# Building Neo4j Applications with Python course notes and links-

# Disclaimer for repos and code-

Course Repo: <https://github.com/neo4j-graphacademy/app-python>

Forked Repo: <https://github.com/amaan784/neo4j-python-app-course/>

Go over the README of the forked repo for instructions and details. The repo all the code for all challenges in the main branch.

# Database Details-

Look at the separate recommendations dataset document for detailed information about the database

OR

Create a sandbox instance by going inside the course and view it

# Module 1: Project Introduction

As part of this course, you will work on a [**pre-built repository**](https://github.com/neo4j-graphacademy/app-python) for the fictional client **Neoflix**. The course is designed to be framework agnostic, so although we have chosen a specific framework, the tasks that you will perform will be the same regardless of your choice of framework.

In the early stages you will learn some of the theory required and then use that knowledge to implement a set of features in the API.

## **Repository Information**

We have built a repository that takes care of the boiler plate, so you can focus on implementing the functionality.

* The project is designed to work with Python version **3.10**
* Packages can be installed with pip
* A web server has been built with [**Flask**](https://flask.palletsprojects.com/en/2.0.x/)
  + Authentication is handled with [**JWT Tokens**](https://jwt.io/) and [**Flask-JWT-Extended**](https://flask-jwt-extended.readthedocs.io/en/stable/)
  + Passwords are encrypted and verified with **[bcrypt](https://github.com/pyca/bcrypt/" \t "_blank)**
  + Testing is performed using **[pytest](https://pytest.org/" \t "_blank)**

## **Get Started**

In the next lesson, we will explore the project in Gitpod and and introduce you to Neo4j Sandbox.

# Exploring the Project

The challenges in this course use a service called **[GitPod](https://gitpod.io/" \t "_blank)** to issue code challenges. We have chosen Gitpod because it means that you can complete the course directly in your browser without having to download or install any additional software.

To use GitPod you must sign in with your GitLab, GitHub or Bitbucket account.

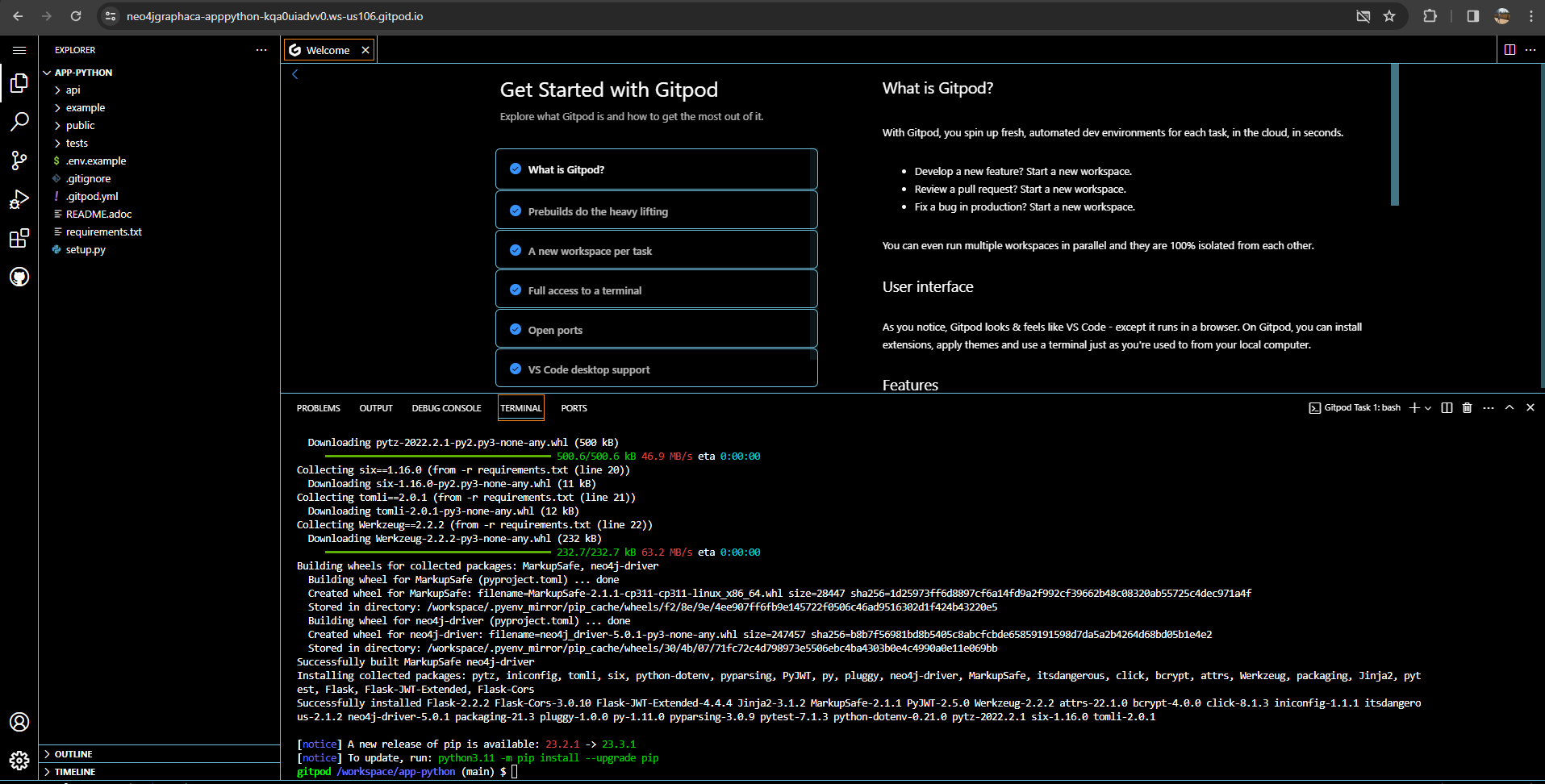
## Explore the Repository

You can open the repository by clicking the link below:

[**Explore Repository in GitPod →**](https://graphacademy.neo4j.com/courses/app-python/0-setup/1-setup/lab/)

Here are some of the important directories in the project:

* example/ - Example code for working with the driver.
* api/ - The application code:
  + dao/ - Data Access Objects which will be modified to communicate with Neo4j
  + middleware/ - Some custom middleware functions that are used by Flask throughout the request lifecycle
  + routes/ - Route handlers that are registered on the server. You shouldn’t need to edit these files.
* public/ - Minified build files for the SPA. **Do not edit these files**.



## Course Challenges

For each challenge, you will be provided with a link to re-open the correct file within the repository. Clicking these links will also set the credentials load environment variables for your [**Neo4j Sandbox**](https://sandbox.neo4j.com/) instance.

## Your Sandbox Instance

As part of this course, a Neo4j Sandbox instance has been created for you with the recommendations dataset. This dataset contains all of the details required to populate the Neoflix website.

## What is Neo4j Sandbox?

Neo4j Sandbox is a free service that allows you to create pre-populated Neo4j instances completely free of charge. Neo4j Sandbox is the perfect environment for experimenting with Neo4j.

You can log into Neo4j Sandbox and create a database with a number of pre-populated datasets by visiting [**sandbox.neo4j.com**](https://sandbox.neo4j.com/).

## Your Sandbox Credentials

**Browser URL**

[**https://582a0da200f97b94876b0f224b1bb9e4.neo4jsandbox.com/browser/**](https://582a0da200f97b94876b0f224b1bb9e4.neo4jsandbox.com/browser/)

**Bolt URI**

bolt://35.174.165.196:7687

**Websocket Bolt URI**

bolt+s://582a0da200f97b94876b0f224b1bb9e4.neo4jsandbox.com:7687

**Username**

neo4j

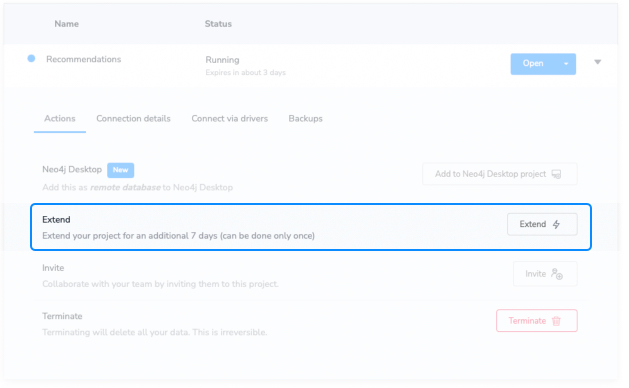
**Password**

dolly-sunrise-cracks

You can open a Neo4j Browser window throughout this course by clicking the [**Toggle Sandbox**](https://graphacademy.neo4j.com/courses/app-python/0-setup/1-setup/) button in the bottom right hand corner of the screen.

**EXTENDING YOUR SANDBOX INSTANCE**

By default, a Neo4j sandbox instance exists for 3 days. You can extend it for another 7 days by going to the [**sandbox site**](https://sandbox.neo4j.com/) and extending it in the details (right-most down arrow) for the recommendations sandbox.



### Accessing Your Sandbox Credentials

Your sandbox credentials can be accessed within the project through the os.getenv() function.

**python**

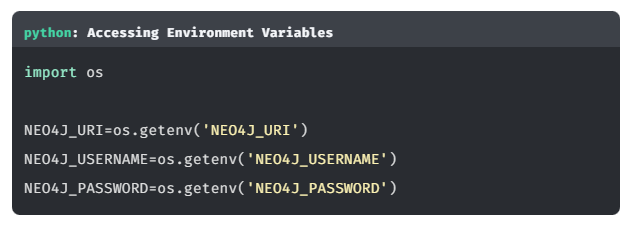
**Accessing Environment Variables**

import os

NEO4J\_URI=os.getenv('NEO4J\_URI')

NEO4J\_USERNAME=os.getenv('NEO4J\_USERNAME')

NEO4J\_PASSWORD=os.getenv('NEO4J\_PASSWORD')



**RUNNING THE PROJECT LOCALLY**

You can also run this project locally by [**cloning the repository**](https://github.com/neo4j-graphacademy/app-python). To configure your Neo4j Sandbox connection, rename the .env.example file in the root folder to .env and replace the NEO4J\_ variables with the values below.

**env**

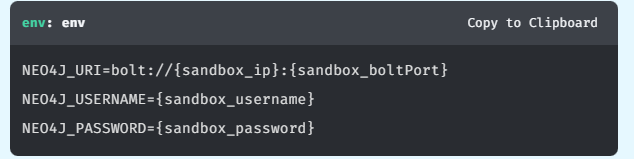
**env**

Copy to Clipboard

NEO4J\_URI=bolt://{sandbox\_ip}:{sandbox\_boltPort}

NEO4J\_USERNAME={sandbox\_username}

NEO4J\_PASSWORD={sandbox\_pas



# Module 2: The Neo4j Python Driver

In this module you will learn about the lifecycle of the Neo4j Driver and how it should be used within your application. By the end of this module, you will have learned to how to:

* Install the Neo4j Python Driver.
* Build a Connection String.
* Create an instance of the Driver.
* Verify that the Driver has successfully connected to Neo4j.

# About the Driver

In the [**Cypher Fundamentals**](https://graphacademy.neo4j.com/courses/cypher-fundamentals/) course, we cover how to query Neo4j using a language called Cypher. To execute a Cypher statement against a Neo4j database you will use an object called a **Driver**.

*The****Driver****object is a thread-safe, application-wide fixture from which all Neo4j interaction derives.*

*The Driver API is****topology independent****, so you can run the same code against a****Neo4j cluster****or a****single DBMS****.*

To connect to and query Neo4j from within a Python application, you use the [**Neo4j Python Driver**](https://neo4j.com/developer/python).

The Neo4j Python Driver is one of five officially supported drivers, the others are Java, JavaScript, .NET, and Go. There are also a wide range of Community Drivers available for other languages including PHP and Ruby.

You should create a **single instance** of the Driver in your application per Neo4j cluster or DBMS, which can then be shared across your application.

## **Installing the Driver**

The Neo4j Python Driver is available through the pip management system and can be installed with the pip command.

**shell**

Copy to Clipboard

pip install neo4j



## **Creating a Driver Instance**

Each driver instance will connect to one DBMS, or Neo4j cluster, depending on the value provided in the connection string.

The neo4j package exports a GraphDatabase object. This object provides a driver() function for creating a new driver instance.

The driver() function requires one mandatory parameter, a **connection string** for the Neo4j cluster or DBMS - for example neo4j://localhost:7687 or neo4j+s://dbhash.databases.neo4j.io:7687.

Additionally, you will also pass a named parameter auth to represent the Neo4j user credentials. You can provide basic username/password authentication by passing the username and password as a tuple.

Here is an example for how to create a driver instance to connect to a Neo4j instance running on localhost on port 7687 with the username neo4j and password neo:

**python**

**Creating a Driver Instance**

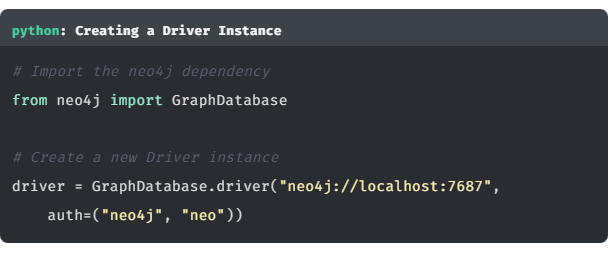
*# Import the neo4j dependency*

from neo4j import GraphDatabase

*# Create a new Driver instance*

driver = GraphDatabase.driver("neo4j://localhost:7687",

auth=("neo4j", "neo"))



The above example creates an unencrypted connection to the Neo4j server at localhost on the default port number of 7687. The driver then attemps to authenticate against the server using a basic authentication with the username neo4j and password neo.

### **Verifying Connectivity**

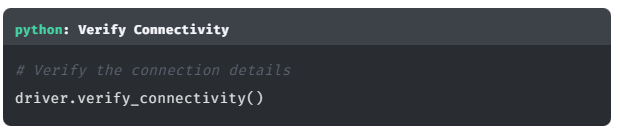
You can verify that the connection details used during driver instantiation are correct by calling the verifyConnectivity() function. This function will raise a Neo4jException with a code property of Neo.ClientError.Security.Unauthorized if a connection could not be made.

**python**

**Verify Connectivity**

*# Verify the connection details*

driver.verify\_connectivity()



# Connection Strings and Authentication

In the previous lesson, you saw some example code for creating a new driver instance. Let’s take a closer look at the driver() function and how it is used to create a driver instance.

**python**

**Creating a Driver Instance**

Copy to Clipboard

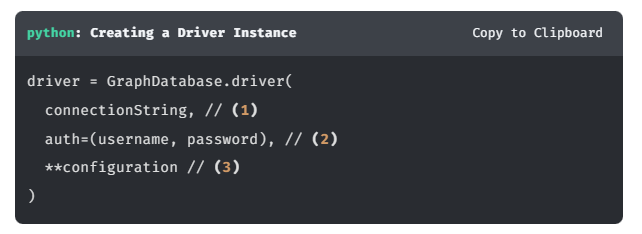
driver = GraphDatabase.driver(

connectionString, // **(1)**

auth=(username, password), // **(2)**

\*\*configuration // **(3)**

)



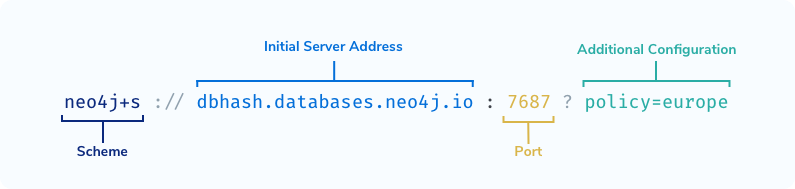
The neo4j.driver() function accepts the following arguments:

1. A connection string
2. An authentication method to use when connecting - in this case, username/password authentication
3. Optionally, you can provide additional configuration as named parameters

Let’s take a look at these points in more detail.

## **1** The Connection String

A connection string typically consists of four elements:



1. The **scheme** used to connect to the Neo4j instance - for example neo4j or neo4j+s (required)
2. The initial server address for the Neo4j DBMS - for example localhost or dbhash.databases.neo4j.io (required)
3. The **port number** that the DBMS is running on (required if the instance is not running on the default port of 7687)
4. Additional connection configuration (for example the routing context)

### Choosing your Scheme

Most likely, you will use a variation of the neo4j scheme within your connection string.

* neo4j - Creates an **unencrypted** connection to the DBMS. If you are connecting to a local DBMS or have not explicitly turned on encryption then this is most likely the option you are looking for.
* neo4j+s - Creates an **encrypted** connection to the DBMS. The driver will verify the authenticity of the certificate and fail to verify connectivity if there is a problem with the certificate.
* neo4j+ssc - Creates an encrypted connection to the DBMS, but will not attempt to verify the authenticity of the certificate.

Variations of the bolt scheme can be used to connect directly to a single DBMS (within a clustered environment or standalone). This can be useful if you have a single server configured for data science or analytics.

* bolt - Creates an unencrypted connection directly to a single DBMS.
* bolt+s - Creates an encrypted connection directly to a single DBMS and verify the certificate.
* bolt+ssc - Creates an encrypted connection to directly to a single DBMS but will not attempt to verify the authenticity of the certificate.

**WHICH SCHEME STRING IS RIGHT FOR YOU?**

You can [**verify the encryption level of your DBMS**](https://neo4j.com/docs/migration-guide/current/upgrade-driver/#_configure_ssl_policy_for_bolt_server_and_https_server) by checking the dbms.connector.bolt.enabled key in neo4j.conf.

If you are connecting to a DBMS hosted on Neo4j Aura, you will always use the neo4j+s scheme.

### Additional Connection Information

Additional connection information can be appended to the connection string after a ?. For example, in a multi-data centre cluster, you may wish to take advantage of locality to reduce latency and improve performance.

In this case, you could configure a routing policy for a set of application servers to connect to Neo4j instances that are within the same data centre.

For more information, see this [**Knowledge Base Article: Consideration about routing tables on multi-data center deployments**](https://neo4j.com/developer/kb/consideration-about-routing-tables-on-multi-data-center-deployments/).

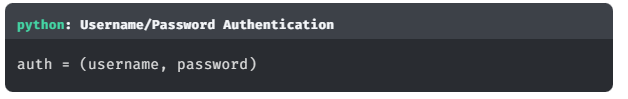
## **2** An Authentication Token

In most cases, you will connect to the DBMS using basic authentication consisting of a username and password. You can pass the username and password as a tuple to the auth parameter.

**python**

**Username/Password Authentication**

auth = (username, password)



For more information on additional authentication methods, see the [**Authentication and authorization**](https://neo4j.com/docs/operations-manual/current/authentication-authorization/) section of the [**Neo4j Operations Manual**](https://neo4j.com/docs/operations-manual/current/).

## **3** Additional Driver Configuration (Optional)

The driver() function also accepts additional configuration parameters.

This object allows you to provide advanced configuration options, for example setting the connection pool size or changing timeout limits.

**python**

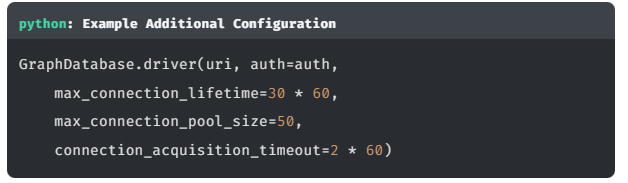
**Example Additional Configuration**

GraphDatabase.driver(uri, auth=auth,

max\_connection\_lifetime=30 \* 60,

max\_connection\_pool\_size=50,

connection\_acquisition\_timeout=2 \* 60)

****

For more information or a full list of configuration options, please [**visit the Neo4j Python Driver manual**](https://neo4j.com/docs/python-manual/current/get-started/).

## What happens next?

The driver will attempt to connect to the DBMS using the supplied credentials. If everything is successful, the driver will then communicate with the DBMS and figure out the best way to execute each query.

You do not need to do any additional configuration when connecting to a single DBMS or a Neo4j cluster. This means you do not have to adapt your application code, regardless of which environment you connect to.

Once the connection has been successfully made, your application can start to interact with the data in the graph.

# Adding the Driver (challenge)

This is your first challenge of this course. Your challenge here is to modify the code to create a new instance of the Driver that can be used across the application.

As we discussed in the [**About the Driver lesson**](https://graphacademy.neo4j.com/courses/app-python/1-driver/1-about/), it is best practice to create a single instance of the driver in our application per Neo4j cluster or DBMS.

Inside api/neo4j.py, you will see an init\_driver() function.

**python**

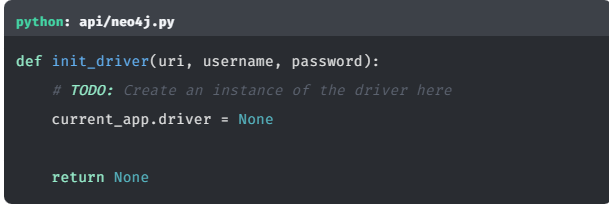
**api/neo4j.py**

def init\_driver(uri, username, password):

*# TODO: Create an instance of the driver here*

current\_app.driver = None

return None



This function should use the uri, username and password parameters supplied to create an instance of the Neo4j Python Driver. The driver instance should then be assigned to the current\_app variable to ensure that it is available throughout the application.

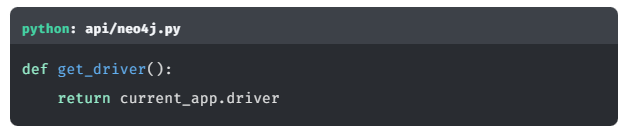
The get\_driver() function also included in neo4j.py will provide access to the driver variable set in the init\_driver() above.

**python**

**api/neo4j.py**

def get\_driver():

return current\_app.driver



## Challenge: Implement the init\_driver function.

Your first challenge is to modify the init\_driver() function in api/neo4j.py to create an instance of the driver and assign it to the driver variable declared above. To do this, we will need to:

1. Install the neo4j dependency using pip.
2. Import the GraphDatabase object from neo4j into neo4j.py.
3. Create the driver instance with the uri, username and password parameters passed to the function and use the verify\_connectivity() method to assert that the credentials are correct.

[**Open**api/neo4j.py**→**](https://graphacademy.neo4j.com/courses/app-python/1-driver/3-connecting/lab/)

### 1. Install the **neo4j** Dependency

First, you will need to install the neo4j dependency. Run the following command in your terminal session:

**sh**

**Install neo4j**

Copy to Clipboard

pip install neo4j

### 2. Importing the Dependency

To include the Driver dependency in our module, you will add an import command to the top of the file. Copy and paste the following code at the top of api/neo4j.py.

**python**

Copy to Clipboard

*# Import the neo4j dependency*

from neo4j import GraphDatabase

### 3. Creating the Driver Instance

Create the driver instance by calling the GraphDatabase.driver() function. The first argument will be the uri passed as the first parameter to the function. The function will also require a named auth argument with the tuple: (username, password)

Replace the init\_driver placeholder function with the following code:

**python**

**api/neo4j.py**

Copy to Clipboard

def init\_driver(uri, username, password):

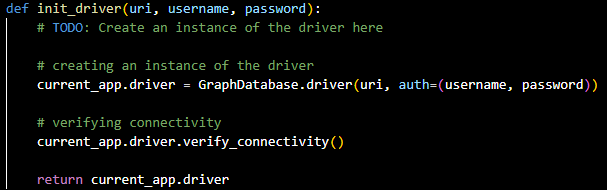
*# Create an instance of the driver*

current\_app.driver = GraphDatabase.driver(uri, auth=(username, password))

*# Verify Connectivity*

current\_app.driver.verify\_connectivity()

return current\_app.driver



The function calls the verify\_connectivity() function on the driver instance to verify that the connection details are correct before returning the newly created driver instance.

If the connection cannot be made for any reason, an Exception will be thrown. If this occurs, your application will be unable to communicate with Neo4j. Manual investigation will be required to diagnose the issue.

If the connection has been successfully verified, the function will return an instance of the driver.

## Testing

To test that this functionality has been correctly implemented, run the following code in a new terminal session:

**sh**

**Running the test**

Copy to Clipboard

pytest tests/01\_connect\_to\_neo4j\_\_test.py

The test file is located at [tests/01\_connect\_to\_neo4j\_\_test.py](https://github.com/neo4j-graphacademy/app-python/blob/main/tests/01_connect_to_neo4j__test.py).

Are you stuck? Click here for help

If you get stuck, you can see a working solution by checking out the 01-connect-to-neo4j branch by running:

**sh**

**Check out the 01-connect-to-neo4j branch**

Copy to Clipboard

git checkout 01-connect-to-neo4j

You may have to commit or stash your changes before checking out this branch. You can also [**click here to expand the Support pane**](https://graphacademy.neo4j.com/courses/app-python/1-driver/3-connecting/).

# Module 3: Interacting with Neo4j

In this module, you will learn how to use the driver within the Neoflix project.

You will learn how to:

* Open a Session and execute a Unit of Work within a Transaction.
* Execute Read and Write queries through the Driver.
* Consume the results returned from Neo4j.
* Handle potential errors thrown by the Driver.

# Sessions and Transactions

## Sessions

Through the Driver, we open **Sessions**.

*A session is a container for a sequence of transactions. Sessions borrow connections from a pool as required and are considered lightweight and disposable.*

It is important to remember that sessions are not the same as database connections. When the Driver connects to the database, it opens up multiple TCP connections that can be borrowed by the session. A query may be sent over multiple connections, and results may be received by the driver over multiple connections.

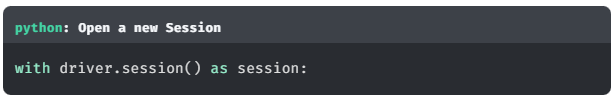
Instead, sessions should be considered a client-side abstraction for grouping units of work, which also handle the underlying connections. The connections themselves are managed internally by the driver and are not directly exposed to the application.

To open a new session, call the session() method on the driver.

**python**

**Open a new Session**

with driver.session() as session:

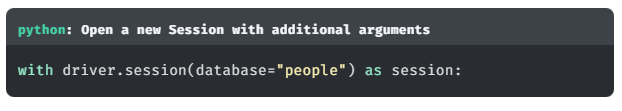


This session method takes an optional configuration argument, which can be used to set the database to run any queries against in a multi-database setup, and the default access mode for any queries run within the transaction.

**python**

**Open a new Session with additional arguments**

with driver.session(database="people") as session:



If no database is supplied, the default database will be used. This is configured in the dbms.default\_database in neo4j.conf, the default value is neo4j. You cannot create multiple databases in Neo4j Aura or in Neo4j Community Edition.

For more information on multi-database setup, see [**Managing Multiple Databases**](https://neo4j.com/developer/manage-multiple-databases/).

The default access mode is set to write, but this can be overwritten by explicitly calling the execute\_read() or execute\_write() functions.

## Transactions

Through a Session, we can run one or more **Transactions**.

*A transaction comprises a unit of work performed against a database. It is treated in a coherent and reliable way, independent of other transactions.*

**ACID TRANSACTIONS**

A transaction, by definition, must be atomic, consistent, isolated, and durable. Many developers are familiar with ACID transactions from their work with relational databases, and as such the ACID consistency model has been the norm for some time.

There are three types of transaction exposed by the driver:

* Auto-commit Transactions
* Read Transactions
* Write Transactions

### Auto-commit Transactions

Auto-commit transactions are a single unit of work that are immediately executed against the DBMS and acknowledged immediately. You can run an auto-commit transaction by calling the run() method on the session object, passing in a Cypher statement as a string and optionally an object containing a set of parameters.

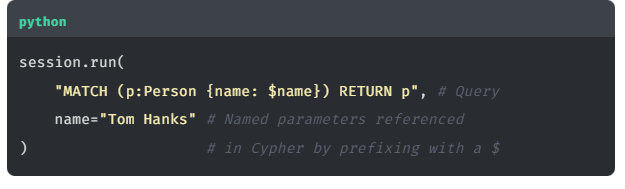
**python**

session.run(

"MATCH (p:Person {name: $name}) RETURN p", *# Query*

name="Tom Hanks" *# Named parameters referenced*

) *# in Cypher by prefixing with a $*



**FOR ONE-OFF QUERIES ONLY**

In the event that there are any transient errors when running a query, the driver will not attempt to retry a query when using session.run(). For this reason, these should only be used for one-off queries and shouldn’t be used in production.

### Read Transactions

When you intend to read data from Neo4j, you should execute a **Read** Transaction. In a clustered environment (including Neo4j Aura), read queries are distributed across the cluster.

The session provides an execute\_read() function, which expects a single parameter, a function that represents the unit of work. The first argument passed to the function will be a transaction object, on which you can call the run() function to execute a Cypher statement. As with the session.run example above, the first argument for the run() function should be a Cypher statement, and any parameters in the Cypher statement should be passed as named parameters.

**python**

**Running a Read Transaction**

*# Define a Unit of work to run within a Transaction (`tx`)*

def get\_movies(tx, title):

return tx.run("""

MATCH (p:Person)-[:ACTED\_IN]->(m:Movie)

WHERE m.title = $title // **(1)**

RETURN p.name AS name

LIMIT 10

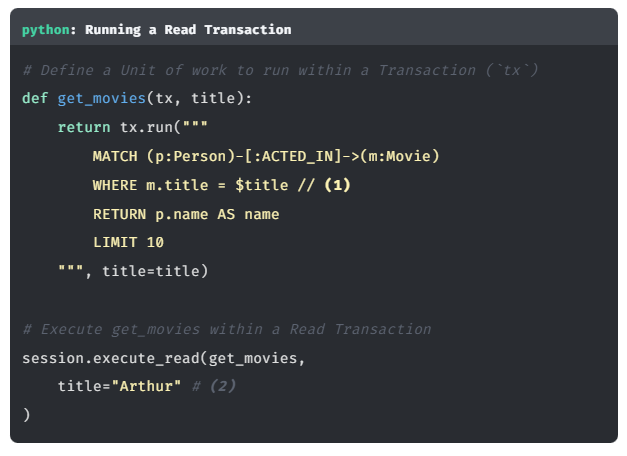
""", title=title)

*# Execute get\_movies within a Read Transaction*

session.execute\_read(get\_movies,

title="Arthur" *#* ***(2)***

)



**PARAMETERIZED QUERIES**

In the query above, the the $ prefix of $title (1) indicates that this value relates to the parameter defined in the second argument (2) of the run() function call.

You do not need to explicitly commit a read transaction.

### Write Transactions

If you intend to write data to the database, you should execute a **Write** Transaction.

In clustered environments, write queries are sent exclusively to the leader of the cluster. The leader of the cluster is then responsible for processing the query and synchronising the transaction across the followers and read-replica servers in the cluster.

The process is identical to running a Read Transaction.

**python**

**Running a Write Transaction**

*# Call tx.run() to execute the query to create a Person node*

def create\_person(tx, name):

return tx.run(

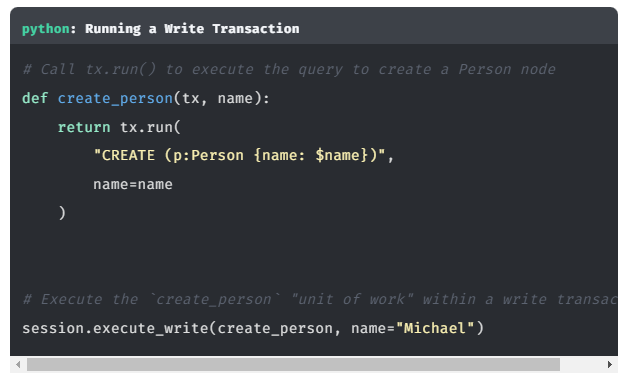
"CREATE (p:Person {name: $name})",

name=name

)

*# Execute the `create\_person` "unit of work" within a write transaction*

session.execute\_write(create\_person, name="Michael")



If anything goes wrong within of the unit of work or there is a problem on Neo4j’s side, the transaction will be automatically rolled back and the database will remain in its previous state. If the unit of work succeeds, the transaction will be automatically committed.

Unlike session.run(), if a transient error is received by the driver, for example a connectivity issue with the DBMS, the driver will automatically retry the unit of work.

## Manually Creating Transactions

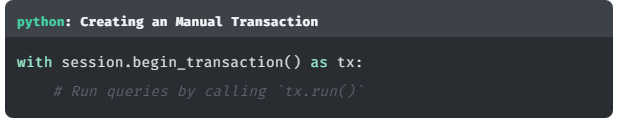
It is also possible to explicitly create a transaction object by calling the begin\_transaction() function on the session.

**python**

**Creating an Manual Transaction**

with session.begin\_transaction() as tx:

*# Run queries by calling `tx.run()`*



This returns a Transaction object identical to the one passed in to the unit of work function when calling execute\_read() or execute\_write().

This method differs from the execute\_read and execute\_write() functions, in that the transaction will have to be manually committed or rolled back depending on the outcome of the unit of work.

You can commit a transaction by calling the tx.commit() function, or roll back the transaction by calling tx.rollback().

**python**

try:

*# Run a query*

tx.run(query, \*\*params)

*# Commit the transaction*

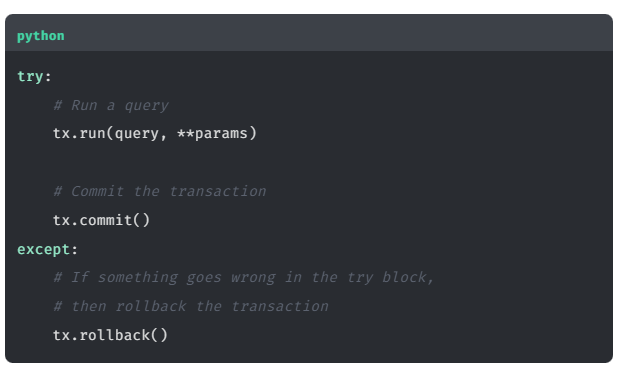
tx.commit()

except:

*# If something goes wrong in the try block,*

*# then rollback the transaction*

tx.rollback()



## Closing the Session

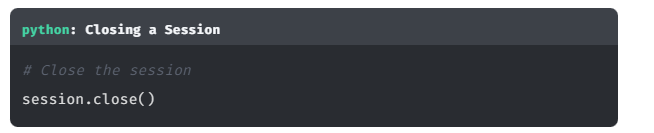
Once you are finished with your session, you call the close() method to release any database connections held by that session.

**python**

**Closing a Session**

*# Close the session*

session.close()



## A Working Example

Click to reveal a complete working example

The following code defines a function that accepts a name parameter, then executes a write transaction to create a :Person node in the people database.

**python**

**Create a Person node in the customers database**

def create\_person\_work(tx, name):

return tx.run("CREATE (p:Person {name: $name}) RETURN p",

name=name).single()

def create\_person(name):

*# Create a Session for the `people` database*

session = driver.session(database="people")

*# Create a node within a write transaction*

record = session.execute\_write(create\_person\_work,

name=name)

*# Get the `p` value from the first record*

person = record["p"]

*# Close the session*

session.close()

*# Return the property from the node*

return person["name"]



# Processing Results

Query results are typically consumed as a stream of records. The drivers provide a way to iterate through that stream.

## Result

Here is an example query which retrieves a list of :Person nodes related to a given Movie.

**python**

**Get Actors by Movie title**

Copy to Clipboard

*# Unit of work*

def get\_actors(tx, movie): *#* ***(1)***

result = tx.run("""

MATCH (p:Person)-[:ACTED\_IN]->(:Movie {title: $title})

RETURN p

""", title=movie)

*# Access the `p` value from each record*

return [ record["p"] for record in result ]

*# Open a Session*

with driver.session() as session:

*# Run the unit of work within a Read Transaction*

actors = session.execute\_read(get\_actors, movie="The Green Mile") *#* ***(2)***

for record in actors:

print(record["p"])

session.close()



The code can be broken down into two elements:

1. The get\_actors() function defines a unit of work to be executed within a transaction, passed as the first argument of the function, in this case referenced as tx
2. The execute\_read() method executes the unit of work within a Read Transaction

The result of the execute\_read() is a **Result** object.

The result object acts as a buffer for an iterable list of records and provides a number of options for accessing those records. Once a result is consumed, it is removed from the buffer.

### Peeking at Results

If you wish to preview a result without consuming it, you can use the peek method.

**python**

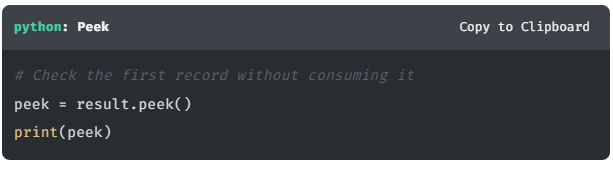
**Peek**

Copy to Clipboard

*# Check the first record without consuming it*

peek = result.peek()

print(peek)



This can be used to preview the first record in the result without removing it from the buffer.

### Keys

To get the keys for each record in the result, you can call the keys() method.

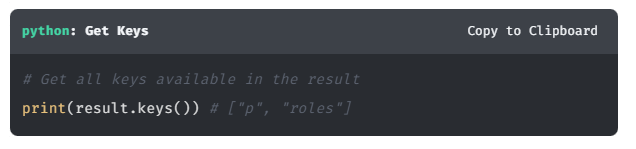
**python**

**Get Keys**

Copy to Clipboard

*# Get all keys available in the result*

print(result.keys()) *# ["p", "roles"]*



### Single Result

If you only expect a single record, you can use the single() method on the result to return the first record.

**python**

**First Result**

Copy to Clipboard

def get\_actors\_single(tx, movie):

result = tx.run("""

MATCH (p:Person)-[:ACTED\_IN]->(:Movie {title: $title})

RETURN p

""", title=movie)

return result.single()



If more than one record is available from the result then a warning will be generated, but the first result will still be returned. If no results are available, then the method call will return None.

### Value

If you wish to extract a single value from the remaining list of results, you can use the value() method.

**python**

**Extract a value**

Copy to Clipboard

def get\_actors\_values(tx, movie):

result = tx.run("""

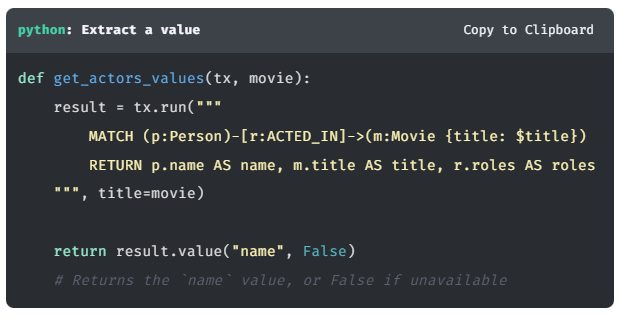
MATCH (p:Person)-[r:ACTED\_IN]->(m:Movie {title: $title})

RETURN p.name AS name, m.title AS title, r.roles AS roles

""", title=movie)

return result.value("name", False)

*# Returns the `name` value, or False if unavailable*



This method expects two parameters:

1. The key or index of the field to return for each remaining record, and returns a list of single values.
2. Optionally, you can provide a default value to be used if the value is None or unavailable.

### Values

If you need to extract more than one item from each record, use the values() method. The method expects one parameter per item requested from the RETURN statement of the query.

**python**

**Extract values**

Copy to Clipboard

def get\_actors\_values(tx, movie):

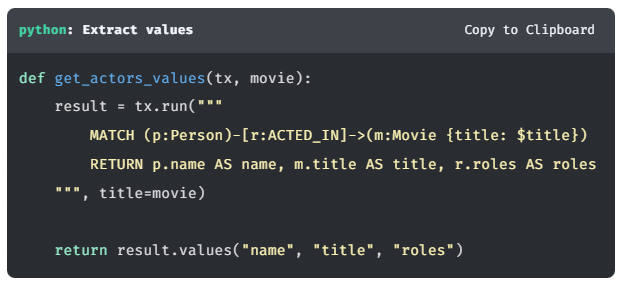
result = tx.run("""

MATCH (p:Person)-[r:ACTED\_IN]->(m:Movie {title: $title})

RETURN p.name AS name, m.title AS title, r.roles AS roles

""", title=movie)

return result.values("name", "title", "roles")



In the above example, a list will be returned, with each entry containing values representing name, title, and roles.

### Consume

The consume() method will consume the remainder of the results and return a **Result Summary**.

**python**

**Result Summary**

Copy to Clipboard

def get\_actors\_consume(tx, name):

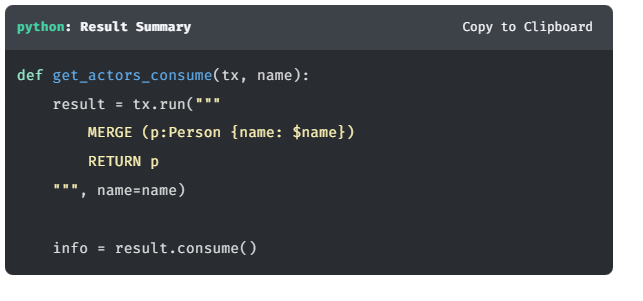
result = tx.run("""

MERGE (p:Person {name: $name})

RETURN p

""", name=name)

info = result.consume()



The Result Summary contains a information about the server, the query, execution times and a counters object which provide statistics about the query.

**python**

**Query Execution Times**

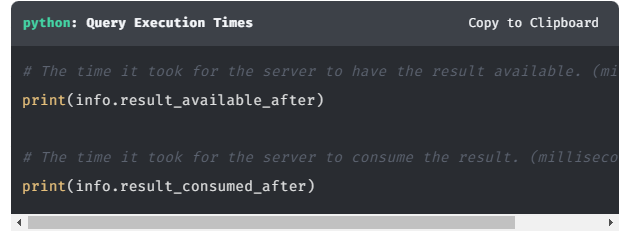
Copy to Clipboard

*# The time it took for the server to have the result available. (milliseconds)*

print(info.result\_available\_after)

*# The time it took for the server to consume the result. (milliseconds)*

print(info.result\_consumed\_after)



The counters object can be used to retrieve the number of nodes, relationships, properties or labels that were affected during a write transaction.

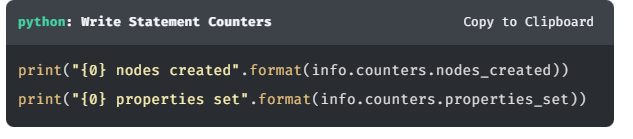
**python**

**Write Statement Counters**

Copy to Clipboard

print("{0} nodes created".format(info.counters.nodes\_created))

print("{0} properties set".format(info.counters.properties\_set))



You can [**read more about the result summary here**](https://neo4j.com/docs/api/python-driver/4.4/api.html#neo4j.ResultSummary).

## Exploring Records

When accessing a record, either within a loop, list comprehension or within a single record, you can use the [] bracket syntax.

The following example extracts the p value from each record in the result buffer.

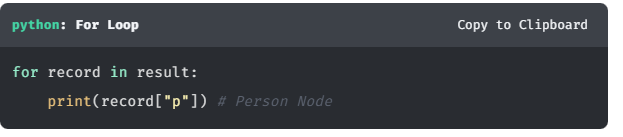
**python**

**For Loop**

Copy to Clipboard

for record in result:

print(record["p"]) *# Person Node*



You can also access a value using its index, as it relates to the value contained in the keys array:

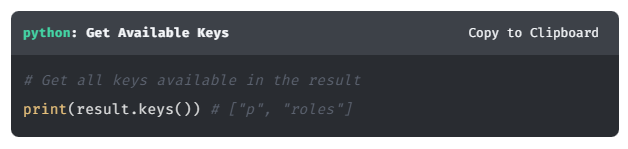
**python**

**Get Available Keys**

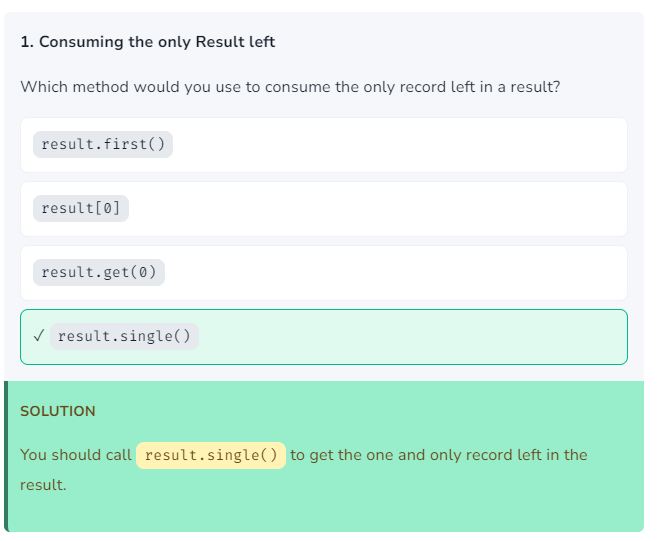
Copy to Clipboard

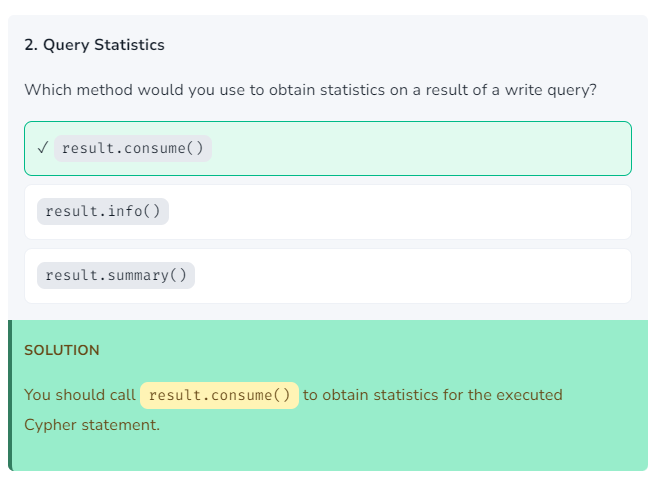
*# Get all keys available in the result*

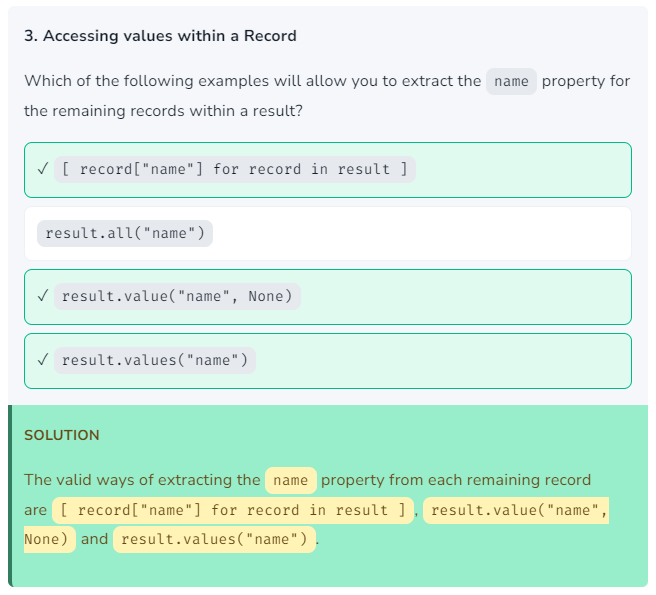
print(result.keys()) *# ["p", "roles"]*



## Quiz: Check Your Understanding







# The Neo4j Type System

At this point, we should take a look at the Cypher type system. As Neo4j is written in Java (the j in Neo4**j** stands for Java after all), there are some discrepancies between the types stored in the Neo4j database and native Python types.

Some values like strings, floats, booleans, and nulls map directly to Python types, but more complex types need special handling.

Python Types to Neo4j Types

| **Python Type** | **Neo4j Cypher Type** | **Notes** |
| --- | --- | --- |
| None | null |  |
| bool | Boolean |  |
| int | Integer |  |
| float | Float |  |
| str | String |  |
| bytearray | Bytes [1] |  |
| list | List |  |
| dict | Map |  |
| neo4j.spatial.Point | Point | See [**Spatial Data Types**](https://graphacademy.neo4j.com/courses/app-python/2-interacting/3-type-system/#_spatial_data_types) |
| neo4j.spatial.CartesianPoint | Point (Cartesian) | See [**Spatial Data Types**](https://graphacademy.neo4j.com/courses/app-python/2-interacting/3-type-system/#_spatial_data_types) |
| neo4j.spatial.WGS84Point | Point (WGS-84) | See [**Spatial Data Types**](https://graphacademy.neo4j.com/courses/app-python/2-interacting/3-type-system/#_spatial_data_types) |
| neo4j.graph.Node | Node | See [**Nodes & Relationships**](https://graphacademy.neo4j.com/courses/app-python/2-interacting/3-type-system/#_nodes_relationships) |
| neo4j.graph.Relationship | Relationship | See [**Nodes & Relationships**](https://graphacademy.neo4j.com/courses/app-python/2-interacting/3-type-system/#_nodes_relationships) |
| neo4j.graph.Path | Path | See [**Nodes & Relationships**](https://graphacademy.neo4j.com/courses/app-python/2-interacting/3-type-system/#_nodes_relationships) |

For more information about Temporal Data Types, see [**Temporal Data Types**](https://graphacademy.neo4j.com/courses/app-python/2-interacting/3-type-system/#_temporal_data_types).

Let’s take a look at some of these types in more detail.

## Nodes & Relationships

Nodes and Relationships are both returned as similar classes.

As an example, let’s take the following code snippet:

**python**

**Return Nodes and Relationships**

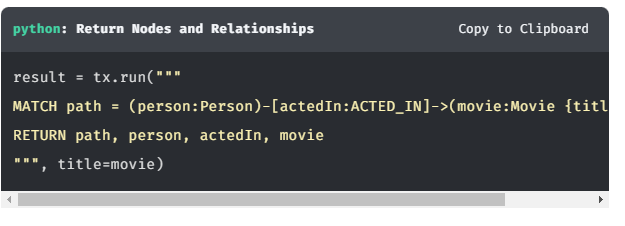
Copy to Clipboard

result = tx.run("""

MATCH path = (person:Person)-[actedIn:ACTED\_IN]->(movie:Movie {title: $title})

RETURN path, person, actedIn, movie

""", title=movie)



The query will return one record for each :Person and :Movie node with an :ACTED\_IN relationship between them.

### Nodes

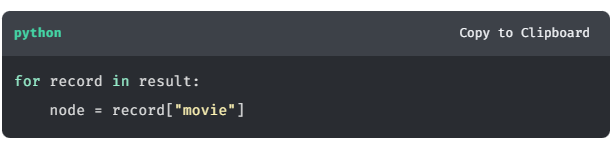
We can retrieve the movie value from a record using the [] brackets method, providing a string that refers to the alias for the :Movie node in the Cypher statement.

**python**

Copy to Clipboard

for record in result:

node = record["movie"]



The value assigned to the node variable will be the instance of a Node. Node is a type provided by the Neo4j Python Driver to hold the information held in Neo4j for the node.

**python**

**Working with Node Objects**

Copy to Clipboard

print(node.id) *#* ***(1)***

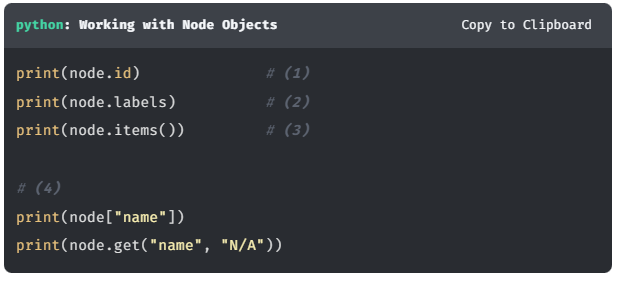
print(node.labels) *#* ***(2)***

print(node.items()) *#* ***(3)***

*#* ***(4)***

print(node["name"])

print(node.get("name", "N/A"))



1. The id property provides access to the node’s Internal ID  
   eg. 1234
2. The labels property is a frozenset containing an array of labels attributed to the Node  
   eg. ['Person', 'Actor']
3. The items() method provides access to the node’s properties as an iterable of all name-value pairs.  
   eg. {name: 'Tom Hanks', tmdbId: '31' }
4. A single property can be retrieved by either using [] brackets or using the get() method. The get() method also allows you to define a default property if none exists.

**INTERNAL IDS**

Internal IDs refer to the position in the Neo4j store files where the record is held. These numbers can be re-used, a best practice is to always look up a node by an indexed property rather than relying on an internal ID.

### Relationships

Relationship objects are similar to a Node in that they provide the same method for accessing the internal ID and properties.

**python**

**Working with Relationship Objects**

Copy to Clipboard

acted\_in = record["actedIn"]

print(acted\_in.id) *#* ***(1)***

print(acted\_in.type) *#* ***(2)***

print(acted\_in.items()) *#* ***(3)***

*# 4*

print(acted\_in["roles"])

print(acted\_in.get("roles", "(Unknown)"))

print(acted\_in.start\_node) *#* ***(5)***

print(acted\_in.end\_node) *#* ***(6)***



1. The id property holds the internal ID of the relationship.  
   eg. 9876
2. The type property holds the relationship type  
   eg. ACTED\_IN
3. The items() method provides access to the relationships’s properties as an iterable of all name-value pairs.  
   eg. {role: 'Woody'}
4. As with Nodes, you can access a single relationship property using brackets or the get() method.
5. start\_node - an integer representing the internal ID for the node at the start of the relationship
6. end\_node - an integer representing the internal ID for the node at the end of the relationship

### Paths

If you return a path of nodes and relationships, they will be returned as an instance of a Path.

**python**

**Working with Path Objects**

Copy to Clipboard

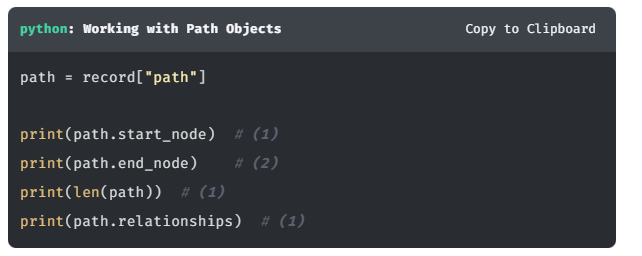
path = record["path"]

print(path.start\_node) *#* ***(1)***

print(path.end\_node) *#* ***(2)***

print(len(path)) *#* ***(1)***

print(path.relationships) *#* ***(1)***



1. start\_node - a Neo4j Integer representing the internal ID for the node at the start of the path
2. end\_node - a Neo4j Integer representing the internal ID for the node at the end of the path
3. len(path) - A count of the number of relationships within the path
4. relationships - An array of Relationship objects within the path.

#### Path Segments

A path is split into segments representing each relationship in the path. For example, say we have a path of (p:Person)-[:ACTED\_IN]→(m:Movie)-[:IN\_GENRE]→(g:Genre), there would be two relationships.

1. (p:Person)-[:ACTED\_IN]→(m:Movie)
2. (m:Movie)-[:IN\_GENRE]→(g:Genre)

The relationships within a path can be iterated over using the iter() function.

**python**

**Iterating over Segments**

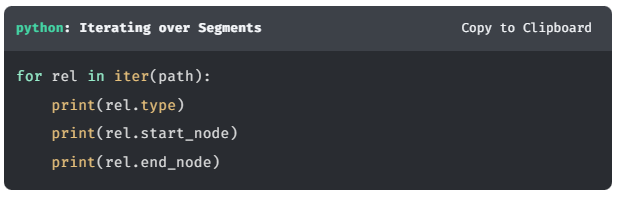
Copy to Clipboard

for rel in iter(path):

print(rel.type)

print(rel.start\_node)

print(rel.end\_node)



## Temporal Data Types

Temporal data types are implemented by the neo4j.time module.

It provides a set of types compliant with ISO-8601 and Cypher, which are similar to those found in the built-in datetime module. Sub-second values are measured to nanosecond precision and the types are compatible with **[pytz](http://pytz.sourceforge.net/" \t "_blank)**.

The table below shows the general mappings between Cypher and the temporal types provided by the driver.

In addition, the built-in temporal types can be passed as parameters and will be mapped appropriately.

| Table 1. Temporal Types | | | |
| --- | --- | --- | --- |
| **Neo4j Cypher Type** | **Python driver type** | **Python built-in type** | tzinfo |
| Date | neo4j.time.Date | datetime.date |  |
| Time | neo4j.time.Time | datetime.time | not None |
| LocalTime | neo4j.time.Time | datetime.time | None |
| DateTime | neo4j.time.DateTime | datetime.datetime | not None |
| LocalDateTime | neo4j.time.DateTime | datetime.datetime | None |
| Duration | neo4j.time.Duration | datetime.timedelta |  |

**python**

**Working with Temporal types**

Copy to Clipboard

*# Create a DateTime instance using individual values*

datetime = neo4j.time.DateTime(year, month, day, hour, minute, second, nanosecond)

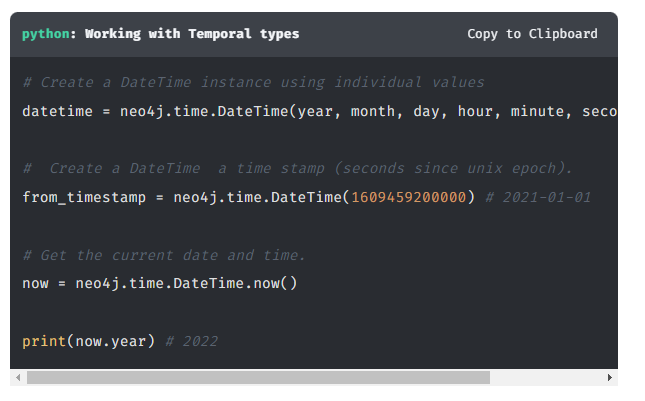
*# Create a DateTime a time stamp (seconds since unix epoch).*

from\_timestamp = neo4j.time.DateTime(1609459200000) *# 2021-01-01*

*# Get the current date and time.*

now = neo4j.time.DateTime.now()

print(now.year) *# 2022*



Each of the above types has a number of attributes for accessing the different, for example year, month, day, and in the case of the types that include a time, hour, minute and second.

For more information, see [**Temporal Data Types&**](https://neo4j.com/docs/api/python-driver/4.4/temporal_types.html).

## Spatial Data Types

Cypher has built-in support for handling spatial values (points), and the underlying database supports storing these point values as properties on nodes and relationships.

### **Points**

| **Cypher Type** | **Python Type** |
| --- | --- |
| Point | neo4j.spatial.Point |
| Point (Cartesian) | neo4j.spatial.CartesianPoint |
| Point (WGS-84) | neo4j.spatial.WGS84Point |

#### CartesianPoint

A Cartesian Point can be created in Cypher by supplying x and y values to the point() function. The optional z value represents the height.

To create a Cartesian Point in Python, you can import the neo4j.spatial.CartesianPoint class.

**python**

**Cartesian**

Copy to Clipboard

*# Using X and Y values*

twoD=CartesianPoint((1.23, 4.56))

print(twoD.x, twoD.y)

*# Using X, Y and Z*

threeD=CartesianPoint((1.23, 4.56, 7.89))

print(threeD.x, threeD.y, threeD.z)



For more information, [**see the Python reference**](https://neo4j.com/docs/api/python-driver/current/api.html#cartesianpoint).

#### WGS84Point

A WGS84 Point can be created in Cypher by supplying latitude and longitude values to the point() function. To create a Cartesian Point in Python, you can import the neo4j.spatial.WGS84Point class.

**python**

**WGS84**

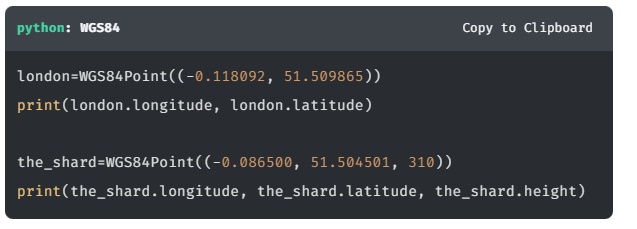
Copy to Clipboard

london=WGS84Point((-0.118092, 51.509865))

print(london.longitude, london.latitude)

the\_shard=WGS84Point((-0.086500, 51.504501, 310))

print(the\_shard.longitude, the\_shard.latitude, the\_shard.height)



For more information, [**see the Python reference**](https://neo4j.com/docs/api/python-driver/current/api.html#wgs84point).

### **Distance**

When using the point.distance function in Cypher, the distance calculated between two points is returned as a float.

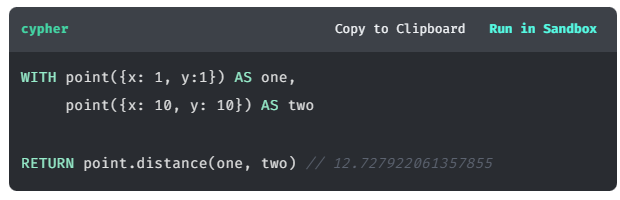
**cypher**

Copy to ClipboardRun in Sandbox

WITH point({x: 1, y:1}) AS one,

point({x: 10, y: 10}) AS two

RETURN point.distance(one, two) *// 12.727922061357855*



For more information on Spatial types, [**see the Cypher Manual**](https://neo4j.com/docs/cypher-manual/current/values-and-types/spatial/).

# The Home Page (challenge)

In this challenge, you will use the knowledge gained so far in this course to add new functionality to the API. You will modify the all() method of the [MovieDAO](https://github.com/neo4j-graphacademy/app-python/blob/main/api/dao/movies.py" \t "_blank) to do the following:

* [**1. Create a Unit of Work**](https://graphacademy.neo4j.com/courses/app-python/2-interacting/4-home-page/#_1_create_a_unit_of_work)
* [**2. Open a new Session**](https://graphacademy.neo4j.com/courses/app-python/2-interacting/4-home-page/#_2_open_a_new_session)
* [**[3. Extract a list of Movies from the Result]**](https://graphacademy.neo4j.com/courses/app-python/2-interacting/4-home-page/#3.%20Extract%20a%20list%20of%20Movies%20from%20the%20Result)
* [**[4. Return the Results]**](https://graphacademy.neo4j.com/courses/app-python/2-interacting/4-home-page/#4.%20Return%20the%20Results)

Once you have completed the challenge, you will be asked to run a unit test to verify that the code has been correctly implemented. If the test runs correctly, the title of the highest rated movie will be logged. You will need this value to verify that the test has run correctly.

## **Exploring the Code**

Before you start, let’s take a look at the code. If you are not interested in exploring the code, you can skip straight to [**Implementing Read Transactions**](https://graphacademy.neo4j.com/courses/app-python/2-interacting/4-home-page/#_implementing_read_transactions).

If you start the application and access the app at [**http://localhost:3000**](http://localhost:3000/), you will see two lists on the home page; one for Popular Movies and one for Latest Releases. Both of these lists are populated by a request to <http://localhost:3000/api/movies> with some additional parameters.

### **Route Handler**

You can find the route handler, the function that handles the request, in the movies blueprint at [api/routes/movies.py](https://github.com/neo4j-graphacademy/app-python/blob/main/api/routes/movies.py" \t "_blank):

**python**

**api/routes/movie.py**

@movie\_routes.get('/')

@jwt\_required(optional=True)

def get\_movies():

*# Extract pagination values from the request*

sort = request.args.get("sort", "title")

order = request.args.get("order", "ASC")

limit = request.args.get("limit", 6, type=int)

skip = request.args.get("skip", 0, type=int)

*# Get User ID from JWT Auth*

user\_id = current\_user["sub"] if current\_user != None else None

*# Create a new MovieDAO Instance*

dao = MovieDAO(current\_app.driver)

*# Retrieve a paginated list of movies*

output = dao.all(sort, order, limit=limit, skip=skip, user\_id=user\_id)

*# Return as JSON*

return jsonify(output)



Within the route handler, you can see that:

1. The sort, order, limit and skip values are extracted from the query string. This allows us to apply sorting and pagination to the results.
2. A new instance of the MovieDAO is created with the Driver instance that you created in [**Adding the Driver**](https://graphacademy.neo4j.com/courses/app-python/1-driver/3-connecting/) passed to the constructor.
3. The results are retrieved via the all() method and are returned by the handler as JSON.

### **The Movie DAO**

The magic happens in the MovieDAO, located at [api/dao/movies.py](https://github.com/neo4j-graphacademy/app-python/blob/main/api/dao/movies.py" \t "_blank). For this route we are concerned with the all() method.

If we take a closer look at the all() method, we can see that it currently returns a hardcoded list of popular movies from another file in the repository.

**python**

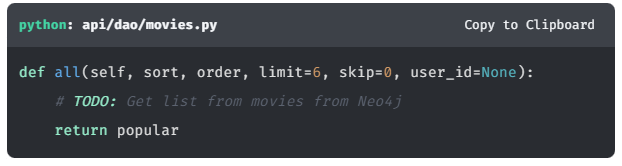
**api/dao/movies.py**

Copy to Clipboard

def all(self, sort, order, limit=6, skip=0, user\_id=None):

*# TODO: Get list from movies from Neo4j*

return popular



You will need to replace these TODO comments with working code to complete the challenge.

## Implementing Read Transactions

As you learned in the Sessions and Transactions lesson, you will write the code to open a new session and run the query within a Read Transaction. Once the query has run, you add code to close the session. Then, finally you add code to extract and return the results.

### **1. Create a Unit of Work**

First, create a new get\_movies to represent the unit of work.

This function will have one mandatory argument passed to it, a Transaction instance that you can use to execute a Cypher statement using the run() method. The function should also have the sort, order, skip, limit and userId parameters passed through as named parameters - these will be referenced in the query by prefixing the name with a dollar sign ($).

**python**

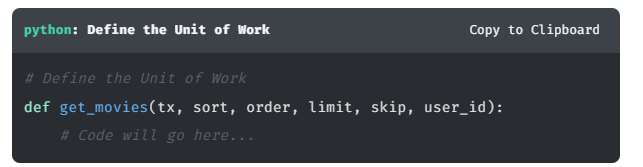
**Define the Unit of Work**

Copy to Clipboard

*# Define the Unit of Work*

def get\_movies(tx, sort, order, limit, skip, user\_id):

*# Code will go here...*



You will need to call the run() method on the tx argument, passing a Cypher statement to retrieve a list of paginated movie results along with named arguments sort, order, skip, limit and userId.

**python**

**Define and Execute a Cypher statement**

Copy to Clipboard

*# Define the cypher statement*

cypher = """

MATCH (m:Movie)

WHERE m.`{0}` IS NOT NULL

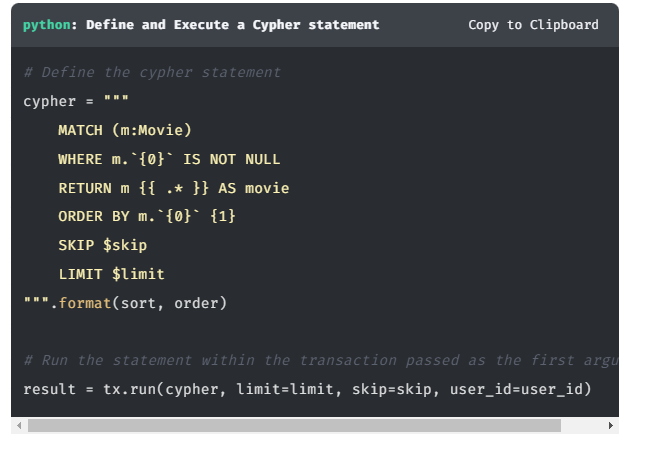
RETURN m {{ .\* }} AS movie

ORDER BY m.`{0}` {1}

SKIP $skip

LIMIT $limit

""".format(sort, order)



*# Run the statement within the transaction passed as the first argument*

result = tx.run(cypher, limit=limit, skip=skip, user\_id=user\_id)

### **Extract a list of Movies from the Result**

The individual movie values can then be extracted from the result variable using a list comprehension.

**python**

Copy to Clipboard

*# Extract a list of Movies from the Result*

return [row.value("movie") for row in result]



**CONSUMING RESULTS**

Make sure that you extract the results within the unit of work. Once the transaction function ends, any results that have not been consumed will be lost.

### **2. Open a new Session**

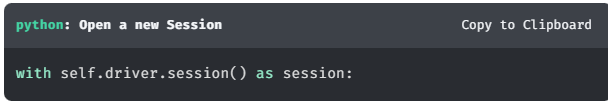
To execute the get\_movies function within a Read Transaction, first open a new session:

**python**

**Open a new Session**

Copy to Clipboard

with self.driver.session() as session:



### **3. Return the Results**

Then the output of the get\_movies function can then be returned.

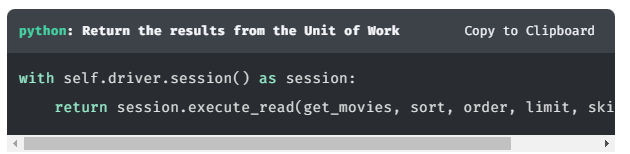
**python**

**Return the results from the Unit of Work**

Copy to Clipboard

with self.driver.session() as session:

return session.execute\_read(get\_movies, sort, order, limit, skip, user\_id)



## **Working Solution**

Click here to reveal the completed all() method

**python**

**api/dao/movies.py**

Copy to Clipboard

def all(self, sort, order, limit=6, skip=0, user\_id=None):

*# Define the Unit of Work*

def get\_movies(tx, sort, order, limit, skip, user\_id):

*# Define the cypher statement*

cypher = """

MATCH (m:Movie)

WHERE m.`{0}` IS NOT NULL

RETURN m {{ .\* }} AS movie

ORDER BY m.`{0}` {1}

SKIP $skip

LIMIT $limit

""".format(sort, order)

*# Run the statement within the transaction passed as the first argument*

result = tx.run(cypher, limit=limit, skip=skip, user\_id=user\_id)

*# Extract a list of Movies from the Result*

return [row.value("movie") for row in result]

with self.driver.session() as session:

return session.execute\_read(get\_movies, sort, order, limit, skip, user\_id)



## **Testing**

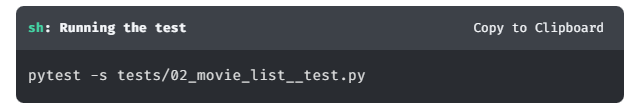
To test that this functionality has been correctly implemented, run the following code in a new terminal session:

**sh**

**Running the test**

Copy to Clipboard

pytest -s tests/02\_movie\_list\_\_test.py



The test file is located at [tests/02\_movie\_list\_\_test.py](https://github.com/neo4j-graphacademy/app-python/blob/main/tests/02_movie_list__test.py).

Are you stuck? Click here for help

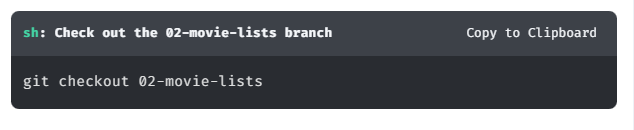
If you get stuck, you can see a working solution by checking out the 02-movie-lists branch by running:

**sh**

**Check out the 02-movie-lists branch**

Copy to Clipboard

git checkout 02-movie-lists



You may have to commit or stash your changes before checking out this branch. You can also [**click here to expand the Support pane**](https://graphacademy.neo4j.com/courses/app-python/2-interacting/4-home-page/).

# Registering a User (challenge)

Now that you have read data from the database, you are now ready to write data to the database.

In this challenge, you will rewrite the register() method in the [AuthDAO](https://github.com/neo4j-graphacademy/app-python/blob/main/api/dao/auth.py" \t "_blank) to do the following:

1. [**Create the Unit of Work function**](https://graphacademy.neo4j.com/courses/app-python/2-interacting/5-registering/#_create_the_unit_of_work_function)
2. [**Execute the function within a new Write Transaction**](https://graphacademy.neo4j.com/courses/app-python/2-interacting/5-registering/#_execute_the_function_within_a_new_write_transaction)
3. [**Return the Result**](https://graphacademy.neo4j.com/courses/app-python/2-interacting/5-registering/#_return_the_result)

This process may start to get a little repetitive, but it worth repeating to get used to working with the Neo4j Python Driver.

## **Registering Users**

A key piece of functionality that the application should provide is for new users to be able to register themselves with the site. This functionality is already built into the front end, but at the moment the credentials are hard coded in the API. This might be fine for demo purposes, but limiting the number of users to one is bad for Neoflix’s bottom line.

The dummy register logic is already written into the register() method of the AuthDAO at api/dao/auth.py. As we can see from the snippet below, at the moment it will only accept an email address of graphacademy@neo4j.com.

**python**

**api/dao/auth.py**

Copy to Clipboard

def register(self, email, plain\_password, name):

encrypted = bcrypt.hashpw(plain\_password.encode("utf8"), bcrypt.gensalt()).decode('utf8')

*# TODO: Handle unique constraint error*

if email != "graphacademy@neo4j.com":

raise ValidationException(

f"An account already exists with the email address {email}",

{"email": "An account already exists with this email"}

)

*# Build a set of claims*

payload = {

"userId": "00000000-0000-0000-0000-000000000000",

"email": email,

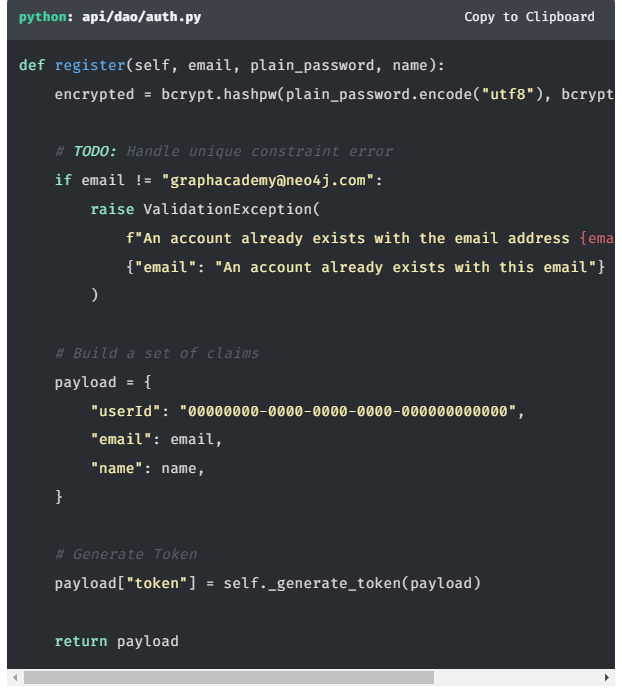
"name": name,

}

*# Generate Token*

payload["token"] = self.\_generate\_token(payload)

return payload



From the last line, you can see that an additional token property is added to the return. This represents the JWT token required to authenticate the user on any future requests. This token is generated further down within this class.

You will replace these TODO comments with working code to complete the challenge.

## **Implementing Write Transactions**

You will follow similar steps to the previous challenge, with the one change that the Cypher statement will be executed within a Write Transaction.

To do so, you will need to call the execute\_write() method on the session object with a function to represent unit of work.

Here are the steps to complete the challenge.

### **Create the Unit of Work function**

Next, define a new function which will call the tx.run() method. The call to tx.run() should include:

1. The Cypher statement as a parameterized string passed as the first argument.
2. The parameters used in the query passed as named parameters: email, encrypted password and name. These will be referenced in the query prefixed with a dollar sign (eg. $email).
3. The query will return a single row, so you can call the .single() method directly on the object to immediately consume the first row.

**python**

**Unit of Work**

Copy to Clipboard

def create\_user(tx, email, encrypted, name):

return tx.run(""" // **(1)**

CREATE (u:User {

userId: randomUuid(),

email: $email,

password: $encrypted,

name: $name

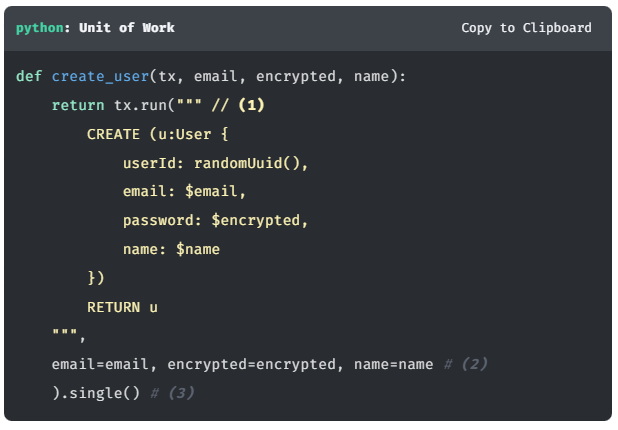
})

RETURN u

""",

email=email, encrypted=encrypted, name=name *#* ***(2)***

).single() *#* ***(3)***



### **Execute the function within a new Write Transaction**

Next, open a new session and use the session.execute\_write() method to execute this unit of work.

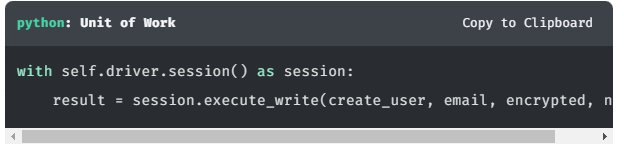
**python**

**Unit of Work**

Copy to Clipboard

with self.driver.session() as session:

result = session.execute\_write(create\_user, email, encrypted, name)



### **Return the Result**

The call above above returns the newly-created :User node as u. As this query creates a single node, it will only ever return one result, so the u value may be taken from the first row.

Use square brackets to extract the u node returned by the Cypher statement above.

The method expects a JWT token to be returned along with the user’s information. The token can be generated by passing a selection of the user’s properties, including the id property, to the \_generate\_token method.

**python**

Copy to Clipboard

user = result['u']

payload = {

"userId": user["userId"],

"email": user["email"],

"name": user["name"],

}

payload["token"] = self.\_generate\_token(payload)

return payload



## **Working Solution**

Click here to reveal the fully-implemented register() method.

**python**

Copy to Clipboard

def register(self, email, plain\_password, name):

encrypted = bcrypt.hashpw(plain\_password.encode("utf8"), bcrypt.gensalt()).decode('utf8')

def create\_user(tx, email, encrypted, name):

return tx.run(""" // **(1)**

CREATE (u:User {

userId: randomUuid(),

email: $email,

password: $encrypted,

name: $name

})

RETURN u

""",

email=email, encrypted=encrypted, name=name *#* ***(2)***

).single() *#* ***(3)***

try:

with self.driver.session() as session:

result = session.execute\_write(create\_user, email, encrypted, name)

user = result['u']

payload = {

"userId": user["userId"],

"email": user["email"],

"name": user["name"],

}

payload["token"] = self.\_generate\_token(payload)

return payload

except ConstraintError as err:

*# Pass error details through to a ValidationException*

raise ValidationException(err.message, {

"email": err.message

})



## **Testing**

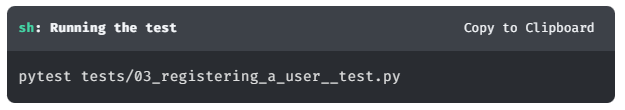
To test that this functionality has been correctly implemented, run the following code in a new terminal session:

**sh**

**Running the test**

Copy to Clipboard

pytest tests/03\_registering\_a\_user\_\_test.py



The test file is located at [tests/03\_registering\_a\_user\_\_test.py](https://github.com/neo4j-graphacademy/app-python/blob/main/tests/03_registering_a_user__test.py).

Are you stuck? Click here for help

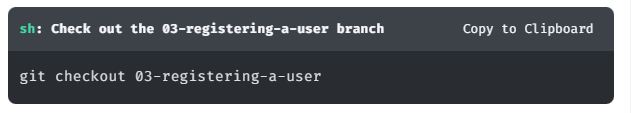
If you get stuck, you can see a working solution by checking out the 03-registering-a-user branch by running:

**sh**

**Check out the 03-registering-a-user branch**

Copy to Clipboard

git checkout 03-registering-a-user



You may have to commit or stash your changes before checking out this branch. You can also [**click here to expand the Support pane**](https://graphacademy.neo4j.com/courses/app-python/2-interacting/5-registering/).



# Handling Driver Errors

When executing a Cypher statement, certain exceptions and error cases may arise. One error could be a transient error that may be resolved if retried, for example a problem connecting to the database instance. Another type of error could be something more permanent, for example a Syntax Error or a Constraint Error.

In the Neo4j Python Driver, an error extending the [neo4j.exceptions.Neo4jError](https://neo4j.com/docs/api/python-driver/current/api.html#neo4j.exceptions.Neo4jError) class will be thrown.

## **Exception Types**

Depending on the nature of the error, you may receive one of the following exceptions:

* neo4j.exceptions.Neo4jError - Raised when the Cypher engine returns an error to the client.
* neo4j.exceptions.ClientError - The Client sent a bad request - changing the request might yield a successful outcome.
  + neo4j.exceptions.CypherSyntaxError - Raised when the Cypher statement contains one or more syntax errors
  + neo4j.exceptions.CypherTypeError - Raised when or more of the data types in the query is incorrect
  + neo4j.exceptions.ConstraintError - Raised when action is rejected due to a constraint violation
  + neo4j.exceptions.AuthError - Raised when authentication failure occurs.
  + neo4j.exceptions.Forbidden - Raised when the action is forbidden for the authenticated user
* neo4j.exceptions.TransientError - The database cannot service the request right now, retrying later might yield a successful outcome
  + neo4j.exceptions.ForbiddenOnReadOnlyDatabase - The write cypher you are requesting cannot be run on a readonly database
  + neo4j.exceptions.NotALeader - The write query cannot be executed on the current server because it is not the leader of the cluster

You can catch the specific exception above within a try/catch block, or catch all Neo4jErrors instances:

**python**

Copy to Clipboard

*# Import the Exception classes from neo4j.exceptions*

from neo4j.exceptions import Neo4jError, ConstraintError

*# Attempt a query*

try:

tx.run(cypher, params)

except ConstraintError as err:

print("Handle constaint violation")

print(err.code) *#* ***(1)***

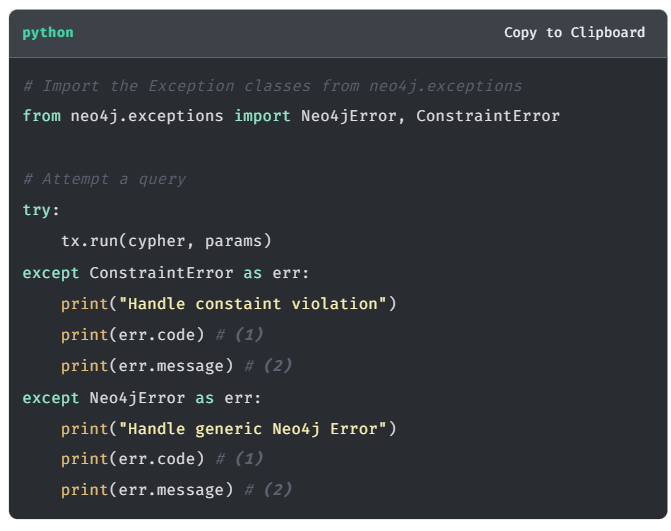
print(err.message) *#* ***(2)***

except Neo4jError as err:

print("Handle generic Neo4j Error")

print(err.code) *#* ***(1)***

print(err.message) *#* ***(2)***



Exceptions contain code (1) and message (2) properties to help you further diagnose the problem.

## **Error Codes**

The Neo4jError includes a code property, which provides higher-level information about the query.

Each status code follows the same format, and includes four parts:

Neo.[Classification].[Category].[Title]

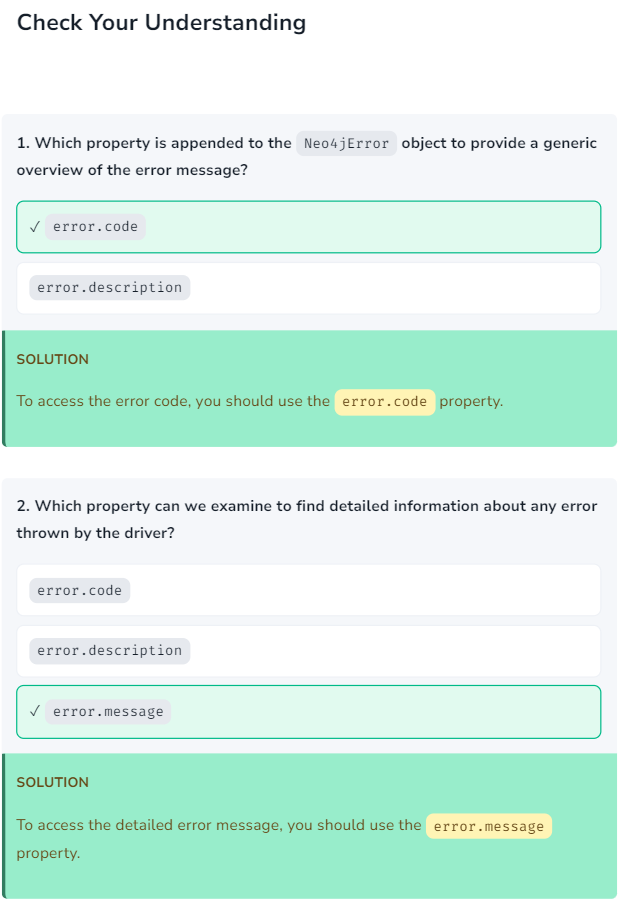
(1) (2) (3) (4)



1. Every Neo4j Error code is prefixed with Neo.
2. The Classification provides a high-level classification of the error - for example, a client-side error or an error with the database.
3. The Category provides a higher-level category for the error - for example, a problem with clustering, a procedure or the database schema.
4. The Title provides specific information about the error that has occurred.

For a comprehensive list of status codes, see [**Status Codes in the Neo4j Documentation**](https://neo4j.com/docs/status-codes/current/).

## **Check Your Understanding**



# Unique Email Addresses (challenge)

In the [**Registering a User challenge**](https://graphacademy.neo4j.com/courses/app-python/2-interacting/5-registering), you updated the register() method in the [AuthDAO](https://github.com/neo4j-graphacademy/app-python/blob/main/api/dao/auth.py" \t "_blank) to create a new User node in the database.

There is still one # TODO comment remaining in this file.

Currently, it is still possible to use the same email address twice, we should take advantage of Unique Constraints in the database to guard against that.

This functionality could be handled by checking the database before running the CREATE Cypher statement, but this could still cause problems if the database is manually updated elsewhere.

Instead, you can pass the responsibility of handling the duplicate user error to the database by creating a Unique Constraint on the :User label, asserting that the email property must be unique.

This will create a potential error case that will need to be handled in the code.

To pass this challenge, you will need to:

1. [**Create a Unique Constraint**](https://graphacademy.neo4j.com/courses/app-python/2-interacting/7-unique-emails/#_create_a_unique_constraint)
2. [**Add a Try/Catch Block**](https://graphacademy.neo4j.com/courses/app-python/2-interacting/7-unique-emails/#_add_a_trycatch_block)

## **Handling Constraint Errors**

If we take a look at register() method, it has been hardcoded to throw a new ValidationError if the email address is anything other than graphacademy@neo4j.com.

**python**

**api/dao/auth.py**

Copy to Clipboard

def register(self, email, plain\_password, name):

encrypted = bcrypt.hashpw(plain\_password.encode("utf8"), bcrypt.gensalt()).decode('utf8')

def create\_user(tx, email, encrypted, name):

return tx.run(""" // **(1)**

CREATE (u:User {

userId: randomUuid(),

email: $email,

password: $encrypted,

name: $name

})

RETURN u

""",

email=email, encrypted=encrypted, name=name *#* ***(2)***

).single() *#* ***(3)***

try:

with self.driver.session() as session:

result = session.execute\_write(create\_user, email, encrypted, name)

user = result['u']

payload = {

"userId": user["userId"],

"email": user["email"],

"name": user["name"],

}

payload["token"] = self.\_generate\_token(payload)

return payload

except ConstraintError as err:

*# Pass error details through to a ValidationException*

raise ValidationException(err.message, {

"email": err.message

})



The code also has no explicit error handling. Any errors will be sent back up the stack and will result in a 500 Bad Request error. Instead, this error should be caught and reformatted in such a way that the server would return a 422 Unprocessable Entity error. This way, the error can be better handled by the UI.

To do this, you will need to rearange the code into try/catch blocks.

**python**

**Try/Catch Example**

Copy to Clipboard

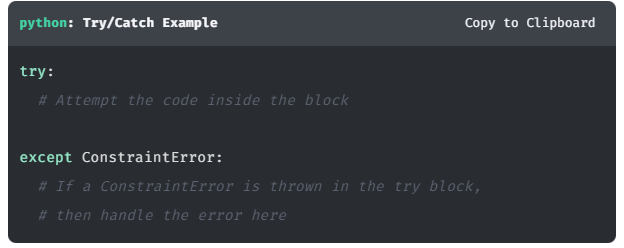
try:

*# Attempt the code inside the block*

except ConstraintError:

*# If a ConstraintError is thrown in the try block,*

*# then handle the error here*



When a user tries to register with an email address that has already been taken, the database will throw an Neo.ClientError.Schema.ConstraintViolation error, at which point the driver will raise a neo4j.exceptions.ConstraintError. Instead of this being treated as an internal server error, it should instead be treated as a 422 Unprocessable Entity. This will allow the front end to handle the error appropriately.

A ValidationError class already exists in the codebase which is handled by a Flask middleware.

## **Completing the Challenge**

To complete this challenge, you will first create a new constraint in your Sandbox database and modify the code to add a try/catch block.

### Create a Unique Constraint

In order to ensure that a property and label combination is unique, you run a CREATE CONSTRAINT query. In this case, we need to ensure that the email property is unique across all nodes with a :User label.

Click the **Run in Sandbox** button to create the constraint on your Sandbox.

**cypher**

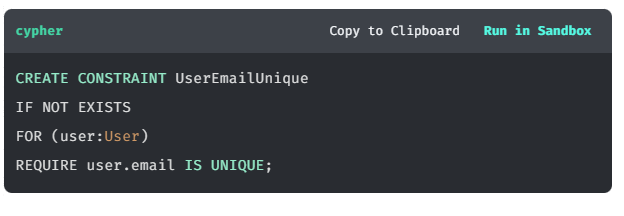
Copy to ClipboardRun in Sandbox

CREATE CONSTRAINT UserEmailUnique

IF NOT EXISTS

FOR (user:User)

REQUIRE user.email IS UNIQUE;



### Add a Try/Catch Block

In the method, we should:

1. **Try** to create a User with the supplied email, password and name.
2. **Catch** a neo4j.exception.ConstraintError error if it is thrown and instead throw an api.exceptions.ValidationException

Your code should look like this:

**python**

Copy to Clipboard

try:

with self.driver.session() as session:

result = session.execute\_write(create\_user, email, encrypted, name)

user = result['u']

payload = {

"userId": user["userId"],

"email": user["email"],

"name": user["name"],

}

payload["token"] = self.\_generate\_token(payload)

return payload

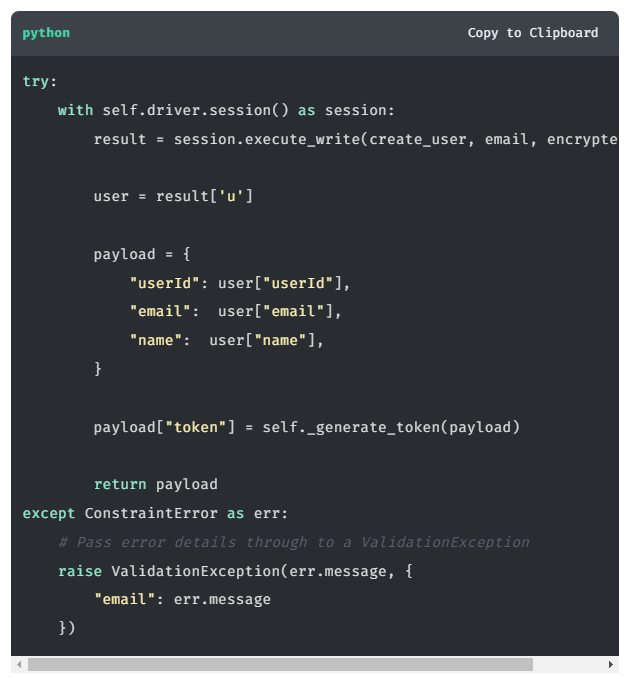
except ConstraintError as err:

*# Pass error details through to a ValidationException*

raise ValidationException(err.message, {

"email": err.message

})



Update the register() method to reflect the changes above, then scroll to [**Testing**](https://graphacademy.neo4j.com/courses/app-python/2-interacting/7-unique-emails/#_testing) to verify that the code works as expected.

## Working Solution

Click here to reveal the fully-implemented register() method.

**python**

Copy to Clipboard

def register(self, email, plain\_password, name):

encrypted = bcrypt.hashpw(plain\_password.encode("utf8"), bcrypt.gensalt()).decode('utf8')

def create\_user(tx, email, encrypted, name):

return tx.run(""" // **(1)**

CREATE (u:User {

userId: randomUuid(),

email: $email,

password: $encrypted,

name: $name

})

RETURN u

""",

email=email, encrypted=encrypted, name=name *#* ***(2)***

).single() *#* ***(3)***

try:

with self.driver.session() as session:

result = session.execute\_write(create\_user, email, encrypted, name)

user = result['u']

payload = {

"userId": user["userId"],

"email": user["email"],

"name": user["name"],

}

payload["token"] = self.\_generate\_token(payload)

return payload

except ConstraintError as err:

*# Pass error details through to a ValidationException*

raise ValidationException(err.message, {

"email": err.message

})



## Testing

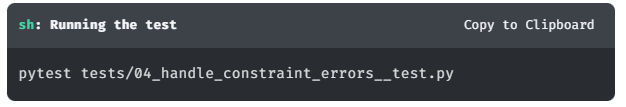
To test that this functionality has been correctly implemented, run the following code in a new terminal session:

**sh**

**Running the test**

Copy to Clipboard

pytest tests/04\_handle\_constraint\_errors\_\_test.py



The test file is located at [tests/04\_handle\_constraint\_errors\_\_test.py](https://github.com/neo4j-graphacademy/app-python/blob/main/tests/04_handle_constraint_errors__test.py).

Are you stuck? Click here for help

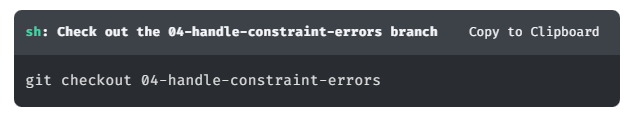
If you get stuck, you can see a working solution by checking out the 04-handle-constraint-errors branch by running:

**sh**

**Check out the 04-handle-constraint-errors branch**

Copy to Clipboard

git checkout 04-handle-constraint-errors



You may have to commit or stash your changes before checking out this branch. You can also [**click here to expand the Support pane**](https://graphacademy.neo4j.com/courses/app-python/2-interacting/7-unique-emails/).

# Authenticating a User (challenge)

At stage of the project, a user can register, but they are still unable to sign in. As with the previous Challenge, the authenticate() method is currently hard coded to accept only the email graphacademy@neo4j.com and password letmein.

In this challenge you will rewrite the authenticate() function of the [AuthDAO](https://github.com/neo4j-graphacademy/app-python/blob/main/api/dao/auth.py" \t "_blank) to find the User node with the corresponding email and compare the password before issuing a JWT token.

But first, let’s take a look at how Authentication works in the application. If you prefer, you can skip straight to [**Implementing Authentication**](https://graphacademy.neo4j.com/courses/app-python/2-interacting/8-authenticating/#_implementing_authentication).

## How Authentication Works

When a user attempts to access an API endpoint that requires authentication, the server checks for a JWT token.

When a user registers or signs in, a JWT token is generated and appended to the User record. This token is then stored by the UI and appended to Bearer to create an authorization header.

The token generation happens in the [AuthDAO](https://github.com/neo4j-graphacademy/app-python/blob/main/api/dao/auth.py" \t "_blank) using the \_generate\_token() method.

When The API receives a request which includes the authorization header, the [**Flask-JWT**](https://pythonhosted.org/Flask-JWT/) library attempts to decode the value and makes the payload available by importing current\_user from flask\_jwt.

### Login Route

When a user submits the login form on the website, a request is sent to [**http://localhost:3000/api/login**](http://localhost:3000/api/login) with a username and password.

## Implementing Authentication

To implement database authentication, you will modify the authenticate method in the AuthDAO.

**python**

**api/dao/auth.py**

Copy to Clipboard

def authenticate(self, email, plain\_password):

*# TODO: Implement Login functionality*

if email == "graphacademy@neo4j.com" and plain\_password == "letmein":

*# Build a set of claims*

payload = {

"userId": "00000000-0000-0000-0000-000000000000",

"email": email,

"name": "GraphAcademy User",

}

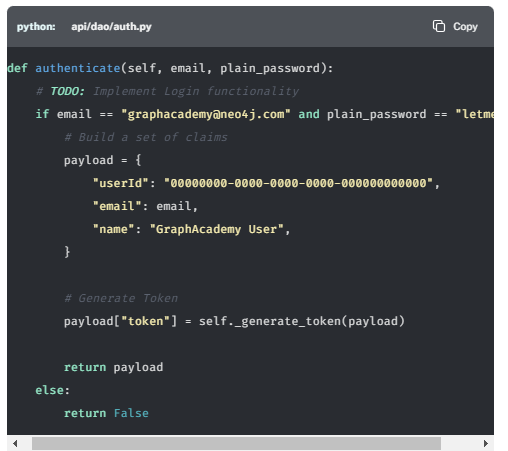
*# Generate Token*

payload["token"] = self.\_generate\_token(payload)

return payload

else:

return False



Your challenge is to update the authenticate() method to perform the following actions:

1. [**Create a transaction function to find the User**](https://graphacademy.neo4j.com/courses/app-python/2-interacting/8-authenticating/#_create_a_transaction_function_to_find_the_user)
2. [**Execute the function within a Read Transaction**](https://graphacademy.neo4j.com/courses/app-python/2-interacting/8-authenticating/#_execute_the_function_within_a_read_transaction)
3. [**Verify the User exists**](https://graphacademy.neo4j.com/courses/app-python/2-interacting/8-authenticating/#_verify_the_user_exists)
4. [**Compare Passwords**](https://graphacademy.neo4j.com/courses/app-python/2-interacting/8-authenticating/#_compare_passwords)
5. [**Return User Details**](https://graphacademy.neo4j.com/courses/app-python/2-interacting/8-authenticating/#_return_user_details)

### Create a transaction function to find the User

The Transaction function should be a simple query that uses Cypher to look up a :User node by the email parameter provided and return a single result.

**python**

**Get User by Email**

Copy to Clipboard

def get\_user(tx, email):

*# Get the result*

result = tx.run("MATCH (u:User {email: $email}) RETURN u",

email=email)

*# Expect a single row*

first = result.single()

*# No records? Return None*

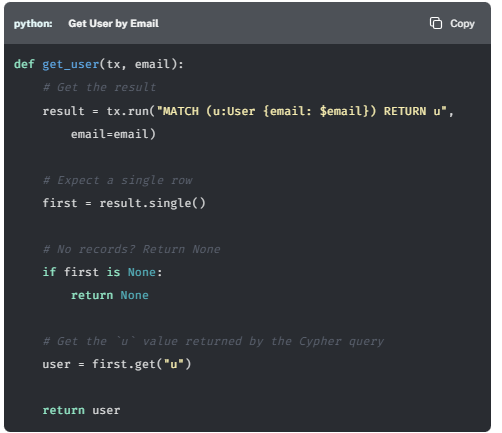
if first is None:

return None

*# Get the `u` value returned by the Cypher query*

user = first.get("u")

return user



### Execute the function within a Read Transaction

After opening up a new session, call the execute\_read method to execute the get\_user function above.

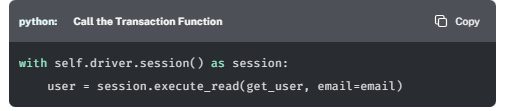
**python**

**Call the Transaction Function**

Copy to Clipboard

with self.driver.session() as session:

user = session.execute\_read(get\_user, email=email)



### Verify the User exists

If the user does not exist, then get\_user will return None. In this case, return False.

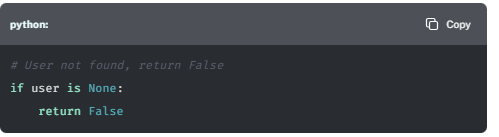
**python**

Copy to Clipboard

*# User not found, return False*

if user is None:

return False



### Compare Passwords

The authenticate() method uses the hashpw function imported from bcrypt to encrypt the password. The library also provides a checkpw function for comparing a plain text value against the previously encrypted value.

If the check fails, return False.

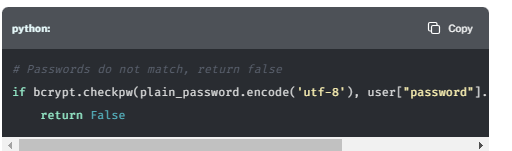
**python**

Copy to Clipboard

*# Passwords do not match, return false*

if bcrypt.checkpw(plain\_password.encode('utf-8'), user["password"].encode('utf-8')) is False:

return False



### Return User Details

Finally, if the user exists and the password comparison returns true, generate a JWT token and return it along with information about the User.

**python**

Copy to Clipboard

*# Generate JWT Token*

payload = {

"userId": user["userId"],

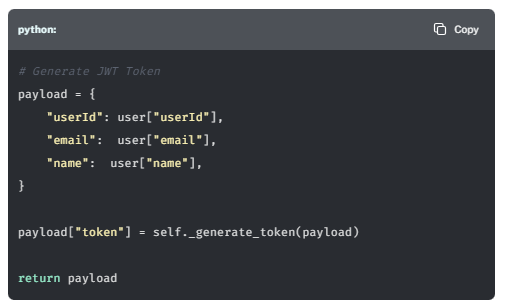
"email": user["email"],

"name": user["name"],

}

payload["token"] = self.\_generate\_token(payload)

return payload



Once you have applied these changes to the authenticate() method, scroll to [**Testing**](https://graphacademy.neo4j.com/courses/app-python/2-interacting/8-authenticating/#_testing) to verify that the method works as expected.

## Working Solution

Click here to reveal the completed authenticate() method:

**python**

**api/dao/auth.py**

Copy to Clipboard

def authenticate(self, email, plain\_password):

def get\_user(tx, email):

*# Get the result*

result = tx.run("MATCH (u:User {email: $email}) RETURN u",

email=email)

*# Expect a single row*

first = result.single()

*# No records? Return None*

if first is None:

return None

*# Get the `u` value returned by the Cypher query*

user = first.get("u")

return user

with self.driver.session() as session:

user = session.execute\_read(get\_user, email=email)

*# User not found, return False*

if user is None:

return False

*# Passwords do not match, return false*

if bcrypt.checkpw(plain\_password.encode('utf-8'), user["password"].encode('utf-8')) is False:

return False

*# Generate JWT Token*

payload = {

"userId": user["userId"],

"email": user["email"],

"name": user["name"],

}

payload["token"] = self.\_generate\_token(payload)

return payload



## Testing

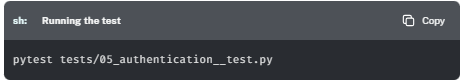
To test that this functionality has been correctly implemented, run the following code in a new terminal session:

**sh**

**Running the test**

Copy to Clipboard

pytest tests/05\_authentication\_\_test.py



The test file is located at [tests/05\_authentication\_\_test.py](https://github.com/neo4j-graphacademy/app-python/blob/main/tests/05_authentication__test.py).

Are you stuck? Click here for help

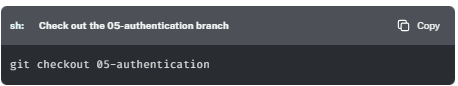
If you get stuck, you can see a working solution by checking out the 05-authentication branch by running:

**sh**

**Check out the 05-authentication branch**

Copy to Clipboard

git checkout 05-authentication



Top of Form

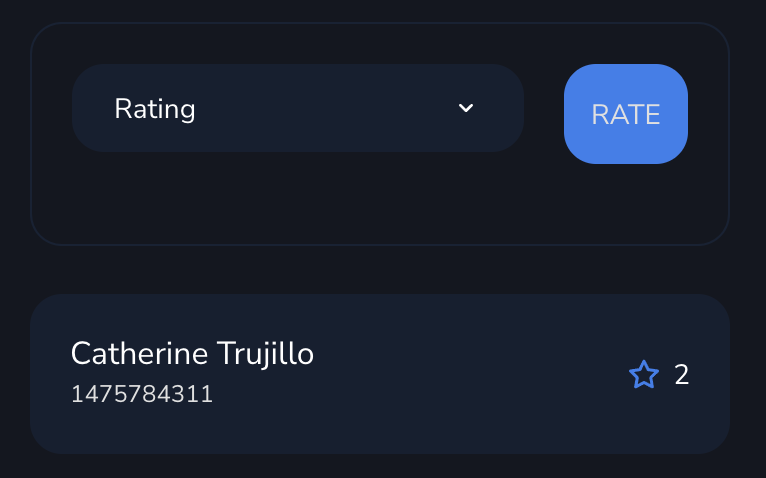


# Rating Movies (Challenge)

The challenges in this course come thick and fast!

In this challenge, you will modify the add() method in the [RatingDTO](https://github.com/neo4j-graphacademy/app-python/blob/main/api/dao/ratings.py" \t "_blank) to save ratings into Neo4j.

## The Request Lifecycle



Before we start, let’s take a look at the request lifecycle when saving a review. If you prefer, you can skip to [**Saving a Rating**](https://graphacademy.neo4j.com/courses/app-python/2-interacting/9-ratings/#_saving_a_rating).

On every Movie page, the user is invited to rate a movie on scale of 1 to 5. The form pictured to the right gives the user the ability to select a rating between 1 and 5 and click submit to save the rating.

When the form is submitted, the website sends a request to /api/account/ratings/{movieId} and the following will happen:

1. The server directs the request to the route handler in [api/routes/account.py](https://github.com/neo4j-graphacademy/app-python/blob/main/api/routes/account.py" \t "_blank), which verifies the user’s JWT token before handling the request.
2. The route handler creates an instance of the [RatingDAO](https://github.com/neo4j-graphacademy/app-python/blob/main/api/dao/ratings.py" \t "_blank).
3. The add() method is called on the RatingDAO, and is passed the ID of the current user, the ID of the movie and a rating from the request body.
4. It is then the responsibility of the add() method to save this information to the database and return an appropriate response.

A rating is represented in the graph a relationship going from a :User to a :Movie node with the type :RATED. The relationship has two properties; the rating (an integer) and a timestamp to represent when the relationship was created.

After the data is saved, the UI expects the movie details to be returned, with an additional property called rating, which will be the rating that the user has given for the movie.

Let’s take a look at the existing method in the RatingDAO.

**python**

**api/dao/ratings.py**

Copy

def add(self, user\_id, movie\_id, rating):

*# TODO: Create function to save the rating in the database*

*# TODO: Call the function within a write transaction*

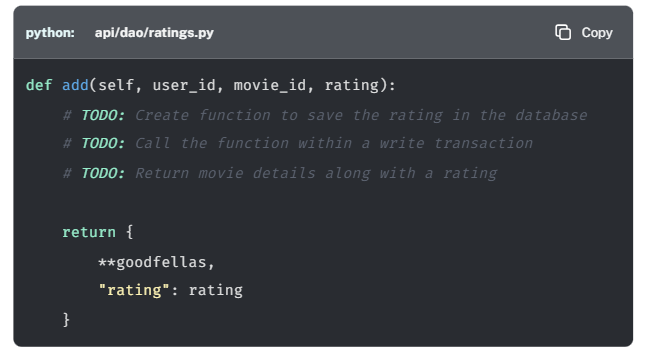
*# TODO: Return movie details along with a rating*

return {

\*\*goodfellas,

"rating": rating

}



Your challenge is to replace the TODO comments in this method with working code.

## Saving a Rating

Complete the following steps to complete the challenge:

1. [**Define a Transaction Function**](https://graphacademy.neo4j.com/courses/app-python/2-interacting/9-ratings/#_define_a_transaction_function)
2. [**Run the Function in a Write Transaction**](https://graphacademy.neo4j.com/courses/app-python/2-interacting/9-ratings/#_run_the_function_in_a_write_transaction)
3. [**Check User and Movie Exist**](https://graphacademy.neo4j.com/courses/app-python/2-interacting/9-ratings/#_check_user_and_movie_exist)
4. [**Return the Results**](https://graphacademy.neo4j.com/courses/app-python/2-interacting/9-ratings/#_return_the_results)

### Define a Transaction Function

The first step is to define a new transaction function. The function should call the run() method on the transaction object passed as the first parameter, using the additional parameters passed to the function as named parameters.

The query should only ever return a single row, so the single() method can be called to instantly consume and return the first row.

**python**

**Unit of Work**

Copy

*# Create function to save the rating in the database*

def create\_rating(tx, user\_id, movie\_id, rating):

return tx.run("""

MATCH (u:User {userId: $user\_id})

MATCH (m:Movie {tmdbId: $movie\_id})

MERGE (u)-[r:RATED]->(m)

SET r.rating = $rating,

r.timestamp = timestamp()

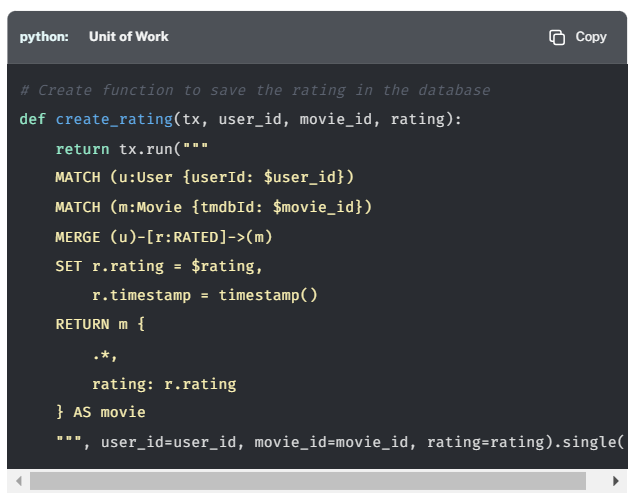
RETURN m {

.\*,

rating: r.rating

} AS movie

""", user\_id=user\_id, movie\_id=movie\_id, rating=rating).single()



By using the MERGE keyword here, we will overwrite an existing rating if one already exists. This way we don’t need to worry about duplicates or deleting existing records.

### Run the Function in a Write Transaction

Within a new database session, call the create\_rating() function, passing the three parameters passed to the method: user\_id, movie\_id and rating.

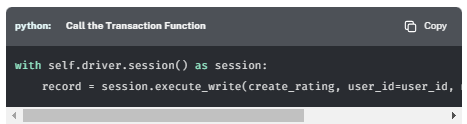
**python**

**Call the Transaction Function**

Copy

with self.driver.session() as session:

record = session.execute\_write(create\_rating, user\_id=user\_id, movie\_id=movie\_id, rating=rating)



### Check User and Movie Exist

As the transaction function calls the single() method, the returned value will be either a Record or None.

If the returned value is None, either the User or Movie could not be found, raise a NotFoundException. This will be handled by a Flask middleware.

**python**

Copy

if record is None:

raise NotFoundException()



### Return the Results

Otherwise, use the brackets method to extract the movie value from the record and return it.

**python**

Copy

return record["movie"]



## Working Solution

Click here to reveal the Working Solution.

**python**

**api/dao/ratings.py**

Copy

def add(self, user\_id, movie\_id, rating):

*# Create function to save the rating in the database*

def create\_rating(tx, user\_id, movie\_id, rating):

return tx.run("""

MATCH (u:User {userId: $user\_id})

MATCH (m:Movie {tmdbId: $movie\_id})

MERGE (u)-[r:RATED]->(m)

SET r.rating = $rating,

r.timestamp = timestamp()

RETURN m {

.\*,

rating: r.rating

} AS movie

""", user\_id=user\_id, movie\_id=movie\_id, rating=rating).single()

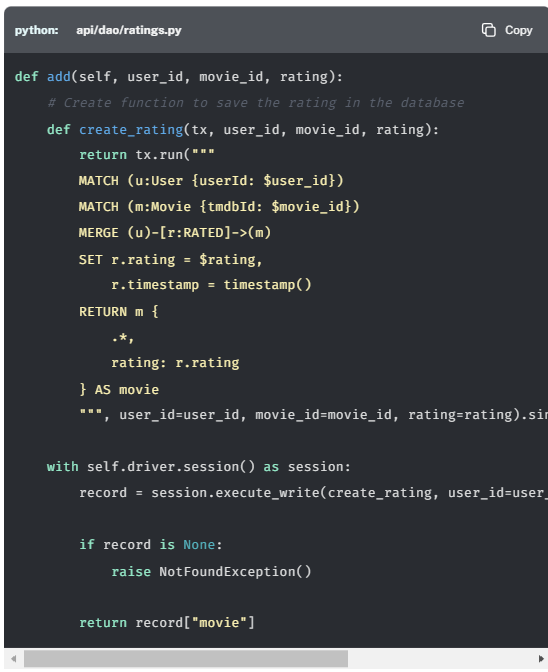
with self.driver.session() as session:

record = session.execute\_write(create\_rating, user\_id=user\_id, movie\_id=movie\_id, rating=rating)

if record is None:

raise NotFoundException()

return record["movie"]



## Testing

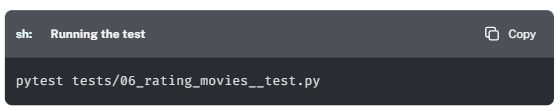
To test that this functionality has been correctly implemented, run the following code in a new terminal session:

**sh**

**Running the test**

Copy

pytest tests/06\_rating\_movies\_\_test.py



The test file is located at [tests/06\_rating\_movies\_\_test.py](https://github.com/neo4j-graphacademy/app-python/blob/main/tests/06_rating_movies__test.py).

Are you stuck? Click here for help

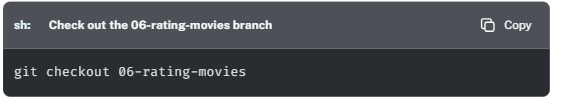
If you get stuck, you can see a working solution by checking out the 06-rating-movies branch by running:

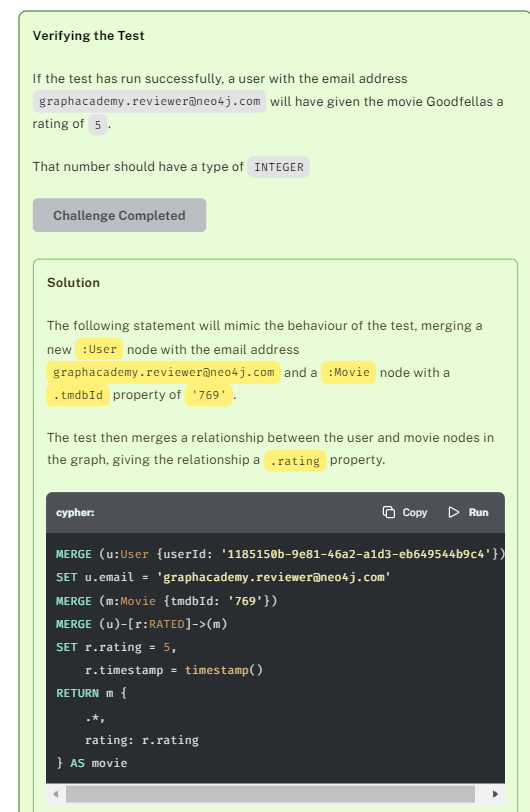
**sh**

**Check out the 06-rating-movies branch**

Copy

git checkout 06-rating-movies



****

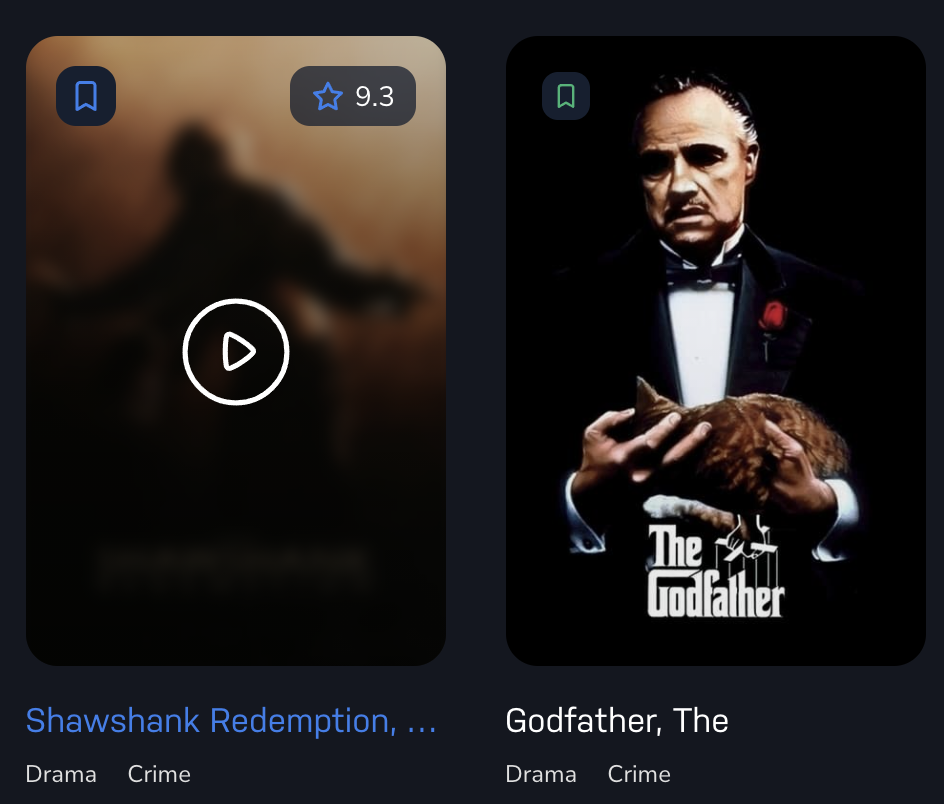
Bottom of Form

# My Favorites List (Challenge)

This challenge has three parts:

1. [**Adding a Movie to My Favorites**](https://graphacademy.neo4j.com/courses/app-python/2-interacting/10-user-favorites/#_adding_a_movie_to_my_favorites)
2. [**Removing a Movie from My Favorites**](https://graphacademy.neo4j.com/courses/app-python/2-interacting/10-user-favorites/#_removing_a_movie_from_my_favorites)
3. [**Listing My Favorites**](https://graphacademy.neo4j.com/courses/app-python/2-interacting/10-user-favorites/#_listing_my_favorites)

## The My Favorites Feature



Clicking the [**My Favorites**](http://localhost:3000/favorites) link in the main navigation will take you to page which contains a list of Movies that you can revisit later.

When a logged in user hovers their mouse cursor over a Movie on the website, a bookmark icon appears in the top right hand corner. Clicking this icon will either add or remove the Movie from the user’s Favorites list.

When adding a Movie to the list, a POST request it sent to the /api/favorites/{id} endpoint. When this happens, the following chain of events will occur:

1. The server directs the request to the route handler in [api/routes/account.py](https://github.com/neo4j-graphacademy/app-python/blob/main/api/routes/account.py" \t "_blank), which verifies the user’s JWT token before handling the request
2. The route handler creates an instance of the [FavoriteDAO](https://github.com/neo4j-graphacademy/app-python/blob/main/api/dao/favorites.py" \t "_blank).
3. The add() method is then called on the FavoriteDAO instance with the user’s ID taken from the JWT token, and the ID of the movie that has been extracted from the request URL.
4. It is then the responsibility of the FavoriteDAO to find the :User and :Movie nodes and create the :HAS\_FAVORITE relationship between them.

Likewise, when it is clicked for a Movie that has already been favorited, a DELETE request is sent to the same URL, and the Movie is removed from the list.

## Adding a Movie to My Favorites

For the first part of this challenge, modify the add() method to open a new database session, run the Cypher statement to create the :HAS\_FAVORITE relationship, close the session and return the movie details along with an additional favorite property.

[**Open**api/dao/auth.py**→**](https://graphacademy.neo4j.com/courses/app-python/2-interacting/10-user-favorites/lab/)

### 1. Define a new Transaction Function

Define a new transaction function to execute the Cypher statement below to create the :HAS\_FAVORITE relationship between the User and the Movie.

Click here to reveal the Cypher statement

**cypher**

CopyRun

MATCH (u:User {userId: $userId})

MATCH (m:Movie {tmdbId: $movieId})

MERGE (u)-[r:HAS\_FAVORITE]->(m)

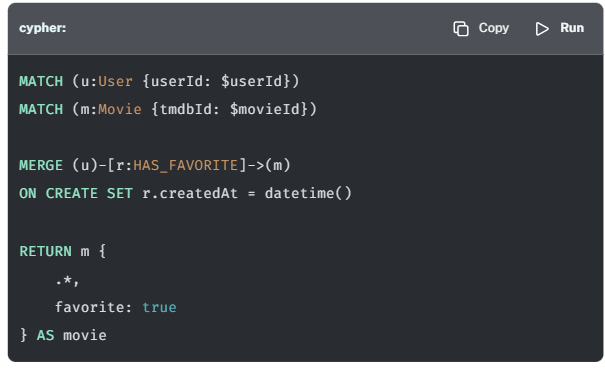
ON CREATE SET r.createdAt = datetime()

RETURN m {

.\*,

favorite: true

} AS movie



The function should expect a transaction object as the first parameter and also accept the user\_id and movie\_id as named parameters to the function.

There should only be one result, so you can call the single() method to instantly consume the first result.

**python**

**Run the Cypher statement**

Copy

*# Define a new transaction function to create a HAS\_FAVORITE relationship*

def add\_to\_favorites(tx, user\_id, movie\_id):

row = tx.run("""

MATCH (u:User {userId: $userId})

MATCH (m:Movie {tmdbId: $movieId})

MERGE (u)-[r:HAS\_FAVORITE]->(m)

ON CREATE SET u.createdAt = datetime()

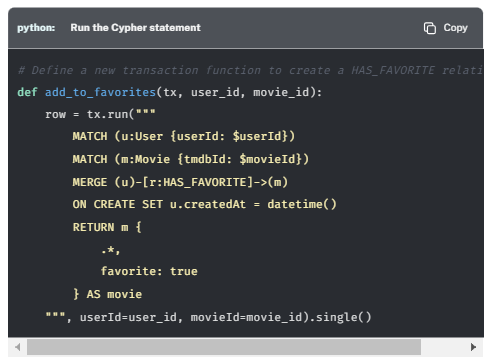
RETURN m {

.\*,

favorite: true

} AS movie

""", userId=user\_id, movieId=movie\_id).single()



If no records are returned, you can safely assume that the either the User or Movie do not exist. In this case, raise a NotFoundException with an appropriate error message.

**python**

Copy

*# If no rows are returnedm throw a NotFoundException*

if row == None:

raise NotFoundException()



Otherwise, use the get() method on the record to return the movie value from the first row.

**python**

Copy

return row.get("movie")



### 2. Call the Transaction Function

In a new session, call the new method in a new Write Transaction and return the results.

**python**

Copy

with self.driver.session() as session:

return session.execute\_write(add\_to\_favorites, user\_id, movie\_id)



### Working Solution

Click here to reveal the completed add() method

**python**

**api/dao/favorites.py**

Copy

def add(self, user\_id, movie\_id):

*# Define a new transaction function to create a HAS\_FAVORITE relationship*

def add\_to\_favorites(tx, user\_id, movie\_id):

row = tx.run("""

MATCH (u:User {userId: $userId})

MATCH (m:Movie {tmdbId: $movieId})

MERGE (u)-[r:HAS\_FAVORITE]->(m)

ON CREATE SET u.createdAt = datetime()

RETURN m {

.\*,

favorite: true

} AS movie

""", userId=user\_id, movieId=movie\_id).single()

*# If no rows are returnedm throw a NotFoundException*

if row == None:

raise NotFoundException()

return row.get("movie")

with self.driver.session() as session:

return session.execute\_write(add\_to\_favorites, user\_id, movie\_id)



## Removing a Movie from My Favorites

The second part of this challenge is to write the code to remove a movie from the My Favorites list.

The code for deleting the :HAS\_FAVORITE relationship will be similar, only the Cypher statement will change.

Instead of two separate MATCH clauses, we can instead attempt to find the pattern within a single clause. If the relationship (with an alias of r) exists, we will delete it and then return the movie information with favorite set to false.

Click here to reveal the Cypher statement

**cypher**

CopyRun

MATCH (u:User {userId: $userId})-[r:HAS\_FAVORITE]->(m:Movie {tmdbId: $movieId})

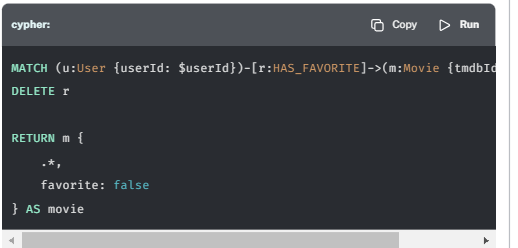
DELETE r

RETURN m {

.\*,

favorite: false

} AS movie



Use the code from the add() method above to implement the remove() function. If you get stuck, you can reveal the completed method below.

### Working Solution

Click here to reveal the completed remove() method

**python**

**api/dao/favorites.py**

Copy

def remove(self, user\_id, movie\_id):

*# Define a transaction function to delete the HAS\_FAVORITE relationship within a Write Transaction*

def remove\_from\_favorites(tx, user\_id, movie\_id):

row = tx.run("""

MATCH (u:User {userId: $userId})-[r:HAS\_FAVORITE]->(m:Movie {tmdbId: $movieId})

DELETE r

RETURN m {

.\*,

favorite: false

} AS movie

""", userId=user\_id, movieId=movie\_id).single()

*# If no rows are returnedm throw a NotFoundException*

if row == None:

raise NotFoundException()

return row.get("movie")

*# Execute the transaction function within a Write Transaction*

with self.driver.session() as session:

*# Return movie details and `favorite` property*

return session.execute\_write(remove\_from\_favorites, user\_id, movie\_id)



## Listing My Favorites

Finally, the all() method in the FavoriteDAO currently returns a hardcoded list of Movies.

**python**

**api/dao/favorites.py**

Copy

def all(self, user\_id, sort = 'title', order = 'ASC', limit = 6, skip = 0):

*# TODO: Open a new session*

*# TODO: Retrieve a list of movies favorited by the user*

return popular



Update this method to return a paginated list of movies that the user has added to their My Favorites list.

Click here to reveal the Cypher statement

**cypher**

CopyRun

MATCH (u:User {userId: $userId})

MATCH (m:Movie {tmdbId: $movieId})

MERGE (u)-[r:HAS\_FAVORITE]->(m)

ON CREATE SET r.createdAt = datetime()

RETURN m {

.\*,

favorite: true

} AS movie



This time, as the query is simpler, you can use a lambda function to represent the transaction function.

**python**

Copy

*# Retrieve a list of movies favorited by the user*

movies = session.execute\_read(lambda tx: tx.run("""

MATCH (u:User {{userId: $userId}})-[r:HAS\_FAVORITE]->(m:Movie)

RETURN m {{

.\*,

favorite: true

}} AS movie

ORDER BY m.`{0}` {1}

SKIP $skip

LIMIT $limit

""".format(sort, order), userId=user\_id, limit=limit, skip=skip).value("movie"))



**Escaped Braces**

You may have noticed that the code block above features double curly braces ({{ and }}) within the MATCH clause rather than the single braces used within the Cypher statement.

MATCH (u:User {{userId: $userId}})

Braces need to be escaped within a Python string, and we do this by using double quotes.

Click here to reveal the completed all() method

**python**

**api/dao/favorites.py**

Copy

def all(self, user\_id, sort = 'title', order = 'ASC', limit = 6, skip = 0):

*# Open a new session*

with self.driver.session() as session:

*# Retrieve a list of movies favorited by the user*

movies = session.execute\_read(lambda tx: tx.run("""

MATCH (u:User {{userId: $userId}})-[r:HAS\_FAVORITE]->(m:Movie)

RETURN m {{

.\*,

favorite: true

}} AS movie

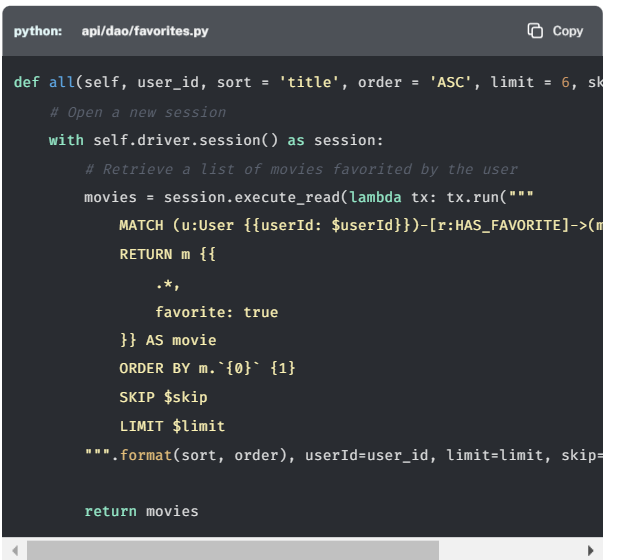
ORDER BY m.`{0}` {1}

SKIP $skip

LIMIT $limit

""".format(sort, order), userId=user\_id, limit=limit, skip=skip).value("movie"))

return movies



## Testing

To test that this functionality has been correctly implemented, run the following code in a new terminal session:

**sh**

**Running the test**

Copy

pytest tests/07\_favorites\_list\_\_test.py



The test file is located at [tests/07\_favorites\_list\_\_test.py](https://github.com/neo4j-graphacademy/app-python/blob/main/tests/07_favorites_list__test.py).

Are you stuck? Click here for help

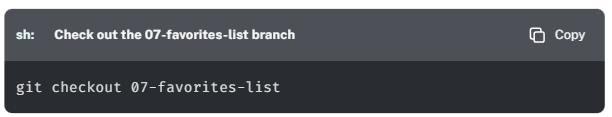
If you get stuck, you can see a working solution by checking out the 07-favorites-list branch by running:

**sh**

**Check out the 07-favorites-list branch**

Copy

git checkout 07-favorites-list



Bottom of Form



# Adding The Favorite Flag (Challenge)

In the previous Challenge, we hardcoded a favorite property on the return to the calls in the FavoriteDAO. We did this because we could guarantee the value based on the query being executed. This won’t be possible in the MovieDAO.

Your next challenge is to modify the [MovieDAO](https://github.com/neo4j-graphacademy/app-python/blob/main/api/dao/movies.py) to dynamically determine whether a movie is on the current user’s My Favorite List.

This challenge has two parts:

1. [**Finding Favorites**](https://graphacademy.neo4j.com/courses/app-python/2-interacting/11-favorite-flag/#_finding_favorites)
2. Implement [**Multiple Transaction Calls**](https://graphacademy.neo4j.com/courses/app-python/2-interacting/11-favorite-flag/#_multiple_transaction_calls)

## Running Multiple Queries within a Transaction

So far, you have only used the tx object within the Unit of Work to run a single query.

This will be fine for the majority of cases, but there may also be scenarios where more than one Query may be required.

### User Favorites

One way that we could find the user’s favorites would be to run a separate MATCH clause within the same query. But as all of the routes that interact with the MovieDAO are used by both anonymous and logged in users, this could add unwanted complexity to the service.

Instead, it would be cleaner to execute two separate queries within the same transaction; one query to fetch a list of the user’s favorites, and another to retrieve a list of movies.

Fortunately, with only a few minor tweaks to the code, we can create a method that can be used to populate the favorite flag in the every other method in the MovieDAO.

## Creating a Reusable Method

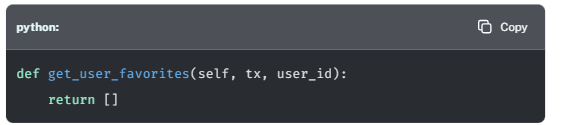
At the bottom of the MovieDAO, a placeholder get\_user\_favorites() method exists which is currently hardcoded to return an empty array.

**python**

Copy

def get\_user\_favorites(self, tx, user\_id):

return []



The purpose of this function is to run a Cypher statement against the Transaction object passed as the first parameter, which will find all of the user’s favorite movies and return a list of tmdbId properties.

Your challenge is to modify this method to retrieve that list of Movie ID’s and then call this function from the Read Transaction in the all() method.

### Finding Favorites

Modify the get\_user\_favorites() method to run the following query against the tx object to return a list of IDs for each movie that the user has added to their favorites.

Click here to reveal the Cypher statement

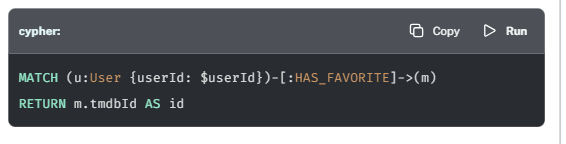
This query should only run if a user is logged in for the current request, and therefore the userId parameter is not None.

**cypher**

CopyRun

MATCH (u:User {userId: $userId})-[:HAS\_FAVORITE]->(m)

RETURN m.tmdbId AS id



#### Working Solution

Click here to reveal the completed get\_user\_favorites() method

.

**python**

Copy

def get\_user\_favorites(self, tx, user\_id):

if user\_id == None:

return []

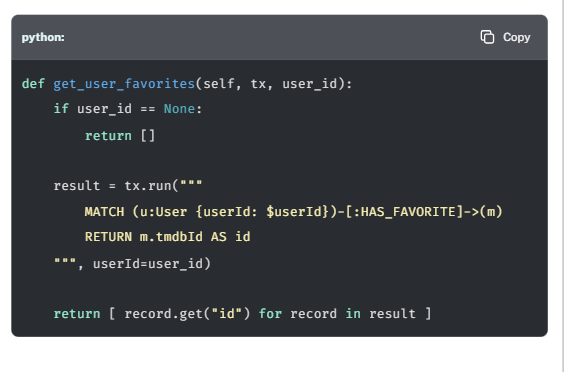
result = tx.run("""

MATCH (u:User {userId: $userId})-[:HAS\_FAVORITE]->(m)

RETURN m.tmdbId AS id

""", userId=user\_id)

return [ record.get("id") for record in result ]



### Multiple Transaction Calls

The get\_movies function defined in the all method of the MoviesDAO already uses tx.run() to run a Cypher statement within the current transaction.

To use the list of Movie IDs in the query, you can simply add a call to the self.get\_user\_favorites() method above the cypher variable. The output of the function can then be passed as a named parameter in the existing call to tx.run().

All that is left to do is to add a new item to m { .\* } in the Cypher statement to check whether the tmdbId property of the current movie is in the favorites array.

**python**

Copy

*# Get User favorites*

favorites = self.get\_user\_favorites(tx, user\_id)

*# Define the cypher statement*

cypher = """

MATCH (m:Movie)

WHERE m.`{0}` IS NOT NULL

RETURN m {{

.\*,

favorite: m.tmdbId IN $favorites

}} AS movie

ORDER BY m.`{0}` {1}

SKIP $skip

LIMIT $limit

""".format(sort, order)

*# Run the statement within the transaction passed as the first argument*

result = tx.run(cypher, limit=limit, skip=skip, user\_id=user\_id, favorites=favorites)



### Comparing Versions

If we take a look at the two versions of the all() method, not much has changed. The favorites array has been passed through as a parameter to the query, and the query now uses the Cypher IN clause to check if the ID is included in the array.

Example 1. Updated Method

**python**

**api/dao/movies.py**

Copy

*# Get User favorites*

favorites = self.get\_user\_favorites(tx, user\_id)

*# Define the cypher statement*

cypher = """

MATCH (m:Movie)

WHERE m.`{0}` IS NOT NULL

RETURN m {{

.\*,

favorite: m.tmdbId IN $favorites

}} AS movie

ORDER BY m.`{0}` {1}

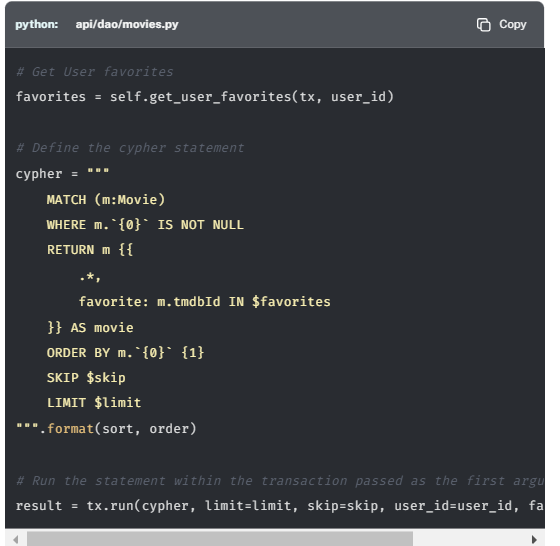
SKIP $skip

LIMIT $limit

""".format(sort, order)

*# Run the statement within the transaction passed as the first argument*

result = tx.run(cypher, limit=limit, skip=skip, user\_id=user\_id, favorites=favorites)



Example 2. Previous Version

**python**

**api/dao/movies.py**

Copy

*# Define the cypher statement*

cypher = """

MATCH (m:Movie)

WHERE m.`{0}` IS NOT NULL

RETURN m {{ .\* }} AS movie

ORDER BY m.`{0}` {1}

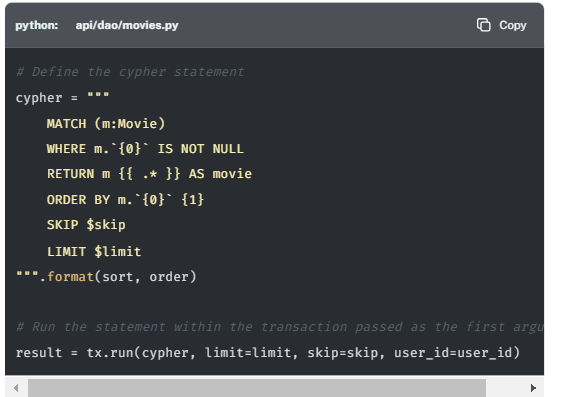
SKIP $skip

LIMIT $limit

""".format(sort, order)

*# Run the statement within the transaction passed as the first argument*

result = tx.run(cypher, limit=limit, skip=skip, user\_id=user\_id)



## Testing

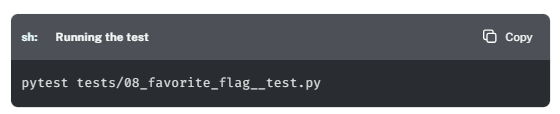
To test that this functionality has been correctly implemented, run the following code in a new terminal session:

**sh**

**Running the test**

Copy

pytest tests/08\_favorite\_flag\_\_test.py



The test file is located at [tests/08\_favorite\_flag\_\_test.py](https://github.com/neo4j-graphacademy/app-python/blob/main/tests/08_favorite_flag__test.py).

Are you stuck? Click here for help

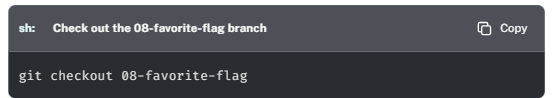
If you get stuck, you can see a working solution by checking out the 08-favorite-flag branch by running:

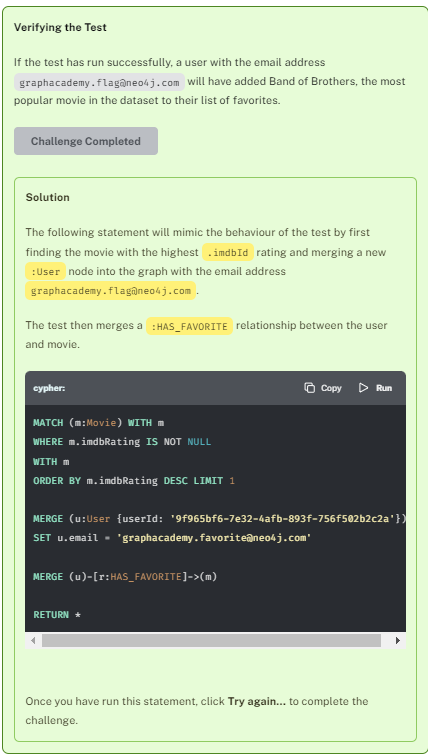
**sh**

**Check out the 08-favorite-flag branch**

Copy

git checkout 08-favorite-flag



****

# Module 4: Project Backlog

Now that we have covered everything that we need to know, you can practise your skills by implementing the remainder of the backlog. There is still functionality to be added to the genre listings, movie listings and people directory.

This module will be slightly different from the previous one.

Instead of an in-depth explanation, you will only be provided with the type of transaction and a Cypher statement. It will then be up to you to use your knowledge to complete the challenge.

If you get stuck at any point, you can reveal the solution under the **Working Solution** header. You can also checkout the relevant branch from the repository at any time.

**All of the lessons in this module are optional and do not count towards your achievement.**

Bottom of Form

# Browsing Genres (Challenge)

**Optional Lesson**

This lesson is optional and will not count towards your achievement. To view the completed code, check out the 09-genre-list branch.

If you click on the **Genres** link in the main navigation, you will be taken to a [**list of Genres**](http://localhost:3000/genres). This list is populated by the API route at [**http://localhost:3000/api/genres**](http://localhost:3000/api/genres), with the list being produced by the all() method within the [GenreDAO](https://github.com/neo4j-graphacademy/app-python/blob/main/api/dao/genres.py" \t "_blank).

**python**

**api/dao/genres.py**

Copy

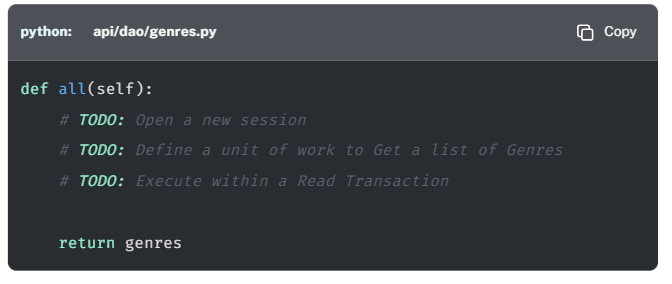
def all(self):

*# TODO: Open a new session*

*# TODO: Define a unit of work to Get a list of Genres*

*# TODO: Execute within a Read Transaction*

return genres



In this challenge, you will modify the method to run the following Cypher statement in a **read transaction**:

**cypher**

**Listing Genres**

CopyRun

MATCH (g:Genre)

WHERE g.name <> '(no genres listed)'

CALL {

WITH g

MATCH (g)<-[:IN\_GENRE]-(m:Movie)

WHERE m.imdbRating IS NOT NULL AND m.poster IS NOT NULL

RETURN m.poster AS poster

ORDER BY m.imdbRating DESC LIMIT 1

}

RETURN g {

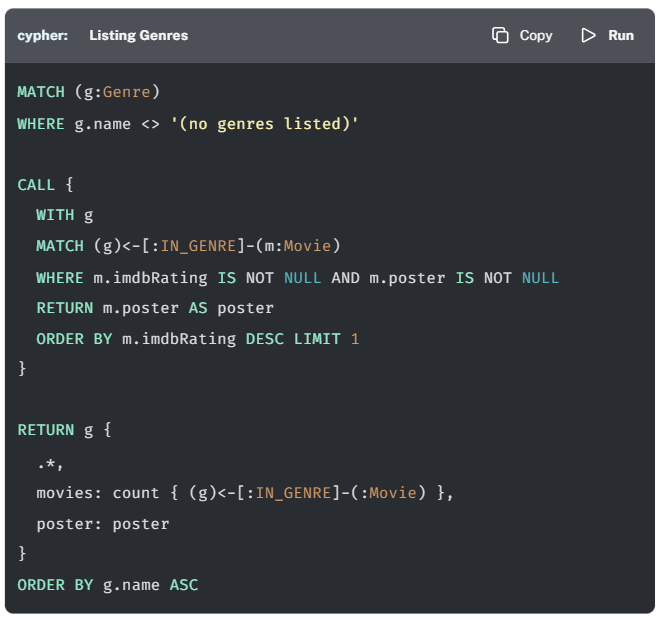
.\*,

movies: count { (g)<-[:IN\_GENRE]-(:Movie) },

poster: poster

}

ORDER BY g.name ASC



## Your Task

* Modify the all() method on the GenreDAO to query Neo4j and return a list of properties ordered by genre name.
* Each genre should have a name, the number of movies listed in that genre (movies), and a poster image (from the most popular movie in that genre).

## Working Solution (incorrect cypher query)

(NOTE: this query mentioned in this solution is incorrect, replace it with the cypher query mentioned just right before) (the only change is the use of count() instead of size()

Click here to view the completed all() method.

**python**

**api/dao/genres.py**

Copy

def all(self):

*# Define a unit of work to Get a list of Genres*

def get\_movies(tx):

result = tx.run("""

MATCH (g:Genre)

WHERE g.name <> '(no genres listed)'

CALL {

WITH g

MATCH (g)<-[:IN\_GENRE]-(m:Movie)

WHERE m.imdbRating IS NOT NULL AND m.poster IS NOT NULL

RETURN m.poster AS poster

ORDER BY m.imdbRating DESC LIMIT 1

}

RETURN g {

.\*,

movies: size((g)<-[:IN\_GENRE]-(:Movie)),

poster: poster

} AS genre

ORDER BY g.name ASC

""")

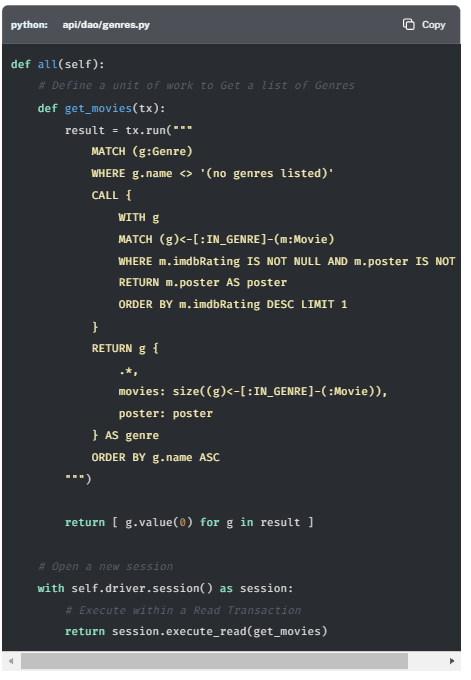
return [ g.value(0) for g in result ]

*# Open a new session*

with self.driver.session() as session:

*# Execute within a Read Transaction*

return session.execute\_read(get\_movies)



## Testing

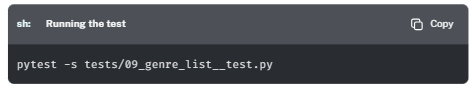
To test that this functionality has been correctly implemented, run the following code in a new terminal session:

**sh**

**Running the test**

Copy

pytest -s tests/09\_genre\_list\_\_test.py



The test file is located at [tests/09\_genre\_list\_\_test.py](https://github.com/neo4j-graphacademy/app-python/blob/main/tests/09_genre_list__test.py).

Are you stuck? Click here for help

If you get stuck, you can see a working solution by checking out the 09-genre-list branch by running:

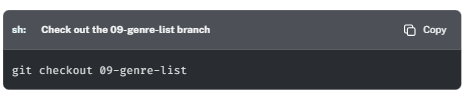
**sh**

**Check out the 09-genre-list branch**

Copy

git checkout 09-genre-list

Bottom of Form



# Finding Genre Details (Challenge)

**Optional Lesson**

This lesson is optional and will not count towards your achievement. To view the completed code, check out the 10-genre-details branch.

When the user clicks a genre in the list, they are taken to a list of movies for that genre. This list is populated by an API request to /api/genres/[name], for example [**http://localhost:3000/api/genres/Comedy**](http://localhost:3000/api/genres/Comedy).

The find() method the the GenreDAO accepts one argument, the name of the genre, and should return the information about the genre, along with a count of movies and a poster image.

**python**

**api/dao/genres.py**

Copy

def find(self, name):

*# TODO: Open a new session*

*# TODO: Define a unit of work to find the genre by it's name*

*# TODO: Execute within a Read Transaction*

return [g for g in genres if g["name"] == name][0]



In this challenge, you will modify the method to run the following Cypher statement in a **read transaction**:

**cypher**

**Finding Genre Details**

CopyRun

MATCH (g:Genre {name: $name})<-[:IN\_GENRE]-(m:Movie)

WHERE m.imdbRating IS NOT NULL AND m.poster IS NOT NULL AND g.name <> '(no genres listed)'

WITH g, m

ORDER BY m.imdbRating DESC

WITH g, head(collect(m)) AS movie

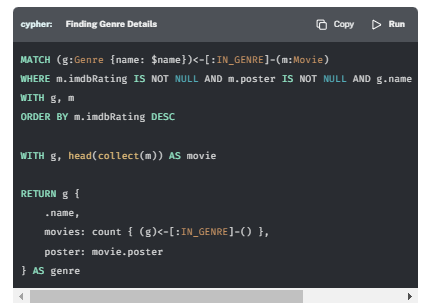
RETURN g {

.name,

movies: count { (g)<-[:IN\_GENRE]-() },

poster: movie.poster

} AS genre



**What does this query do?**

This query uses the MATCH clause for a Genre node with the name passed through with the function call as a parameter. The query then finds the highest rated movie with a poster property and uses that image as the background to the card in the UI.

The size() function uses a precalculated value stored against the Genre node to return the number of incoming IN\_GENRE relationships the Genre node.

## Your Task

* Modify the find() method on the GenreDAO to call the Neo4j database and return details for a genre.
* The name variable should be passed to the run() call as a parameter.
* If no records are found, a NotFoundException should be raised.

## Working Solution (incorrect cypher query)

(NOTE: this query mentioned in this solution is incorrect, replace it with the cypher query mentioned just right before) (the only change is the use of count() instead of size()

Click here to view the completed find() method.

**python**

**src/services/genre/service.js**

Copy

def find(self, name):

*# Define a unit of work to find the genre by it's name*

def find\_genre(tx, name):

first = tx.run("""

MATCH (g:Genre {name: $name})<-[:IN\_GENRE]-(m:Movie)

WHERE m.imdbRating IS NOT NULL AND m.poster IS NOT NULL AND g.name <> '(no genres listed)'

WITH g, m

ORDER BY m.imdbRating DESC

WITH g, head(collect(m)) AS movie

RETURN g {

.name,

movies: size((g)<-[:IN\_GENRE]-()),

poster: movie.poster

} AS genre

""", name=name).single()

*# If no records are found raise a NotFoundException*

if first == None:

raise NotFoundException()

return first.get("genre")

*# Open a new session*

with self.driver.session() as session:

*# Execute within a Read Transaction*

return session.execute\_read(find\_genre, name)



## Testing

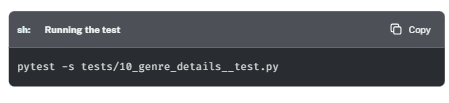
To test that this functionality has been correctly implemented, run the following code in a new terminal session:

**sh**

**Running the test**

Copy

pytest -s tests/10\_genre\_details\_\_test.py



The test file is located at [tests/10\_genre\_details\_\_test.py](https://github.com/neo4j-graphacademy/app-python/blob/main/tests/10_genre_details__test.py).

Are you stuck? Click here for help

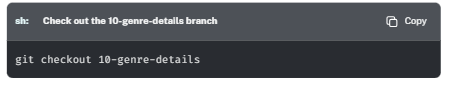
If you get stuck, you can see a working solution by checking out the 10-genre-details branch by running:

**sh**

**Check out the 10-genre-details branch**

Copy

git checkout 10-genre-details



# Movie Lists and Pagination (Challenge)

**Optional Lesson**

This lesson is optional and will not count towards your achievement. To view the completed code, check out the 11-movie-lists branch.

In this challenge, you will implement the remaining methods in the [MovieDAO](https://github.com/neo4j-graphacademy/app-python/blob/main/api/dao/movies.py" \t "_blank) for retrieving a list of movies:

* [**get\_by\_genre()**](https://graphacademy.neo4j.com/courses/app-python/3-backlog/3-movie-lists/#get_by_genre) - should return a paginated list of movies that are listed in a particular Genre
* [**get\_for\_actor()**](https://graphacademy.neo4j.com/courses/app-python/3-backlog/3-movie-lists/#get_for_actor) - should return a paginated list of movies that a particular Person has acted in
* [**get\_for\_director()**](https://graphacademy.neo4j.com/courses/app-python/3-backlog/3-movie-lists/#get_for_director) - should return a paginated list of movies that a particular Person has directed

These methods are very similar to the all() method which returns a full list, but with a different pattern in the MATCH clause.

For each subtask, we will provide you with the pattern required to run in the Cypher statement, plus any additional parameters that are required for the query to run.

All you need to do is take the code from the all() method of the MovieDAO and modify the pattern in the first line of the Cypher statement in the first argument of the tx.run() call, then if necessary add the additional parameters to the second argument.

If you get stuck at any point, you can click to reveal the completed solution or skip to the bottom for instructions on how to checkout the branch with the working solution.

## get\_by\_genre()

The get\_by\_genre() method returns a paginated list of movies that are listed within a certain genre.

**python**

**api/dao/movies.py**

Copy

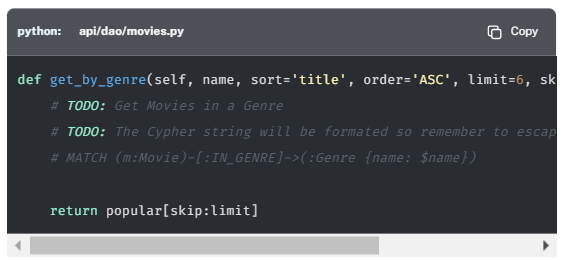
def get\_by\_genre(self, name, sort='title', order='ASC', limit=6, skip=0, user\_id=None):

*# TODO: Get Movies in a Genre*

*# TODO: The Cypher string will be formated so remember to escape the braces: {{name: $name}}*

*# MATCH (m:Movie)-[:IN\_GENRE]->(:Genre {name: $name})*

return popular[skip:limit]



Update the get\_by\_genre() method to use the following Cypher Pattern:

**cypher**

CopyRun

MATCH (m:Movie)-[:IN\_GENRE]->(:Genre {name: $name})



The tx.run() call will need to include the name property, which represents the name of the genre.

Click to reveal the completed get\_by\_genre() method

Find the get\_by\_genre() method in api/dao/movies.py and modify the function to find a list of movies by genre.

**python**

**api/dao/movies.py**

Copy

def get\_by\_genre(self, name, sort='title', order='ASC', limit=6, skip=0, user\_id=None):

*# Get Movies in a Genre*

def get\_movies\_in\_genre(tx, sort, order, limit, skip, user\_id):

favorites = self.get\_user\_favorites(tx, user\_id)

cypher = """

MATCH (m:Movie)-[:IN\_GENRE]->(:Genre {{name: $name}})

WHERE m.`{0}` IS NOT NULL

RETURN m {{

.\*,

favorite: m.tmdbId in $favorites

}} AS movie

ORDER BY m.`{0}` {1}

SKIP $skip

LIMIT $limit

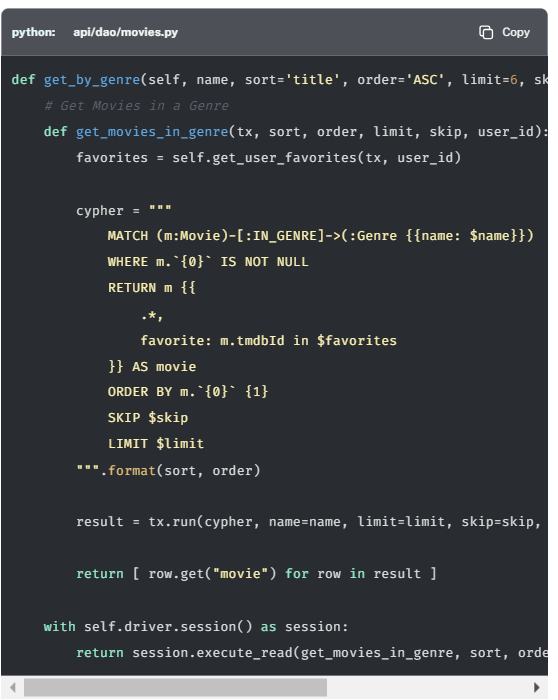
""".format(sort, order)

result = tx.run(cypher, name=name, limit=limit, skip=skip, user\_id=user\_id, favorites=favorites)

return [ row.get("movie") for row in result ]

with self.driver.session() as session:

return session.execute\_read(get\_movies\_in\_genre, sort, order, limit=limit, skip=skip, user\_id=user\_id)



## get\_for\_actor()

The get\_for\_actor() method returns a paginated list of movies that a person has acted in.

**python**

**api/dao/movies.py**

Copy

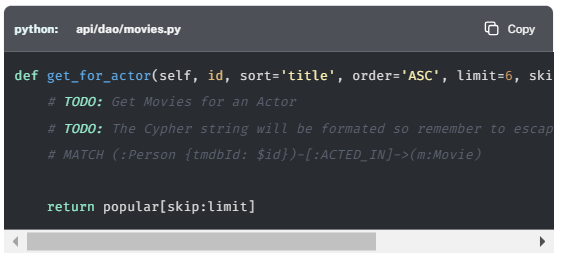
def get\_for\_actor(self, id, sort='title', order='ASC', limit=6, skip=0, user\_id=None):

*# TODO: Get Movies for an Actor*

*# TODO: The Cypher string will be formated so remember to escape the braces: {{tmdbId: $id}}*

*# MATCH (:Person {tmdbId: $id})-[:ACTED\_IN]->(m:Movie)*

return popular[skip:limit]



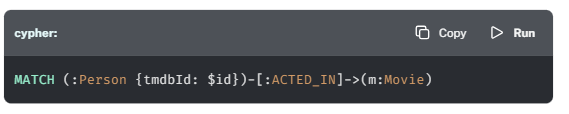
Find the get\_for\_actor() method in api/dao/movies.py and modify the function to find a list of movies by actor.

The pattern required in the MATCH clause will be:

**cypher**

CopyRun

MATCH (:Person {tmdbId: $id})-[:ACTED\_IN]->(m:Movie)



You will have to include the additional parameter id - The tmdbId property relating to the actor.

Click to reveal the completed getForActor() method

**python**

**api/dao/movies.py**

Copy

def get\_for\_actor(self, id, sort='title', order='ASC', limit=6, skip=0, user\_id=None):

*# Get Movies for an Actor*

def get\_movies\_for\_actor(tx, id, sort, order, limit, skip, user\_id):

favorites = self.get\_user\_favorites(tx, user\_id)

cypher = """

MATCH (:Person {{tmdbId: $id}})-[:ACTED\_IN]->(m:Movie)

WHERE m.`{0}` IS NOT NULL

RETURN m {{

.\*,

favorite: m.tmdbId in $favorites

}} AS movie

ORDER BY m.`{0}` {1}

SKIP $skip

LIMIT $limit

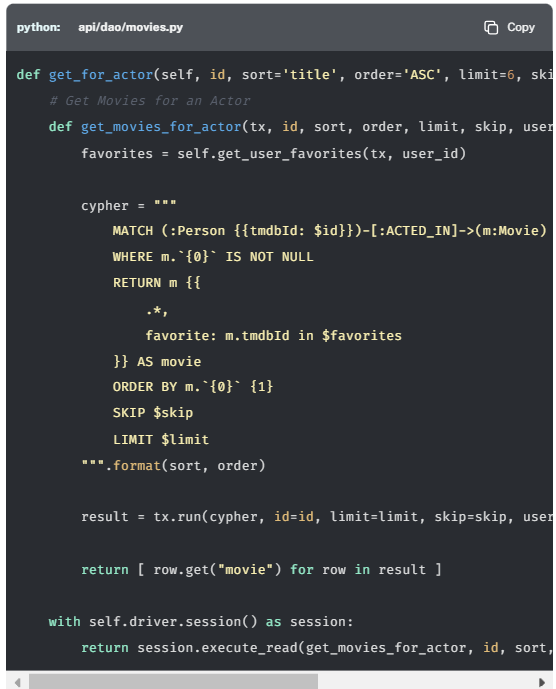
""".format(sort, order)

result = tx.run(cypher, id=id, limit=limit, skip=skip, user\_id=user\_id, favorites=favorites)

return [ row.get("movie") for row in result ]

with self.driver.session() as session:

return session.execute\_read(get\_movies\_for\_actor, id, sort, order, limit=limit, skip=skip, user\_id=user\_id)



## get\_for\_director()

The get\_for\_director() method returns a paginated list of movies that a person has directed.

**python**

**api/dao/movies.py**

Copy

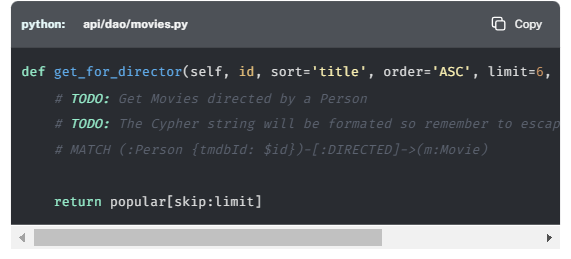
def get\_for\_director(self, id, sort='title', order='ASC', limit=6, skip=0, user\_id=None):

*# TODO: Get Movies directed by a Person*

*# TODO: The Cypher string will be formated so remember to escape the braces: {{name: $name}}*

*# MATCH (:Person {tmdbId: $id})-[:DIRECTED]->(m:Movie)*

return popular[skip:limit]



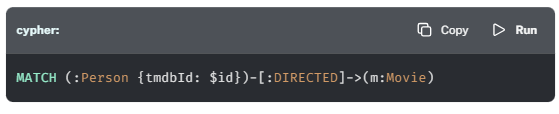
Find the getForDirector() method in api/dao/movies.py and modify the function to find a list of movies by director.

The pattern required in the MATCH clause will be:

**cypher**

CopyRun

MATCH (:Person {tmdbId: $id})-[:DIRECTED]->(m:Movie)



You will have to include the additional parameter id - The tmdbId property relating to the director.

Click to reveal the completed get\_for\_director() method

**python**

**api/dao/movies.py**

Copy

def get\_for\_director(self, id, sort='title', order='ASC', limit=6, skip=0, user\_id=None):

*# Get Movies directed by a Person*

def get\_movies\_for\_director(tx, id, sort, order, limit, skip, user\_id):

favorites = self.get\_user\_favorites(tx, user\_id)

cypher = """

MATCH (:Person {{tmdbId: $id}})-[:DIRECTED]->(m:Movie)

WHERE m.`{0}` IS NOT NULL

RETURN m {{

.\*,

favorite: m.tmdbId in $favorites

}} AS movie

ORDER BY m.`{0}` {1}

SKIP $skip

LIMIT $limit

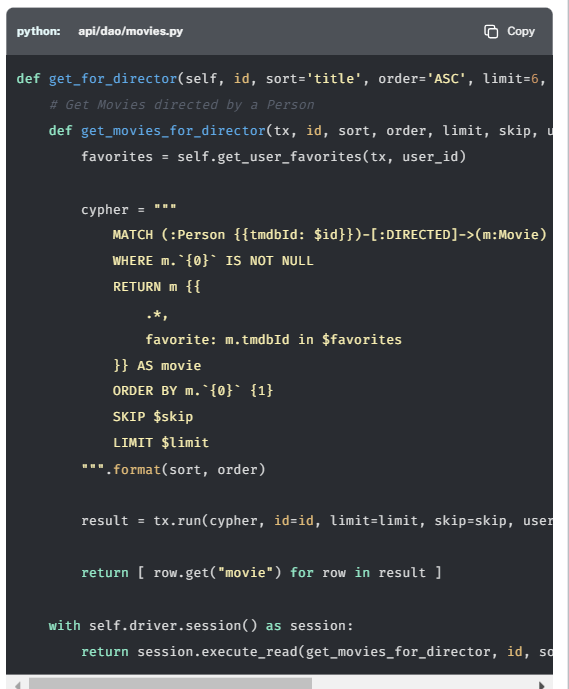
""".format(sort, order)

result = tx.run(cypher, id=id, limit=limit, skip=skip, user\_id=user\_id, favorites=favorites)

return [ row.get("movie") for row in result ]

with self.driver.session() as session:

return session.execute\_read(get\_movies\_for\_director, id, sort, order, limit=limit, skip=skip, user\_id=user\_id)



## Testing

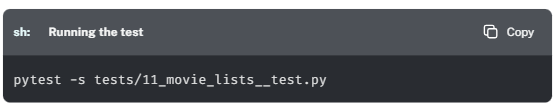
To test that this functionality has been correctly implemented, run the following code in a new terminal session:

**sh**

**Running the test**

Copy

pytest -s tests/11\_movie\_lists\_\_test.py



The test file is located at [tests/11\_movie\_lists\_\_test.py](https://github.com/neo4j-graphacademy/app-python/blob/main/tests/11_movie_lists__test.py).

Are you stuck? Click here for help

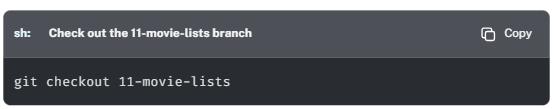
If you get stuck, you can see a working solution by checking out the 11-movie-lists branch by running:

**sh**

**Check out the 11-movie-lists branch**

Copy

git checkout 11-movie-lists



# Movie Details (Challenge)

**Optional Lesson**

This lesson is optional and will not count towards your achievement. To view the completed code, check out the 12-movie-details branch.

There are two methods remaining in the [MovieDAO](https://github.com/neo4j-graphacademy/app-python/blob/main/api/dao/movies.py" \t "_blank) that are currently returning hardcoded data.

* [**find\_by\_id()**](https://graphacademy.neo4j.com/courses/app-python/3-backlog/4-movie-view/#find_by_id) - should return information about a movie, including a list of actors, director, and genres.
* [**get\_similar\_movies()**](https://graphacademy.neo4j.com/courses/app-python/3-backlog/4-movie-view/#get_similar_movies) - should return a list of similar movies, ordered by a score generated by Neoflix’s recommendation algorithm.

In this challenge, you will update these methods to query Neo4j.

First, let’s take a look at how these methods are used.

## Movie Page

If you click on any movie, you’ll see that the API currently only returns information about the film Goodfellas.

The page itself is populated by three API calls:

1. The details about the movie are loaded via the api/movies/{id} endpoint. This endpoint gets its data from the **[find\_by\_id()](https://graphacademy.neo4j.com/courses/app-python/3-backlog/4-movie-view/" \l "find_by_id)** method.
2. The similar movies list is loaded by a call to the api/movies/{id}/similar endpoint, which gets its data from the **[get\_similar\_movies()](https://graphacademy.neo4j.com/courses/app-python/3-backlog/4-movie-view/" \l "get_similar_movies)** method.
3. The ratings on the right hand side of the page are loaded by a call to the api/movies/{id}/ratings endpoint. You will update this method in the next lesson.

## find\_by\_id()

Movie information is retrieved by the find\_by\_id() method.

**python**

**api/dao/movies.py**

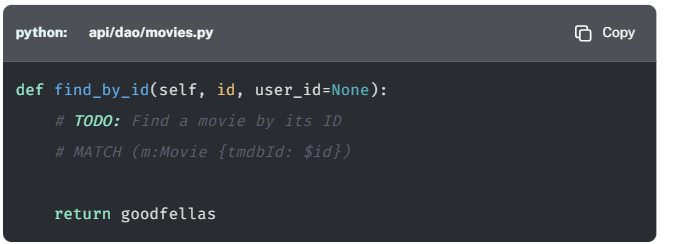
Copy

def find\_by\_id(self, id, user\_id=None):

*# TODO: Find a movie by its ID*

*# MATCH (m:Movie {tmdbId: $id})*

return goodfellas



Most of the important information is held as properties on the Movie node, but the payload should also return some additional information. We can use a Cypher subquery and the count() aggregate function to get the number of ratings for the movie and use a [**Pattern Comprehension**](https://neo4j.com/docs/cypher-manual/current/values-and-types/lists/#cypher-pattern-comprehension) to get a list of actors and directors.

As with the other movie methods in the MovieDAO, we should also provide a favorite value to represent whether the user has added this movie to their My Favorites list or not.

The following Cypher statement should be run within a **read transaction**:

**cypher**

**Movie Details**

CopyRun

MATCH (m:Movie {tmdbId: $id})

RETURN m {

.\*,

actors: [ (a)-[r:ACTED\_IN]->(m) | a { .\*, role: r.role } ],

directors: [ (d)-[:DIRECTED]->(m) | d { .\* } ],

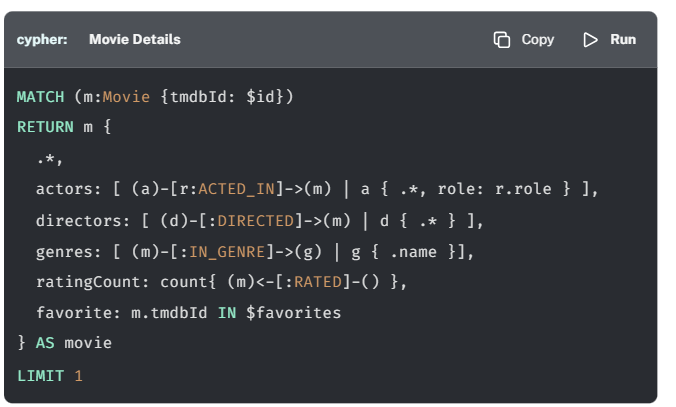
genres: [ (m)-[:IN\_GENRE]->(g) | g { .name }],

ratingCount: count{ (m)<-[:RATED]-() },

favorite: m.tmdbId IN $favorites

} AS movie

LIMIT 1



### Your Task

* Modify the find\_by\_id() method to query Neo4j and return details for the requested movie.
* The returned object should include a list of actors, directors, genres, and a boolean flag to represent whether the movie exists on the current user’s My Favorites List.
* If no records are returned, the method should raise a NotFoundException.

Click to reveal the completed find\_by\_id() method

### Working Solution

**python**

**api/dao/movies.py**

Copy

def find\_by\_id(self, id, user\_id=None):

*# Find a movie by its ID*

def find\_movie\_by\_id(tx, id, user\_id = None):

favorites = self.get\_user\_favorites(tx, user\_id)

cypher = """

MATCH (m:Movie {tmdbId: $id})

RETURN m {

.\*,

actors: [ (a)-[r:ACTED\_IN]->(m) | a { .\*, role: r.role } ],

directors: [ (d)-[:DIRECTED]->(m) | d { .\* } ],

genres: [ (m)-[:IN\_GENRE]->(g) | g { .name }],

favorite: m.tmdbId IN $favorites

} AS movie

LIMIT 1

"""

first = tx.run(cypher, id=id, favorites=favorites).single()

if first == None:

raise NotFoundException()

return first.get("movie")

with self.driver.session() as session:

return session.execute\_read(find\_movie\_by\_id, id, user\_id)



## get\_similar\_movies()

Similar movies are found using the get\_similar\_movies() method.

To provide a simple set of similar movies, the Cypher statement below uses the number of neighbors in common and their IMDB rating to generate a similarity score.

The following Cypher statement should be run within a **read transaction**:

**cypher**

**Find Similar Movies**

CopyRun

MATCH (:Movie {tmdbId: $id})-[:IN\_GENRE|ACTED\_IN|DIRECTED]->()<-[:IN\_GENRE|ACTED\_IN|DIRECTED]-(m)

WHERE m.imdbRating IS NOT NULL

WITH m, count(\*) AS inCommon

WITH m, inCommon, m.imdbRating \* inCommon AS score

ORDER BY score DESC

SKIP $skip

LIMIT $limit

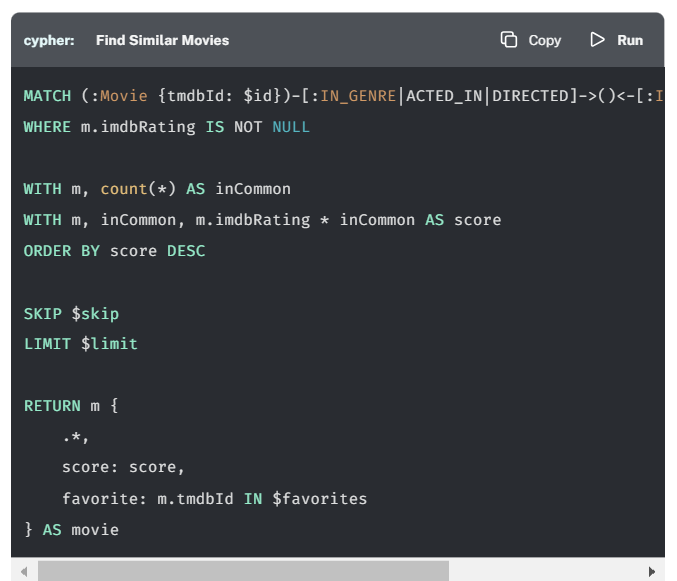
RETURN m {

.\*,

score: score,

favorite: m.tmdbId IN $favorites

} AS movie



### Your Task

* Modify the get\_similar\_movies() method to query Neo4j and return a list of similar movies.
* The returned objects should include a list of actors, directors, genres, and a boolean flag to represent whether the movie exists on the current user’s My Favorites List.

Click here to reveal the completed get\_similar\_movies() method

### Working Solution

**python**

**api/dao/movies.py**

Copy

def get\_similar\_movies(self, id, limit=6, skip=0, user\_id=None):

*# Get similar movies*

def find\_similar\_movies(tx, id, limit, skip, user\_id):

favorites = self.get\_user\_favorites(tx, user\_id)

cypher = """

MATCH (:Movie {tmdbId: $id})-[:IN\_GENRE|ACTED\_IN|DIRECTED]->()<-[:IN\_GENRE|ACTED\_IN|DIRECTED]-(m)

WHERE m.imdbRating IS NOT NULL

WITH m, count(\*) AS inCommon

WITH m, inCommon, m.imdbRating \* inCommon AS score

ORDER BY score DESC

SKIP $skip

LIMIT $limit

RETURN m {

.\*,

score: score,

favorite: m.tmdbId IN $favorites

} AS movie

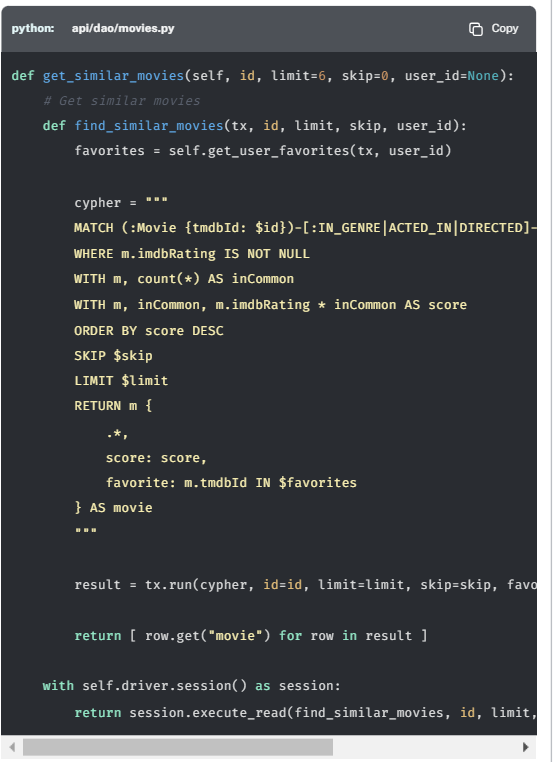
"""

result = tx.run(cypher, id=id, limit=limit, skip=skip, favorites=favorites)

return [ row.get("movie") for row in result ]

with self.driver.session() as session:

return session.execute\_read(find\_similar\_movies, id, limit, skip, user\_id)



## Testing

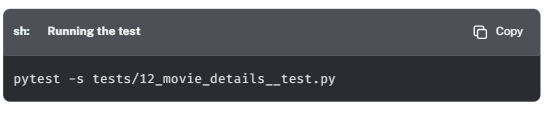
To test that this functionality has been correctly implemented, run the following code in a new terminal session:

**sh**

**Running the test**

Copy

pytest -s tests/12\_movie\_details\_\_test.py



The test file is located at [tests/12\_movie\_details\_\_test.py](https://github.com/neo4j-graphacademy/app-python/blob/main/tests/12_movie_details__test.py).

Are you stuck? Click here for help

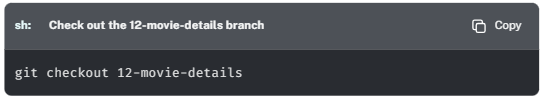
If you get stuck, you can see a working solution by checking out the 12-movie-details branch by running:

**sh**

**Check out the 12-movie-details branch**

Copy

git checkout 12-movie-details



# Listing Ratings (Challenge)

**Optional Lesson**

This lesson is optional and will not count towards your achievement. To view the completed code, check out the 13-listing-ratings branch.

The last remaining element on the Movie page is the list of ratings on the right-hand side of the page. Although the count in the header is now accurate, the ratings list being returned by the API is still hardcoded.

The list of ratings is populated by the for\_movie() method in the [RatingDAO](https://github.com/neo4j-graphacademy/app-python/blob/main/api/dao/ratings.py" \t "_blank), which currently returns a hardcoded list of ratings.

**python**

**api/dao/ratings.py**

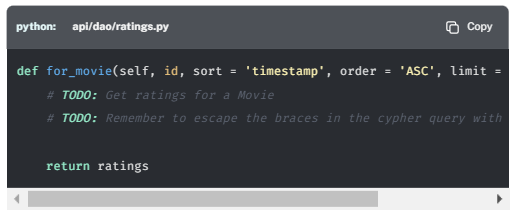
Copy

def for\_movie(self, id, sort = 'timestamp', order = 'ASC', limit = 6, skip = 0):

*# TODO: Get ratings for a Movie*

*# TODO: Remember to escape the braces in the cypher query with double braces: {{ }}*

return ratings



In this challenge, you will run the following Cypher statement in a **read transaction**:

**cypher**

**Get Ratings**

CopyRun

MATCH (u:User)-[r:RATED]->(m:Movie {tmdbId: $id})

RETURN r {

.rating,

.timestamp,

user: u {

.userId, .name

}

} AS review

ORDER BY r.timestamp DESC

SKIP $skip

LIMIT $limit



## Your Task

* Modify the for\_movie() method to retrieve the ratings from Neo4j.
* Remember to use the .format() method on the string to replace the r.timestamp value from the query above with the sort parameter supplied to the method and escape the braces in the Cypher statement using double braces ({{ and }}).

Click to reveal the completed for\_movie() method

## Working Solution

**python**

**api/dao/ratings.py**

Copy

def for\_movie(self, id, sort = 'timestamp', order = 'ASC', limit = 6, skip = 0):

*# Get ratings for a Movie*

def get\_movie\_ratings(tx, id, sort, order, limit):

cypher = """

MATCH (u:User)-[r:RATED]->(m:Movie {{tmdbId: $id}})

RETURN r {{

.rating,

.timestamp,

user: u {{

.userId, .name

}}

}} AS review

ORDER BY r.`{0}` {1}

SKIP $skip

LIMIT $limit

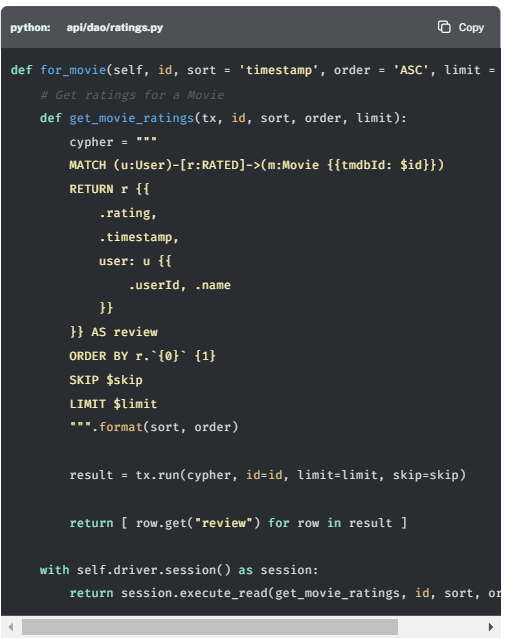
""".format(sort, order)

result = tx.run(cypher, id=id, limit=limit, skip=skip)

return [ row.get("review") for row in result ]

with self.driver.session() as session:

return session.execute\_read(get\_movie\_ratings, id, sort, order, limit)



## Testing

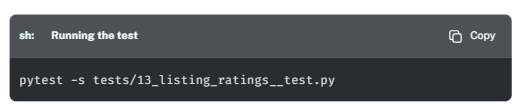
To test that this functionality has been correctly implemented, run the following code in a new terminal session:

**sh**

**Running the test**

Copy

pytest -s tests/13\_listing\_ratings\_\_test.py



The test file is located at [tests/13\_listing\_ratings\_\_test.py](https://github.com/neo4j-graphacademy/app-python/blob/main/tests/13_listing_ratings__test.py).

Are you stuck? Click here for help

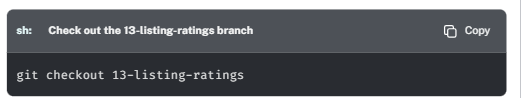
If you get stuck, you can see a working solution by checking out the 13-listing-ratings branch by running:

**sh**

**Check out the 13-listing-ratings branch**

Copy

git checkout 13-listing-ratings



# 14) Listing People (Challenge)

**Optional Lesson**

This lesson is optional and will not count towards your achievement. To view the completed code, check out the 14-person-list branch.

If you open click on the **People** link in the main navigation, you will see [**a paginated list of actors and directors**](http://localhost:3000/people/).

## People Search

A list of people can be retrieved by calling the [**http://localhost:3000/people/**](http://localhost:3000/people/) endpoint. The list is populated by the all() method in the [PeopleDAO](https://github.com/neo4j-graphacademy/app-python/blob/main/api/dao/people.py" \t "_blank).

**python**

**api/dao/people.py**

Copy

def all(self, q, sort = 'name', order = 'ASC', limit = 6, skip = 0):

*# TODO: Get a list of people from the database*

*# TODO: Remember to use double braces to replace the braces in the Cypher query {{ }}*

return people[skip:limit]



This person lists vary slightly from the movie lists because this features a **Search by Name** input. When this value is set, an additional q parameter is passed to tx.run(), and if defined, the query adds an additional CONTAINS predicate.

**cypher**

**Find People with CONTAINS predicate**

CopyRun

MATCH (p:Person)

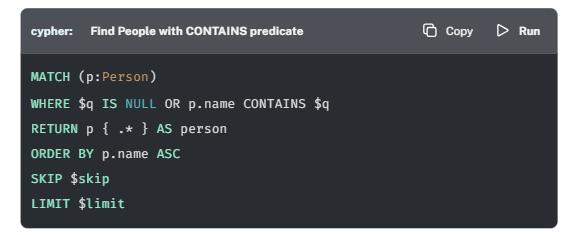
WHERE $q IS NULL OR p.name CONTAINS $q

RETURN p { .\* } AS person

ORDER BY p.name ASC

SKIP $skip

LIMIT $limit



**Using Text Indexes**

The STARTS WITH, ENDS WITH and CONTAINS predicates are **case-sensitive**. If you are looking for a case-insensitive search, you should consider setting up a [**full-text schema index**](https://neo4j.com/docs/cypher-manual/current/indexes-for-full-text-search/).

## Your Task

* Modify the all() method to query Neo4j and return a list of people.
* If the q parameter is defined, add a WHERE clause to check that the name property contains the value provided.

## Working Solution

Click to reveal the completed all() method

**python**

**api/dao/people.py**

Copy

def all(self, q, sort = 'name', order = 'ASC', limit = 6, skip = 0):

*# Get a list of people from the database*

def get\_all\_people(tx, q, sort, order, limit, skip):

cypher = "MATCH (p:Person) "

*# If q is set, use it to filter on the name property*

if q is not None:

cypher += "WHERE p.name CONTAINS $q"

cypher += """

RETURN p {{ .\* }} AS person

ORDER BY p.`{0}` {1}

SKIP $skip

LIMIT $limit

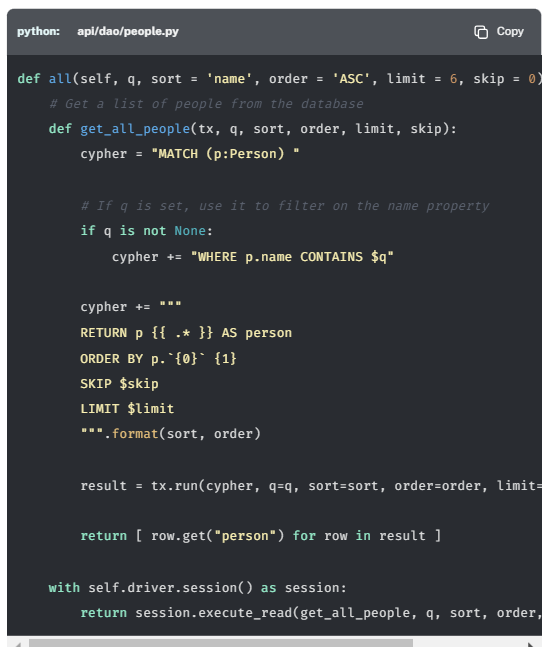
""".format(sort, order)

result = tx.run(cypher, q=q, sort=sort, order=order, limit=limit, skip=skip)

return [ row.get("person") for row in result ]

with self.driver.session() as session:

return session.execute\_read(get\_all\_people, q, sort, order, limit, skip)



## Testing

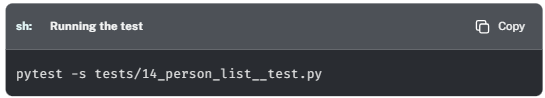
To test that this functionality has been correctly implemented, run the following code in a new terminal session:

**sh**

**Running the test**

Copy

pytest -s tests/14\_person\_list\_\_test.py



The test file is located at [tests/14\_person\_list\_\_test.py](https://github.com/neo4j-graphacademy/app-python/blob/main/tests/14_person_list__test.py).

Are you stuck? Click here for help

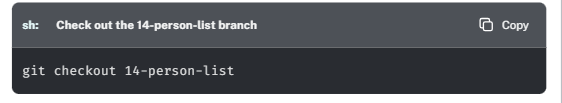
If you get stuck, you can see a working solution by checking out the 14-person-list branch by running:

**sh**

**Check out the 14-person-list branch**

Copy

git checkout 14-person-list



# 15) Person Profile (Challenge)

**Optional Lesson**

This lesson is optional and will not count towards your achievement. To view the completed code, check out the 15-person-profile branch.

If you click on a Person card anywhere on the website, you will be taken to a Person profile. This API call is the same regardless of whether the person is an actor, director, or both.

We have already implemented the methods that populate the methods in the MovieDAO to get a list of movies that the person has either acted in or directed.

But if you take a look at the [PeopleDAO](https://github.com/neo4j-graphacademy/app-python/blob/main/api/dao/people.py" \t "_blank), you will see two methods that need to be implemented to complete this page:

* [**find\_by\_id()**](https://graphacademy.neo4j.com/courses/app-python/3-backlog/7-person-view/#find_by_id) - should find a person by their ID.
* [**get\_similar\_people()**](https://graphacademy.neo4j.com/courses/app-python/3-backlog/7-person-view/#get_similar_people) - should return a list of similar people who are commonly connected to the same movies.

## find\_by\_id()

The find\_by\_id() method is currently hardcoded to return information about Al Pacino.

**python**

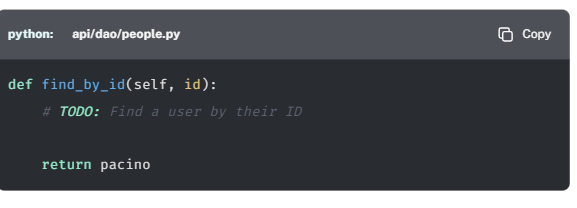
**api/dao/people.py**

Copy

def find\_by\_id(self, id):

*# TODO: Find a user by their ID*

return pacino



You will update this method to run the following cypher statement in a **read transaction**.

**cypher**

**Get Person information**

CopyRun

MATCH (p:Person {tmdbId: $id})

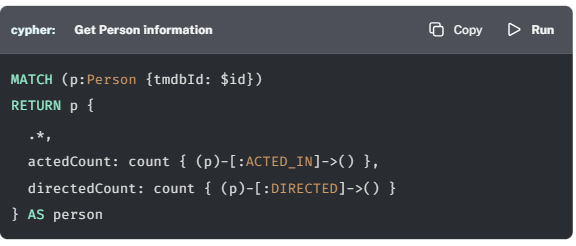
RETURN p {

.\*,

actedCount: count { (p)-[:ACTED\_IN]->() },

directedCount: count { (p)-[:DIRECTED]->() }

} AS person



The query will return the properties for the person with the corresponding tmdbId, along with a count of the number of movies that the person has acted in and directed.

**Parameters in Neo4j Browser**

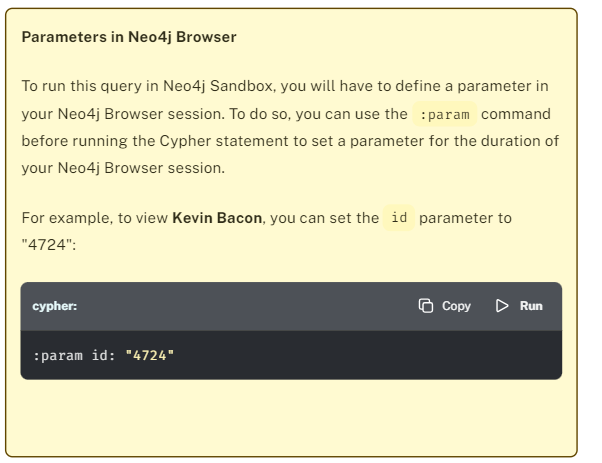
To run this query in Neo4j Sandbox, you will have to define a parameter in your Neo4j Browser session. To do so, you can use the :param command before running the Cypher statement to set a parameter for the duration of your Neo4j Browser session.

For example, to view **Kevin Bacon**, you can set the id parameter to "4724":

**cypher**

CopyRun

:param id: "4724"



### Your Task

* Modify the find\_by\_id() method to query Neo4j and return details for the requested person.
* The returned object should include counts of the number of movies that the person has acted in or directed.
* If no records are returned, the method will raise a NotFoundException.

Click here to reveal the final find\_by\_id() method.

### Working Solution (incorrect cypher query)

(NOTE: this query mentioned in this solution is incorrect, replace it with the cypher query mentioned just right before) (the only change is the use of count() instead of size()

**python**

Copy

def find\_by\_id(self, id):

*# Find a user by their ID*

def get\_person(tx, id):

row = tx.run("""

MATCH (p:Person {tmdbId: $id})

RETURN p {

.\*,

actedCount: size((p)-[:ACTED\_IN]->()),

directedCount: size((p)-[:DIRECTED]->())

} AS person

""", id=id).single()

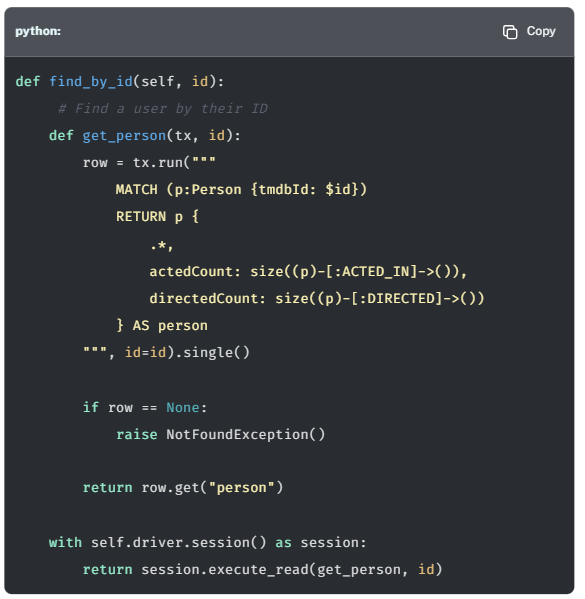
if row == None:

raise NotFoundException()

return row.get("person")

with self.driver.session() as session:

return session.execute\_read(get\_person, id)



## get\_similar\_people()

The get\_similar\_people() method should return a paginated list of similar people based on their second degree connections - either people who have acted in or have directed the person.

**python**

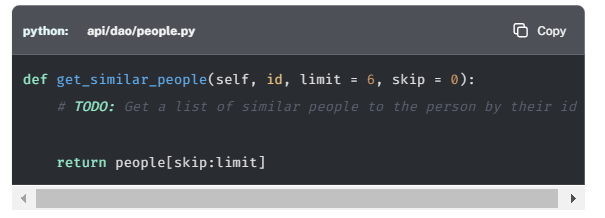
**api/dao/people.py**

Copy

def get\_similar\_people(self, id, limit = 6, skip = 0):

*# TODO: Get a list of similar people to the person by their id*

return people[skip:limit]



There could be a more clever algorithm for finding similar people by weighting the type of relationship differently, but for now, the following query will find a list of similar people based on the number of relationships in common.

**cypher**

**Get Similar People**

CopyRun

MATCH (:Person {tmdbId: $id})-[:ACTED\_IN|DIRECTED]->(m)<-[r:ACTED\_IN|DIRECTED]-(p)

WITH p, collect(m {.tmdbId, .title, type: type(r)}) AS inCommon

RETURN p {

.\*,

actedCount: count { (p)-[:ACTED\_IN]->() },

directedCount: count {(p)-[:DIRECTED]->() },

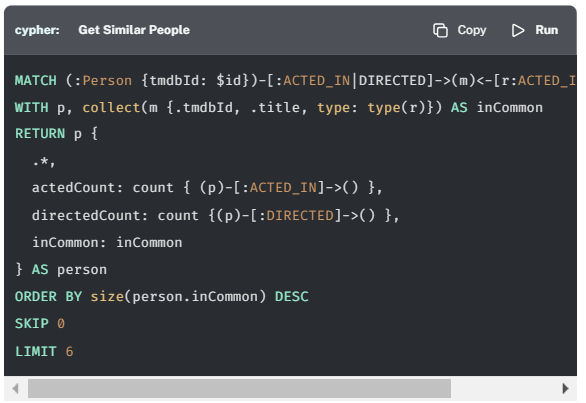
inCommon: inCommon

} AS person

ORDER BY size(person.inCommon) DESC

SKIP 0

LIMIT 6



**Parameters in Neo4j Browser**

To run this query in Neo4j Sandbox, you will have to define a parameter in your Neo4j Browser session. To do so, you can use the :param command before running the Cypher statement to set a parameter for the duration of your Neo4j Browser session.

For example, to view **Kevin Bacon**, you can set the id parameter to "4724":

**cypher**

CopyRun

:param id: "4724"

### Your Task

* Modify the get\_similar\_people() method to query Neo4j and return a list of similar people.
* The returned objects should include a list of actors and an inCommon property to show how the two people are related.

Click here to reveal the final get\_similar\_people() method.

### Working Solution (incorrect cypher query)

(NOTE: this query mentioned in this solution is incorrect, replace it with the cypher query mentioned just right before) (the only change is the use of count() instead of size()

**python**

Copy

def get\_similar\_people(self, id, limit = 6, skip = 0):

*# Get a list of similar people to the person by their id*

def get\_similar\_people(tx, id, skip, limit):

result = tx.run("""

MATCH (:Person {tmdbId: $id})-[:ACTED\_IN|DIRECTED]->(m)<-[r:ACTED\_IN|DIRECTED]-(p)

RETURN p {

.\*,

actedCount: size((p)-[:ACTED\_IN]->()),

directedCount: size((p)-[:DIRECTED]->()),

inCommon: collect(m {.tmdbId, .title, type: type(r)})

} AS person

ORDER BY size(person.inCommon) DESC

SKIP $skip

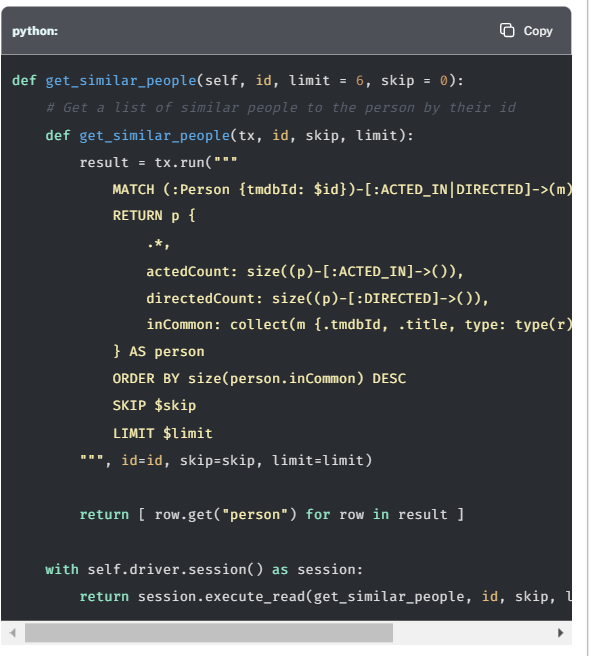
LIMIT $limit

""", id=id, skip=skip, limit=limit)

return [ row.get("person") for row in result ]

with self.driver.session() as session:

return session.execute\_read(get\_similar\_people, id, skip, limit)



## Testing

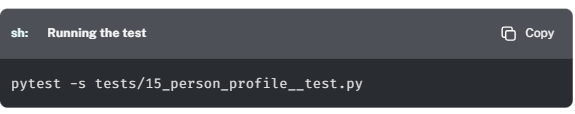
To test that this functionality has been correctly implemented, run the following code in a new terminal session:

**sh**

**Running the test**

Copy

pytest -s tests/15\_person\_profile\_\_test.py



The test file is located at [tests/15\_person\_profile\_\_test.py](https://github.com/neo4j-graphacademy/app-python/blob/main/tests/15_person_profile__test.py).

Are you stuck? Click here for help

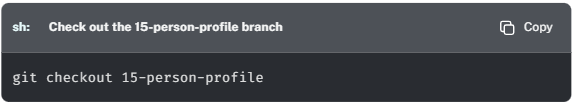
If you get stuck, you can see a working solution by checking out the 15-person-profile branch by running:

**sh**

**Check out the 15-person-profile branch**

Copy

git checkout 15-person-profile



# Course Summary

Congratulations on completing this course!

In this course, you have modified an existing project to query the [**Recommendations dataset**](https://sandbox.neo4j.com/?usecase=recommendations).

In the first module, **Project Setup** you configured the project and stored the Neo4j Sandbox credentials within the project’s environment variables for use at a later stage.

In the second module, **The Neo4j Python Driver**, you learned all about the Neo4j Driver and how it should be used within an application. This included installing the neo4j-driver dependency, building a connection string, creating a Driver instance using the driver() method and verifying that the credentials used to create the driver instance were correct.

In the third module, **Interacting with Neo4j**, you learned how to open new database sessions, execute read and write transactions and how to consume the results. You also learned how to handle potential errors thrown by the driver, and gained hands on experience by implementing these skills within the project.

In the final module, **Project Backlog**, you honed your skills by modifying the remaining service classes to retrieve.

You should now have a fully working application that you can share with your friends or colleagues, and all of the knowledge required to build one of your own.

## Continue Learning

Now that you have completed the course, we have terminated your Sandbox instance. If you would like to continue learning, you can either [**create a new Neo4j Sandbox instance**](https://sandbox.neo4j.com/?usecase=recommendations) or [**download the Recommendations dataset**](https://github.com/neo4j-graph-examples/recommendations/data) and follow the instructions in the [**Your Neo4j Sandbox**](https://graphacademy.neo4j.com/courses/app-python/0-setup/2-sandbox/) to update the credentials used by the driver.

You can find instructions on how to [**import a Neo4j dump using the**neo4j-admin load**command**](https://neo4j.com/docs/operations-manual/current/backup-restore/restore-dump/) in the [**Neo4j Operations Manual**](https://neo4j.com/docs/operations-manual/current/).

## Resources

* **There are many resources available to you for learning more about Neo4j**  
  [**https://neo4j.com/developer/resources/**](https://neo4j.com/developer/resources/)
* **Neo4j Community Site where you can ask or answer questions about Neo4j and discuss with other users:**  
  [**https://community.neo4j.com**](https://community.neo4j.com/)
* **Neo4j documentation:**  
  [**https://neo4j.com/docs/**](https://neo4j.com/docs/)
* **Neo4j Sandboxes for experimenting with graphs:**  
  [**https://sandbox.neo4j.com/?ref=graph-academy**](https://sandbox.neo4j.com/?ref=graph-academy)
* **Videos on the Neo4j YouTube channel:**  
  [**https://www.youtube.com/channel/UCvze3hU6OZBkB1vkhH2lH9Q**](https://www.youtube.com/channel/UCvze3hU6OZBkB1vkhH2lH9Q)
* **Become a Neo4j certified developer:**  
  [**https://graphacademy.neo4j.com/categories/certification/**](https://graphacademy.neo4j.com/categories/certification/)
* **GitHub repository:**  
  [**https://github.com/neo4j-contrib**](https://github.com/neo4j-contrib)
* **Neo4j events all over the world:**  
  [**https://neo4j.com/events/world/all/**](https://neo4j.com/events/world/all/)
* **Graph Gists for learning more use cases for Neo4j:**  
  [**https://neo4j.com/graphgists/**](https://neo4j.com/graphgists/)
* **Attend a Neo4j meetup:**  
  [**https://www.meetup.com/topics/neo4j/**](https://www.meetup.com/topics/neo4j/)
* **View questions/answers raised about Neo4j:**  
  [**https://stackoverflow.com/tags/neo4j/hot**](https://stackoverflow.com/tags/neo4j/hot)