TL;DR: Cursor For Testing and QA

Step	Description	Example Prompt / Code
1	Set Up Your Project Structure	Organize `/tests` and `/mocks` folders for clarity.
2	Generate Your First Test	`Generate unit tests for the UserService module covering standard operations, edge cases, and invalid inputs. Add assertions and mock dependencies as needed.`
3	Run and Refine Tests	Cursor outputs ready-to-run test files; expand for additional scenarios.
4	Verify Coverage and Consistency	`Prompt Cursor to review your modules for untested functions or missing branches.`

Why Use Cursor For Testing and QA

Cursor accelerates testing by generating tests, catching errors early, automating workflows, and maintaining consistent QA standards.

Key Benefits

- Faster test creation: unit, integration, regression tests automatically.
- Early error detection: catch edge cases and logic flaws.
- Workflow automation: reduce manual work.
- Consistent standards: uniform test structures.
- Improved coverage: proactively identify gaps.

Where Cursor Fits in QA Lifecycle

- Unit Testing
- Integration Testing
- Regression Testing
- Debugging

Example Prompt:

`Generate unit tests for the PaymentProcessor module covering standard scenarios, edge cases, invalid inputs, and mock external services.`

Setting Up Cursor For Your QA Workflow

Organize Project and Test Folders

Recommended folder structure:

Folder	Purpose
/tests	Store all unit and integration tests
/mocks	Reusable mocks/stubs for dependencies
/data	Sample input/output files for edge cases
/scripts	Automation scripts for CI/CD

Configure Testing Frameworks

- **pytest:** unit testing with fixtures, parametrization, assertions
- **unittest:** Python's built-in testing framework
- **pytest-mock:** mock external dependencies

Verify Setup Prompt:

`Check the project setup for UserModule. Suggest improvements in folder structure, test coverage, and framework configuration.`

Automating Unit Tests With Cursor

Generate Standard Unit Tests

Prompt:

`Generate unit tests for the AuthService module. Test login, registration, and token validation.`

Example Output:

```
def test_login_success():
    auth_service = AuthService()
    token = auth_service.login("user@example.com", "password123")
    assert token is not None

def test_registration_creates_user():
    auth_service = AuthService()
```

```
    user = auth_service.register("new@example.com", "pass123")
    assert user.email == "new@example.com"
```

Handle Edge Cases

Prompt:

`Generate tests for edge cases in PaymentGateway. Cover null inputs, invalid card numbers, and API failures.`

Example Output:

```
def test_payment_invalid_card():
    result = PaymentGateway.charge("1234", 100)
    assert result["status"] == "failed"
```

Improve Test Coverage

Prompt:

`Review UserService and generate tests for untested functions or branches.`

Unit Test Checklist

- Assertions for all outcomes
- Mock external dependencies
- Handle edge cases/invalid inputs
- Follow consistent naming conventions
- Validate timing-sensitive logic

Automating Integration Tests

Create Full Module Integration Tests

Prompt:

`Generate integration tests for OrderService and InventoryService. Validate order creation, stock deduction, and error handling.`

Example Output:

```
def test_order_creation_and_inventory():
    order_service = OrderService()
    inventory_service = InventoryService()
    order = order_service.create_order(user_id=1, items=[{"id": 101, "qty": 2}])
```

```
    assert order["status"] == "success"
    stock = inventory_service.get_stock(item_id=101)
    assert stock["quantity"] == 98
```

Validate Data Flows

Prompt:

`Check data flow from PaymentService to OrderService. Generate tests to validate successful transactions and rollback scenarios.`

Example Output:

```
def test_payment_and_order_sync():
    payment_service = PaymentService()
    order_service = OrderService()
    payment_result = payment_service.charge(user_id=1, amount=50)
    assert payment_result["status"] == "success"
```

Mocked vs. Real Integration

Mocked Services Example:

```
def test_payment_with_mocked_gateway(mocker):
    mocker.patch("PaymentGateway.charge", return_value={"status": "success"})
    result = PaymentService().charge(user_id=1, amount=100)
    assert result["status"] == "success"
```

Sandbox / Non-Production Example:

```
def test_payment_sandbox_gateway():
    result = PaymentService().charge(user_id=1, amount=100, sandbox=True)
    assert result["status"] in ["success", "failed"]
```

Debugging and Issue Resolution With Cursor

Structured Debugging Workflow

1. **Collect failing tests and logs**
   Prompt: `Here are the failing test logs. Identify the likely causes and suggest potential fixes.`

2. **Analyze failures with structured prompts**
   Prompts:
3. `Analyze this stack trace and suggest what part of the code may be causing the failure.`
4. `Based on these error logs, propose possible fixes.`
5. **Detect flaky tests**
   Prompt: `Review these test logs and identify potential timing or asynchronous issues.`
6. **Create a triage runbook**
   Prompt: `Group these failing tests by root cause and suggest an order to apply fixes.`
7. **Apply fixes and rerun verification**
8. **Document resolutions**
   Prompt: `Create a summary of all recent test failures, root causes, and applied fixes.`

Enhancing Code Quality and Testability

Automated Code Review

Prompt:
`Review the OrderService module. Suggest changes to reduce side effects, simplify functions, and enhance testability.`

Generate Test Documentation

Prompt:
`Generate a test documentation file for UserModule. Describe folder structure, types of tests, and execution instructions.`

Refactoring Patterns

- Dependency injection
- Modular code
- Small, focused functions

Security and QA Checks

Prompt:
`Check PaymentProcessor for untested security scenarios, covering input validation, authorization, and potential data leaks.`

End-to-End Workflow For Testing and QA

Step	Prompt / Action
1. Audit Test Coverage	`Analyze the UserService module and highlight untested functions or branches.`
2. Generate Unit Tests	`Generate unit tests for PaymentService, including edge and invalid input scenarios.`
3. Generate Integration Tests	`Generate integration tests for OrderService and InventoryService.`
4. Validate CI/CD Configuration	`Review CI/CD setup for CheckoutModule. Suggest additions to run unit and integration tests.`
5. Build Regression Suite	`Generate a regression suite for UserService covering recent changes.`
6. Document QA Conventions	`Generate test style guide and folder naming conventions for the QA team.`

Cursor Prompts, Templates, and Examples

Unit Test Prompt Template

```
Generate unit tests for the AuthService module. Include:
- Standard scenarios (login, registration)
- Edge cases (invalid inputs, null values)
- Mock external dependencies
- Assertions for all outcomes
```

Integration Test Prompt Template

```
Generate integration tests for OrderService and InventoryService. Include:
- API communication validation
- Error handling for failed requests
- Data flow between services
```

Mocking Template

```
Create reusable mocks for PaymentGateway:
- Successful transactions
```

```
- Network errors
- Invalid card inputs
```

Debugging Template

```
Analyze failing tests for CheckoutService. Identify root causes and suggest
fixes for:
- Assertion failures
- Async timing issues
- Race conditions
```

Test Review Template

```
Review all existing tests for UserService. Highlight:
- Missing test cases
- Edge scenarios not covered
- Inconsistent assertions or mocks
```

QA Metrics and Performance Impact

Metric	Before Cursor	After Cursor	Improvement
Time to generate unit tests	8 hours	30 minutes	94% faster
Test coverage	65%	90%	+25%
Bugs detected early	20 per cycle	35 per cycle	+75%
Manual test writing effort	High	Low	Significant
Regression suite setup	5 hours	45 minutes	85% faster