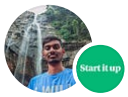
Member-only story

Python Testing With a Mock Database



Minul Lamahewage · Follow
Published in The Startup
4 min read · Jul 19, 2020

Listen Share More

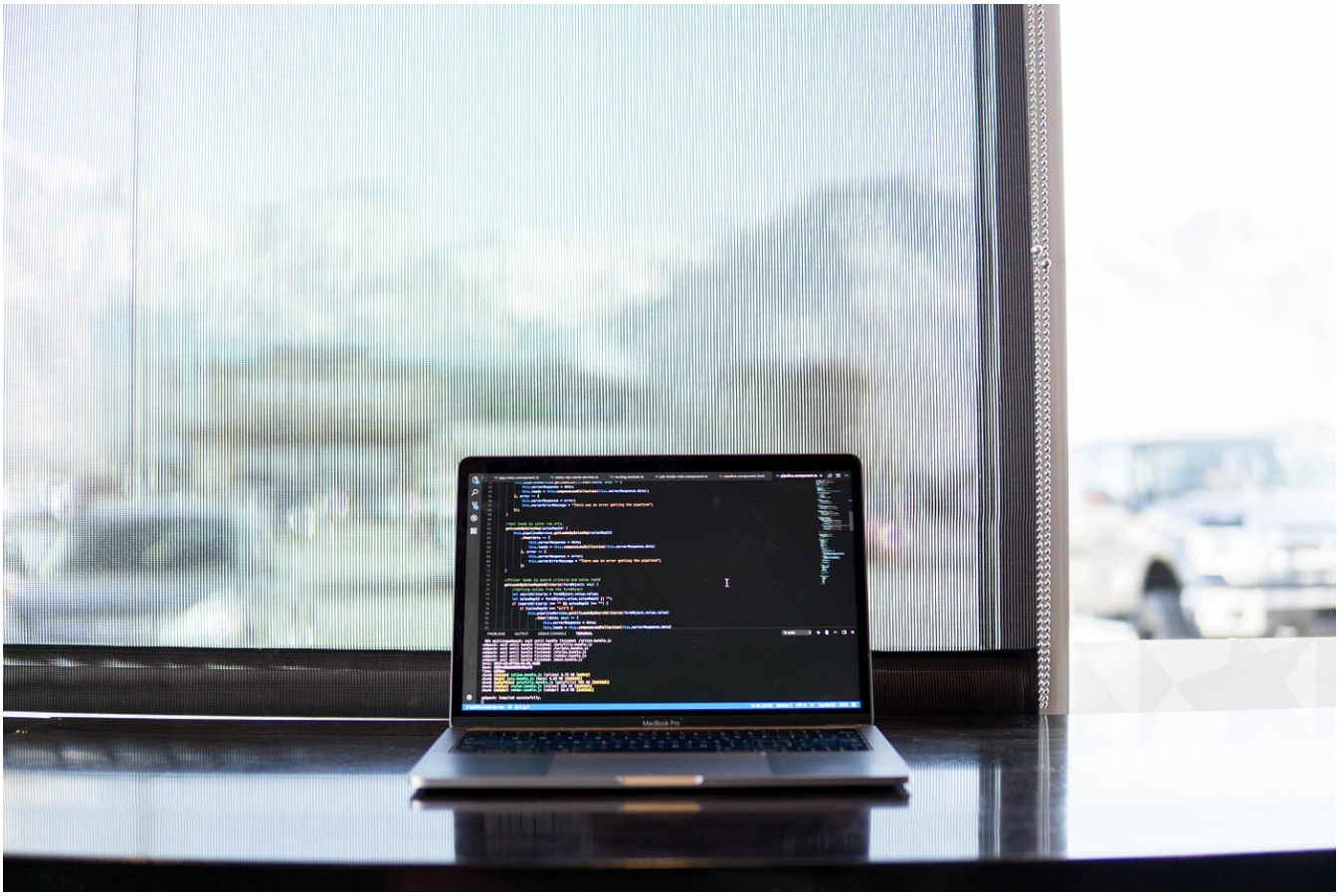


Photo by Maxwell Nelson on Unsplash

There are plenty of tutorials on the internet on using unittest but something I couldn't find while doing a project was a tutorial on how to mock a database for testing.

I believe a similar method can be used for pytest as well.

When doing continual testing as the software is developed or improved upon, testing has to be done to ensure expected functionality.
There may be methods or functions that can alter the data in the database. When testing these functions, it's best to use a separate database. It's most definitely not recommended to use the production database while testing.

When testing is automated it's not possible to manually change the database that each function is using. So, it's necessary to patch in the test database to the production database. For that, we use the patch function available in the mock package. This is available in the Python standard library, available as unittest.mock, but for this tutorial, we'll be using the mock package.

Requirements

You need the unittest package, patch from mock, and a mysql connector.

Setting up

I have two central functions that directly connect to the database, so all the patching will be done to these functions.

```
1  import mysql.connector
2  from mysql.connector import errorcode
3
4  config = {
5      'host': MYSQL_HOST,
6      'user': MYSQL_USER,
7      'password': MYSQL_PASSWORD,
8      'database': MYSQL_DB
9  }
10
11 def db_read(query, params=None):
12     try:
13         cnx = mysql.connector.connect(**config)
14         cursor = cnx.cursor(dictionary=True)
15         if params:
16             cursor.execute(query, params)
17         else:
18             cursor.execute(query)
19
```

```
        entries = cursor.fetchall()
        cursor.close()
        cnx.close()

        content = []

        for entry in entries:
            content.append(entry)

        return content

    except mysql.connector.Error as err:
        if err.errno == errorcode.ER_ACCESS_DENIED_ERROR:
            print("User authorization error")
        elif err.errno == errorcode.ER_BAD_DB_ERROR:
            print("Database doesn't exist")
        else:
            print(err)
    else:
        cnx.close()
    finally:
        if cnx.is_connected():
            cursor.close()
            cnx.close()
            print("Connection closed")

def db_write(query, params=None):
    try:
        cnx = mysql.connector.connect(**config)
        cursor = cnx.cursor(dictionary=True)
        try:
            cursor.execute(query, params)
            cnx.commit()
            cursor.close()
            cnx.close()
            return True

        except MySQLdb._exceptions.IntegrityError:
            cursor.close()
            cnx.close()
            return False

    except mysql.connector.Error as err:
        if err.errno == errorcode.ER_ACCESS_DENIED_ERROR:
            print("User authorization error")
        elif err.errno == errorcode.ER_BAD_DB_ERROR:
            print("Database doesn't exist")
```

```
67          else:
68              print(err)
69          return False
70      else:
71          cnx.close()
72          return False
73      finally:
74          if cnx.is_connected():
75              cursor.close()
76              cnx.close()
77              print("Connection closed")
```

The above code is part of the production code. Therefore the config information in it will be of the production database.

During testing, we will be patching the config variable with the details of the test database.

So, with that out of way, let's look at how we can set up the mock database.

Creating the MockDB class

For this, you need to have a MySQL server running on the system you wish to run the test on.
Every time a test suite is run, a temporary database is created and once the tests are completed, it is dropped.

First the imports

```
import mysql.connector
from mysql.connector import errorcode
from unittest import TestCase
from mock import patch
import utils
```

Here utils is the code given above named as utils.py

Since you'll probably need to use the same test database for several test cases, it is easier to create a superclass. This superclass itself needs to be a subclass of the unittest.TestCase class.

```
class MockDB(TestCase):
```

Through this, MockDB class inherits four methods
1. SetUpClass()
2. SetUp()
3.TearDownClass()
4.TearDown()

method runs after every test within that test case. These are instance methods.

SetUpClass() and TearDownClass() are class methods. They run once for a single test case. That is, SetUpClass() will run before all the tests in the relevant test case and TearDownClass() will run after all the tests are done.

You can choose to use either of these two methods depending on the requirement. If you need the database to set up for each test then the first method is best suited for you. If you can manage with the database being set up only once then the second method is better suited. One thing to note is that the first method will result in slower tests. Depending on the number of tests you have, it will continue to become slower. The second method in comparison will be significantly faster. For this tutorial, I'll be using the second method.

Creating the SetUpClass method

```
@classmethod
def setUpClass(cls):
  cnx = mysql.connector.connect(
      host=MYSQL_HOST,
      user=MYSQL_USER,
      password=MYSQL_PASSWORD,
      port = MYSQL_PORT
  )
  cursor = cnx.cursor(dictionary=True)
```

setup1.py hosted with ❤ by GitHub view raw

First, we define our database connection. Then we check if the database is already created and drop it if it has been.

```
# drop database if it already exists
  try:
      cursor.execute("DROP DATABASE {}".format(MYSQL_DB))
      cursor.close()
      print("DB dropped")
  except mysql.connector.Error as err:
      print("{}{}".format(MYSQL_DB, err))
```

setup2.py hosted with ❤ by GitHub view raw

Afterwards, we create the database, create the necessary tables and insert data to those tables.

```
cursor = cnx.cursor(dictionary=True)
#create database
  try:
      cursor.execute(
          "CREATE DATABASE {} DEFAULT CHARACTER SET 'utf8'".format(MYSQL_DB))
  except mysql.connector.Error as err:
      print("Failed creating database: {}".format(err))
      exit(1)
  cnx.database = MYSQL_DB

#create table

  query = """CREATE TABLE `test_table` (
            `id` varchar(30) NOT NULL PRIMARY KEY ,
            `text` text NOT NULL,
            `int` int NOT NULL
          )"""
  try:
      cursor.execute(query)
      cnx.commit()
  except mysql.connector.Error as err:
      if err.errno == errorcode.ER_TABLE_EXISTS_ERROR:
          print("test_table already exists.")
      else:
          print(err.msg)
  else:
      print("OK")

#insert data

```

```
    insert_data_query = """INSERT INTO `test_table` (`id`, `text`, `int`) VALUES
                        ('1', 'test_text', 1),
                        ('2', 'test_text_2',2)"""
    try:
        cursor.execute(insert_data_query)
        cnx.commit()
    except mysql.connector.Error as err:
        print("Data insertion to test_table failed \n" + err)
    cursor.close()
    cnx.close()
```

setup3.py hosted with ❤ by GitHub view raw

Patching config

Once setting up the test database is done, we need to create a config variable with the test database information to patch the production database.

```
testconfig ={
        'host': MYSQL_HOST,
        'user': MYSQL_USER,
        'password': MYSQL_PASSWORD,
        'database': MYSQL_DB
    }
    cls.mock_db_config = patch.dict(utils.config, testconfig)
```

setup4.py hosted with ❤ by GitHub view raw

testconfig holds the details of the test database. mock_db_config is used to patch the config variable in the utils file with testconfig. Since these are dictionaries, patch.dict is used.

Creating the TearDownClass method

Next, we need the tearDownClass method. All this needs to do is drop the test database.

```
@classmethod
def tearDownClass(cls):
    cnx = mysql.connector.connect(
        host=MYSQL_HOST,
        user=MYSQL_USER,
        password=MYSQL_PASSWORD
    )
    cursor = cnx.cursor(dictionary=True)

    # drop test database
```

```
    try:
        cursor.execute("DROP DATABASE {}".format(MYSQL_DB))
        cnx.commit()
        cursor.close()
    except mysql.connector.Error as err:
        print("Database {} does not exists. Dropping db failed".format(MYSQL_DB))
    cnx.close()
```

teardown.py hosted with ❤ by GitHub view raw

MockDB Class

Here is the entire MockDB class.

```
from unittest import TestCase
import mysql.connector
from mysql.connector import errorcode
from mock import patch
import utils


MYSQL_USER = "root"
MYSQL_PASSWORD = ""
MYSQL_DB = "testdb"
MYSQL_HOST = "localhost"
MYSQL_PORT = "3306"


class MockDB(TestCase):

    @classmethod
    def setUpClass(cls):
        cnx = mysql.connector.connect(
            host=MYSQL_HOST,
            user=MYSQL_USER,
            password=MYSQL_PASSWORD,
            port = MYSQL_PORT
        )
        cursor = cnx.cursor(dictionary=True)

        # drop database if it already exists
        try:
            cursor.execute("DROP DATABASE {}".format(MYSQL_DB))
            cursor.close()
            print("DB dropped")
        except mysql.connector.Error as err:
            print("{}{}".format(MYSQL_DB, err))

```

```
        cursor = cnx.cursor(dictionary=True)
        try:
            cursor.execute(
                "CREATE DATABASE {} DEFAULT CHARACTER SET 'utf8'".format(MYSQL_DB))
        except mysql.connector.Error as err:
            print("Failed creating database: {}".format(err))
            exit(1)
        cnx.database = MYSQL_DB

        query = """CREATE TABLE `test_table` (
                  `id` varchar(30) NOT NULL PRIMARY KEY ,
                  `text` text NOT NULL,
                  `int` int NOT NULL
                )"""
        try:
            cursor.execute(query)
            cnx.commit()
        except mysql.connector.Error as err:
            if err.errno == errorcode.ER_TABLE_EXISTS_ERROR:
                print("test_table already exists.")
            else:
                print(err.msg)
        else:
            print("OK")

        insert_data_query = """INSERT INTO `test_table` (`id`, `text`, `int`) VALUES
                            ('1', 'test_text', 1),
                            ('2', 'test_text_2',2)"""
        try:
            cursor.execute(insert_data_query)
            cnx.commit()
        except mysql.connector.Error as err:
            print("Data insertion to test_table failed \n" + err)
        cursor.close()
        cnx.close()

        testconfig ={
            'host': MYSQL_HOST,
            'user': MYSQL_USER,
            'password': MYSQL_PASSWORD,
            'database': MYSQL_DB
        }
        cls.mock_db_config = patch.dict(utils.config, testconfig)

    @classmethod
    def tearDownClass(cls):
        cnx = mysql.connector.connect(
            host=MYSQL_HOST,
```

```
            user=MYSQL_USER,
            password=MYSQL_PASSWORD
        )
        cursor = cnx.cursor(dictionary=True)

        # drop test database
        try:
            cursor.execute("DROP DATABASE {}".format(MYSQL_DB))
            cnx.commit()
            cursor.close()
        except mysql.connector.Error as err:
            print("Database {} does not exists. Dropping db failed".format(MYSQL_DB))
        cnx.close()


```

Once MockDB class is done, we can inherit this class to create test cases.

Testing

We'll write tests to test the functions in utils.py.

We need to import the file we are planning to test, the MockDB class and patch from import

```
import utils
from mock_db import MockDB
from mock import patch
```

When using the patch for the config variable, the following has to be used,

```
with self.mock_db_config:
```

Any code that is executed within this, will use the patched config variables.

Here is the completed test_utils code with some sample tests

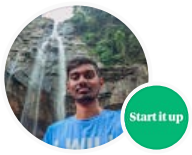
```
from mock_db import MockDB
from mock import patch
import utils

class TestUtils(MockDB):
```

```
5  class TestUtils(MockDB):
6
7      def test_db_write(self):
8          with self.mock_db_config:
9              self.assertEqual(utils.db_write("""INSERT INTO `test_table` (`id`, `text`,
10                                 ('3', 'test_text_3', 3)"""), True)
11             self.assertEqual(utils.db_write("""INSERT INTO `test_table` (`id`, `text`,
12                                 ('1', 'test_text_3', 3)"""), False)
13             self.assertEqual(utils.db_write("""DELETE FROM `test_table` WHERE id='1' "'
14             self.assertEqual(utils.db_write("""DELETE FROM `test_table` WHERE id='4' "'
```

test_utils.py hosted with ♥ by GitHub view raw

I hope this provided you with a basic understanding of how to use a mock database for testing with unittest. This tutorial is aimed at providing users with a foundation on mocking a database for testing. Your implementation of this will depend on factors such as the use of an ORM for example.

Unittest Database Mock Testing Python



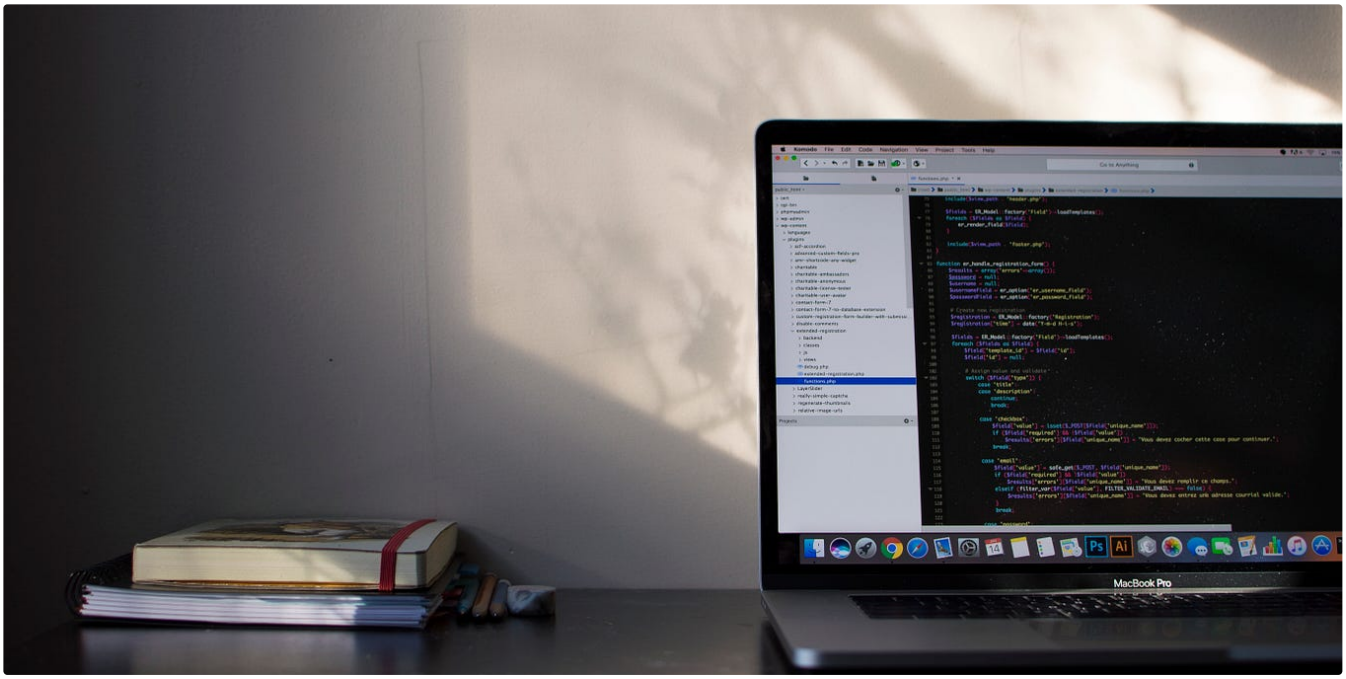
Follow

Written by Minul Lamahewage

10 Followers · Writer for The Startup

3rd year Computer Science and Engineering undergraduate at University of Moratuwa, Sri Lanka

More from Minul Lamahewage and The Startup



Minul Lamahewage in Arimac

Refactoring Vue.js code

Refactoring code is never fun but it can be less of a pain. Following general methods for various problems you may face while trying to...

4 min read · Nov 13, 2020

37 1

Tim Denning in The Startup

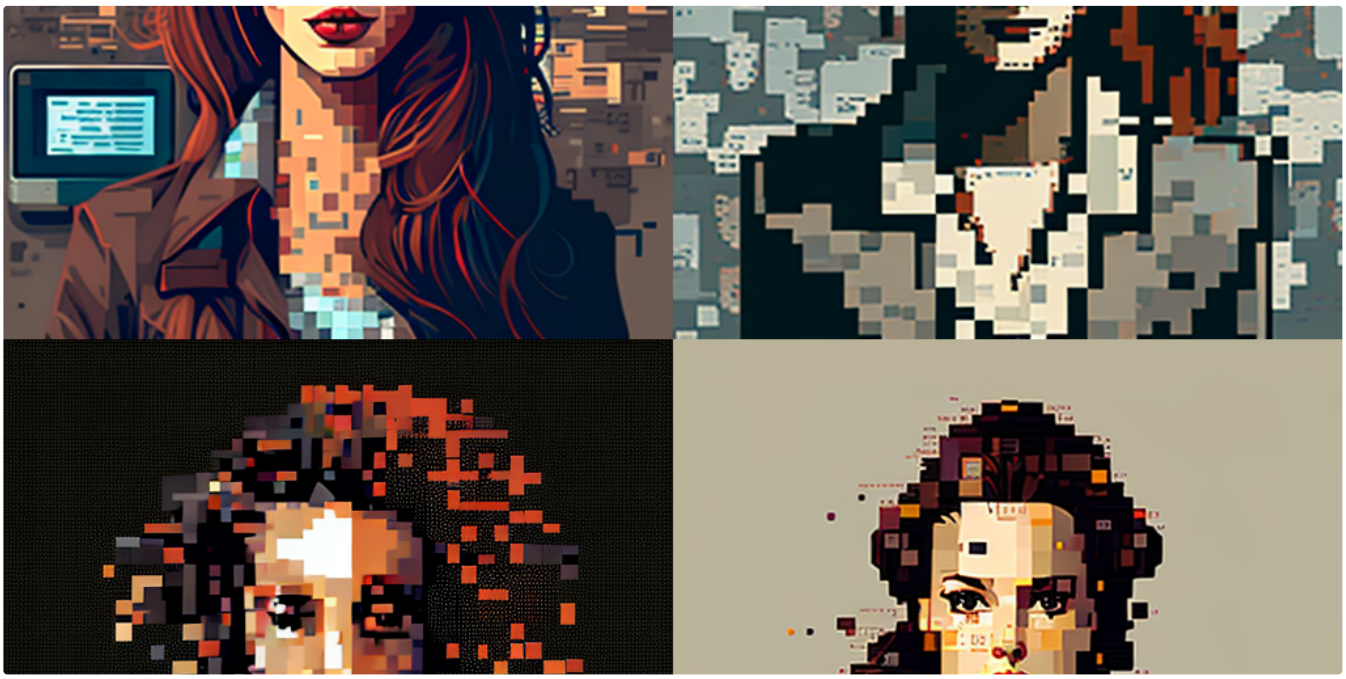
I've Interviewed 100s of Wealthy People — Here Are 9 Money Lessons They

Taught Me

I wish they taught these in school

· 5 min read · Jun 23

3.6K 78



Zulie Rane in The Startup

If You Want to Be a Creator, Delete All (But Two) Social Media Platforms

In October 2022, during the whole Elon Musk debacle, I finally deleted Twitter from my phone. Around the same time, I also logged out of...

· 8 min read · Apr 19

34K 738

Nitin Sharma in The Startup

Goodbye ChatGPT And Bard: Here Are (Newly Released) AI Tools That Will Blow Your Mind

I bet you don't know any of these tools.

· 8 min read · Jun 14

pandaquests in Level Up Coding

How to Unit Test in ReactJS

Unit testing is an important aspect of software development that helps ensure that individual units of code, such as functions and...

✦ · 8 min read · Jan 15

50

Renato Boemer in Towards Data Science

Ensure Model Reliability with Unit Testing

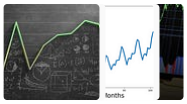
Catch errors early on in the development process

· 4 min read · Jan 25

165 4

Lists

Coding & Development
11 stories · 47 saves



Predictive Modeling w/ Python
18 stories · 110 saves

ChatGPT
21 stories · 43 saves

Practical Guides to Machine Learning
10 stories · 123 saves

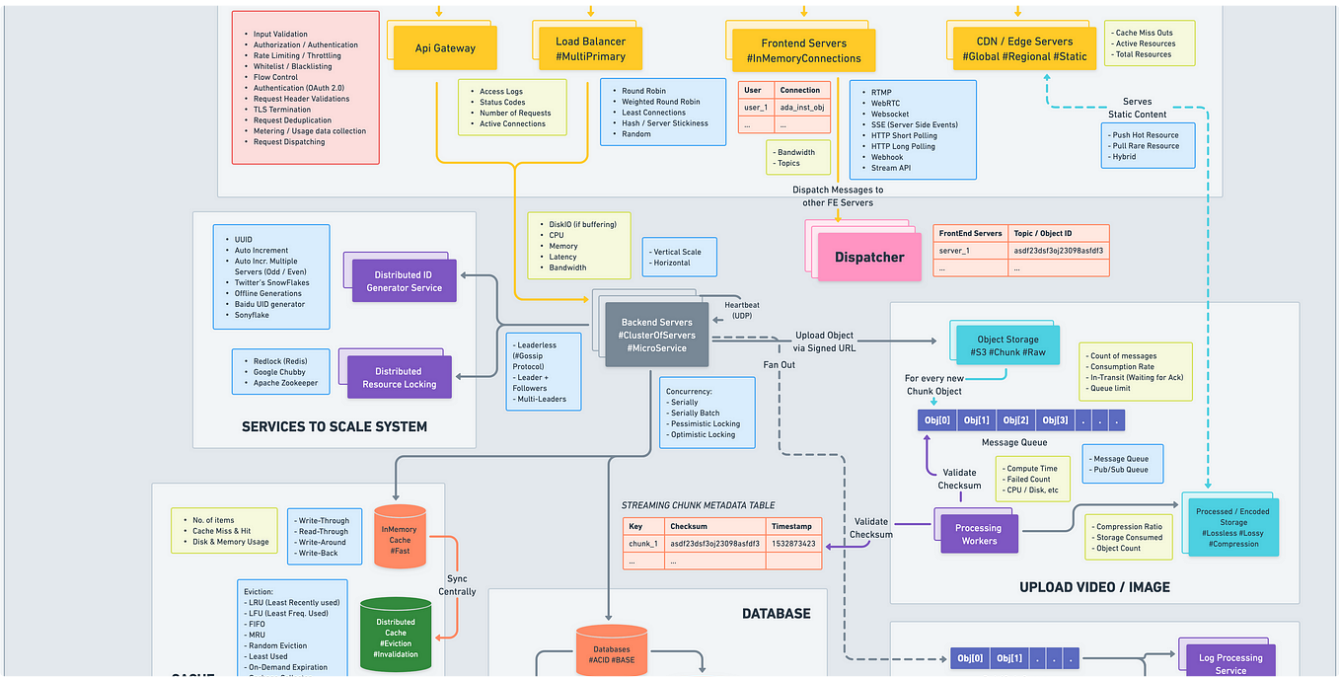
Xiaoxu Gao in Towards Data Science

From Novice to Expert: How to Write a Configuration file in Python

Treat config file like your production code

· 9 min read · Dec 28, 2020

1.3K 14



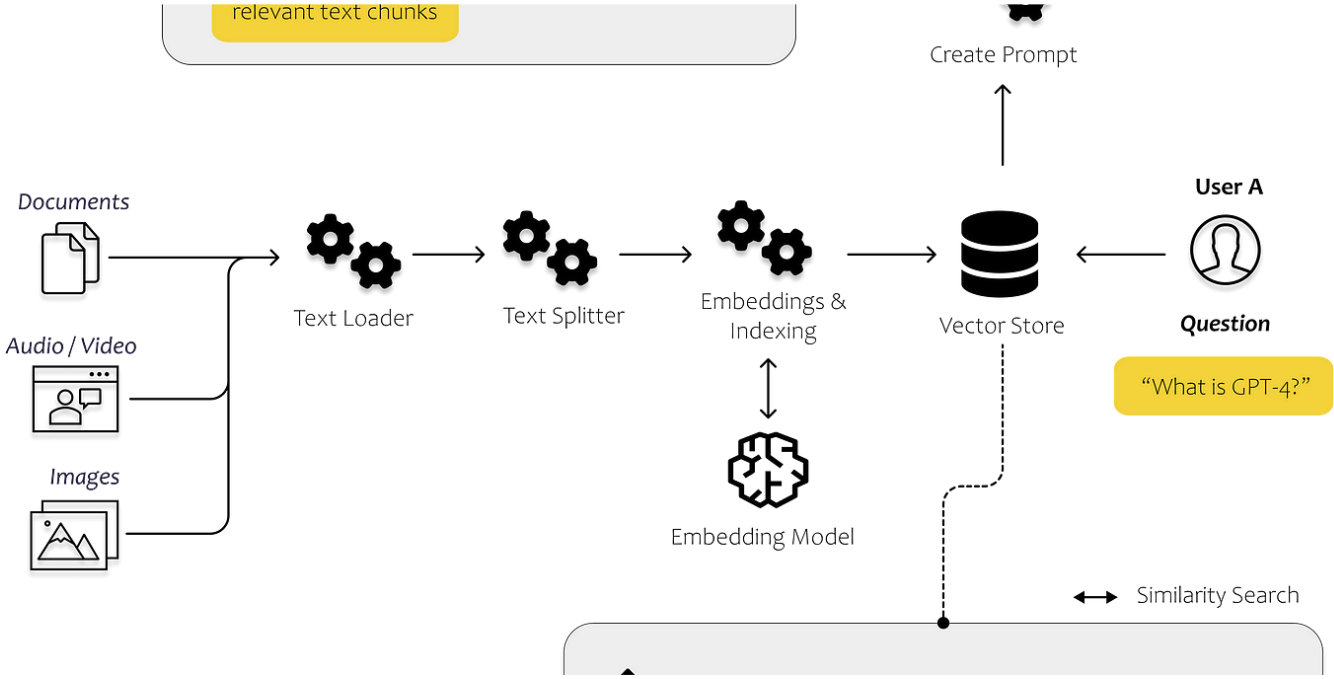
Love Sharma in ByteByteGo System Design Alliance

System Design Blueprint: The Ultimate Guide

Developing a robust, scalable, and efficient system can be daunting. However, understanding the key concepts and components can make the...

· 9 min read · Apr 20

5.9K 49



Dominik Polzer in Towards Data Science

Tomer Gabay in Towards Data Science

5 Python Tricks That Distinguish Senior Developers From Juniors

Illustrated through differences in approaches to Advent of Code puzzles

✦ · 6 min read · Jan 16

816 20