Lightning-Fast Apex Test Project Template

This template is designed to be used as a **starting point for any new Salesforce project** where test execution speed, determinism, and maintainability are first-class requirements.

Each section includes an explanation of the code, what it does, and how it should be used, with full source code included.

---

1. Core Principles (Read First)

Explanation: This section lists the guiding principles for building fast, maintainable tests and architecture.

What this does: It sets expectations for how production and test code should be structured to maximize test speed, reliability, and maintainability.

```
1. No database unless unavoidable
2. Mock before you insert
3. Unit tests > service tests > integration tests
4. SeeAllData=false always
5. Architecture enables speed
```

---

2. Recommended Folder Structure

Explanation: Defines the recommended layout of classes and tests in Salesforce DX projects.

What this does: Helps developers quickly locate production code, unit tests, service tests, integration tests, mocks, and test data factories, enforcing a clean separation of concerns.

```
force-app/
  main/
    default/
      classes/
        core/
          interfaces/
          services/
          selectors/
          domain/
          utils/
        test/
          factories/
          mocks/
          unit/
          service/
          integration/
```

---

3. Production Code Conventions

Explanation: Sets rules for how production code should be written, with a focus on testability.

What this does: Ensures code is modular, easy to mock, and follows dependency injection principles.

3.1 Interfaces First

Explanation: Every external dependency is wrapped in an interface.

What this does: Allows mocking in tests and avoids hard-coded dependencies.

```
public interface IAccountSelector {
    Account getById(Id accountId);
}

public interface IClock {
    Datetime now();
}
```

3.2 Selectors (All SOQL Lives Here)

Explanation: Place all SOQL queries in selector classes.

What this does: Prevents business logic classes from directly querying Salesforce, enabling easier testing and mocking.

```
public class AccountSelector implements IAccountSelector {
    public Account getById(Id accountId) {
        return [SELECT Id, Name FROM Account WHERE Id = :accountId];
    }
}
```

3.3 Services (Business Logic)

Explanation: Business logic is encapsulated in services.

What this does: Keeps logic separate from database queries and triggers, making it easier to test and maintain.

```
public class AccountService {
    private final IAccountSelector selector;
    public AccountService(IAccountSelector selector) {
        this.selector = selector;
    }

    public Account getAccount(Id accountId) {
        return selector.getById(accountId);
    }
}
```

3.4 Dependency Injection Pattern

Explanation: Demonstrates constructor-based injection for services.

What this does: Allows the use of mocks in tests and production implementations in real code.

```
AccountService svc = new AccountService(new AccountSelector()); // Production
AccountService testSvc = new AccountService(new MockAccountSelector()); //
Test
```

---

4. Test Architecture

Explanation: Provides the structure for organizing tests into unit, service, integration, mocks, and factories.

What this does: Ensures that tests are organized by type and purpose, making them easier to maintain and run quickly.

```
unit/      // Fast, mock-only tests
service/   // Minimal DML and SOQL
integration/ // Full stack tests with async/callouts
mocks/     // Mock implementations
factories/ // Test data creation
```

---

5. Test Data Factory

Explanation: Contains methods to create minimal, controlled test data.

What this does: Reduces unnecessary DML and SOQL in tests, producing fast, predictable tests.

```
@isTest
public class TestDataFactory {
    public static Account account(Boolean doInsert) {
        Account a = new Account(Name = 'Test');
        if (doInsert) insert a;
        return a;
    }
}
```

6. Mocks

Explanation: Provides example mocks for selectors and other dependencies.

What this does: Allows tests to run without hitting the database or external systems.

6.1 Selector Mock Example

Explanation: A mock implementation of IAccountSelector.

What this does: Returns controlled, deterministic data in tests.

```
@isTest
public class MockAccountSelector implements IAccountSelector {
    public Account getById(Id accountId) {
        return new Account(Id = accountId, Name = 'Mock');
    }
}
```

7. Unit Test Template (Fastest Tests)

Explanation: Example of a unit test using mocks only.

What this does: Demonstrates a test that runs in milliseconds with no database dependency.

```
@isTest
private class AccountServiceTest {
    @isTest
    static void returnsMockedAccount() {
        AccountService svc = new AccountService(new MockAccountSelector());
        Account acc = svc.getAccount(Id.valueOf('001000000000001'));
        System.assertEquals('Mock', acc.Name);
    }
}
```

8. Service Test Template (Light DML)

Explanation: Example of testing service logic with minimal DML.

What this does: Tests business logic with real database interaction, but only as necessary, keeping tests fast.

```
@isTest
private class AccountTriggerHandlerTest {
    @isTest
    static void updatesFieldOnInsert() {
        Account a = TestDataFactory.account(true);
        a.Description = 'Updated';
        update a;

        Account saved = [SELECT Description FROM Account WHERE Id = :a.Id];
        System.assertEquals('Updated', saved.Description);
    }
}
```

---

9. Integration Test Template (Use Sparingly)

Explanation: Template for full stack tests, including async processes and callouts.

What this does: Ensures that end-to-end scenarios are tested without bloating the test suite with slow tests.

```
@isTest
private class AccountIntegrationTest {
    @isTest
    static void endToEndScenario() {
        Test.startTest();
        // async / callouts / batch
        Test.stopTest();

        System.assert(true);
    }
}
```

---

10. Base Mocking Framework (Required)

Explanation: Provides the core Service Registry and mock registration mechanism.

What this does: Centralizes dependency management, allowing production and test code to swap implementations easily.

10.1 Service Registry

Explanation: Manages mappings between interfaces and concrete implementations.

What this does: Ensures all dependencies can be resolved dynamically in production and tests.

```
public class ServiceRegistry {
    private static Map<Type, Object> services = new Map<Type, Object>();

    public static void register(Type iface, Object impl) {
        services.put(iface, impl);
    }

    public static Object resolve(Type iface) {
        if (services.containsKey(iface)) {
            return services.get(iface);
        }
        throw new ServiceException('No service registered for ' + iface);
    }

    public class ServiceException extends Exception {}
}
```

10.2 Production Registration

Explanation: Registers default production implementations.

What this does: Ensures that services in production resolve the correct concrete implementations.

```
public class ProductionServices {
    public static void registerAll() {
        ServiceRegistry.register(IAccountSelector.class, new
AccountSelector());
        ServiceRegistry.register(IClock.class, new SystemClock());
    }
}
```

10.3 Mock Registration

Explanation: Registers mocks for use in tests.

What this does: Allows tests to inject mock objects instead of hitting real services.

```
@isTest
public class TestServiceRegistry {
    public static void registerMock(Type iface, Object mock) {
        ServiceRegistry.register(iface, mock);
    }
}
```

10.4 Platform Service Abstractions

Explanation: Wraps platform services like System.now in interfaces.

What this does: Enables deterministic, mockable access to platform functionality in tests.

```
public interface IClock { Datetime now(); }
public class SystemClock implements IClock { public Datetime now() { return
System.now(); } }
@isTest public class FixedClock implements IClock { public Datetime now()
{ return Datetime.newInstance(2024,1,1); } }
```

---

11. Trigger + Handler Example Using Service Registry

Explanation: Shows how triggers can use the Service Registry to resolve dependencies.

What this does: Ensures triggers are testable and do not contain hard-coded service calls.

11.1 Trigger Handler

Explanation: Business logic for after-insert trigger.

What this does: Uses the registry to get the selector and update account fields.

```
public class AccountTriggerHandler {
    private final IAccountSelector selector;

    public AccountTriggerHandler() {
        this.selector = (IAccountSelector)
ServiceRegistry.resolve(IAccountSelector.class);
    }

    public void handleAfterInsert(List<Account> newAccounts) {
        for (Account acc : newAccounts) {
            Account fullAcc = selector.getById(acc.Id);
            acc.Description = 'Processed: ' + fullAcc.Name;
        }
    }
}
```

11.2 Trigger

Explanation: Wire-up for the trigger handler.

What this does: Executes handler logic on Account insert events.

```
trigger AccountTrigger on Account (after insert) {
    new AccountTriggerHandler().handleAfterInsert(Trigger.new);
}
```

11.3 Test

Explanation: Demonstrates a test using a mock selector.

What this does: Validates trigger logic without relying on real selectors.

```
@isTest
private class AccountTriggerHandlerIntegrationTest {
    @isTest
    static void afterInsertUsesMockSelector() {
        TestServiceRegistry.registerMock(IAccountSelector.class, new
MockAccountSelector());

        Account acc = TestDataFactory.account(false);
        insert acc;

        Account saved = [SELECT Description FROM Account WHERE Id = :acc.Id];
        System.assertEquals('Processed: Mock', saved.Description);
    }
}
```

---

12. Async Mocking Pattern (Queueable / Batch)

Explanation: Shows how to mock asynchronous jobs in tests.

What this does: Allows testing of Queueable or Batchable jobs without executing real business logic or long-running operations.

```
public interface IQueueableJob { void execute(); }

public class AccountQueueable implements Queueable, IQueueableJob {
    public void execute(QueueableContext ctx) { /* Business logic */ }
}

@isTest
public class MockQueueable implements IQueueableJob {
    public Boolean executed = false;
    public void execute() { executed = true; }
}

@isTest
static void testQueueableExecution() {
    MockQueueable mockJob = new MockQueueable();
    System.enqueueJob(mockJob);
    Test.startTest();
    Test.stopTest();
    System.assertEquals(true, mockJob.executed);
}
```

13. HTTP Callout Mock Base Class

Explanation: Provides a base class for HTTP callout mocks.

What this does: Enables testing callouts in Apex without hitting real endpoints, ensuring deterministic responses.

```
@isTest
global class BaseHttpCalloutMock implements HttpCalloutMock {
    global HTTPResponse respond(HTTPRequest req) {
        HttpResponse res = new HttpResponse();
        res.setHeader('Content-Type', 'application/json');
        res.setBody('{}');
        res.setStatusCode(200);
        return res;
    }
}

@isTest
static void testCallout() {
    Test.setMock(HttpCalloutMock.class, new BaseHttpCalloutMock());

    Http http = new Http();
    HttpRequest req = new HttpRequest();
    req.setEndpoint('https://example.com');
    req.setMethod('GET');

    HttpResponse res = http.send(req);
    System.assertEquals(200, res.getStatusCode());
    System.assertEquals('{}', res.getBody());
}
```

---

14. @testSetup Policy

Explanation: Guidelines for when and how to use @testSetup.

What this does: Encourages minimal use of shared setup to avoid slow or brittle tests.

```
Default: Do not use unless:
- Data is immutable
- Used across many test classes
- Required by platform metadata (RecordTypes)
```

---

15. Performance Guardrails

Explanation: Recommended limits for DML, SOQL, and CPU usage in tests.

What this does: Keeps test suite fast and predictable.

```
DML <= 5 per test
SOQL <= 5 per test
CPU < 50ms per test
```

---

16. CI / Code Review Rules

Explanation: Provides automated and manual rules to enforce good practices, especially registry usage.

What this does: Ensures developers do not bypass the Service Registry and maintain testable architecture.

```
- All services must resolve dependencies via ServiceRegistry
- No direct 'new' calls for selectors, callouts, or platform services
- Unit tests must use TestServiceRegistry.registerMock for mocks
- PMD rules: new AccountSelector() forbidden outside ProductionServices
- CI scan: detect HttpRequest calls without Test.setMock
```

---

17. Final Reminder

Explanation: Reinforces the importance of fast, maintainable tests.

What this does: Encourages developers to prioritize speed and quality in the test suite, shaping architecture decisions.