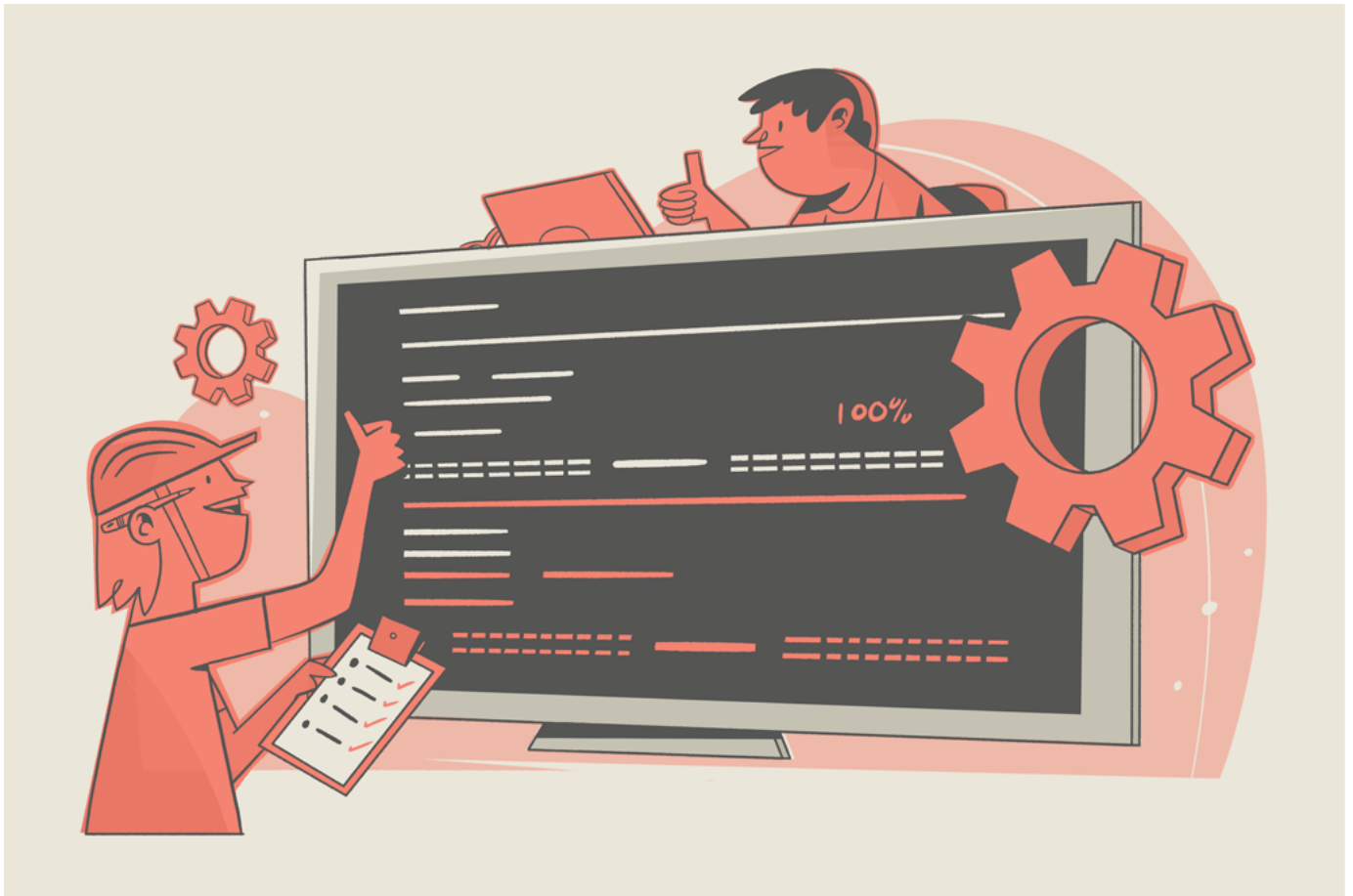


 What if ChatGPT was part of your technical interview?

A Guide To Database Unit Testing with **Pytest** and SQLAlchemy

November 10, 2022 • Development

By EzzEddin Abdullah



Testing is a decisive phase in your [systems development lifecycle](#). This is an important step to make your software reliable and maintainable in the future. In this tutorial, you will have a practical guide to **unit testing**, one type of software testing by which individual units of your code are tested to determine if they are fit for use.

We will discuss how to do unit testing using [pytest](#), which is a Python testing tool. This tool has useful features to help write better programs. We will test a database created by

SQLAlchemy ORM.

At the end of this tutorial, you will learn how to **test** SQLAlchemy ORM **using fixtures and the classic way to implement fixtures** (setup and teardown methods).

As always, to grasp the ideas introduced in this tutorial, you need to experiment with your code by yourself, whether in your local machine or here in the [CoderPad sandbox](#). The sandbox has a ready-to-use environment to try out your experiments.

You can select **Python3**, click on the three dots in the top left corner, and then select SQLAlchemy (Postgres) from the adapters. You're free to choose a MySQL adapter, but in this tutorial, we will use the Postgres engine in SQLAlchemy.

Before you start, you need to know more about **how to interact with databases using SQLAlchemy with Postgres and revise transactions in SQLAlchemy**.

Create the database models

We will use the simple database models introduced in this [SQLAlchemy tutorial](#).

For this tutorial, we will dump all the code in the Pad on the left window. But for a production use case, you need to structure your files into two files, for example:

- `models.py` to have the database models
- `test_blog.py` to have the testing suite

So the database models would have the following:

```
from sqlalchemy import create_engine, Column, Integer, String, DateTime,
Text, ForeignKey
from sqlalchemy.engine import URL
from sqlalchemy.orm import declarative_base, relationship, sessionmaker
from datetime import datetime
import pytest
```

```
Base = declarative_base()
```

```
class Author(Base):
    __tablename__ = 'authors'

    id = Column(Integer(), primary_key=True)
    firstname = Column(String(100))
    lastname = Column(String(100))
    email = Column(String(255), nullable=False)
    joined = Column(DateTime(), default=datetime.now)
```

```
articles = relationship('Article', backref='author')

class Article(Base):
    __tablename__ = 'articles'

    id = Column(Integer(), primary_key=True)
    slug = Column(String(100), nullable=False)
    title = Column(String(100), nullable=False)
    created_on = Column(DateTime(), default=datetime.now)
    updated_on = Column(DateTime(), default=datetime.now,
onupdate=datetime.now)
    content = Column(Text)
    author_id = Column(Integer(), ForeignKey('authors.id'))

url = URL.create(
    drivername="postgresql",
    username="coderpad",
    host="/tmp/postgresql/socket",
    database="coderpad"
)

engine = create_engine(url)
Session = sessionmaker(bind=engine)
```

So we have two tables, authors and articles, and we want to test a few units of a transactional database in SQLAlchemy.



In the previous section, we defined the `Session` class, which we will use in the testing class. You can define a `class` called `TestBlog`, which contains the test functions. Each test function should start with `test_` so that pytest can understand it's a function that contains test cases.

Fixtures classic way

`Fixtures` is a powerful feature in `pytest`. While testing a database created in an SQLAlchemy ORM, you need to ensure that the session is open at each test function call with just one setup. You don't need to instantiate it at every test function. You'd also need to close the session after all unit tests are done.

The `classic way` of using Python database fixtures in `pytest` is to `use setup and teardown functions`. These functions are useful to avoid repeating code at every test function. This is useful because hitting the database multiple times would be discouraged, especially if you're testing a large application.

Here is how to write these methods:

```
class TestBlog:
    def setup_class(self):
        Base.metadata.create_all(engine)
        self.session = Session()
        self.valid_author = Author(
            firstname="Ezzeddin",
            lastname="Aybak",
            email="aybak_email@gmail.com"
        )

    def teardown_class(self):
        self.session.rollback()
        self.session.close()
```

The `setup_class` is called before all test methods in the `TestBlog`, while the `teardown_class` is called after all test methods. Both are called at the class level.

The setup method creates all your tables; authors and articles tables. It then defines the session, which is the `Session` object in SQLAlchemy in which the conversation with the database occurs. It also has the `valid_author` object, which we would frequently use in the following test functions.

However, the teardown method is the clean-up phase after all test methods. It rolls back the changes to the database and then closes the session, so there wouldn't be any conversation channel between the SQLAlchemy and your database anymore.

Testing a success test case

Let's start with testing the content retrieved by the database of the valid author:

```
class TestBlog:
    #...

    def test_author_valid(self):
        self.session.add(self.valid_author)
        self.session.commit()
        aybak =
self.session.query(Author).filter_by(lastname="Aybak").first()
        assert aybak.firstname == "Ezzeddin"
        assert aybak.lastname != "Abdullah"
        assert aybak.email == "aybak_email@gmail.com"
```

Here, we use the session to add the valid author object (defined in the setup function). We then commit that change to the database and query that author. At the end of the test function, we typically use an assert statement to [verify test expectations](#).

Testing a failure test case

Let's add another test method under the `TestBlog` class to [test an integrity error](#), which is an exception raised when there is relational data integrity is affected:

```
from sqlalchemy.exc import IntegrityError

# ...
class TestBlog:
    # ...
    @pytest.mark.xfail(raises=IntegrityError)
    def test_author_no_email(self):
        author = Author(
            firstname="James",
            lastname="Clear"
        )
        self.session.add(author)
        try:
            self.session.commit()
        except IntegrityError:
            self.session.rollback()
```

In this test function, we use a `pytest.mark.xfail` decorator to mark this `test_author_no_email` test function that [can fail for some reason](#). The reason to fail here is to have an `IntegrityError` while committing an author object with no email. That's because when we defined the `authors` table, we made a restriction on the email attribute to not have a **NULL** record.

In this test function, we use a `try/except` statement to be able to [roll back the transaction](#) where the `IntegrityError` exception occurs while committing to the DB.

Testing a referenced table

To reference a foreign key, you need to have the referenced object defined in the same test function, or you [can define that object in the setup phase](#), where you can import it to other test functions. In our case, we defined the `valid_author` object, in the setup function, which we referenced in the articles table:

```

class TestBlog:
    # ...
    def test_article_valid(self):
        valid_article = Article(
            slug="sample-slug",
            title="Title of the Valid Article",
            content="Lorem ipsum dolor sit amet, consectetur adipiscing
elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut
enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut
aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint
occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit
anim id est laborum.",
            author=self.valid_author
        )
        self.session.add(valid_article)
        self.session.commit()
        sample_article =
self.session.query(Article).filter_by(slug="sample-slug").first()
        assert sample_article.title == "Title of the Valid Article"
        assert len(sample_article.content.split(" ")) > 50

```

Here, we define an object for a valid article with the referenced valid author defined in the setup method. We then add and commit it to the database.

Finally, we verify our tests. For example, we verify the title and then verify the article's length to be greater than 50 words.

Running pytest

To invoke pytest, you have [some options](#) mentioned in the pytest documentation. In this tutorial, you can use the following line in the sandbox console:

```
pytest.main(['-v'])
```

We use the -v option here to increase verbosity and show each test function result.

USE QUESTION

As you can see in your sandbox, there are 3 test cases for each test function. The first and third test cases succeeded with a PASSED keyword. However, the second test case succeeded with an XPASS keyword to **indicate that pytest caught the expected error successfully.**

Using fixtures

Instead of the classic way to use a fixture, let's look at how to define a fixture itself. But why do we need fixtures anyway?

In addition to the benefits of the classic way of defining fixtures, fixtures themselves are useful because they **lead to dependency inversion**. Dependency inversion is a design


pattern useful when a test function receives other objects it depends on. It aims to separate the concerns which lead to **loosely coupled programs**.

Let's see **fixtures** in action and **get rid of the `setup_class` and `teardown_class` methods**. Now, define the following **before the `TestBlog` class**:

```
@pytest.fixture(scope="module")
def db_session():
    Base.metadata.create_all(engine)
    session = Session()
    yield session
    session.rollback()
    session.close()

@pytest.fixture(scope="module")
def valid_author():
    valid_author = Author(
        firstname="Ezzeddin",
        lastname="Aybak",
        email="aybak_email@gmail.com"
    )
    return valid_author
```

These fixtures are defined at the module level. That's why there is a scope option inside the `pytest.fixture` decorator. A fixture is called based on the scope.

 A **scope** is what you can use to share fixtures across classes, modules, packages, or sessions. In our case, the `db_session` and `valid_author` fixtures can only be called across this module.

The first fixture function creates all the tables metadata and then yields the session object whenever it's called. Each test function calling the session will receive the same session instance, **thus saving time**. So that fixture function is invoked once per the test module.

Using the yield statement is **recommended**. After the yield line, the session is rolled back and then closed, equivalent to the teardown code.

The second fixture function returns the valid author object whenever a test function calls it.

Note: The separation of concerns here is clear. Instead of the classic setup function, which contained both the session and the valid author objects, we now have two separate fixture functions for each.

Calling fixtures

To call a fixture inside a test function, you need to pass it as an argument to that function. Here is the new `TestBlog` class with the associated test functions:

```
class TestBlog:
    def test_author_valid(self, db_session, valid_author):
        db_session.add(valid_author)
        db_session.commit()
        aybak =
db_session.query(Author).filter_by(lastname="Aybak").first()
        assert aybak.firstname == "Ezzeddin"
        assert aybak.lastname != "Abdullah"
        assert aybak.email == "aybak_email@gmail.com"

    @pytest.mark.xfail(raises=IntegrityError)
    def test_author_no_email(self, db_session):
        author = Author(
            firstname="James",
            lastname="Clear"
        )
        db_session.add(author)
        try:
            db_session.commit()
        except IntegrityError:
            db_session.rollback()

    def test_article_valid(self, db_session, valid_author):
        valid_article = Article(
            slug="sample-slug",
            title="Title of the Valid Article",
            content="Lorem ipsum dolor sit amet, consectetur adipiscing
elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut
enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut
aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint
occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit
anim id est laborum.",
            author=valid_author
        )
        db_session.add(valid_article)
        db_session.commit()
        sample_article = db_session.query(Article).filter_by(slug="sample-
slug").first()
        assert sample_article.title == "Title of the Valid Article"
        assert len(sample_article.content.split(" ")) > 50
```

As you can see, `db_session` and `valid_author` are passed into each test function as arguments. The `db_session` fixture replaces each `self.session`, and each `self.valid_author` is now replaced by the `valid_author` fixture.

Verify the tests in the sandbox below by running `pytest.main(['-v'])` in the console:

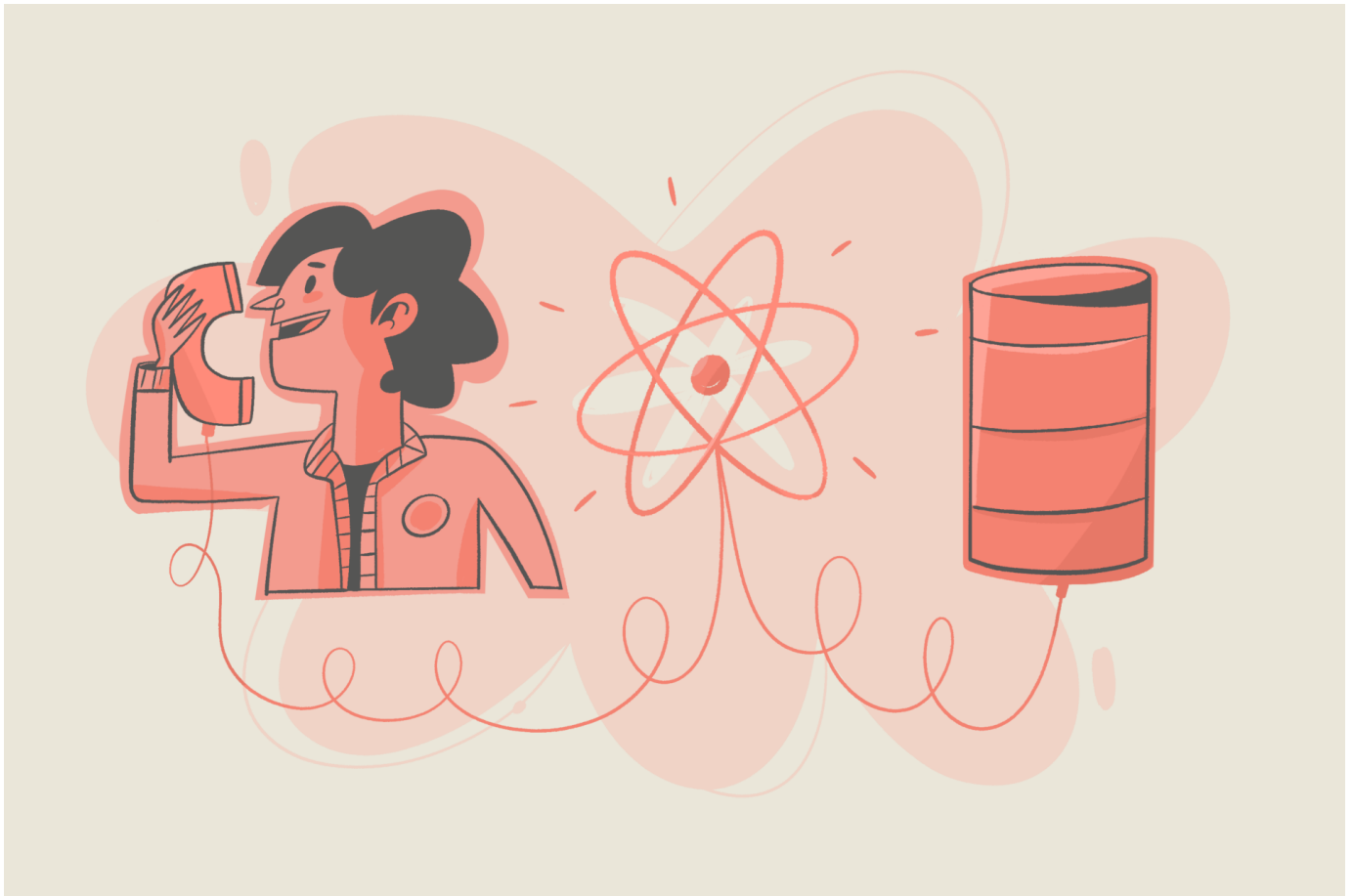
Wrapping up

This tutorial has covered how to do unit testing for a transactional database in SQLAlchemy using pytest. You started with creating the database models and then went through the

classic way of using pytest fixtures. Lastly, you learned how to use fixtures and why they are useful for writing efficient tests.

I'm Ezz. I'm an AWS Certified Machine Learning Specialist and a Data Platform Engineer. I help SaaS companies rank on Google. Check out my [website](#) for more.

Related Posts



A Guide To Using React's useCallback() Hook

April 25, 2023 • [Development](#)

By Mercy Kibet

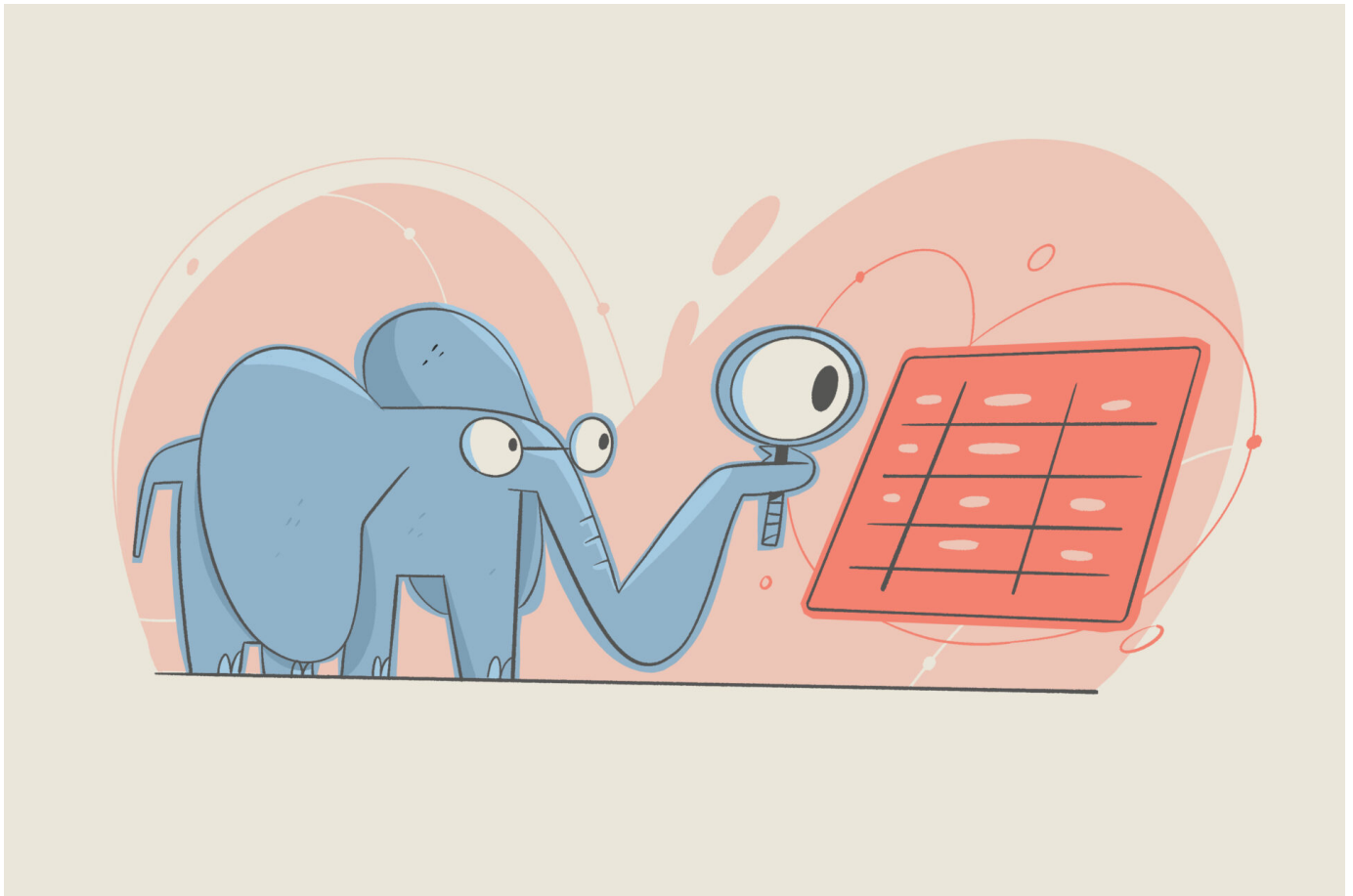
Are you looking to optimize your React applications? Have you heard of the useCallback hook? useCallback is a powerful React hook that helps you optimize your application by preventing unnecessary re-renders. In this post, we'll dive into React's useCallback() hook, define referential equality and callback functions and how to tie it all together.



MySQL vs. PostgreSQL: How Do They Compare?

April 17, 2023 • [Development](#)

A database is a convenient tool that allows users to access information, maintain records, and apply commonly used operations, such as insertion, modification, deletion, and data organization. Learn about MySQL vs PostgreSQL, their features, advantages, and disadvantages. Finally, check out which one is the best choice for you.



PostgreSQL LIKE Operator: A Detailed Guide

April 17, 2023 • [Development](#)

By Carlos Schults

When using databases, searching for values that match a given pattern is a familiar necessity. Learn how you can retrieve entries with queries including a particular string using Postgres LIKE operator.

Need a better way to interview candidates? Try CoderPad.

[Sign up free](#)

[Request demo](#)

PLATFORM

Pricing

Languages and Frameworks

Live Collaborative Coding & Online IDE

Take-Home Projects

Drawing Mode

Focus Time

Integrations

Single Sign-On (SSO)

RESOURCES

Blog

FAQs

User Guides

Interview Questions

Docs

Events

Enterprise

Customers

Webinars

University Recruiting

COMPANY

About Us

Careers

Press

Contact Support

Request a Demo

[Terms of Service](#)

[Privacy Policy](#)

Questions? **Contact support**
Status

© 2023 CoderPad, Inc

CoderPad is a service mark of CoderPad, Inc.