Unit and Integration Testing

SoftUni Team

Technical Trainers

Software University

https://softuni.bg

Have a Question?

Software University

sli.do

#python-web

Table of Contents

Unit and Integration Testing

Unit and Integration Testing

- **Unit** and **integration testing** are **crucial** parts of the **development process**
- They **ensure** that **individual components** (**unit tests**) and **integrated parts** of the system (**integration tests**) work as expected

Benefits and Usage of Unit Tests

- **Isolated Code**:
  - **Unit tests** are designed to **test isolated units** of code **without relying** on **external dependencies**
  - This **isolation ensures** that any **failures** are **due to issues within** the **unit** being tested **rather** than **external factors**

Benefits and Usage of Unit Tests

- Fast Execution:
  - Unit tests typically have fast execution times since they **don't** involve **interactions** with external systems like **databases** or **APIs**
  - Fast execution allows developers to quickly **identify** and **fix issues** during development cycles

Benefits and Usage of Unit Tests

- **Abundance**:
  - Due to their **focused nature**, **unit tests** tend to be more **numerous** compared to other types of tests
  - A **larger number** of **unit tests** provides **better coverage** of the **codebase**, ensuring **more comprehensive testing**

Benefits and Usage of Unit Tests

- **Ideal for Pure Functions**:
  - **Unit tests** are **particularly practical** for **testing pure functions**, which **produce** the **same output** for a given **input**, with **no side effects**
  - They **validate** the **behavior** of **functions** in **isolation**, making it **easier** to **pinpoint** and **resolve issues**

Benefits and Usage of Unit Tests

- **String Transformations**:
  - **Unit tests** are **well-suited** for **testing string transformation** functions
  - They can **verify** that **functions correctly manipulate strings** according to the **specified logic**, ensuring the **desired output**

Benefits and Usage of Unit Tests

- **Validators**:
  - **Unit tests** are **effective** for **validating input data** using **custom validators**
  - They **ensure** that **validators correctly identify valid** and **invalid inputs** according to the **defined criteria**

Integration Tests

- **Integration tests** evaluate the **entire application flow** by **testing** the **integration** of **multiple functions** or **components**
- These tests **verify** that **different parts** of the system work **together** as expected, **simulating real-world** scenarios

Benefits and Usage of Integration Tests

- **Comprehensive Coverage**:
  - **Integration tests** provide comprehensive **coverage** by **examining** the **interaction** between **various components**, including **databases**, **APIs**, and **external** services
  - They ensure that the application **behaves correctly across different layers** and **subsystems**

Benefits and Usage of Integration Tests

- **Practical for end-to-end scenarios**:
  - **Integration tests** are practical for testing **system workflows** or **end-to-end scenarios** that involve **multiple interactions** within the application
  - They **validate** the **complete user journey**, ensuring that **all features** and **functionalities** work **seamlessly together**

Drawbacks of Integration Tests

- **Slower Execution**:
  - **Integration tests** typically have **slower execution times** compared to unit tests due to their **broader scope** and **involvement** of **external dependencies**
  - **Slower execution** can **impact** development **cycles** but is **necessary** to **validate** the **integration** of **complex system components**

Drawbacks of Integration Tests

- **Limited in Number**:
  - **Integration tests** are generally **fewer** in **number** compared to unit tests **due to** their **comprehensive nature**
  - While **fewer** in **quantity**, they play a **critical role** in **validating** the **overall functionality** and **behavior** of the application

Example Use Cases of Integration Tests

- **User Registration**:
  - Test the **entire user registration process**, including **form submission, data validation**, and **database persistence**
- **Course Signup After Payment**:
  - **Validate** the **flow** from **selecting** a **course** to **completing payment** and **accessing** course **content**

Best Practices

Best Practices

- Test **All Potentially Breakable** Code:
  - It's **essential** to test **any code** that could **potentially break** to **maintain** the **stability** and **reliability** of your application
- Test **Granularity**:
  - Each test should **focus** on a **single function** (for **unit tests**) or a **specific flow** (for **integration tests**) to ensure **clarity** and **effectiveness**

Best Practices

- **Single Assertion**:
  - **Limit** each test to **asserting only one case** to maintain **simplicity** and **clarity**
- Keep it **Simple**:
  - Tests should be **simple** and **straightforward**, avoiding **unnecessary complexity** or **dependencies**

Best Practices

- Follow the **Triple-A Rule**:
  - Adhere to the **Arrange**, **Act**, **Assert pattern** in your test setup
  - **Arrange**: **Set** up the **preconditions** and **inputs** for the test
  - **Act**: **Execute** the **code** or **function** being tested
  - **Assert**: **Verify** the **expected outcome** or **behavior** of the code

Structuring and Organizing Tests

Structuring and Organizing Tests

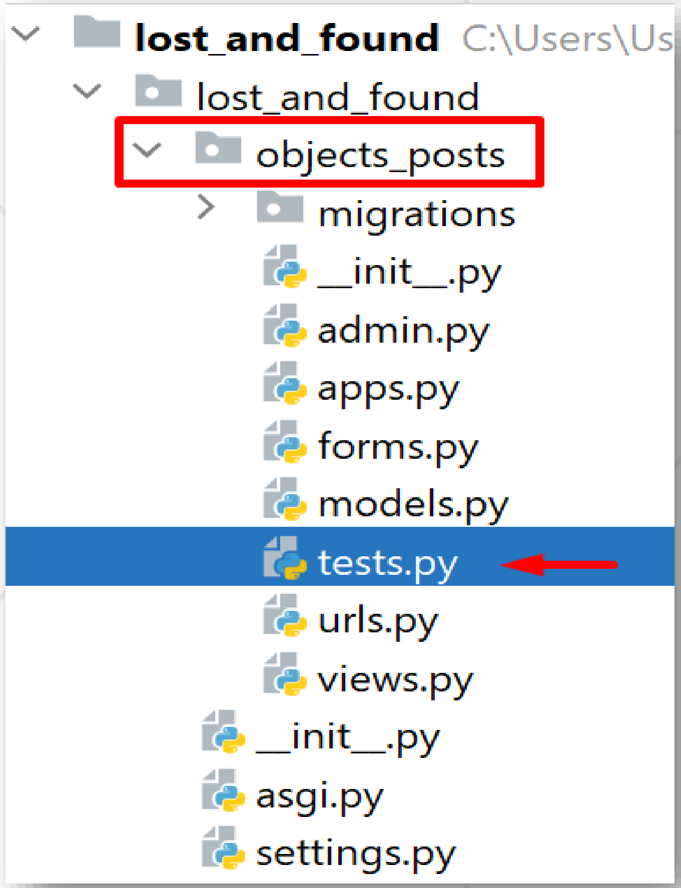
- There are **different approaches** to **structuring** and **organizing tests**
  - Using the **app's** `tests.py` file
    - **Group** tests within **each** Django **app's** `tests.py` file based on the **functionality** they test (**models**, **views**, **forms**, etc.)
    - This approach keeps **tests closely associated** with the **code** they are **testing**, making it **easier** to **maintain** and **understand**

Structuring and Organizing Tests

- Organizing test files **within** a **project's** `tests` **folder**
  - Create a **dedicated** '`tests`' **folder** at the **project level** to contain test files for **different** functionalities
  - **Organize** test files into subfolders following the **project's app** naming **convention**
  - This **structure** allows for a more systematic organization of **tests** across **multiple apps** within the project

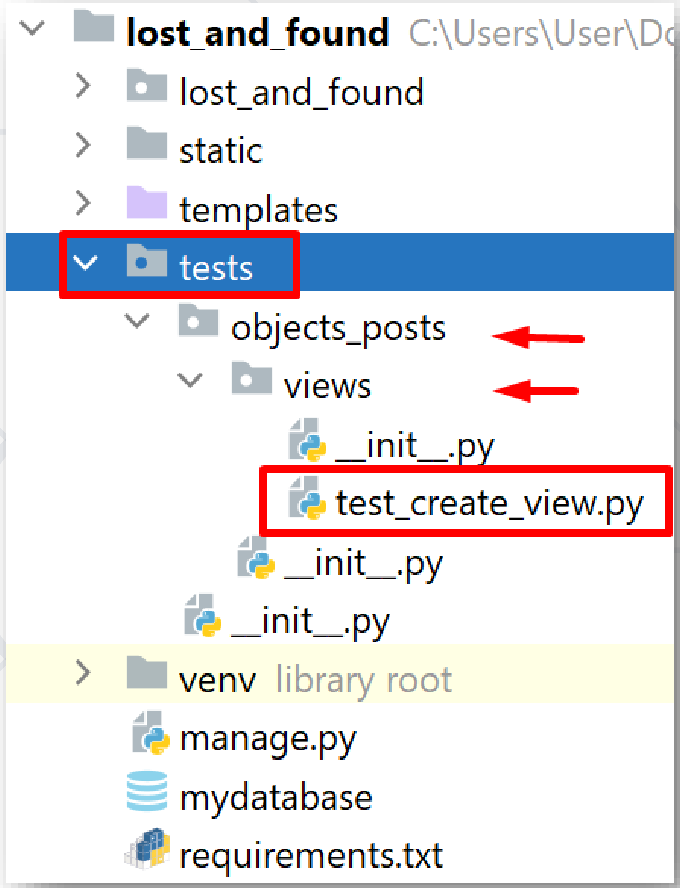
Structure Examples

- Each app tests in a `tests.py` file
- All tests in a `tests` folder



```
lost_and_found  C:\Users\Us
  lost_and_found
    objects_posts
      migrations
      __init__.py
      admin.py
      apps.py
      forms.py
      models.py
      tests.py
      urls.py
      views.py
    __init__.py
    asgi.py
    settings.py
```

Use **tests.py** file for each app



```
lost_and_found  C:\Users\User\Do
  lost_and_found
  static
  templates
  tests
    objects_posts
      views
        __init__.py
        test_create_view.py
      __init__.py
    __init__.py
  venv  library root
  manage.py
  mydatabase
  requirements.txt
```

Create a **tests** folder and organize the code into subfolders

Testing Django Components

What to Test?

What to Test?

- Test **All Custom** Code
  - Models and Custom Managers
  - Forms
  - Views
  - Other Custom Code
- **Exceptions**:
  - **Built-in** Code (e.g., Django's built-in methods) like:
    - `Model.objects.all()`
  - Code from **Third-Party** Libraries

Testing Models

- **Purpose of Testing**:
  - Testing **model definitions** **primarily focus** on **validation logic** rather than database-specific details
  - The aim is to **ensure** that the **defined fields** and **validation rules** **behave** as expected
- **Validation Testing**:
  - **Validate** the **behavior** of **custom validation** **logic** applied to model fields
  - Test **scenarios** where **invalid data** **should** **trigger** **validation** **errors**

Testing Models

- **Exceptions:**
  - **Built-in Validators**
    - Django's built-in validators are **thoroughly tested** and can be **assumed** to **work** as **intended**
    - Testing these **validators** in **isolation** may **not be necessary unless custom validation logic interacts** with **them** in a specific way

Testing Models

```
class Profile(models.Model):
    name = models.CharField(max_length=30)
    age = models.IntegerField(validators=(
        MinValueValidator(0),
        MaxValueValidator(150)
    ))
    egn = models.CharField(max_length=10,
validators=[egn_validator])
```

- Testing **name** and **age** fields may **not be necessary** since they use **built-in validators**
- Focus **testing** efforts on the **egn** field, which contains **custom validation logic**

Testing Custom Validators in Models

- Testing with **valid data** should **save** the **instance** to the DB

```
class ProfileModelTestCase(TestCase):
    def test_profile_create_when_valid_egn__should_create(self):
        # Arrange
        valid_egn = '0506221234'
        p = Profile(name='Valid String', age=21, egn=valid_egn)

        # Act
        p.full_clean()
        p.save()

        # Assert
        self.assertIsNotNone(p)
```

Testing Custom Validators in Models

- Testing with **invalid** **data** should **raise** a **ValidationError**

```python
class ProfileModelTestCase(TestCase):
    def test_profile_create_when_invalid_egn__should_raise(self):
        # Arrange
        invalid_egn = '0506a21234'
        p = Profile(
            name='Valid String',
            age=21,
            egn=invalid_egn
        )
        # Act & Assert
        with self.assertRaises(ValidationError):
            p.full_clean()
```

Testing Forms

- Testing **forms** share **similarities** with testing **models**
  - Particularly in testing **custom logic** associated with them

```
def 
test_profileForm_whenValid__returnsT
rue(self):
    valid_data = {
        'name': 'Valid String',
        'age': 21,
        'egn': '0506221234',
    }
    form = ProfileForm(**valid_data)
    self.assertTrue(form.is_valid())
```

```
def 
test_profileForm_whenInvalid__returnsF
alse(self):
    invalid_data = {
        'name': 'Valid String',
        'age': 21,
        'egn': '05062a1234',
    }
    form = ProfileForm(**invalid_data)
    self.assertFalse(form.is_valid())
```

Testing Views

- Views are **tested** using Django's **test** **`Client`**
- Tests **send** **requests** to views by URL and **assert** various aspects of the **response**, including **templates**, **context**, **redirects**, and **status codes**
- The **test client** can also be used to **simulate** **user authentication** and **persist** **sessions** for **authenticated** views

```
class ProfileViewTests(TestCase):
    def setUp(self) :
        self.test_client = Client()
```

Testing Views - GET Requests

- Verify that the **response** **renders** the **expected template**:

```
def test_getProfilesIndex__shouldRenderTemplate(self):
    response = self.test_client.get(reverse('index'))
    self.assertTemplateUsed(response, 'testing/index.html')
```

- Verify the **correctness** of the **context data**:

```
def test_getProfilesIndex__shouldReturnCorrectContext(self):
    response = self.test_client.get(reverse('index'))
    profiles = response.context['profiles']
    # Add regular asserts to check the context data as needed
```

Testing Views - POST Requests

- Test **Redirects**

```
def test_profilesIndex_whenValidData__shouldCreateAndRedirectToIndex(self):
    # Arrange
    url = reverse('index')
    valid_data = {
        'name': 'Valid String',
        'age': 21,
        'egn': '0506231234',
    }
    # Act
    response = self.test_client.post(url, valid_data)
    # Assert
    self.assertRedirects(response, url)
```

The test client sends a POST request to the index view with the provided data

Verifies that the response redirects to the expected URL after successful processing of the POST request

Testing Isolated Code with Unit Tests

- A **custom validator**:

```
def egn_validator(value: str):
    if not all(d.isdigit() for d in value):
        raise ValidationError('EGN should contain only digits')
```

- **Unit tests**:

```
def test_egnValidator_whenAllIsDigit__shouldDoNothing(self):
    result = egn_validator('1234567890')
    self.assertIsNone(result)

def test_egnValidator_whenOneNonDigit__shouldRaise(self):
    with self.assertRaises(ValidationError) as context:
        egn_validator('12345678s0')
    self.assertIsNotNone(context.exception)
```

Integration Testing Perspective

- **Model**, **form**, and **view** tests are typically considered **integration tests** within the context of Django testing
  - They **inherently depend on** Django itself, making it **impractical** to treat them as unit tests
- However, **validation tests** within **models** or **forms** can **potentially** be **treated** as **unit tests**
  - If they do **not have external dependencies** or if their external dependencies **can** be **effectively mocked**

Redundancy in Test Coverage

- As the **complexity** of **integration** **tests** **increases**, the **need** for smaller **unit** **tests** and **redundant** **integration tests** **diminishes**
  - **View** **tests** often **cover** **forms** and **models**, **reducing** the **necessity** for **separate** **tests** for these components
- The **goal** is to strike a **balance** between **comprehensive** **test** **coverage** and **avoiding** **redundancy** in test cases

Live Demo

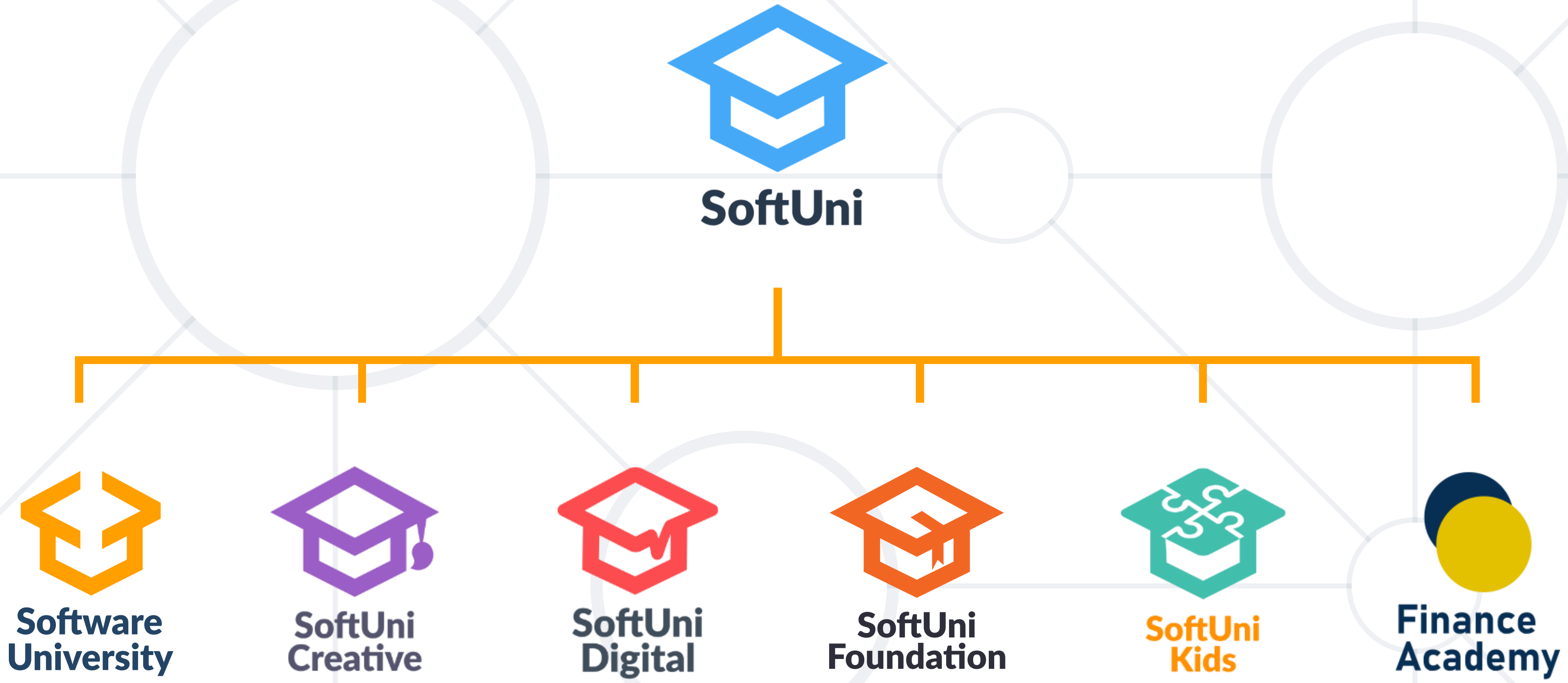
Testing Django Components

Summary

- **Unit Testing**
  - Focuses on **isolated tests** that target specific **functions** or **components**
- **Integration Testing**
  - Involves **larger tests** that **assess user behavior** and the **functionality** of the **entire application**

Software University

Questions?



SoftUni Diamond Partners

Software University

SmartIT

Coca-Cola HBC Bulgaria

SOFTWARE GROUP

DXC TECHNOLOGY

createX

Postbank

Решения за твоето утре

SUPER HOSTING .BG

DRAFT KINGS

VIVACOM

INDEAVR

Serving the high achievers

AMBITIONED

PHAR VISION

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, softuni.org
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity

License