To mock or not to mock?

What is a test double?

A test double is to code what a stunt double is to an actor.

Types of test doubles

Types of test doubles

Mocks 🎭

Types of test doubles

Mocks 🎭

Stubs 🪤

Types of test doubles

Mocks 🎭

Stubs 🪤

Fakes 🏗️

Types of test doubles

Mocks 🎭

Stubs 🪤

Fakes 🏗️

Spies 🕵️

What is a mock? 🎭

A type of test double that we can use to replace real objects in our code.

Replace real object ➡️ Control behavior ➡️ Track interactions (call arguments)

In `unittest` framework ➡️ create mocks with either `Mock` or `MagicMock`.

example_mock.py

```python
import unittest.mock as mock

mock_object = mock.Mock()

mock_object.some_method(x=1, y=2) # returns a new mock object
mock_object.some_method.assert_called_once()  # after the call, track interactions with the mock

mock_object.some_method(3, y=4)  # call again
mock_object.some_method.assert_has_calls([mock.call(x=1, y=2), mock.call(3, y=4)]) # assert many calls
```

What is a mock? 🎭

A type of test double that we can use to replace real objects in our code.

Replace real object ➡️ Control behavior ➡️ Track interactions (call arguments)

In `unittest` framework ➡️ create mocks with either `Mock` or `MagicMock`.

example_mock.py

```python
import unittest.mock as mock

mock_object = mock.Mock()

mock_object.some_method(x=1, y=2) # returns a new mock object
mock_object.some_method.assert_called_once()  # after the call, track interactions with the mock

mock_object.some_method(3, y=4)  # call again
mock_object.some_method.assert_has_calls([mock.call(x=1, y=2), mock.call(3, y=4)]) # assert many calls
```

What is a mock? 🎭

A type of test double that we can use to replace real objects in our code.

Replace real object ➡️ Control behavior ➡️ Track interactions (call arguments)

In `unittest` framework ➡️ create mocks with either `Mock` or `MagicMock` .

example_mock.py

```python
import unittest.mock as mock

mock_object = mock.Mock()

mock_object.some_method(x=1, y=2) # returns a new mock object
mock_object.some_method.assert_called_once()  # after the call, track interactions with the mock

mock_object.some_method(3, y=4)  # call again
mock_object.some_method.assert_has_calls([mock.call(x=1, y=2), mock.call(3, y=4)]) # assert many calls
```

Difference between `MagicMock` and `Mock`

`MagicMock` supports "magic methods", while `Mock` does not.

Difference between `MagicMock` and `Mock`

`MagicMock` supports "magic methods", while `Mock` does not.

Prefer `MagicMock` when using **magic methods**, like `Sequence` (`list`, `tuple`) or a context manager that defines `__enter__` & `__exit__`.

example_magic_mock_pass.py ✅

```python
import unittest.mock as mock

mock_object = mock.MagicMock()

# MagicMock has __enter__ & __exit__ magic methods 👍
with mock_object as m:
    pass

# any other magic methods are also available 👍
len(mock_object)

# track interactions with the mock object
mock_object.__enter__.assert_called_once()
mock_object.__len__.assert_called_once()
```

example_mock_fail.py ❌

```python
import unittest.mock as mock

mock_object = mock.Mock()

# Mock does not have __enter__ & __exit__
with mock_object as m:
    pass

# TypeError💥 'Mock' object does not support
# the context manager protocol
```

How to replace an object with a test double?

Does the function we are testing **own** the dependency we want to replace?
➡️ Then use `mock.patch` **where the dependency is used** not where it is defined.

test_example_with_dep.py

```python
import example_with_dep
import unittest.mock as mock

# @mock.patch("io.FileIO")  ❌ does not work
@mock.patch("example_with_dep.FileIO") # ✅ works
def test_read_file(mock_io: mock.MagicMock):
  example_with_dep.read_file("some-file.txt")

  # assert that we entered the with block
  mock_io.return_value.__enter__.assert_called_once()
```

example_with_dep.py

```python
from io import FileIO

# function directly uses FileIO
def read_file(filename: str) -> bytes:
  with FileIO(filename) as f:
    return f.read()
```

How to replace an object with a test double?

Does the function we are testing **own** the dependency we want to replace?
➡️ Then use `mock.patch` **where the dependency is used** not where it is defined.

test_example_with_dep.py

```python
import example_with_dep
import unittest.mock as mock

# @mock.patch("io.FileIO")  ❌  does not work
@mock.patch("example_with_dep.FileIO") # ✅  works
def test_read_file(mock_io: mock.MagicMock):
  example_with_dep.read_file("some-file.txt")

  # assert that we entered the with block
  mock_io.return_value.__enter__.assert_called_once()
```

example_with_dep.py

```python
from io import FileIO

# function directly uses FileIO
def read_file(filename: str) -> bytes:
  with FileIO(filename) as f:
    return f.read()
```

How to replace an object with a test double?

Does the function we are testing **own** the dependency we want to replace?
➡️ Then use `mock.patch` **where the dependency is used** not where it is defined.

test_example_with_dep.py

```python
import example_with_dep
import unittest.mock as mock

# @mock.patch("io.FileIO")  ❌ does not work
@mock.patch("example_with_dep.FileIO") # ✅ works
def test_read_file(mock_io: mock.MagicMock):
  example_with_dep.read_file("some-file.txt")

  # assert that we entered the with block
  mock_io.return_value.__enter__.assert_called_once()
```

example_with_dep.py

```python
from io import FileIO

# function directly uses FileIO
def read_file(filename: str) → bytes:
  with FileIO(filename) as f:
    return f.read()
```

Why global patching does not work?

1. `from io import FileIO` binds local reference to `FileIO` in the `example_with_dep` module.
2. Our patch targets `io.FileIO`, but `read_file` uses the local reference.
3. We would have to patch before importing or `reload(example_with_dep)` - both not a good practice.

Better way: use Dependency Injection

If the function we are testing expects the dependency as a parameter.
➡️ pass the `MagicMock` instead, no need to use `mock.patch`

test_example_with_di.py

```python
import example_with_di
import unittest.mock as mock

def test_read_file():
  mock_io = mock.MagicMock(spec=FileIO)

  example_with_di.read_file(mock_io)

  # assert that we entered the with block
  mock_io.read.assert_called_once()

  mock_io.arbitrary_method() # raises AttributeError 💥
  # Mock object has no attribute 'arbitrary_method'
```

example_with_di.py

```python
from io import FileIO

# function directly uses FileIO
def read_file(opened_file: FileIO) → bytes:
  # just operate on the file object
  # caller manages opening/closing
  return opened_file.read()
```

Better way: use Dependency Injection

If the function we are testing expects the dependency as a parameter.

➡️ pass the `MagicMock` instead, no need to use `mock.patch`

test_example_with_di.py

```python
import example_with_di
import unittest.mock as mock

def test_read_file():
    mock_io = mock.MagicMock(spec=FileIO)

    example_with_di.read_file(mock_io)

    # assert that we entered the with block
    mock_io.read.assert_called_once()

    mock_io.arbitrary_method() # raises AttributeError 💥
    # Mock object has no attribute 'arbitrary_method'
```

example_with_di.py

```python
from io import FileIO

# function directly uses FileIO
def read_file(opened_file: FileIO) → bytes:
    # just operate on the file object
    # caller manages opening/closing
    return opened_file.read()
```

Extra: Other ways to patch

Besides the decorator, we can also use `mock.patch` as a context manager.

You can also use `mock.patch.object` to patch attributes on an **already imported** object.

test_example_with_dep_context.py

```python
import example_with_dep
import unittest.mock as mock

def test_read_file():
  with mock.patch(
    "example_with_dep.FileIO", autospec=True
  ) as mock_:
    example_with_dep.read_file("some-file.txt")

    # assert that we entered the with block
    mock_.return_value.__enter__.assert_called_once()
```

test_example_with_dep_object.py

```python
import example_with_dep
import unittest.mock as mock

def test_read_file():
  with mock.patch.object(
    example_with_dep, "FileIO", autospec=True
  ) as mock_io:
    example_with_dep.read_file("some-file.txt")

    # assert that we entered the with block
    mock_.return_value.__enter__.assert_called_once()
```

Use `autospec` to automatically follow `FileIO`'s protocol.

Extra: Other ways to patch

Besides the decorator, we can also use `mock.patch` as a context manager.

You can also use `mock.patch.object` to patch attributes on an **already imported** object.

test_example_with_dep_context.py

```python
import example_with_dep
import unittest.mock as mock

def test_read_file():
  with mock.patch(
    "example_with_dep.FileIO", autospec=True
  ) as mock_:
    example_with_dep.read_file("some-file.txt")

    # assert that we entered the with block
    mock_.return_value.__enter__.assert_called_once()
```

test_example_with_dep_object.py

```python
import example_with_dep
import unittest.mock as mock

def test_read_file():
  with mock.patch.object(
    example_with_dep, "FileIO", autospec=True
  ) as mock_io:
    example_with_dep.read_file("some-file.txt")

    # assert that we entered the with block
    mock_.return_value.__enter__.assert_called_once()
```

Use `autospec` to automatically follow `FileIO`'s protocol.

Extra: Other ways to patch

Besides the decorator, we can also use `mock.patch` as a context manager.

You can also use `mock.patch.object` to patch attributes on an **already imported** object.

test_example_with_dep_context.py

```python
import example_with_dep
import unittest.mock as mock

def test_read_file():
    with mock.patch(
        "example_with_dep.FileIO", autospec=True
    ) as mock_:
        example_with_dep.read_file("some-file.txt")

        # assert that we entered the with block
        mock_.return_value.__enter__.assert_called_once()
```

test_example_with_dep_object.py

```python
import example_with_dep
import unittest.mock as mock

def test_read_file():
    with mock.patch.object(
        example_with_dep, "FileIO", autospec=True
    ) as mock_io:
        example_with_dep.read_file("some-file.txt")

        # assert that we entered the with block
        mock_.return_value.__enter__.assert_called_once()
```

Use `autospec` to automatically follow `FileIO`'s protocol.

What is a stub? 🪤 How does it differ from a mock?

It provides predefined responses to function calls, but does not track interactions.

We can use a `Mock` or `MagicMock` object for stubbing by fixing the `return_value` of a method.

These objects can double as both a mock and a stub.

example_stub_fixed.py

```python
import unittest.mock as mock

mock_object = mock.Mock()

mock_object.some_method.return_value = "some value"
assert mock_object.some_method() == "some value"

mock_object.some_method.side_effect = [1, 2]
assert mock_object.some_method() == 1
assert mock_object.some_method() == 2

# we can also raise exceptions
mock_object.raise_method.side_effect = ValueError("some value")
mock_object.raise_method()  # raises ValueError 💥
```

What is a stub? 🪤 How does it differ from a mock?

It provides predefined responses to function calls, but does not track interactions.

We can use a `Mock` or `MagicMock` object for stubbing by fixing the `return_value` of a method.

These objects can double as both a mock and a stub.

example_stub_fixed.py

```python
import unittest.mock as mock

mock_object = mock.Mock()

mock_object.some_method.return_value = "some value"
assert mock_object.some_method() == "some value"

mock_object.some_method.side_effect = [1, 2]
assert mock_object.some_method() == 1
assert mock_object.some_method() == 2

# we can also raise exceptions
mock_object.raise_method.side_effect = ValueError("some value")
mock_object.raise_method()  # raises ValueError 💥
```

What is a stub? 🪤 How does it differ from a mock?

It provides predefined responses to function calls, but does not track interactions.

We can use a `Mock` or `MagicMock` object for stubbing by fixing the `return_value` of a method.

These objects can double as both a mock and a stub.

example_stub_fixed.py

```python
import unittest.mock as mock

mock_object = mock.Mock()

mock_object.some_method.return_value = "some value"
assert mock_object.some_method() == "some value"

mock_object.some_method.side_effect = [1, 2]
assert mock_object.some_method() == 1
assert mock_object.some_method() == 2

# we can also raise exceptions
mock_object.raise_method.side_effect = ValueError("some value")
mock_object.raise_method()  # raises ValueError 💥
```

What is a stub? 🪤 How does it differ from a mock?

It provides predefined responses to function calls, but does not track interactions.

We can use a `Mock` or `MagicMock` object for stubbing by fixing the `return_value` of a method.

These objects can double as both a mock and a stub.

example_stub_fixed.py

```python
import unittest.mock as mock

mock_object = mock.Mock()

mock_object.some_method.return_value = "some value"
assert mock_object.some_method() == "some value"

mock_object.some_method.side_effect = [1, 2]
assert mock_object.some_method() == 1
assert mock_object.some_method() == 2

# we can also raise exceptions
mock_object.raise_method.side_effect = ValueError("some value")
mock_object.raise_method()  # raises ValueError 💥
```

An example: generating HTTP headers for a request

You are given an in-house authentication library to get a token, with the following signature:

authlib.py

```python
def authenticate(account_id: str, resource_id: Optional[str] = None) → str:
    """
    Calls a remote authentication service to get a token for a specific resource.

    Args
    ————
        account_id: str, example "some-project-dev"
        resource_id: Optional[str], a target service id that
            we want to authneticate for, e.g. "STORAGE-SERVICE-XXXXXX"

    Returns
    ———————
        str, a token
    """
    ...
```

An example: generating HTTP headers for a request

We want to include the token in the `Authorization` header of our HTTP requests, along with some static headers. These headers are part of a client library we are writing.

headers.py

```python
class Configuration:
 user_id: str
 resource_id: Optional[str] = None

def get_headers(config: Configuration) → dict:
 token = authlib.authenticate(
   config.user_id,
   config.resource_id
 )

 return {
   "Content-Type": "application/json",
   "Authorization": f"Bearer {token}"
 }
```

An example: generating HTTP headers for a request

We want to include the token in the `Authorization` header of our HTTP requests, along with some static headers. These headers are part of a client library we are writing.

headers.py

```python
class Configuration:
 user_id: str
 resource_id: Optional[str] = None

def get_headers(config: Configuration) → dict:
 token = authlib.authenticate(
   config.user_id,
   config.resource_id
 )

 return {
   "Content-Type": "application/json",
   "Authorization": f"Bearer {token}"
 }
```

An example: generating HTTP headers for a request

We want to include the token in the `Authorization` header of our HTTP requests, along with some static headers. These headers are part of a client library we are writing.

headers.py

```python
class Configuration:
 user_id: str
 resource_id: Optional[str] = None

def get_headers(config: Configuration) → dict:
 token = authlib.authenticate(
   config.user_id,
   config.resource_id
 )

 return {
   "Content-Type": "application/json",
   "Authorization": f"Bearer {token}"
 }
```

Mocks + stubs: verify behaviour and results

We stub `authenticate` to control the token with expected results

test_headers.py

```python
@mock.patch("headers.authlib.authenticate")
def test_get_headers(self, mock_auth):
   mock_auth.return_value = "token"

   expected = {
     "Content-Type": "application/json",
     "Authorization": "Bearer token"
   }

   actual = get_headers(mock.Mock())
   self.assertEqual(actual, expected)  # validate result
```

headers.py

```python
import authlib

@dataclass
class Configuration:
 user_id: str
 resource_id: Optional[str] = None

def get_headers(config: Configuration) → dict:
 token = authlib.authenticate(
    config.user_id
    # a bug 🐞 here
 )

 return {
   "Content-Type": "application/json",
   "Authorization": f"Bearer {token}"
 }
```

Mocks + stubs: verify behaviour and results

We stub `authenticate` to control the token with expected results, then assert the arguments of the call.

test_headers.py

```python
@mock.patch("headers.authlib.authenticate")
def test_get_headers(self, mock_auth):
    config = {
      "user_id": "a_machine",
      "resource_id": "STORAGE-SERVICE-XXXXXX",
    }
    mock_auth.return_value = "token"

    expected = {
      "Content-Type": "application/json",
      "Authorization": "Bearer token"
    }

    actual = get_headers(mock.Mock(**config))
    self.assertEqual(actual, expected)  # validate result

    # validate interaction with 3rd party
    mock_auth.assert_called_once_with(
      config["user_id"],
      config["resource_id"]
    )
```

headers.py

```python
import authlib

@dataclass
class Configuration:
 user_id: str
 resource_id: Optional[str] = None

def get_headers(config: Configuration) → dict:
 token = authlib.authenticate(
   config.user_id,
   config.resource_id  # was missing
 )

 return {
   "Content-Type": "application/json",
   "Authorization": f"Bearer {token}"
 }
```

What is a fake? 🏗️

A fake is a working implementation, but it is kept lightweight for testing purposes.

Avoids complex dependencies, I/O operations, or external services ➕ no need maintaining stub states.

storage.py

```python
# real dependency (talks to a service)
# assume StorageProvider implements connection methods
class StorageClient(Mapping, StorageProvider):
  def __enter__(self):
    self.connect()
    return self

  def __exit__(self, exc_type, exc_value, traceback):
    self.disconnect()

  def __getitem__(self, key: str):
    return self.read_from_service(key)

  def __setitem__(self, key: str, value):
    self.write_to_service(key, value)
```

fake_storage.py

```python
# fake dependency (in-memory implementation)
class FakeStorageClient(Mapping):
  def __init__(self):
    self.__store = {}

  def __enter__(self):
    return self

  def __exit__(self, exc_type, exc_value, traceback):
    pass

  def __getitem__(self, key: str):
    return self.__store.get(key)

  def __setitem__(self, key: str, value):
    self.__store[key] = value
```

An example: SerDe on top of StorageClient

Imagine we build serialization + chunking on top of the StorageClient

service.py

```python
import json
from storage import StorageClient

def save_object(storage: StorageClient, key: str, obj: dict, *, chunk_size: int = 8) -> None:
  """Serialize to JSON, split into fixed-size chunks, store parts + index."""
  data = json.dumps(obj).encode("utf-8")  # encode
  chunks = [data[i:i+chunk_size] for i in range(0, len(data), chunk_size)]
  for i, chunk in enumerate(chunks):
      storage[f"{key}/chunk/{i}"] = chunk
  storage[f"{key}/chunks_length"] = len(chunks)

def load_object(storage: StorageClient, key: str) -> dict:
    """Read index, reassemble chunks, deserialize."""
    n = storage[f"{key}/chunks_length"]
    data = b"".join(storage[f"{key}/chunk/{i}"] for i in range(n))
    return json.loads(data.decode("utf-8"))
```

An example: SerDe on top of StorageClient

Imagine we build serialization + chunking on top of the StorageClient

service.py

```python
import json
from storage import StorageClient

def save_object(storage: StorageClient, key: str, obj: dict, *, chunk_size: int = 8) -> None:
  """Serialize to JSON, split into fixed-size chunks, store parts + index."""
  data = json.dumps(obj).encode("utf-8")  # encode
  chunks = [data[i:i+chunk_size] for i in range(0, len(data), chunk_size)]
  for i, chunk in enumerate(chunks):
      storage[f"{key}/chunk/{i}"] = chunk
  storage[f"{key}/chunks_length"] = len(chunks)

def load_object(storage: StorageClient, key: str) -> dict:
    """Read index, reassemble chunks, deserialize."""
    n = storage[f"{key}/chunks_length"]
    data = b"".join(storage[f"{key}/chunk/{i}"] for i in range(n))
    return json.loads(data.decode("utf-8"))
```

An example: SerDe on top of StorageClient

Imagine we build serialization + chunking on top of the StorageClient

service.py

```python
import json
from storage import StorageClient

def save_object(storage: StorageClient, key: str, obj: dict, *, chunk_size: int = 8) -> None:
  """Serialize to JSON, split into fixed-size chunks, store parts + index."""
  data = json.dumps(obj).encode("utf-8")  # encode
  chunks = [data[i:i+chunk_size] for i in range(0, len(data), chunk_size)]
  for i, chunk in enumerate(chunks):
      storage[f"{key}/chunk/{i}"] = chunk
  storage[f"{key}/chunks_length"] = len(chunks)

def load_object(storage: StorageClient, key: str) -> dict:
    """Read index, reassemble chunks, deserialize."""
    n = storage[f"{key}/chunks_length"]
    data = b"".join(storage[f"{key}/chunk/{i}"] for i in range(n))
    return json.loads(data.decode("utf-8"))
```

An example: SerDe on top of StorageClient

Imagine we build serialization + chunking on top of the StorageClient

service.py

```python
import json
from storage import StorageClient

def save_object(storage: StorageClient, key: str, obj: dict, *, chunk_size: int = 8) -> None:
  """Serialize to JSON, split into fixed-size chunks, store parts + index."""
  data = json.dumps(obj).encode("utf-8")  # encode
  chunks = [data[i:i+chunk_size] for i in range(0, len(data), chunk_size)]
  for i, chunk in enumerate(chunks):
      storage[f"{key}/chunk/{i}"] = chunk
  storage[f"{key}/chunks_length"] = len(chunks)

def load_object(storage: StorageClient, key: str) -> dict:
    """Read index, reassemble chunks, deserialize."""
    n = storage[f"{key}/chunks_length"]
    data = b"".join(storage[f"{key}/chunk/{i}"] for i in range(n))
    return json.loads(data.decode("utf-8"))
```

An example: SerDe on top of StorageClient

Imagine we build serialization + chunking on top of the StorageClient

service.py

```python
import json
from storage import StorageClient

def save_object(storage: StorageClient, key: str, obj: dict, *, chunk_size: int = 8) -> None:
  """Serialize to JSON, split into fixed-size chunks, store parts + index."""
  data = json.dumps(obj).encode("utf-8")  # encode
  chunks = [data[i:i+chunk_size] for i in range(0, len(data), chunk_size)]
  for i, chunk in enumerate(chunks):
      storage[f"{key}/chunk/{i}"] = chunk
  storage[f"{key}/chunks_length"] = len(chunks)

def load_object(storage: StorageClient, key: str) -> dict:
    """Read index, reassemble chunks, deserialize."""
    n = storage[f"{key}/chunks_length"]
    data = b"".join(storage[f"{key}/chunk/{i}"] for i in range(n))
    return json.loads(data.decode("utf-8"))
```

An example: SerDe on top of StorageClient

Imagine we build serialization + chunking on top of the StorageClient

service.py

```python
import json
from storage import StorageClient

def save_object(storage: StorageClient, key: str, obj: dict, *, chunk_size: int = 8) -> None:
  """Serialize to JSON, split into fixed-size chunks, store parts + index."""
  data = json.dumps(obj).encode("utf-8")  # encode
  chunks = [data[i:i+chunk_size] for i in range(0, len(data), chunk_size)]
  for i, chunk in enumerate(chunks):
      storage[f"{key}/chunk/{i}"] = chunk
  storage[f"{key}/chunks_length"] = len(chunks)

def load_object(storage: StorageClient, key: str) -> dict:
    """Read index, reassemble chunks, deserialize."""
    n = storage[f"{key}/chunks_length"]
    data = b"".join(storage[f"{key}/chunk/{i}"] for i in range(n))
    return json.loads(data.decode("utf-8"))
```

An example: SerDe on top of StorageClient

Imagine we build serialization + chunking on top of the StorageClient

service.py

```python
import json
from storage import StorageClient

def save_object(storage: StorageClient, key: str, obj: dict, *, chunk_size: int = 8) -> None:
  """Serialize to JSON, split into fixed-size chunks, store parts + index."""
  data = json.dumps(obj).encode("utf-8")  # encode
  chunks = [data[i:i+chunk_size] for i in range(0, len(data), chunk_size)]
  for i, chunk in enumerate(chunks):
      storage[f"{key}/chunk/{i}"] = chunk
  storage[f"{key}/chunks_length"] = len(chunks)

def load_object(storage: StorageClient, key: str) -> dict:
    """Read index, reassemble chunks, deserialize."""
    n = storage[f"{key}/chunks_length"]
    data = b"".join(storage[f"{key}/chunk/{i}"] for i in range(n))
    return json.loads(data.decode("utf-8"))
```

An example: SerDe on top of StorageClient

Imagine we build serialization + chunking on top of the StorageClient

service.py

```python
import json
from storage import StorageClient

def save_object(storage: StorageClient, key: str, obj: dict, *, chunk_size: int = 8) -> None:
 """Serialize to JSON, split into fixed-size chunks, store parts + index."""
 data = json.dumps(obj).encode("utf-8")  # encode
 chunks = [data[i:i+chunk_size] for i in range(0, len(data), chunk_size)]
 for i, chunk in enumerate(chunks):
   storage[f"{key}/chunk/{i}"] = chunk
 storage[f"{key}/chunks_length"] = len(chunks)

def load_object(storage: StorageClient, key: str) -> dict:
  """Read index, reassemble chunks, deserialize."""
  n = storage[f"{key}/chunks_length"]
  data = b"".join(storage[f"{key}/chunk/{i}"] for i in range(n))
  return json.loads(data.decode("utf-8"))
```

An example: SerDe on top of StorageClient

Imagine we build serialization + chunking on top of the StorageClient

service.py

```python
import json
from storage import StorageClient

def save_object(storage: StorageClient, key: str, obj: dict, *, chunk_size: int = 8) -> None:
  """Serialize to JSON, split into fixed-size chunks, store parts + index."""
  data = json.dumps(obj).encode("utf-8")  # encode
  chunks = [data[i:i+chunk_size] for i in range(0, len(data), chunk_size)]
  for i, chunk in enumerate(chunks):
      storage[f"{key}/chunk/{i}"] = chunk
  storage[f"{key}/chunks_length"] = len(chunks)

def load_object(storage: StorageClient, key: str) -> dict:
    """Read index, reassemble chunks, deserialize."""
    n = storage[f"{key}/chunks_length"]
    data = b"".join(storage[f"{key}/chunk/{i}"] for i in range(n))
    return json.loads(data.decode("utf-8"))
```

Using stubs to test this gets complicated

test_with_stub.py

```python
from storage
from service import save_json, load_json

def test_save_load_round_trip():
  # Stub only: no state, everything pre-scripted
  storage = mock.Mock(spec=StorageClient)
  payload = {"name": "Alice", "age": 30, "bio": "lipsum" * 100}
  data = json.dumps(payload).encode("utf-8")
  parts = [data[i:i+8] for i in range(0, len(data), 8)]  # duplicate chunking logic 😬

  # omitted for brevity, we need to stub __getitem__ / __setitem__ for index and chunks
  # can be several lines of implementation just for the sake of stubbing
  storage.__getitem__.side_effect = ...
  storage.__setitem__.side_effect = ...

  save_json(storage, "user", payload, chunk_size=8)
  assert load_json(storage, "user") == payload
```

Using stubs to test this gets complicated

test_with_stub.py

```python
from storage
from service import save_json, load_json

def test_save_load_round_trip():
  # Stub only: no state, everything pre-scripted
  storage = mock.Mock(spec=StorageClient)
  payload = {"name": "Alice", "age": 30, "bio": "lipsum" * 100}
  data = json.dumps(payload).encode("utf-8")
  parts = [data[i:i+8] for i in range(0, len(data), 8)]  # duplicate chunking logic 😬

  # omitted for brevity, we need to stub __getitem__ / __setitem__ for index and chunks
  # can be several lines of implementation just for the sake of stubbing
  storage.__getitem__.side_effect = ...
  storage.__setitem__.side_effect = ...

  save_json(storage, "user", payload, chunk_size=8)
  assert load_json(storage, "user") == payload
```

Using stubs to test this gets complicated

test_with_stub.py

```python
from storage
from service import save_json, load_json

def test_save_load_round_trip():
  # Stub only: no state, everything pre-scripted
  storage = mock.Mock(spec=StorageClient)
  payload = {"name": "Alice", "age": 30, "bio": "lipsum" * 100}
  data = json.dumps(payload).encode("utf-8")
  parts = [data[i:i+8] for i in range(0, len(data), 8)]  # duplicate chunking logic 😬

  # omitted for brevity, we need to stub __getitem__ / __setitem__ for index and chunks
  # can be several lines of implementation just for the sake of stubbing
  storage.__getitem__.side_effect = ...
  storage.__setitem__.side_effect = ...

  save_json(storage, "user", payload, chunk_size=8)
  assert load_json(storage, "user") == payload
```

Faking with dependency injection

FakeStoragClient is a drop in replacement for StorageClient.

Just inject the FakeStorageClient, you can also wrap it through a Mock, to track interactions

test_with_fake.py

```python
from fake_storage import FakeStorageClient
from service import save_json, load_json

def test_save_load_round_trip():
  fake = FakeStorageClient()
  save_json(mock_fake, "user_001", {"name": "Alice", "age": 30, "bio": "lipsum" * 100})
  assert load_json(mock_fake, "user_001") == {"name": "Alice", "age": 30, "bio": "lipsum" * 100}


def test_save_load_round_trip_fake_plus_mock():
  fake = FakeStorageClient()
  mock_fake = mock.MagicMock(wraps=fake, autospec=True)

  save_json(mock_fake, "user_001", {"name": "Alice", "age": 30, "bio": "lipsum" * 100})
  assert load_json(mock_fake, "user_001") == {"name": "Alice", "age": 30, "bio": "lipsum" * 100}

  mock_fake.__getitem__.assert_called_with("user_001/chunks_length")
```

Faking with dependency injection

FakeStoragClient is a drop in replacement for StorageClient.

Just inject the FakeStorageClient, you can also wrap it through a Mock, to track interactions

test_with_fake.py

```python
from fake_storage import FakeStorageClient
from service import save_json, load_json

def test_save_load_round_trip():
  fake = FakeStorageClient()
  save_json(mock_fake, "user_001", {"name": "Alice", "age": 30, "bio": "lipsum" * 100})
  assert load_json(mock_fake, "user_001") == {"name": "Alice", "age": 30, "bio": "lipsum" * 100}


def test_save_load_round_trip_fake_plus_mock():
  fake = FakeStorageClient()
  mock_fake = mock.MagicMock(wraps=fake, autospec=True)

  save_json(mock_fake, "user_001", {"name": "Alice", "age": 30, "bio": "lipsum" * 100})
  assert load_json(mock_fake, "user_001") == {"name": "Alice", "age": 30, "bio": "lipsum" * 100}

  mock_fake.__getitem__.assert_called_with("user_001/chunks_length")
```

What is a spy? 🕵️

A spy is a test double that wraps a real object, allowing us to monitor its interactions while still using its actual implementation.

We can use a `MagicMock` / `Mock` with the `wraps` argument to create a spy.

example_spy.py

```python
import unittest.mock as mock

class DollarConverter:
  rates = {
    "USD": 1,
    "EUR": 0.9,
    "GBP": 0.8,
  } # static data

  def convert(self, amount: float, currency: str) → float:
    return rates.get(currency, 0) * amount

spy_object = mock.Mock(wraps=DollarConverter(), autospec=True)
euros = spy_object.convert(10, "EUR")

assert euros == 9  # uses the real method
spy_object.convert.assert_called_once_with(10, "EUR")  # tracks the call
```

What is a spy? 🕵️

A spy is a test double that wraps a real object, allowing us to monitor its interactions while still using its actual implementation.

We can use a `MagicMock` / `Mock` with the `wraps` argument to create a spy.

example_spy.py

```python
import unittest.mock as mock

class DollarConverter:
  rates = {
    "USD": 1,
    "EUR": 0.9,
    "GBP": 0.8,
  } # static data

  def convert(self, amount: float, currency: str) -> float:
    return rates.get(currency, 0) * amount

spy_object = mock.Mock(wraps=DollarConverter(), autospec=True)
euros = spy_object.convert(10, "EUR")

assert euros == 9  # uses the real method
spy_object.convert.assert_called_once_with(10, "EUR")  # tracks the call
```

What is a spy? 🕵️

A spy is a test double that wraps a real object, allowing us to monitor its interactions while still using its actual implementation.

We can use a `MagicMock` / `Mock` with the `wraps` argument to create a spy.

example_spy.py

```python
import unittest.mock as mock

class DollarConverter:
  rates = {
    "USD": 1,
    "EUR": 0.9,
    "GBP": 0.8,
  } # static data

  def convert(self, amount: float, currency: str) -> float:
    return rates.get(currency, 0) * amount

spy_object = mock.Mock(wraps=DollarConverter(), autospec=True)
euros = spy_object.convert(10, "EUR")

assert euros == 9  # uses the real method
spy_object.convert.assert_called_once_with(10, "EUR")  # tracks the call
```

What is a spy? 🕵️

A spy is a test double that wraps a real object, allowing us to monitor its interactions while still using its actual implementation.

We can use a `MagicMock` / `Mock` with the `wraps` argument to create a spy.

example_spy.py

```python
import unittest.mock as mock

class DollarConverter:
  rates = {
    "USD": 1,
    "EUR": 0.9,
    "GBP": 0.8,
  } # static data

  def convert(self, amount: float, currency: str) -> float:
    return rates.get(currency, 0) * amount

spy_object = mock.Mock(wraps=DollarConverter(), autospec=True)
euros = spy_object.convert(10, "EUR")

assert euros == 9  # uses the real method
spy_object.convert.assert_called_once_with(10, "EUR")  # tracks the call
```

Takeaway

Takeaway

Mock 🎭
verifies behavior

Takeaway

Mock 🎭
verifies behavior

Stub 🪤
controls outputs

Takeaway

Mock 🎭
verifies behavior

Stub 📦
controls outputs

Fake 🏗️
working minimal replacement

Takeaway

Mock 🎭
verifies behavior

Stub 🪤
controls outputs

Fake 🏗️
working minimal replacement

Spy 🕵️
wraps real code for inspection

Takeaway

Takeaway

👉 Prefer `MagicMock` over `Mock` unless you are sure you don't need magic methods

Takeaway

- 👉 Prefer `MagicMock` over `Mock` unless you are sure you don't need magic methods
- 👉 Use `autospec` / `spec` to make the mock strictly follow the real object's protocol

Takeaway

- 👉 Prefer `MagicMock` over `Mock` unless you are sure you don't need magic methods
- 👉 Use `autospec` / `spec` to make the mock strictly follow the real object's protocol
- 👉 Prefer Dependency Injection over patching when possible, use patching when you interact with 3rd party code, that is not modular

Takeaway

- 👉 Prefer `MagicMock` over `Mock` unless you are sure you don't need magic methods
- 👉 Use `autospec` / `spec` to make the mock strictly follow the real object's protocol
- 👉 Prefer Dependency Injection over patching when possible, use patching when you interact with 3rd party code, that is not modular
- 👉 If you need to patch, **patch where the dependency is used**

Takeaway

- 👉 Prefer `MagicMock` over `Mock` unless you are sure you don't need magic methods
- 👉 Use `autospec` / `spec` to make the mock strictly follow the real object's protocol
- 👉 Prefer Dependency Injection over patching when possible, use patching when you interact with 3rd party code, that is not modular
- 👉 If you need to patch, **patch where the dependency is used**
- 👉 In `unittest`, `Mock` & `MagicMock` can be used in a way that combines all types of test doubles, to achieve completeness

Thank you very much! 😊