Flutter Mock Testing

Testing with Fake Objects

Isolation, Control, and Reliability in Testing

Test Doubles

Mock 🎭

- Imagine you ask a toy robot:
  "Did you wave at me?"
- The robot pretends to wave and keeps track:
  ✅ Yes, I waved!

Mocks pretend and check if they did what you asked.

- Simulates behavior
- Verifies interactions
- Returns predefined responses

Stub 📝

- Imagine asking a teddy bear:
  "What's 2 + 2?"
- Teddy just says: "4"
- No thinking, just reading from a script.

Stubs always give the same answer you wrote for them.

- Provides predetermined responses
- No behavior verification

Fake ⛱️

- Instead of a real playground, you make a small toy playground inside your room.
- It works like the real one, but simpler.

Fakes are tiny working versions (like a toy kitchen or in-memory database).

- Working implementation
- Simplified for testing (e.g., in-memory database)

Comparison of the Three Appoaches

Testing is like playing pretend with toys!

- **Mock** 🎭 = Pretender → checks if it did what you asked
- **Stub** 📝 = Script Reader → always gives the same answer
- **Fake** 🎪 = Mini Playground → small, simple but working version

We use a Mock in this course, but the Mock can use Stubs in it.

Mocking

Mocking creates fake objects that simulate real dependencies:

Real-world analogy:

Like using a **crash test dummy** instead of a real person:

- **Safe** - No real harm if something goes wrong
- **Controlled** - Behaves exactly as we specify
- **Consistent** - Same behavior every time
- **Fast** - No external dependencies

Why Use Mocks?

✅ Isolation:

- Test one component at a time
- Remove external dependencies (databases, APIs, files)

✅ Control:

- Predict exactly how dependencies behave
- Test error scenarios safely

✅ **Speed:**

- No network calls or database operations
- Tests run instantly

✅ **Reliability:**

- Tests don't fail due to external issues
- Consistent results every time

Testing Without Mocks vs With Mocks

❌ Without Mocks:

```
test('should save todo', () async {
  final repository = InMemoryTodoRepository();
  final todo = Todo(id: '1', title: 'Test');

  await repository.saveTodo(todo);
  final todos = await repository.getAllTodos();

  expect(todos.contains(todo), true);
});
```

Depends on real repository implementation

✅ With Mocks:

```
test('should save todo', () async {
  final mockRepo = MockTodoRepository();
  final todo = Todo(id: '1', title: 'Test');

  await mockRepo.saveTodo(todo);

  verify(mockRepo.saveTodo(todo)).called(1);
});
```

Tests just the interaction, not implementation

Using Mock in Flutter

Setting Up Mockito

1. Add dependencies to `pubspec.yaml`:

```yaml
dev_dependencies:
  flutter_test:
    sdk: flutter
  mockito: ^5.4.4      # Mocking framework
  build_runner: ^2.4.9 # Code generation
```

2. Generate mocks:

```
flutter packages pub run build_runner build
```

Creating Mock Objects

Step 1: Annotate your test file

```dart
import 'package:mockito/annotations.dart';
import 'package:todo/repositories/todo_repository.dart';

// Generate mocks for these classes
// @GenerateNiceMocks provides default return values automatically
@GenerateNiceMocks([MockSpec<TodoService>()])
import 'simple_mock_test.mocks.dart';
```

From the "4 Tests/todo/test/mocks/simple_mock_test.dart".

Step 2: Use the generated mock

```
void main() {
  late MockTodoRepository mockRepository;

  setUp(() {
    mockRepository = MockTodoRepository();
  });
}
```

Generate the Mock and run the tests

- Run `dart run build_runner build`

In the same directory "simple_mock_test.mocks.dart" will be generated.

- Run `flutter test test/mocks/simple_simple_mock_test.dart` to run Mock tests.

Basic Mock Creation Test

```
test('should create a mock repository', () {
  // Arrange & Act
  final mock = MockTodoService();

  // Assert
  expect(mock, isA<TodoService>());
  expect(mock, isA<MockTodoService>());
});
```

Key Points:

- Mock implements the original interface
- Can be used wherever the real object is expected
- Provides all methods but with no default behavior

Understanding the Mock

Mock TodoService:

- Has NO database, NO storage, NO real data
- It's completely fake/empty
- getAllTodos() has no idea what to return by default

The same interface

We need to include the `todo_service.dart` because the mock must have the exact same interface (same methods, same parameters, same return types).

```
import 'package:todo/services/todo_service.dart';
```

Setting Mock Behavior: when().thenReturn()

Basic behavior setup:

We can make the Mock to return the Stub (expectedTodos in this example).

```
test('should return predefined todos', () async {
  // Arrange - Define what the mock should return
  final expectedTodos = [
    Todo(id: '1', title: 'Mock Todo 1'),
    Todo(id: '2', title: 'Mock Todo 2', isCompleted: true),
  ];

  when(mockRepository.getAllTodos())
      .thenAnswer((_) async => expectedTodos);
```

From now on, when getAllTodos() are called, the Stub is returned.

```
  // Act - Call the mocked method
  final result = await mockRepository.getAllTodos();

  // Assert - Verify the result
  expect(result, expectedTodos);
  expect(result.length, 2);
});
```

That's the point of mocks - they have no real logic, so you control exactly what they return for testing.

Mock Behavior Patterns

1. Simple return value:

```
when(mockRepository.getAllTodos())
    .thenAnswer((_) async => []);
```

2. Multiple return values:

```
when(mockRepository.getAllTodos())
    .thenAnswer((_) async => [Todo(id: '1', title: 'First')])
    .thenAnswer((_) async => [Todo(id: '2', title: 'Second')]);
```

3. Throw exceptions:

```
when(mockRepository.getAllTodos())
    .thenThrow(Exception('Network error'));
```

Verifying Mock Interactions

Basic verification:

```
test('should verify that saveTodo was called', () async {
  // Arrange
  final todo = Todo(id: '1', title: 'Test Todo');

  // Act
  await mockRepository.saveTodo(todo);

  // Assert - Verify the method was called
  verify(mockRepository.saveTodo(todo)).called(1);
});
```

`verify()` checks that a method was called with specific parameters.

- This does NOTHING except record "saveTodo was called with this todo".

```
// Act
await mockRepository.saveTodo(todo);
```

- Then it checks the record: "Was saveTodo called once?"

```
// Assert
verify(mockRepository.saveTodo(todo)).called(1);
```

Verification Patterns

1. Verify specific parameters:

```
verify(mockRepository.deleteTodo('test-id')).called(1);
```

2. Verify method was never called:

```
verifyNever(mockRepository.getAllTodos());
```

3. Verify multiple calls:

```
verify(mockRepository.saveTodo(todo1)).called(1);
verify(mockRepository.saveTodo(todo2)).called(1);
verify(mockRepository.getAllTodos()).called(1);
```

Argument Matchers

1. Match any argument:

```
when(mockRepository.deleteTodo(any))
    .thenAnswer((_) async {});

// Both calls will work
await mockRepository.deleteTodo('any-id');
await mockRepository.deleteTodo('another-id');

verify(mockRepository.deleteTodo('any-id')).called(1);
```

2. Match with conditions:

```
when(mockRepository.deleteTodo(argThat(startsWith('test'))))
    .thenAnswer((_) async {});
```

Error Simulation

Testing network failures:

```
test('should handle network error', () async {
  // Arrange - Mock throws exception
  when(mockRepository.getAllTodos())
      .thenThrow(Exception('Network error'));

  // Act & Assert - Expect exception
  expect(
    () async => await mockRepository.getAllTodos(),
    throwsA(isA<Exception>()),
  );
});
```

💡 **Mocks let you test error handling without real failures**

Different Error Types

```
test('should handle different error types', () async {
  // Arrange
  when(mockRepository.saveTodo(any))
      .thenThrow(ArgumentError('Invalid todo'));

  // Act & Assert
  final todo = Todo(id: '1', title: 'Test');
  expect(
    () async => await mockRepository.saveTodo(todo),
    throwsA(isA<ArgumentError>()),
  );
});
```

Practical Mock Example

Complete workflow test:

```
test('should simulate a complete workflow', () async {
  // Arrange - Set up realistic scenario
  final existingTodos = [Todo(id: '1', title: 'Existing Todo')];
  final newTodo = Todo(id: '2', title: 'New Todo');

  when(mockRepository.getAllTodos())
      .thenAnswer((_) async => existingTodos);
  when(mockRepository.saveTodo(any))
      .thenAnswer((_) async {});

  // Act - Simulate app workflow
  final currentTodos = await mockRepository.getAllTodos();
  await mockRepository.saveTodo(newTodo);

  // Assert - Verify both behavior and interactions
  expect(currentTodos.length, 1);
  verify(mockRepository.getAllTodos()).called(1);
  verify(mockRepository.saveTodo(newTodo)).called(1);
});
```

Mock vs Real Object Comparison

Aspect	Real Object	Mock Object
Speed	Slow (I/O operations)	Fast (in-memory)
Control	Hard to control	Complete control
Reliability	May fail externally	Always predictable
Testing	Tests implementation	Tests interaction
Setup	Complex setup needed	Simple setup

When to Use Mocks

✅ **Use mocks for:**

- External services (APIs, databases)
- File system operations
- Network requests
- Time-dependent operations
- Complex dependencies

❌ **Don't mock:**

- Simple value objects (Models)
- The system under test itself
- Everything (over-mocking makes tests brittle)

Mock Testing Best Practices

✅ **DO:**

- Mock external dependencies only
- Use descriptive test names
- Verify important interactions
- Test both success and error cases
- Keep mocks simple

❌ **DON'T:**

- Mock everything
- Create overly complex mock behaviors
- Forget to verify important calls
- Use mocks for simple value objects

Common Mock Patterns

1. Repository Pattern:

```
@GenerateMocks([TodoRepository])
// Mock data access layer
```

2. Service Pattern:

```
@GenerateMocks([NetworkService, AuthService])
// Mock external services
```

3. Provider Pattern:

```
@GenerateMocks([UserProvider, SettingsProvider])
// Mock state providers
```

Running Mock Tests

Generate mocks first:

```
dart run build_runner build
```

Run tests:

```
flutter test test/mocks/simple_mock_test.dart
flutter test test/mocks/  # All mock tests
```

Watch for changes:

```
flutter packages pub run build_runner watch
```

Mock vs Other Testing Types

Unit Tests with Mocks:

- Test single component in isolation
- Fast and reliable
- Test business logic

Integration Tests:

- Test components working together
- Use real implementations
- Test data flow

Widget Tests:

- Test UI components

Real-World Mock Example

Testing a ViewModel with Repository:

```
class TodoViewModel {
  final TodoRepository repository;
  TodoViewModel(this.repository);

  Future<void> loadTodos() async {
    final todos = await repository.getAllTodos();
    // Update UI state
  }
}
```

```dart
// Test
import 'package:todo/view_models/todo_view_model.dart';   // ← Test this
@GenerateNiceMocks([MockSpec<TodoRepository>()])  // ← Mock the dependency
import 'todo_view_model_test.mocks.dart';

test('should load todos from repository', () async {
  final mockRepo = MockTodoRepository();
  final viewModel = TodoViewModel(mockRepo);

  when(mockRepo.getAllTodos()).thenAnswer((_) async => []);

  await viewModel.loadTodos();

  verify(mockRepo.getAllTodos()).called(1);
});
```

Advanced Mock Features

Argument capturing:

```
final captured = verify(mockRepository.saveTodo(captureAny)).captured;
expect(captured.first.title, 'Expected Title');
```

Call counting:

```
verify(mockRepository.getAllTodos()).called(greaterThan(1));
```

Sequential returns:

```
when(mockRepository.getAllTodos())
    .thenAnswer((_) async => [])
    .thenAnswer((_) async => [todo]);
```

Debugging Mock Issues

Common problems:

1. **Mock not generated:**
   - Run `build_runner build`
   - Check import path
2. **Verification fails:**
   - Check exact parameter matching
   - Use `any` for flexible matching
3. **Unexpected behavior:**
   - Ensure `when()` is called before test
   - Check method signatures match

Summary

- **Mocks** create controlled fake objects for testing
- **Mockito** provides easy mock generation and setup
- **when().thenReturn()** defines mock behavior
- **verify()** checks method interactions
- **Mocks enable** fast, reliable, isolated testing
- **Use mocks** for external dependencies, not simple objects

Remember: Mock external dependencies to create fast, reliable, isolated tests!