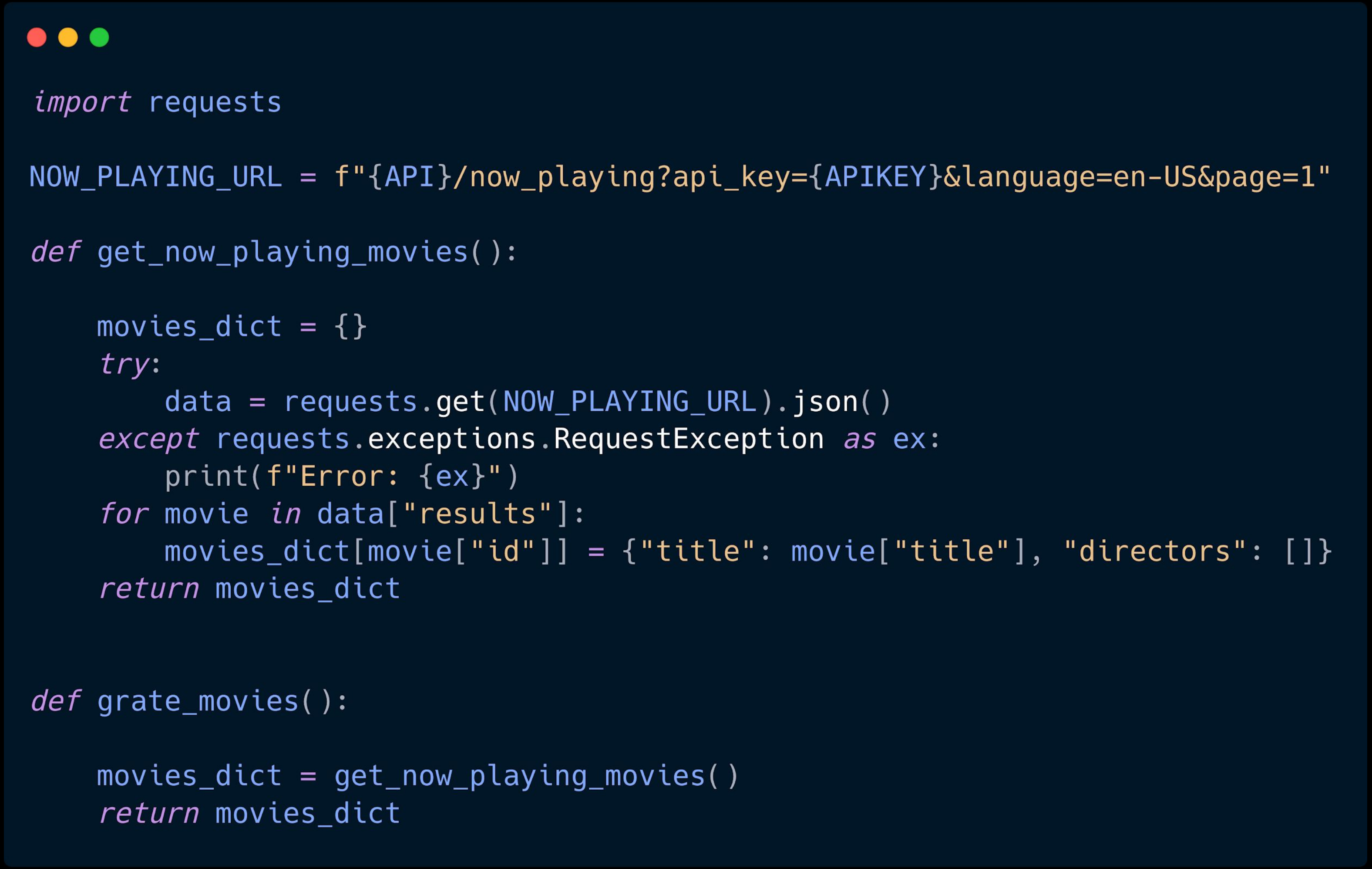


Physical Distancing. Image created by Kyle Mueller.

Photo by Myke Simon on Unsplash

Grate movie 🎥 application

using TMDB API
a simple version

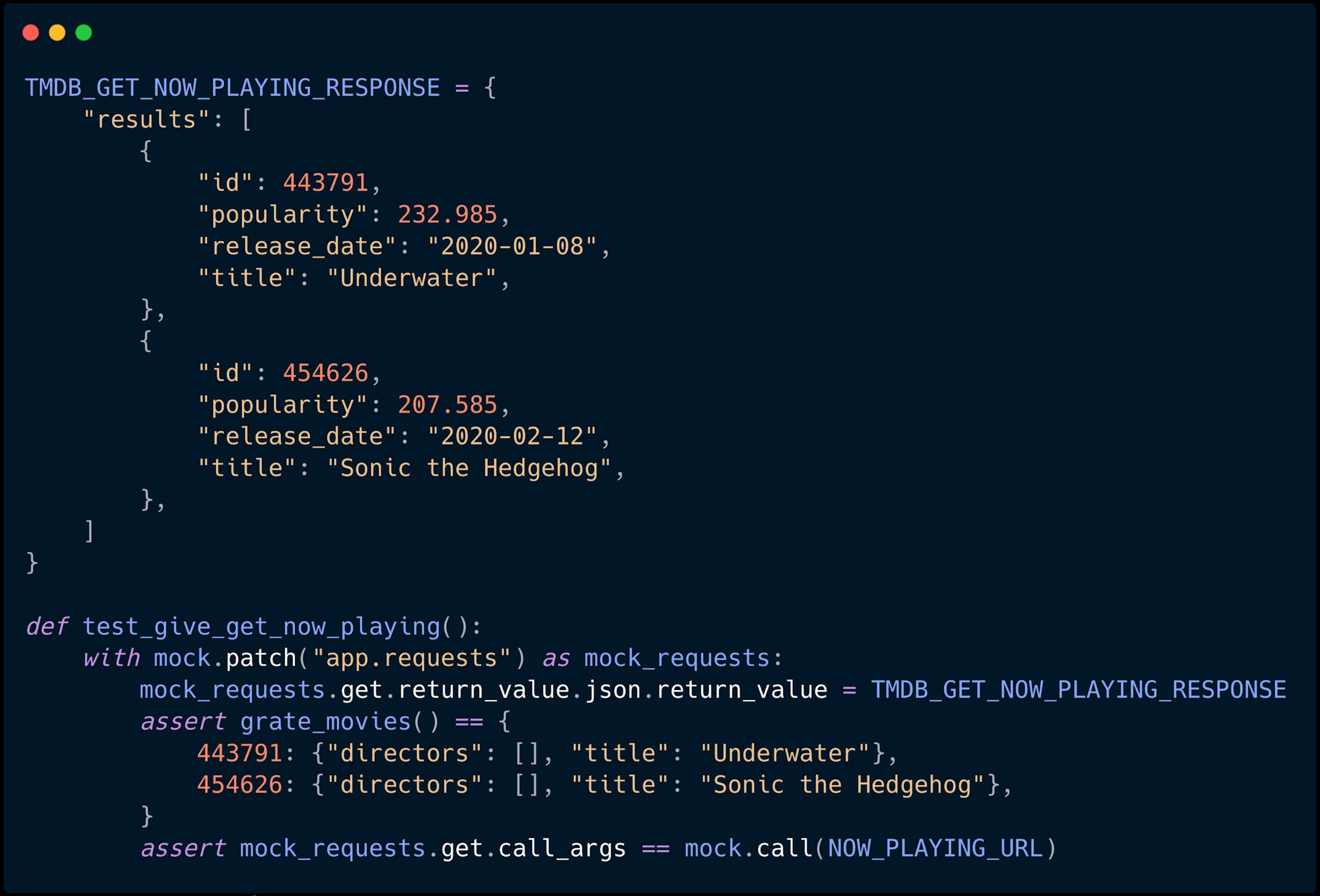


```python
import requests

NOW_PLAYING_URL = f"{API}/now_playing?api_key={APIKEY}&language=en-US&page=1"

def get_now_playing_movies():

    movies_dict = {}
    try:
        data = requests.get(NOW_PLAYING_URL).json()
    except requests.exceptions.RequestException as ex:
        print(f"Error: {ex}")
    for movie in data["results"]:
        movies_dict[movie["id"]] = {"title": movie["title"], "directors": []}
    return movies_dict


def grate_movies():

    movies_dict = get_now_playing_movies()
    return movies_dict
```

Grate movie 🎥 application

using TMDB API

a simple test



```python
TMDB_GET_NOW_PLAYING_RESPONSE = {
    "results": [
        {
            "id": 443791,
            "popularity": 232.985,
            "release_date": "2020-01-08",
            "title": "Underwater",
        },
        {
            "id": 454626,
            "popularity": 207.585,
            "release_date": "2020-02-12",
            "title": "Sonic the Hedgehog",
        },
    ]
}

def test_give_get_now_playing():
    with mock.patch("app.requests") as mock_requests:
        mock_requests.get.return_value.json.return_value = TMDB_GET_NOW_PLAYING_RESPONSE
        assert grate_movies() == {
            443791: {"directors": [], "title": "Underwater"},
            454626: {"directors": [], "title": "Sonic the Hedgehog"},
        }
        assert mock_requests.get.call_args == mock.call(NOW_PLAYING_URL)
```

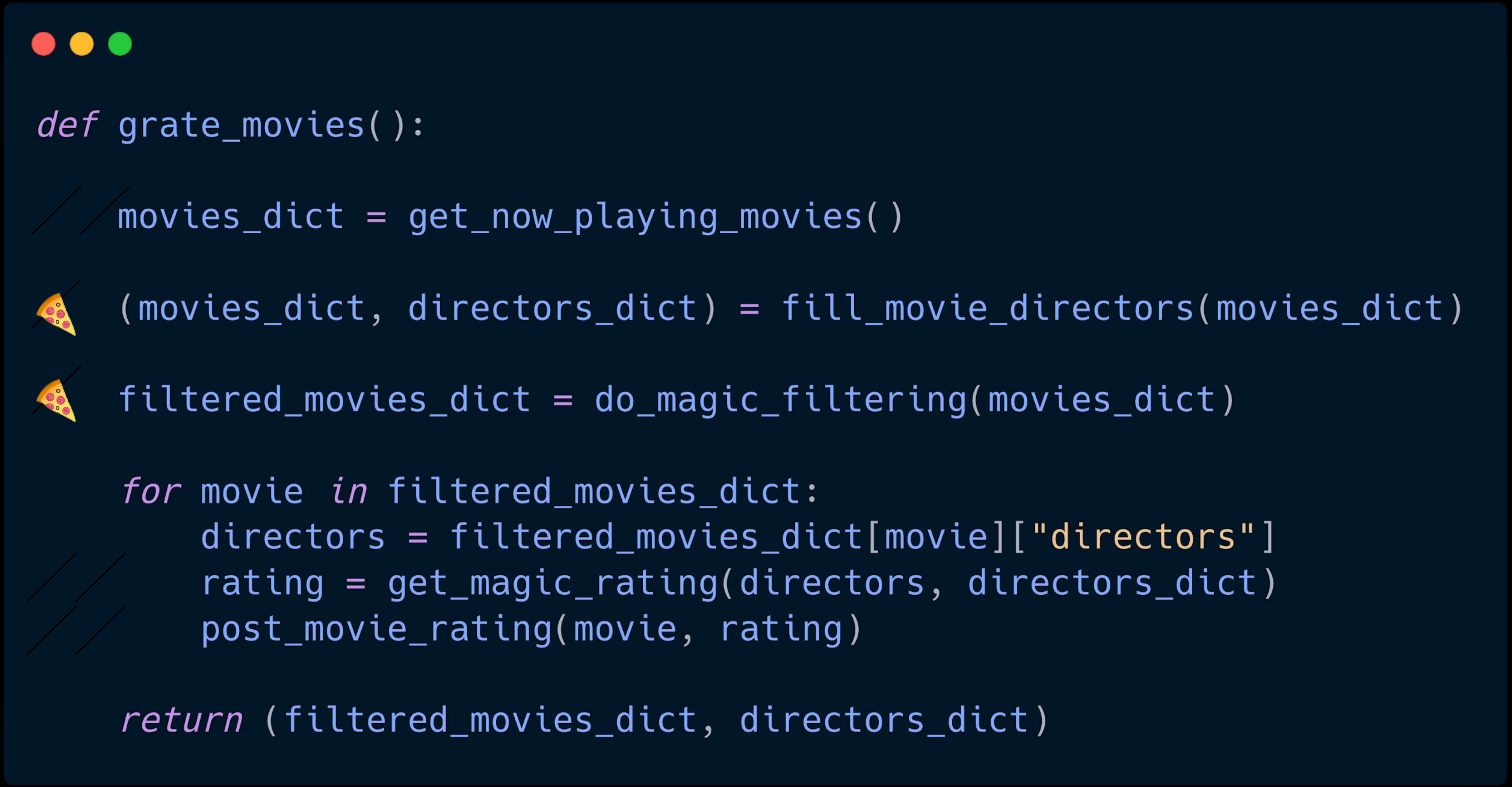
Grate movie 🎥 application

using TMDB API
a rich version

```python
def grate_movies():

    movies_dict = get_now_playing_movies()

    (movies_dict, directors_dict) = fill_movie_directors(movies_dict)

    filtered_movies_dict = do_magic_filtering(movies_dict)

    for movie in filtered_movies_dict:
        directors = filtered_movies_dict[movie]["directors"]
        rating = get_magic_rating(directors, directors_dict)
        post_movie_rating(movie, rating)

    return (filtered_movies_dict, directors_dict)
```

Grate movie 🎥 application

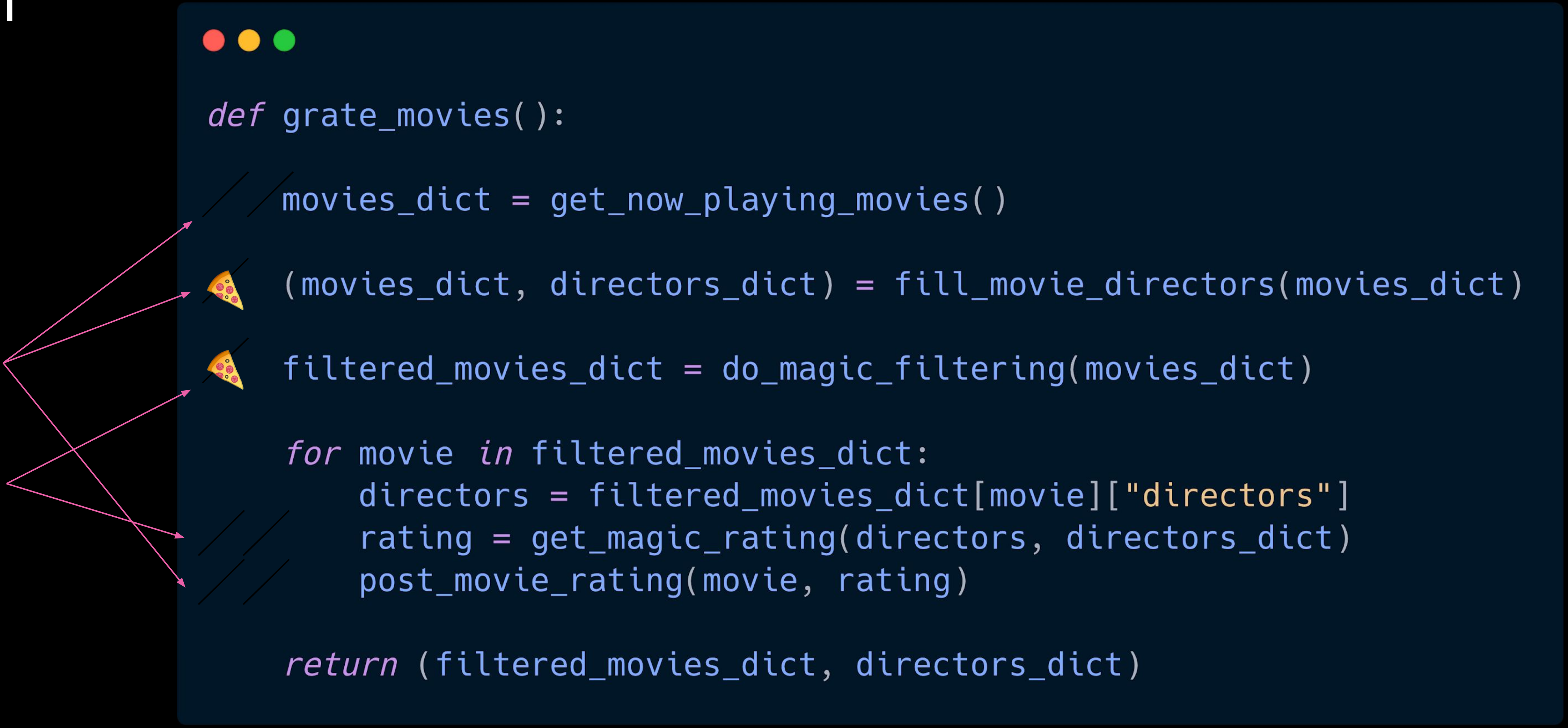
using TMDB API
a rich version



```python
def grate_movies():

    movies_dict = get_now_playing_movies()

🍕  (movies_dict, directors_dict) = fill_movie_directors(movies_dict)

🍕  filtered_movies_dict = do_magic_filtering(movies_dict)

    for movie in filtered_movies_dict:
        directors = filtered_movies_dict[movie]["directors"]
        rating = get_magic_rating(directors, directors_dict)
        post_movie_rating(movie, rating)

    return (filtered_movies_dict, directors_dict)
```

Grate movie 🎥 application

using TMDB API
a rich version

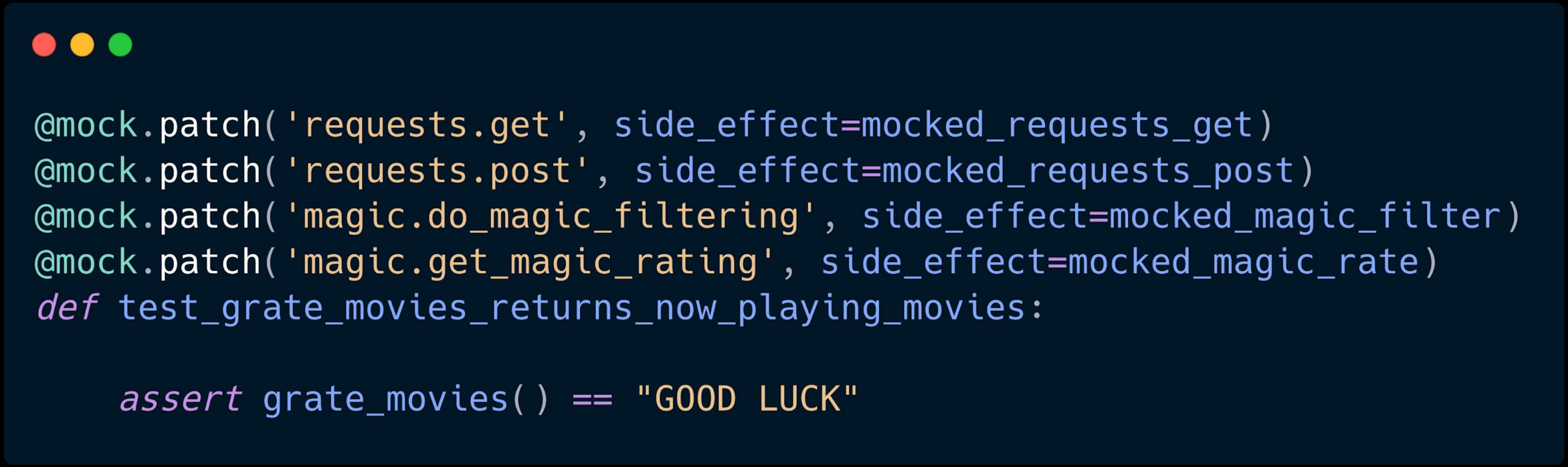


```
def grate_movies():
    movies_dict = get_now_playing_movies()
🍕  (movies_dict, directors_dict) = fill_movie_directors(movies_dict)
🍕  filtered_movies_dict = do_magic_filtering(movies_dict)

    for movie in filtered_movies_dict:
        directors = filtered_movies_dict[movie]["directors"]
        rating = get_magic_rating(directors, directors_dict)
        post_movie_rating(movie, rating)

    return (filtered_movies_dict, directors_dict)
```

Patch it up!?

Grate movie 🎥 application

using TMDB API
a rich test



```python
@mock.patch('requests.get', side_effect=mocked_requests_get)
@mock.patch('requests.post', side_effect=mocked_requests_post)
@mock.patch('magic.do_magic_filtering', side_effect=mocked_magic_filter)
@mock.patch('magic.get_magic_rating', side_effect=mocked_magic_rate)
def test_grate_movies_returns_now_playing_movies:

    assert grate_movies() == "GOOD LUCK"
```

Grate movie 🎥 application

using TMDB API
more tests?

What about

- different arguments
- other responses
- empty responses
- failures and exceptions

Every time you use mock.patch
it means you have a design flaw
in your architecture

Someone on stackoverflow

Don't mock what you don't own

Many wise developers

Back to the roots

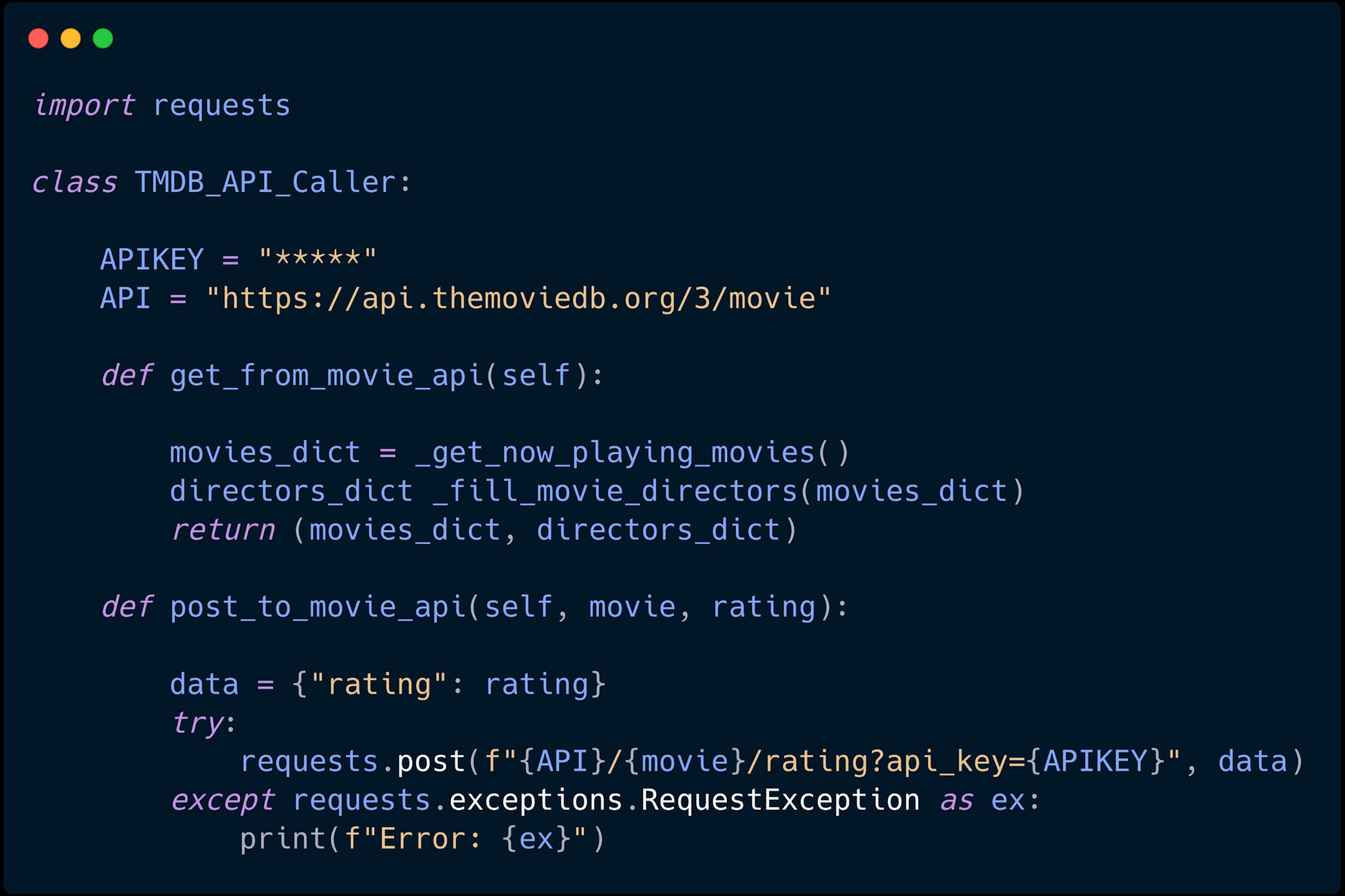
Photo by veeterzy on Unsplash

Dependency Injection

🔌 🎁

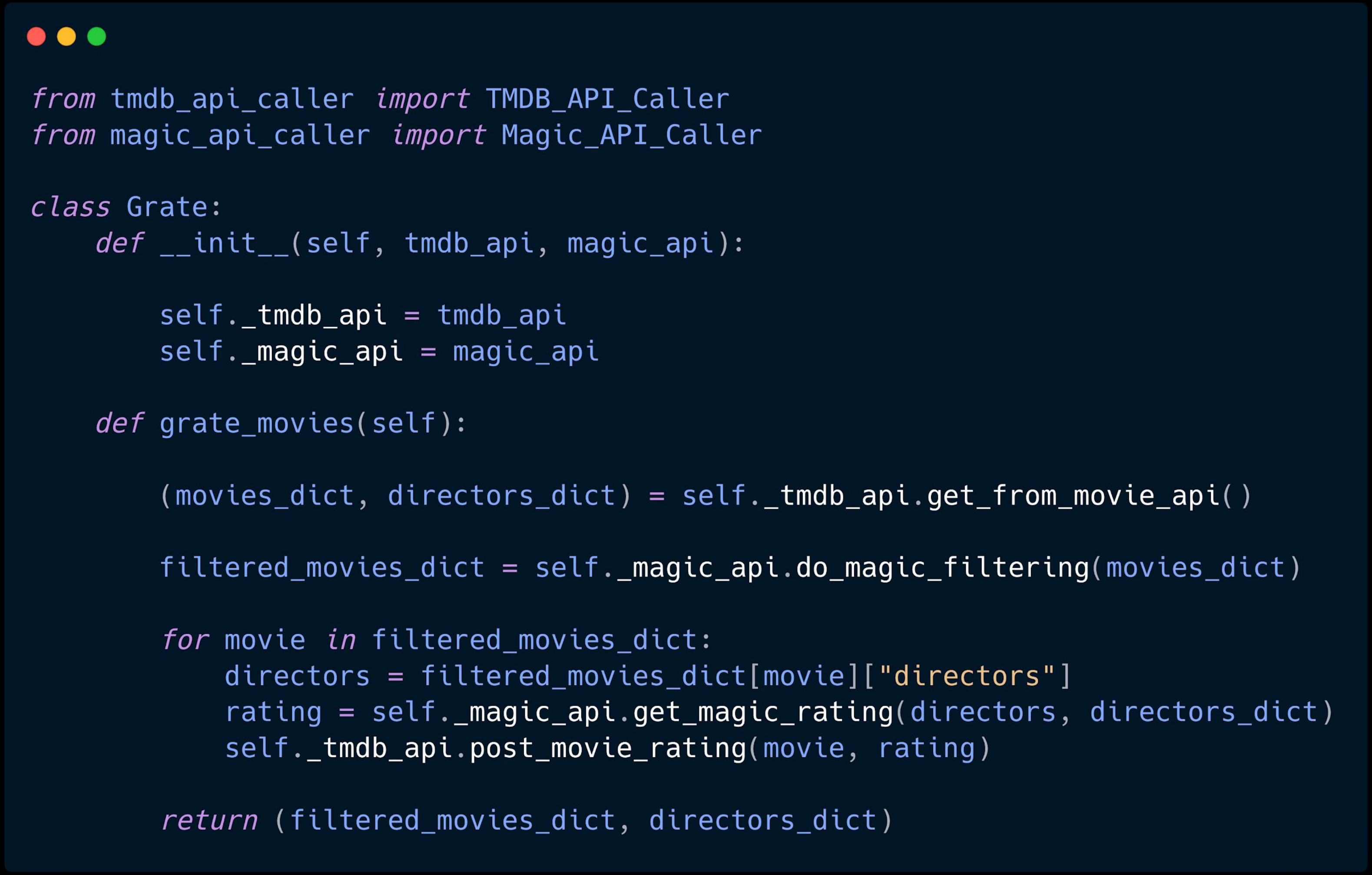
The Adapter Pattern (thin wrapper)

Wrapper



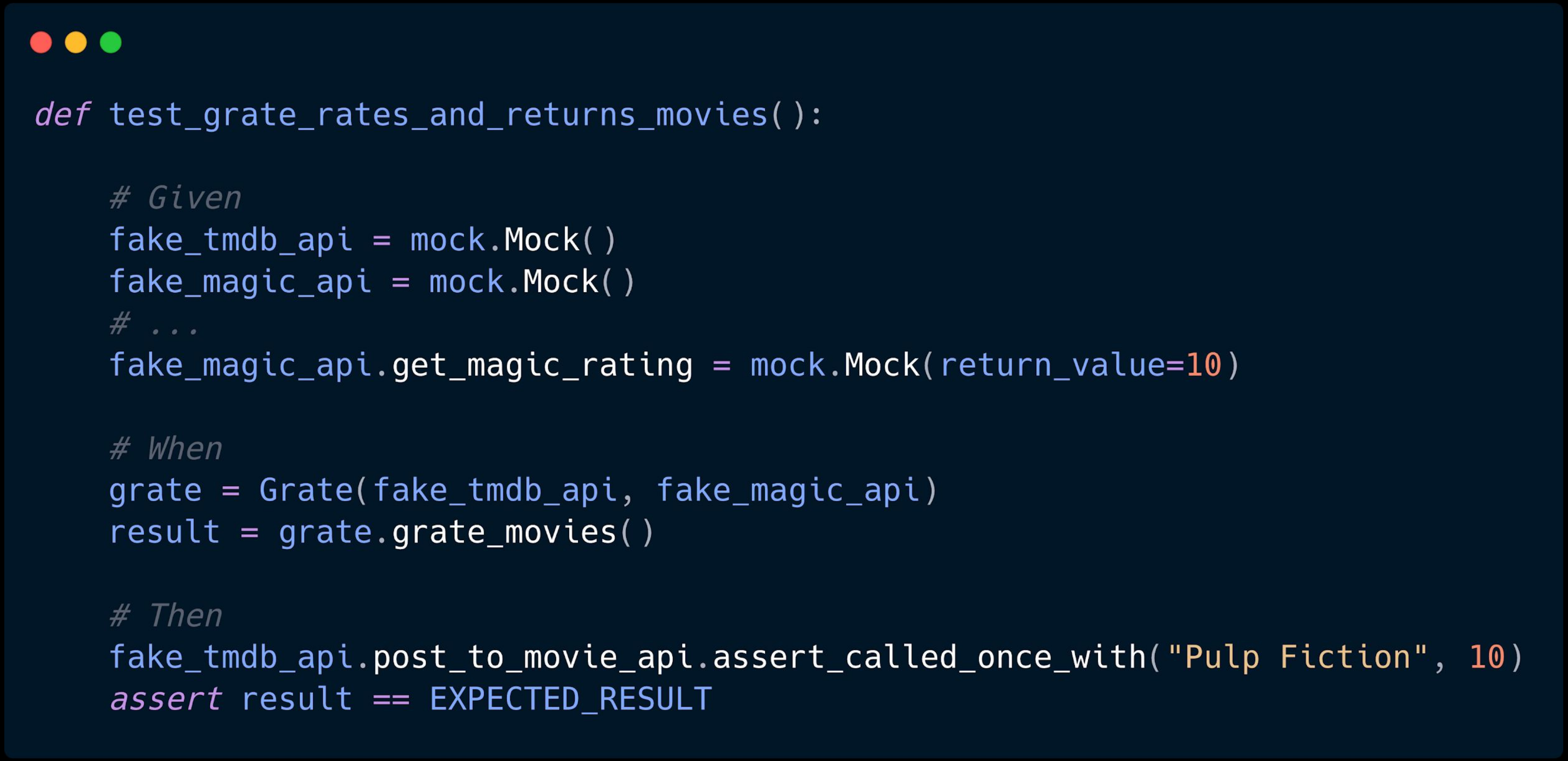
```python
import requests

class TMDB_API_Caller:

    APIKEY = "*****"
    API = "https://api.themoviedb.org/3/movie"

    def get_from_movie_api(self):

        movies_dict = _get_now_playing_movies()
        directors_dict _fill_movie_directors(movies_dict)
        return (movies_dict, directors_dict)

    def post_to_movie_api(self, movie, rating):

        data = {"rating": rating}
        try:
            requests.post(f"{API}/{movie}/rating?api_key={APIKEY}", data)
        except requests.exceptions.RequestException as ex:
            print(f"Error: {ex}")
```

Dependency Injection



```python
from tmdb_api_caller import TMDB_API_Caller
from magic_api_caller import Magic_API_Caller

class Grate:
    def __init__(self, tmdb_api, magic_api):

        self._tmdb_api = tmdb_api
        self._magic_api = magic_api

    def grate_movies(self):

        (movies_dict, directors_dict) = self._tmdb_api.get_from_movie_api()

        filtered_movies_dict = self._magic_api.do_magic_filtering(movies_dict)

        for movie in filtered_movies_dict:
            directors = filtered_movies_dict[movie]["directors"]
            rating = self._magic_api.get_magic_rating(directors, directors_dict)
            self._tmdb_api.post_movie_rating(movie, rating)

        return (filtered_movies_dict, directors_dict)
```

Happy tests 🤗



```python
def test_grate_rates_and_returns_movies():

    # Given
    fake_tmdb_api = mock.Mock()
    fake_magic_api = mock.Mock()
    # ...
    fake_magic_api.get_magic_rating = mock.Mock(return_value=10)

    # When
    grate = Grate(fake_tmdb_api, fake_magic_api)
    result = grate.grate_movies()

    # Then
    fake_tmdb_api.post_to_movie_api.assert_called_once_with("Pulp Fiction", 10)
    assert result == EXPECTED_RESULT
```

🧪 readable, extensible, flexible tests

🧺 no danger to forget a mock.patch

🤷‍♀️ no care for specific API implementation details

👩‍💼 test only the business logic, not the I/O

🎨 more thought through design

What about integration tests? 🧩

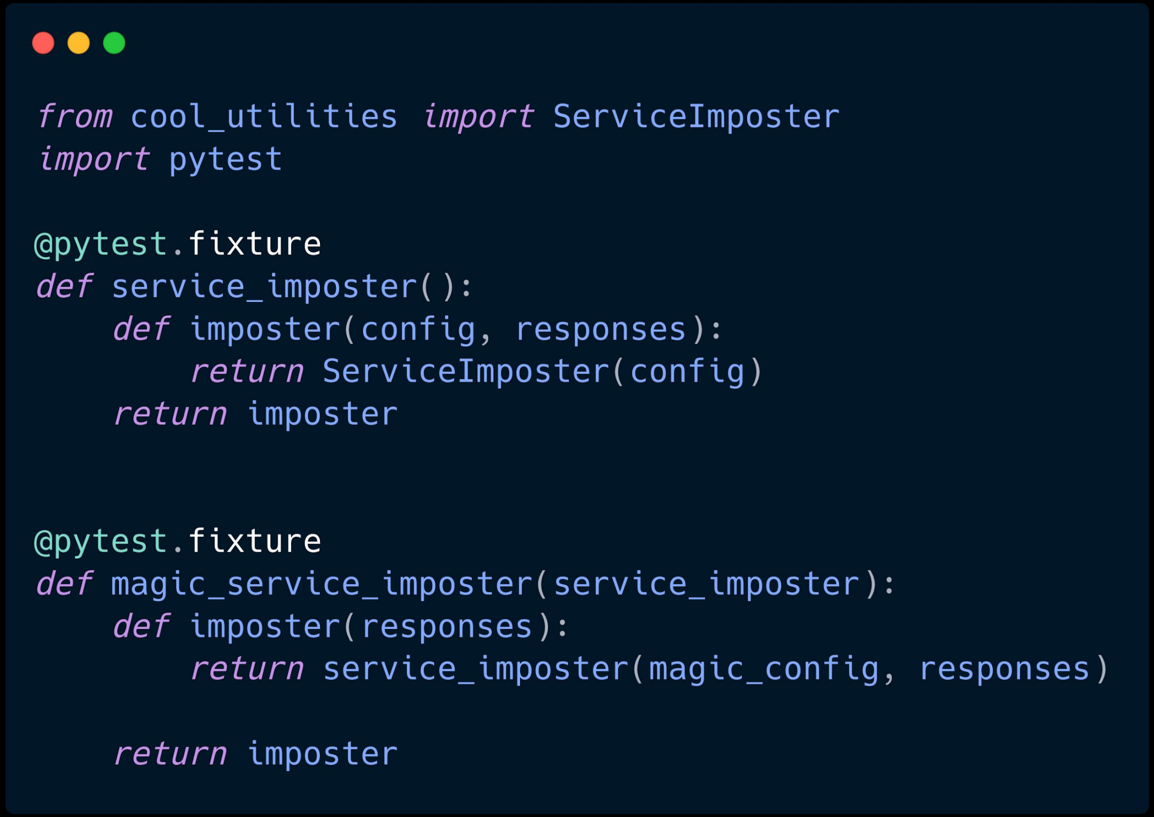
What about integration tests? 🧩

Imposters

aka verified fakes

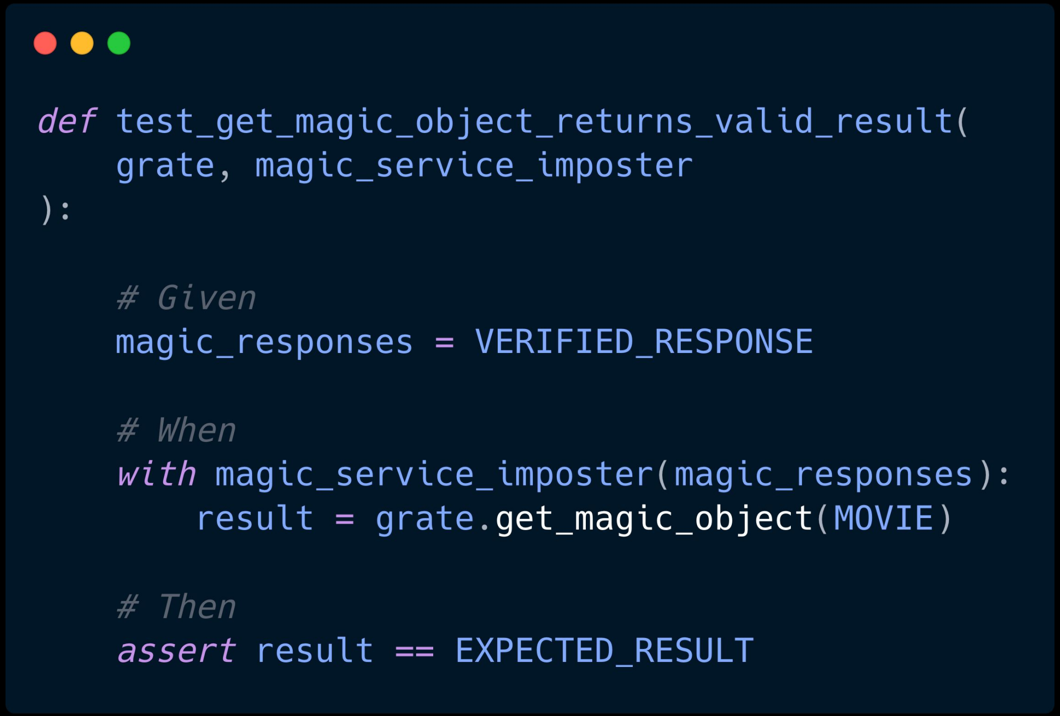
aka a fake API generator with verification for I/O

The utility



```python
from cool_utilities import ServiceImposter
import pytest

@pytest.fixture
def service_imposter():
    def imposter(config, responses):
        return ServiceImposter(config)
    return imposter


@pytest.fixture
def magic_service_imposter(service_imposter):
    def imposter(responses):
        return service_imposter(magic_config, responses)

    return imposter
```

The test



```python
def test_get_magic_object_returns_valid_result(
    grate, magic_service_imposter
):

    # Given
    magic_responses = VERIFIED_RESPONSE

    # When
    with magic_service_imposter(magic_responses):
        result = grate.get_magic_object(MOVIE)

    # Then
    assert result == EXPECTED_RESULT
```

It should just work.

Someone who regretted saying that

Photo by Alice Donovan Rouse on Unsplash

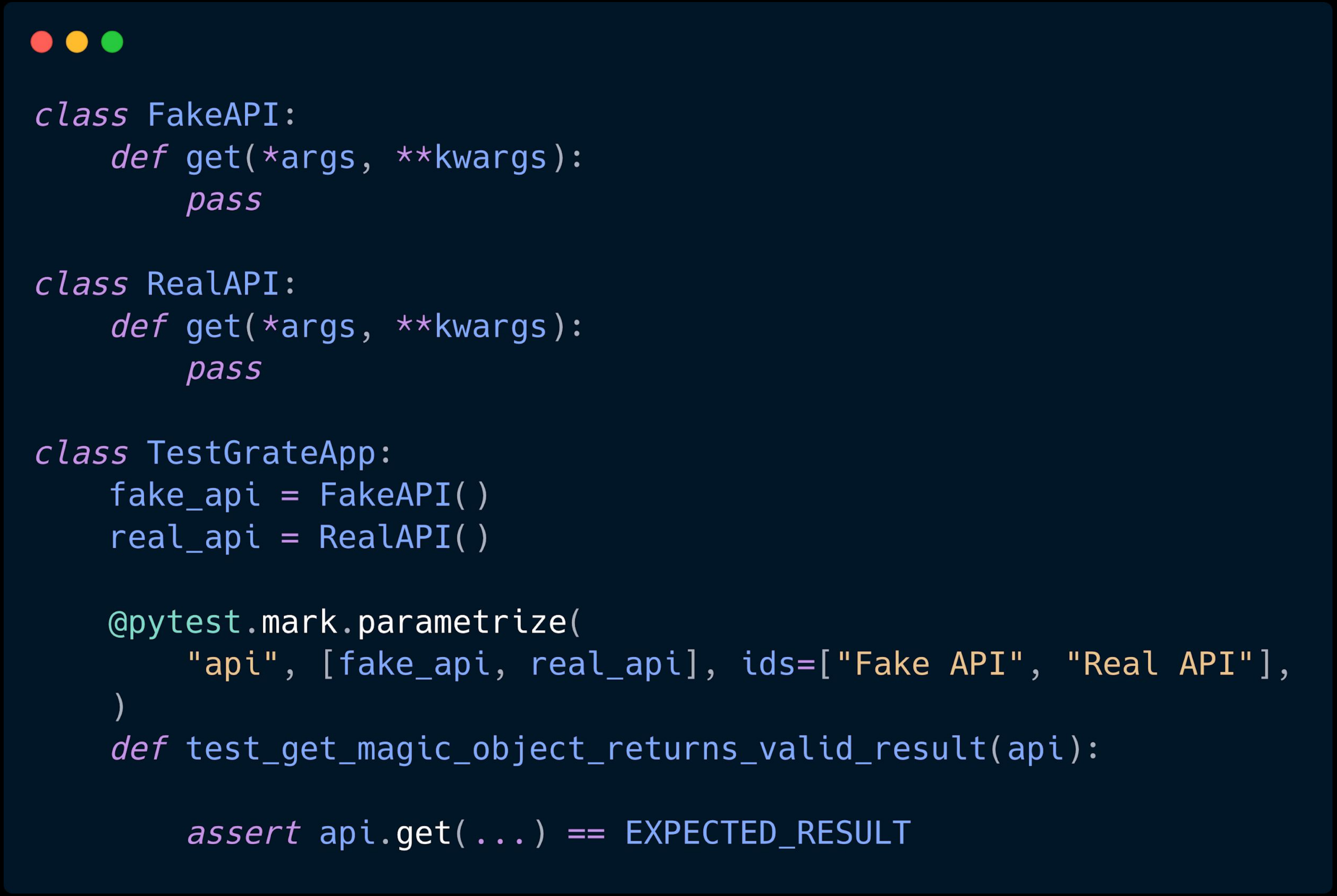
Plug and play

the different interfaces (fake and real)

Photo by Juan Di Nella on Unsplash

Plug and play

the different interfaces (fake and real)



```python
class FakeAPI:
    def get(*args, **kwargs):
        pass

class RealAPI:
    def get(*args, **kwargs):
        pass

class TestGrateApp:
    fake_api = FakeAPI()
    real_api = RealAPI()

    @pytest.mark.parametrize(
        "api", [fake_api, real_api], ids=["Fake API", "Real API"],
    )
    def test_get_magic_object_returns_valid_result(api):

        assert api.get(...) == EXPECTED_RESULT
```

What makes a good software design?

- functionable
- performant
- robust
- testable
- abstract
- extensible
- ...

What makes a good software design?

- functionable
- performant
- robust
- testable
- abstract
- extensible
- ...

Testability is a good enough reason to affect your design decisions

Thank you for listening!

Reach out to me at:
@olgamatoula

Also...

TechAtBloomberg.com

Bloomberg Engineering

We are hiring!

http://www.bloomberg.com/careers

Reach out to me at:

@olgamatoula

TechAtBloomberg.com

Bloomberg Engineering