# Copilot: Faked http request – response by test url patterns

### 🧠 Bonus Tip: Why Fake?

Faking responses is useful when:

* You want to isolate logic without relying on the database.
* You’re testing error handling or edge cases.
* You want to simulate external API calls or slow queries.

Let me know if you want to mock a specific part of your view or model—I can help tailor the code.

## My Prompt

other alternative the client.get returns the specific result which I want, withouth mock import and models (no database)

Ah, you're aiming for a lightweight test setup—no database, no mocking, just a client.get() that returns a controlled response. While Django's self.client.get() is designed to hit actual views and go through the full request-response cycle, you can still simulate specific behavior by using **custom views** and **temporary URL overrides**. Here's how to do it:

### 🧪 Use a Custom View + Temporary URLConf

You can define a minimal view inside your test file and override the URL configuration during the test. This lets you return any response you want—no database, no mocking.

#### ✅ Step-by-Step Example

from django.test import TestCase, override\_settings

from django.http import JsonResponse

from django.urls import path

# Define a fake view directly in your test file

def fake\_element\_view(request, id):

return JsonResponse({"name": "Fake Element"}, status=200)

# Temporary URLConf for testing

test\_urls = [

path("elements/<int:id>/", fake\_element\_view, name="requests\_byIds"),

]

class FakeResponseTest(TestCase):

@override\_settings(ROOT\_URLCONF=\_\_name\_\_) # Point to current module

def test\_fake\_response(self):

response = self.client.get("/elements/123/")

self.assertEqual(response.status\_code, 200)

self.assertJSONEqual(response.content, {"name": "Fake Element"})

### 🧠 Why This Works

* You bypass the database entirely.
* You don’t need mock or any external libraries.
* You control the response directly in the view logic.
* It’s perfect for testing frontend expectations or isolating logic.

If you want to simulate different status codes (like 404 or 500), just change the return value in fake\_element\_view. Want to simulate a delay or malformed JSON? You can do that too.

Let me know if you want to simulate a more complex response—like headers, cookies, or error formats.

## Next prompt

where test\_urls is used in test case?

Great catch—let’s clarify that part. In the example I gave earlier, test\_urls was defined as a list of URL patterns, but it wasn’t directly used in the test case. That’s because Django doesn’t accept a list of URL patterns directly in override\_settings. Instead, you need to define a **temporary URLConf module** that Django can import.

Here’s how to do it properly:

### ✅ Correct Way to Use a Custom URLConf in Tests

#### 1. **Create a Temporary URLConf Module**

You can define it inside your test file like this:

# test\_urls.py (or inside your test file if you're using \_\_name\_\_)

from django.urls import path

from django.http import JsonResponse

def fake\_element\_view(request, id):

return JsonResponse({"name": "Fake Element"}, status=200)

urlpatterns = [

path("elements/<int:id>/", fake\_element\_view, name="requests\_byIds"),

]

#### 2. **Use** override\_settings **to Point to That Module**

In your test case:

from django.test import TestCase, override\_settings

@override\_settings(ROOT\_URLCONF="myapp.tests.test\_urls") # Replace with actual module path

class FakeResponseTest(TestCase):

def test\_fake\_response(self):

response = self.client.get("/elements/123/")

self.assertEqual(response.status\_code, 200)

self.assertJSONEqual(response.content, {"name": "Fake Element"})

### 🧠 Why This Works

* ROOT\_URLCONF tells Django which URL patterns to use during the test.
* You’re swapping out your real app’s URLs with a controlled test-only version.
* This lets you simulate any response you want without touching your database or view logic.

If you want to keep everything inside one file (no separate module), you can use ROOT\_URLCONF=\_\_name\_\_ **only if** your test file defines urlpatterns at the module level.

Want me to help you refactor your test file to use this pattern cleanly? Just show me your current structure.

## Prompt Error

File "D:\Projects\REST-servers-genAI\django-server\crudServer\tests.py", line 46, in test\_get\_element\_by\_id response = self.client.get(reverse("api/element", args=[self.initial\_element.id])) File "D:\ProgramData\Miniconda3\envs\crud-py39\lib\site-packages\django\urls\base.py", line 30, in reverse resolver = get\_resolver(urlconf) File "D:\ProgramData\Miniconda3\envs\crud-py39\lib\site-packages\django\urls\resolvers.py", line 108, in get\_resolver return \_get\_cached\_resolver(urlconf) TypeError: unhashable type: 'list'

Ah, this traceback is pointing to a subtle but common mistake in your use of Django’s reverse() function. The key error is:

**TypeError: unhashable type: 'list'**

This means somewhere in your call to reverse(), you're passing a list where Django expects a string or hashable value.

### 🔍 Problem Breakdown

Your code:

reverse("api/element", args=[self.initial\_element.id])

The issue is likely with "api/element" — Django expects the **name of a URL pattern**, not a path string.

### ✅ Correct Usage of reverse()

Make sure your urls.py has something like:

path("api/element/<int:id>/", views.get\_element\_by\_id, name="element-detail")

Then in your test:

reverse("element-detail", args=[self.initial\_element.id])

### 🛠️ Fix Your Test Code

Update this line:

response = self.client.get(reverse("element-detail", args=[self.initial\_element.id]))

Make sure "element-detail" matches the name= in your urls.py.

### 🧠 Bonus Tip

If you’re unsure what names are available, run:

python manage.py show\_urls

(You’ll need django-extensions or similar tools for that.)

Or just inspect your urls.py and confirm the name assigned to the route.

Let me know if you want help refactoring your URL patterns or organizing your test cases more cleanly—I’ve got some neat tricks for that too.

## My Prompt

How to setup @override\_settings(ROOT\_URLCONF="test\_urls") if the test\_urls in the string as local variable, not module

Ah, great question! Django’s @override\_settings expects ROOT\_URLCONF to be a **string that points to a Python module**, not a variable holding URL patterns or a dynamic string. So if you have a local variable named test\_urls, you can’t directly pass it to ROOT\_URLCONF—but there’s a clever workaround.

### 🧩 Problem Recap

Let’s say you have something like this in your test file:

test\_urls = [

path("api/test/", test\_view, name="test-view"),

]

This won’t work with:

@override\_settings(ROOT\_URLCONF=test\_urls) # ❌ TypeError: unhashable type: 'list'

### ✅ Solution: Create a Temporary URLConf Module

You need to define a **real Python module** that Django can import. Here's how to do it:

#### 1. **Create a** test\_urls.py **file** in your test directory (or somewhere importable):

# test\_urls.py

from django.urls import path

from your\_app.views import test\_view

urlpatterns = [

path("api/test/", test\_view, name="test-view"),

]

#### 2. **Use** @override\_settings **with the module name as a string**:

from django.test import override\_settings

@override\_settings(ROOT\_URLCONF="your\_test\_package.test\_urls")

def test\_your\_view(client):

response = client.get("/api/test/")

assert response.status\_code == 200

### 🧪 Bonus: Dynamically Create a URLConf (Advanced)

If you **really** want to avoid creating a separate file, you can dynamically create a module using Python’s types and sys.modules. Here's a hacky but valid approach:

import types

import sys

from django.urls import path

from your\_app.views import test\_view

# Create a fake module

test\_urls\_module = types.ModuleType("test\_urls")

test\_urls\_module.urlpatterns = [

path("api/test/", test\_view, name="test-view"),

]

# Inject it into sys.modules

sys.modules["test\_urls"] = test\_urls\_module

# Now override settings

@override\_settings(ROOT\_URLCONF="test\_urls")

def test\_dynamic\_urlconf(client):

response = client.get("/api/test/")

assert response.status\_code == 200

This trick lets Django treat your in-memory URLConf as a real module. Just be cautious with naming collisions in sys.modules.

Let me know if you want help organizing your test suite or setting up isolated routing for specific test cases. I can help you make it clean and scalable.