DevOps

Testing in DevOps

Chandara Chea

23/10/2025
Week 2

What is Software Testing

Software testing is the process of evaluating and verifying that a software product or application functions correctly, securely and efficiently according to its specific requirements.

Primary Benefits: Delivering high-quality software by identifying bugs and improving performance.

Modern Role:

- Testing is deeply embedded in modern development practices (driven by Agile, DevOps, and continuous integration/continuous delivery (CI/CD) pipelines).
- It is no longer a final step; it begins at the design planning phase and continues after deployment

Modern Software Testing

- Shift-Left Testing: Testing begins earlier in the development cycle to help teams uncover issues sooner.
- Shift-Right Testing: Focuses on monitoring and validation in production, allowing teams to adapt to real-world usage quickly.
- These approaches support faster releases and reduce risk in rapidly changing IT environments.
- Evolution Drivers: Modern strategies evolve with advances in automation, artificial intelligence (AI), and cloud-native architectures like microservices.
- AI-Enabled Testing Market:
  - Valued at USD 856.7 million in 2024.
  - Projected to grow to USD 3,824.0 million by 2032, exhibiting a CAGR of 20.9%.

History of Software Testing

- The Beginning:
  - Began alongside software engineering, emerging just after World War II.
  - Computer scientist Tom Kilburn is credited with writing the first piece of software (debuted June 21, 1948).
- Early Years (Decades following WWII): Debugging was the primary testing method.
- 1980s Shift (Quality Assurance):
  - Development teams began looking beyond isolating and fixing bugs.
  - Started testing applications in real-world settings to ensure broader functionality and reliability.
  - Testing became an integral part of the Software Development Lifecycle (SDLC)

History of Software Testing

- 1990s and 2000s:
  - Rise of automated testing and practices like test-driven development (TDD).
  - Modular programming (OOP) made it easier to write focused unit tests.
  - Expansion of mobile/web applications demanded new strategies (performance, usability, and security testing).
- The Last Decade (Agile/DevOps):
  - Testing has become continuous, automated, and integrated into every phase of development and deployment.
  - Organizations leverage proprietary and open source tools (e.g., Katalon Studio, Selenium) for quality assurance

Why Testing is Important (The Cost of Defects)

- Avoid Costly Failures
  - Software defects can cause major business and financial losses.
  - Example: Delta Air Lines (2024) — a faulty update caused global system crashes
    → $500M loss.
- Protect Brand Reputation
- Detect Issues Early - early testing helps find and fix:
  - Architectural flaws
  - Poor design
  - Invalid functionality
  - Security vulnerabilities
  - Scalability problems
- Improve Quality & Reliability
  - Well-tested systems perform better and meet user expectations.
  - Leads to fewer errors, higher satisfaction, and market advantage.

Testing in DevOps

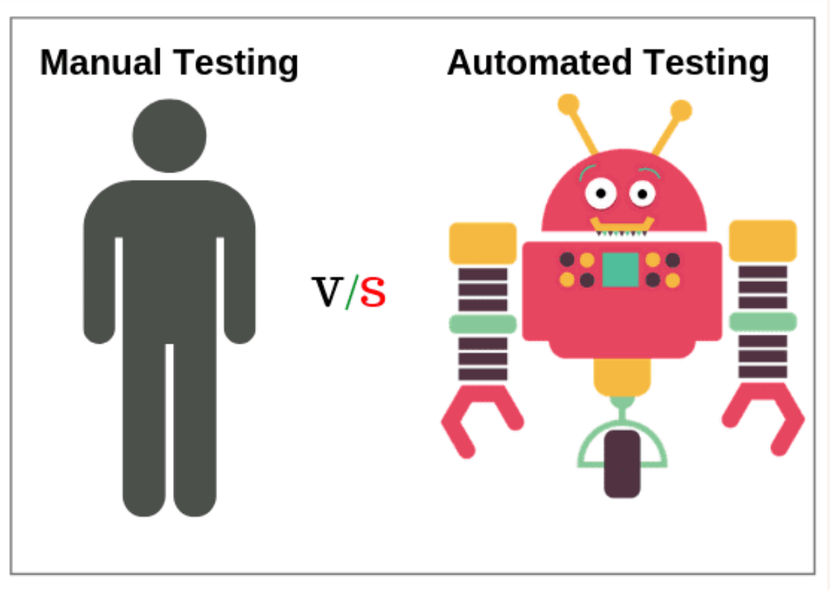
Testing in DevOps is a fundamental aspect of ensuring continuous delivery of high-quality software. Unlike traditional testing, which often occurs at the end of the development cycle, DevOps integrates testing throughout the entire software development lifecycle (SDLC), from initial coding to deployment and operations.

This continuous approach, known as **Continuous Testing**, aims to detect and address issues early, minimizing risks and accelerating delivery.

Software testing may either be a manual or an automated process.

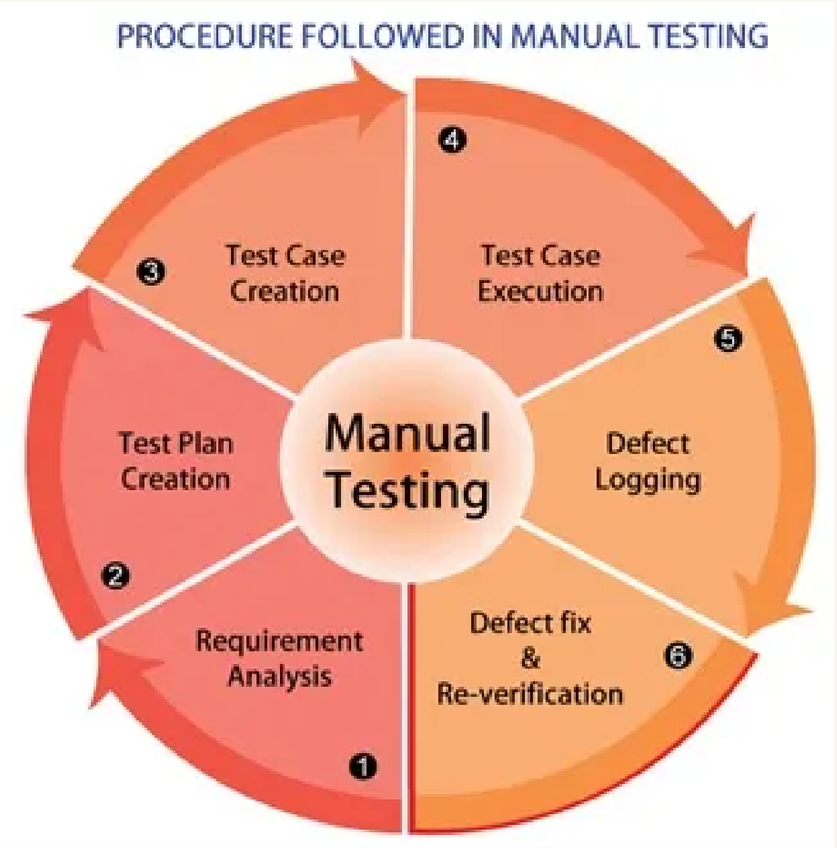
Manual Testing vs Automated Testing

- Software testing falls primarily into 2 broad categories:
  - Manual testing
  - Automated testing



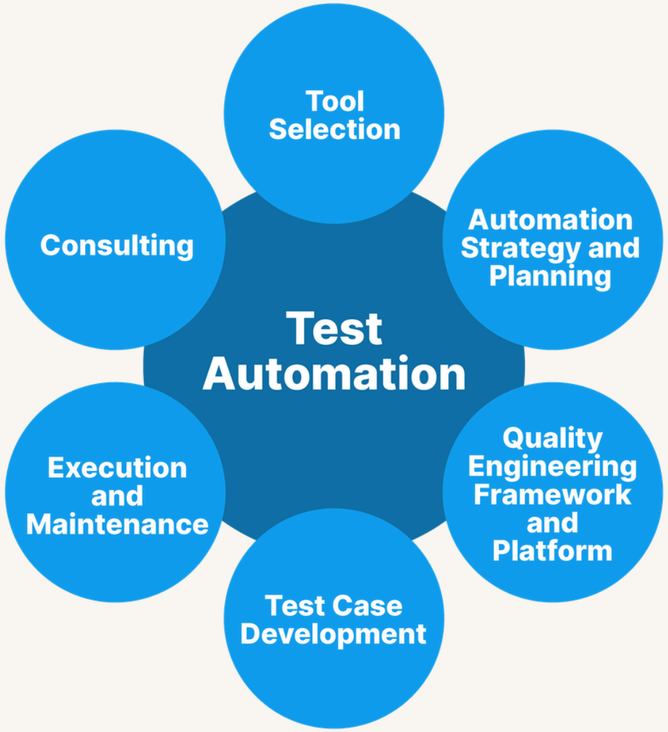
Manual Testing

- Led by a team or individual who will manually operate a software product and ensure it behaves as expected.
- Done in person, by clicking through the application or interacting with the software and APIs with the appropriate tooling.
- Very expensive since it requires someone to setup an environment and execute the tests themselves
- Prone to human error as the tester might make typos or omit steps in the test script



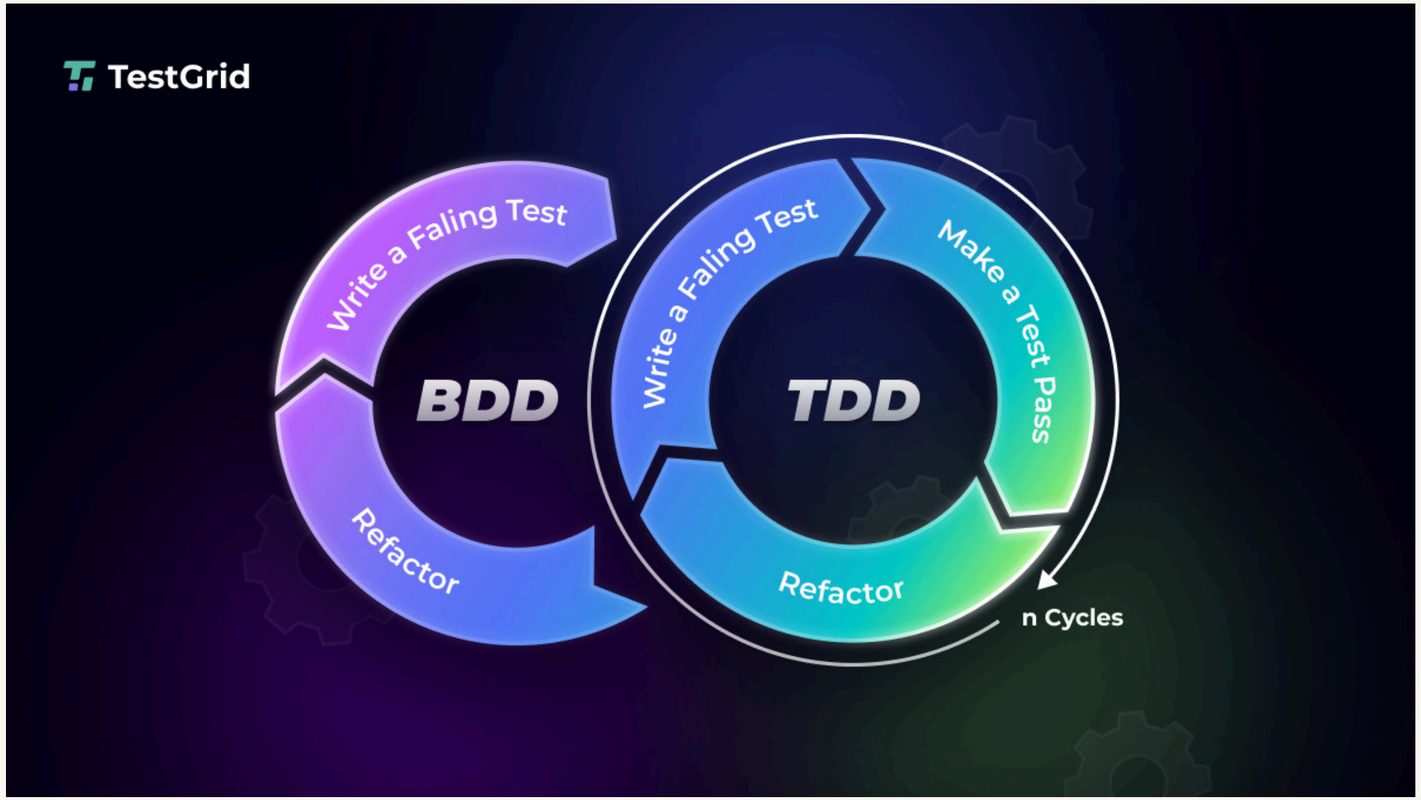
Automated Testing

- Is the process of automating manual testing throughout the software development life-cycle.
- Uses scripts and tools to execute tests automatically.
- Beneficial for repetitive tasks and larger systems that require running the same tests multiple times.
- Ensure software testing’s consistency, speeds up testing, reduces human error, and improves efficiency over time

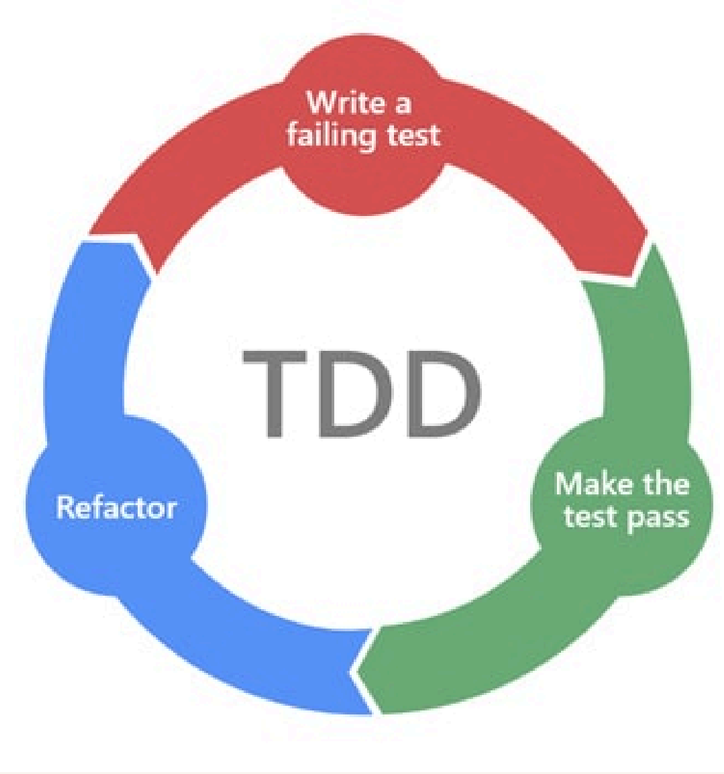


Software Testing Approach

There are several different ways to approach software testing, and the two most commonly applied are **Test-Driven Development (TDD)** and **Behavior-Driven Development (BDD).**



Test Driven Development (TDD)



Test-driven development is an iterative approach to development that involves writing automated tests prior to writing the code.

This gives developers the context they need to improve their code and is designed to offer continuous feedback so that bugs can be identified and fixed as soon as possible.

Behavior Driven Development (BDD)



Behavior-driven development is a testing approach that defines various methods to develop an application feature based on its behavior.

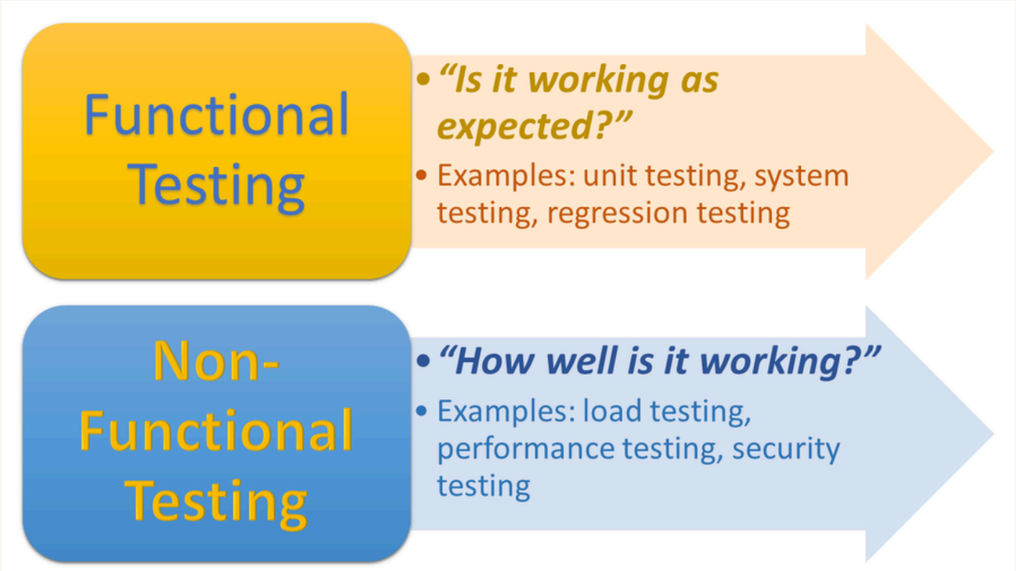
The emphasis is on defining the behavior in simple everyday language so that everyone on the team can understand the feature (including non-technical team members).

- Given the user has entered a valid username and password,
- When the user clicks the “Login” button,
- Then, the system should display a “Login Successful” message.

Types of Software Testing

There are many different types of software testing that fall under the levels discussed earlier, and they can typically be divided into two main categories:

- Functional testing verifies whether a software application behaves according to specified requirements.
- Non-functional testing assesses how the software performs under various conditions, such as load, stress or across different environments.



Functional Testing

Functional testing assesses overall compatibility between different software modules, which has a direct bearing on other related aspects, such as expected user interactions and the viability of the user interface.

The test environment provided by functional testing is considered an example of **black-box testing**, in which testers don't gain insight to the inner machinations of the system (as is found in **white-box testing**)—and are instead given a "Pass/Fail" report on whether the system and its pipelines function as desired.

Functional Testing

Functional testing usually follows a testing process requiring **six steps**:

1. Identify the various functions that the software needs to execute.
2. Create input data according to functional requirements.
3. Establish expected outputs per functional specifications.
4. Conduct test cases.
5. Evaluate how actual outputs compare to expected outputs.
6. Determine whether the software application works satisfactorily per expectations.

Type of Functional Testing

- **White-box testing**: involves testing based on knowledge of the internal structure, logic and functions of the software being tested.
- **Black-box testing:** a tester does not have any information about the internal workings of the software system.
- **Ad hoc testing:** testers try to break or find bugs in an application without following predefined tests or documentation.
- **API testing**: verifies that the interfaces between software components function correctly and reliably. API testing is an essential part of API management, the software and processes that support an API's lifecycle.
- **Exploratory testing**: helps software testers uncover hard-to-predict scenarios and situations that can lead to software errors.

Type of Functional Testing

- **Regression testing:** checks whether new features break or degrade existing functionality. It ensures that recent changes haven’t introduced new defects.
- **Sanity testing**: evaluates whether specific functionalities work as expected. Testers can use it to verify menus, functions and commands at the surface level when there is no time for a full regression test.
- **Smoke testing**: is a preliminary software testing process that checks whether the basic functions of an application work correctly. It helps ensure that the build is stable enough for further testing.
- **User acceptance testing (UAT):** is a specific type of acceptance testing performed by the end users to confirm the system meets their needs and works in real-world scenarios.

Non-Functional Testing

Non-functional testing evaluates application behaviors that aren’t usually considered vital to ensure functionality. Nonetheless, delivering a well-organized and pleasant user experience is now considered an essential part of software development.

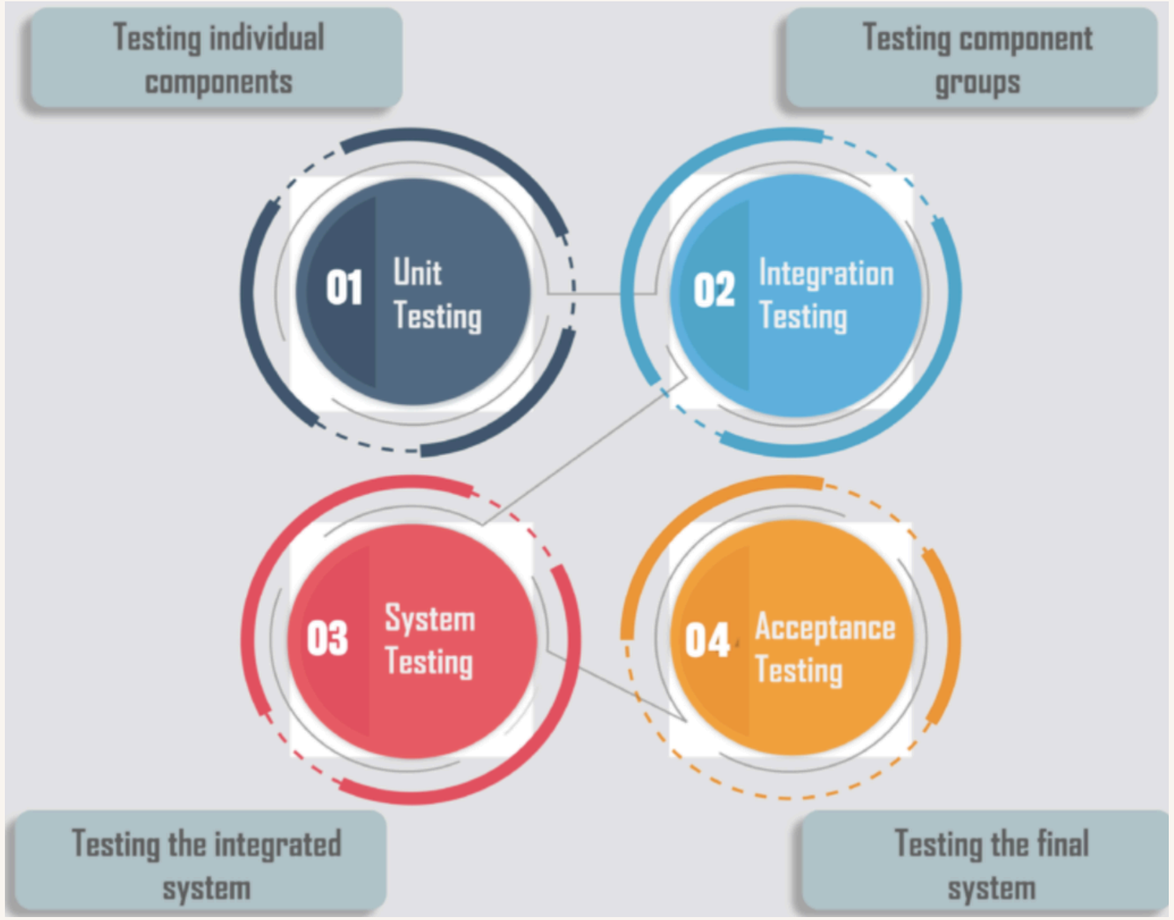
Non-functional testing can reveal potential system problems, especially when the software is maximized to demonstrate greater scalability.

Type of Non-Functional Testing

- **Recovery testing**: verifies how the software responds and recovers from failures, ensuring that data and processes are restored correctly.
- **Performance testing**: refers to how the software runs under different workloads.
- **Load testing**: a type of performance testing— evaluates performance under real-life load balancing conditions.
- **Stress testing**: examines the amount of strain the system can withstand before it fails.
- **Security testing**: validates whether software is open to hackers or other malicious types of vulnerabilities.
- **Usability testing**: validates how well a customer can use a system's user interface to complete a task efficiently and intuitively.
- **Compatibility testing:** checks whether a software application functions as expected across various devices, operating systems, browsers and network environments.

Levels of software testing

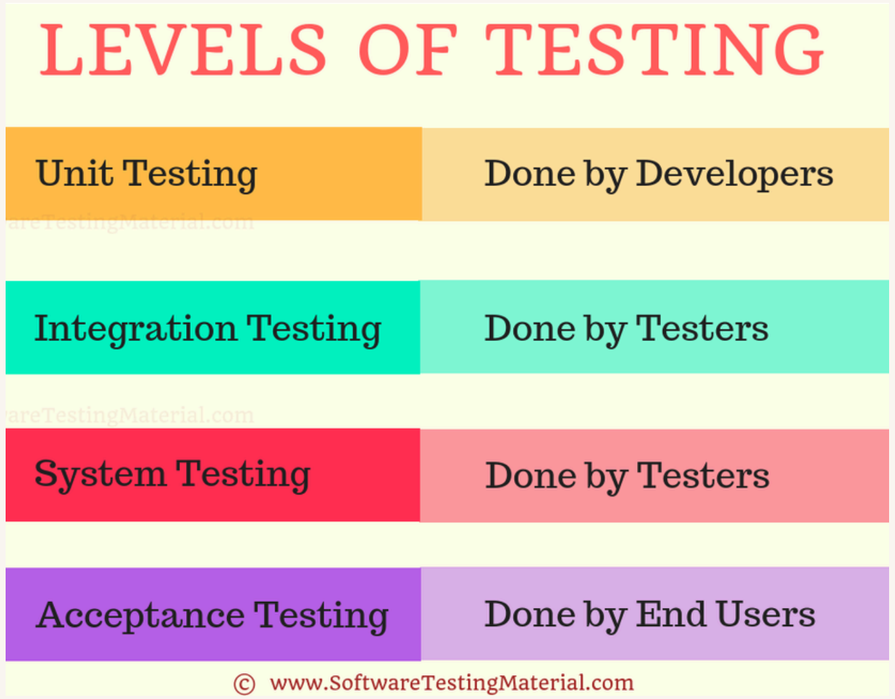
In general, software testing occurs at four different levels—or stages—within the software development lifecycle, each focusing on specific parts of the application.



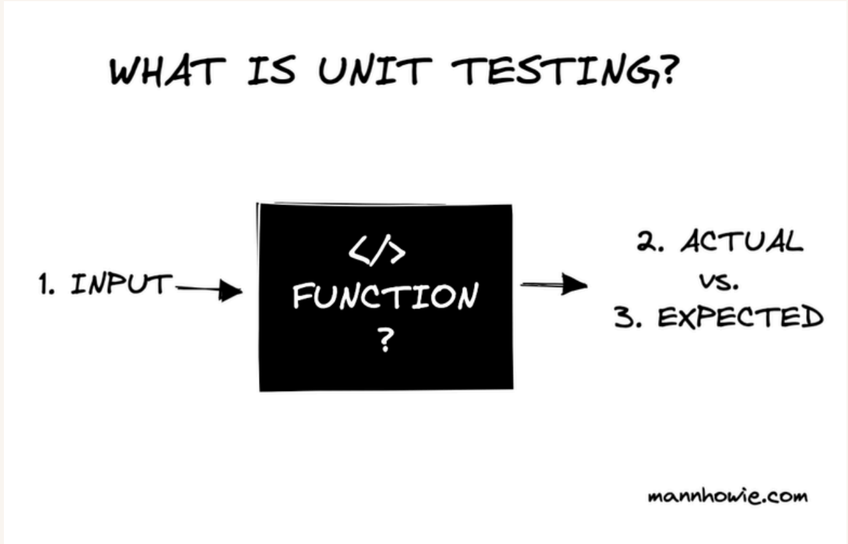
Unit Testing

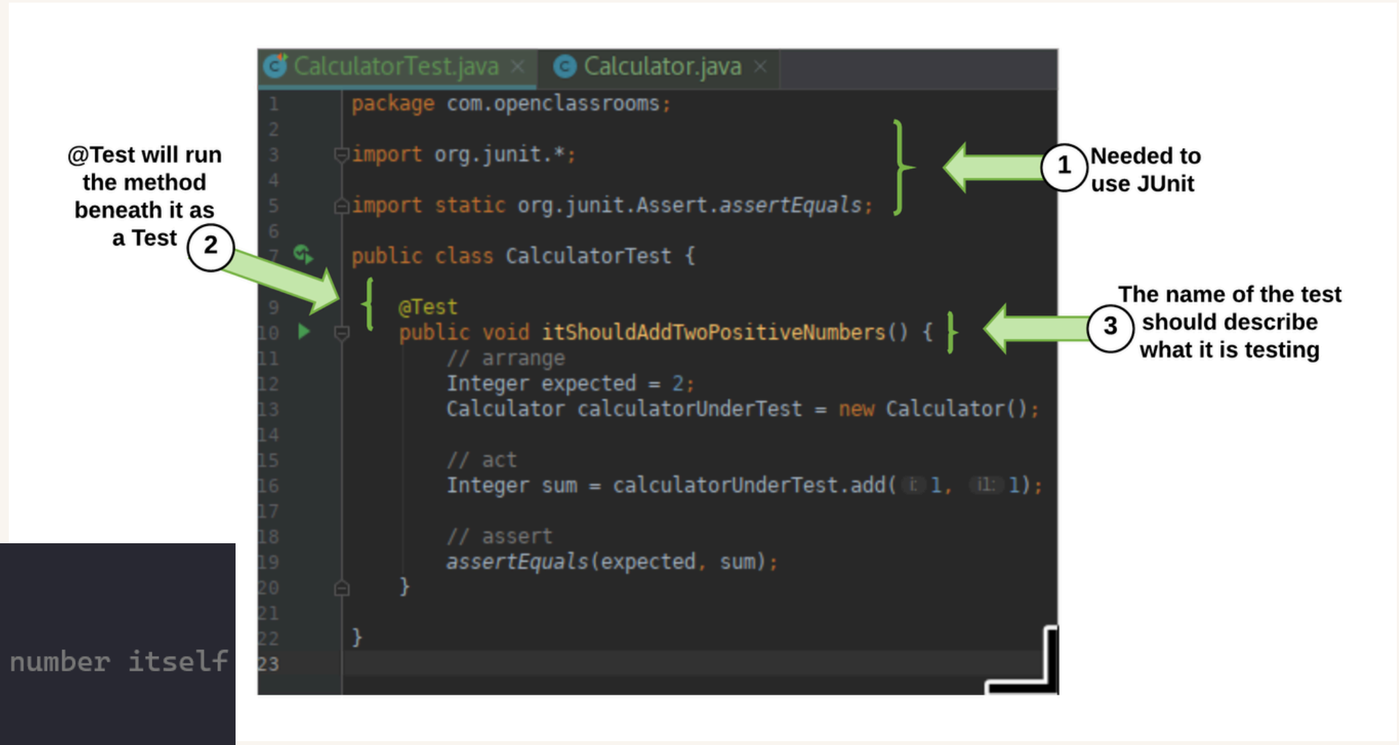
Unit tests are very low level and close to the source of an application. They consist in testing individual methods and functions of the classes, components, or modules used by your software.

Unit testing involves isolating units so that functionality can be confirmed before units are integrated with other parts of the application.



Unit Tests






```
> jest

 PASS  ./calculator.test.js
  √ string with a single number should result in the number itself

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.561 s
Ran all test suites.
```

Unit Testing

Five Sequential Steps of Unit Testing

1. **Identify the unit**: chooses the unit test code (function, class, or method) to be analyzed.
2. **Select an approach**: choose between manual testing or automated unit testing through a framework.
3. **Establish the test environment**:
    a. ensure the environment meets all requirements, including test data, dependencies, and mock objects.
    b. use an Integrated Development Environment (IDE), which contains all the tools necessary for writing, building, testing, and debugging code.
4. **Create and use test cases**: selects a unit testing framework and writes the test cases.
5. **Debug and resolve issues**: debug the code to confirm its root cause, repair the issue, and then run unit tests again to ensure bugs have been remediated.

Unit Testing

Common Unit Testing Tools (Frameworks)

- **Jest**: JavaScript framework for testing JavaScript and React components.
- **JUnit**: Java™ framework. Known for better code organization, more thorough error detection, and improving software quality. Can also be used for integration and functional testing.
- **Mocha**: Open source framework for JavaScript code. .
- **NUnit**: Open source framework for the .NET platform (C#, VB.NET, F#).
- **Pytest**: Framework for writing and executing Python tests.







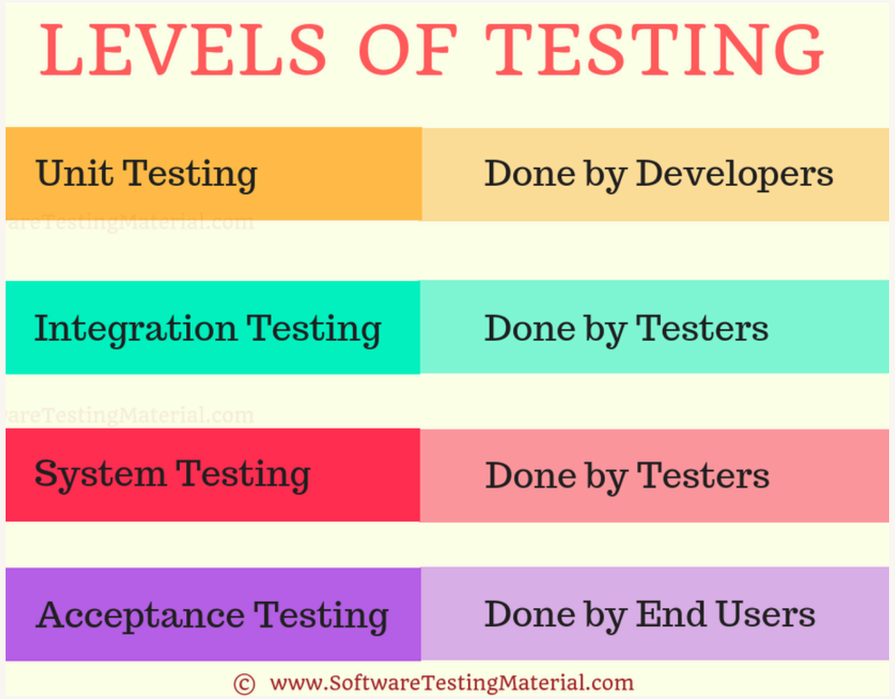




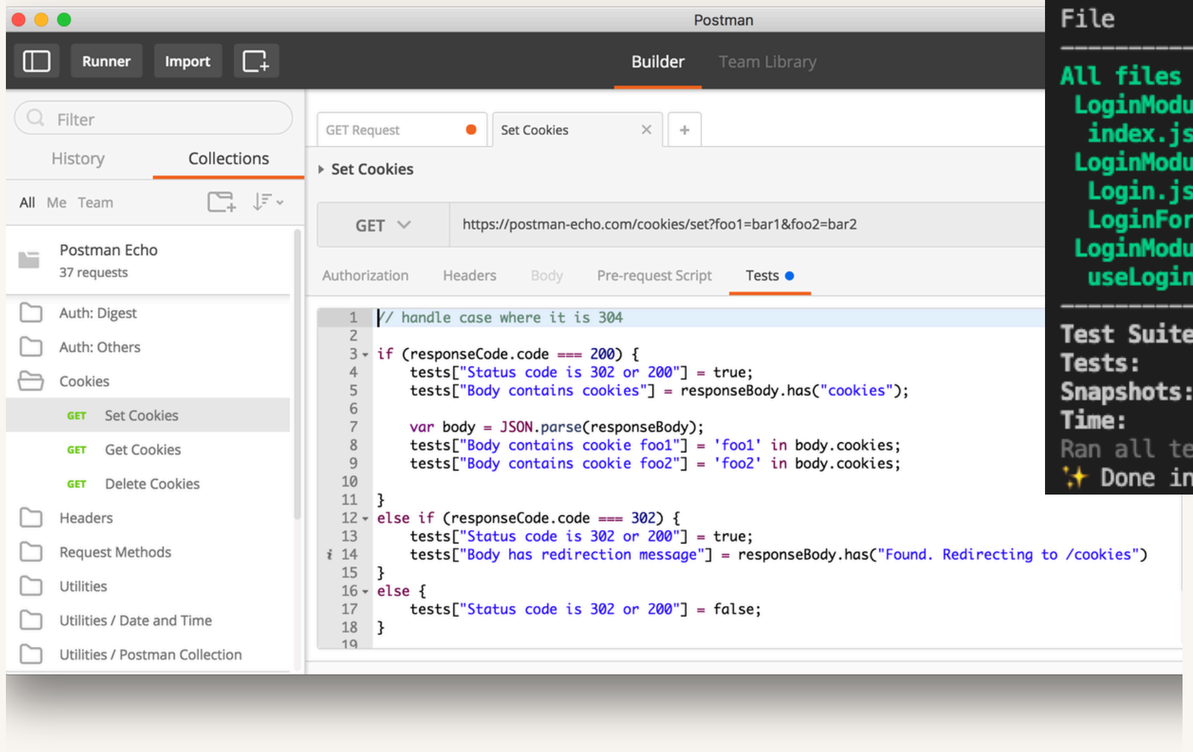
Integration Testing

Integration tests verify that different modules or services used by your application work well together. For example, it can be testing the interaction with the database or making sure that microservices work together as expected.

These types of tests are more expensive to run as they require multiple parts of the application to be up and running.



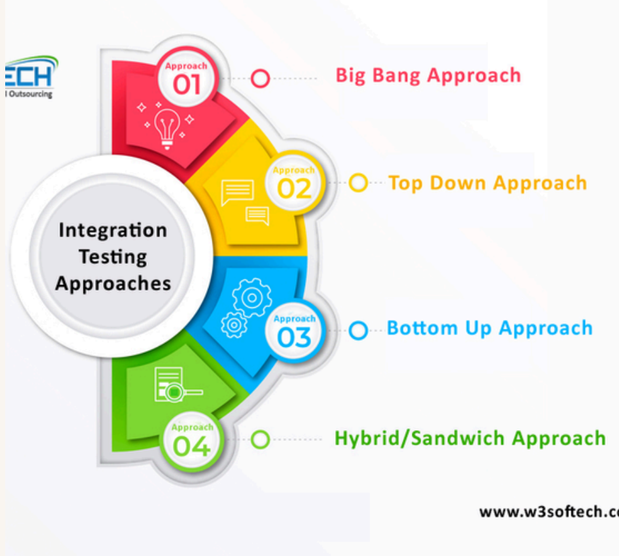
Integration Testing




```
~/fun-with-tests (master ✓) ▷ yarn test:coverage:integration
yarn run v1.17.0
$ react-scripts test --watchAll=false --env=jest-environment-jsdom-sixteen --coverage --te
stPathPattern="integration.test.js" --verbose
 PASS  src/LoginModule/index.integration.test.js
  LoginModule
    ✓ initial state (172ms)
    ✓ successful login (111ms)
    ✓ error login (94ms)

-------------------------|----------|----------|----------|----------|-------------------
File                     |  % Stmts | % Branch |  % Funcs |  % Lines | Uncovered Line #s
-------------------------|----------|----------|----------|----------|-------------------
All files                |      100 |      100 |      100 |      100 |
 LoginModule             |      100 |      100 |      100 |      100 |
  index.js               |      100 |      100 |      100 |      100 |
 LoginModule/components  |      100 |      100 |      100 |      100 |
  Login.js               |      100 |      100 |      100 |      100 |
  LoginForm.js           |      100 |      100 |      100 |      100 |
 LoginModule/hooks       |      100 |      100 |      100 |      100 |
  useLogin.js            |      100 |      100 |      100 |      100 |
-------------------------|----------|----------|----------|----------|-------------------
Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        2.483s
Ran all test suites matching /integration.test.js/i.
✨  Done in 3.16s.
```

Integration Testing Techiques



There are numerous types of integration testing techniques, but these are the most used to evaluate software systems.

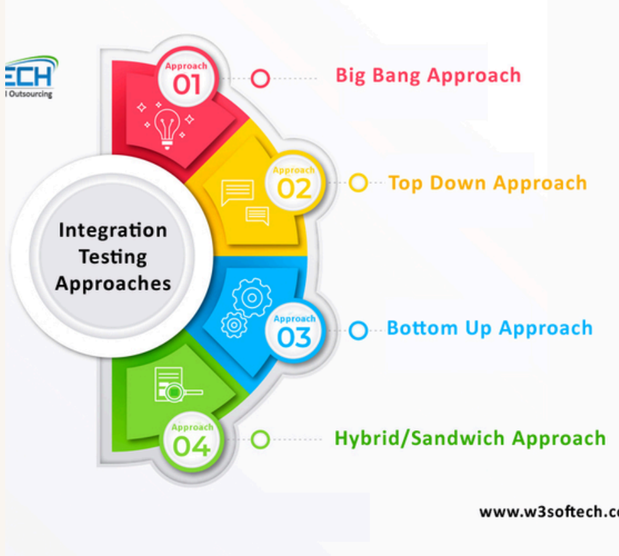
Top-down integration testing:

- Focuses on the main module and its workings first, then evaluates the submodules and subroutines.
- Advantage: Can be used early in the process.
- Testers use stubs (placeholders) as a substitute for lower-level modules that have not been fully identified.

Bottom-up integration testing:

- Switches the order: submodules and subroutines are evaluated first, then the main module.
- Uses temporary modules called drivers as substitutes for high-level components that haven’t yet been identified.

Integration Testing Techiques



Mixed Integration Testing (Sandwiched Integration):

- Combines the top-down and bottom-up methods.
- Advantage: Overcomes the enforced process sequence limitations of the other two approaches.
- Testing can begin with either the main module or with the submodules, as user needs dictate

Big Bang Integration Testing

- All individual units, components, and modules within a system are integrated and tested together at one time.
- The system is treated as if it were a single unit.
- Advantage: Can provide a quick answer if the system works with all its various parts.
- Limitation: If the system does not work as expected, big bang testing cannot reveal which individual parts are failing to work in unison

Integration Testing Methods

- **API integration testing**: Checks how different Application Programming Interfaces (APIs) within a system work together to share data. Goal: APIs function flawlessly as a single organized unit.
- **Automated integration testing**: Uses specialized tools and scripts to run test cases. It increases efficiency and agility, and is a key element of continuous integration (CI).
- **Functional testing (FIT)**: Validates that different modules or components can interact as needed. It is useful for locating defects early; performed after unit testing but before full system testing.
- **End-to-end testing (E2E):** Checks the functions of the entire system, from start to finish. It mimics real-world test scenarios. Typically appears later, after integration testing but before user acceptance testing.

Integration Testing Methods

- **Regression testing:** An after-the-fact test that checks whether new changes (like new features) made during integration have inadvertently introduced glitches to other parts of the system.
- **Black box testing:** Testers prefer a simpler and quicker answer about whether systems work harmoniously; they do not examine the specific codes used within different modules.
- **White box testing:** The direct contrast to black box; testers examine relevant code during the process in hopes of pinpointing problems and making corrective changes (debugging). This approach is almost certainly more time-consuming.

Integration Testing Tools

Citrus:

- An open source Java™ framework
- Manages API usage (transactions) and generates testing messages.

Katalon:

- Katalon Studio's automation testing software incorporates the open source framework **Selenium**
- Allows testers to write test scripts in various programming languages (e.g., JavaScript, NodeJS, Python).

Postman:

- Excels at API integration testing.
- Enables collaboration and accommodates automation.
- Allows users to make a request and receive a corresponding URL without saving the test.

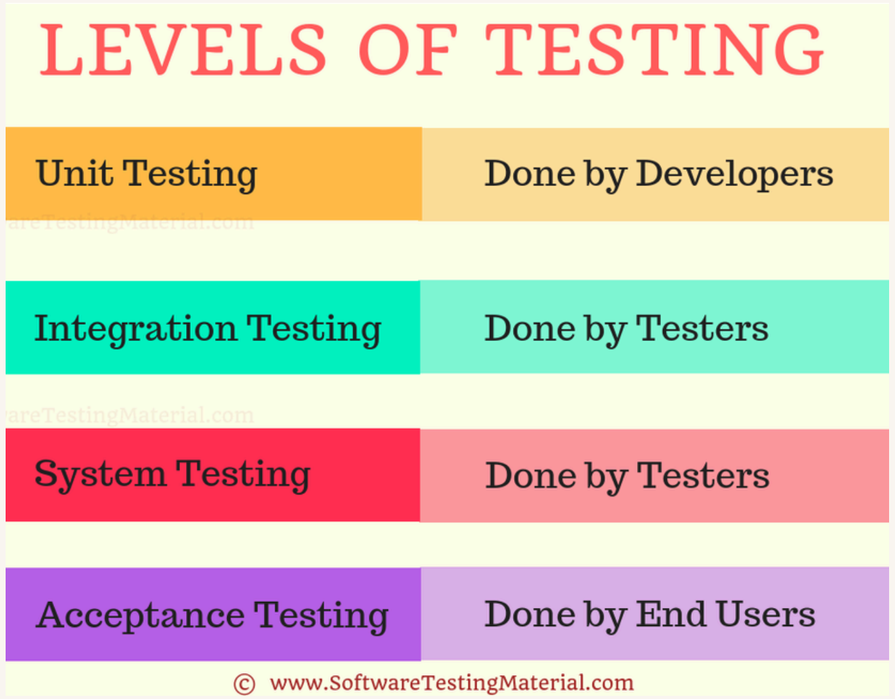
SoapUI:

- Provides an open source tool for testing web applications and executing integration testing.
- Features a graphical interface that supports the creation of test cases and working with test data.

System Testing

System Testing is a form of black box testing, meaning testers focus on overall functionality and performance rather than the internal code or components. Its goal is to ensure the software capably supports the business requirements of the users who will be relying upon it.

At this point, you're no longer checking how each part is built — you're checking if the whole system works as expected from a user's point of view.



System Testing

The process reflects the same production environment that will be used, utilizing test data mimicking real-world situations

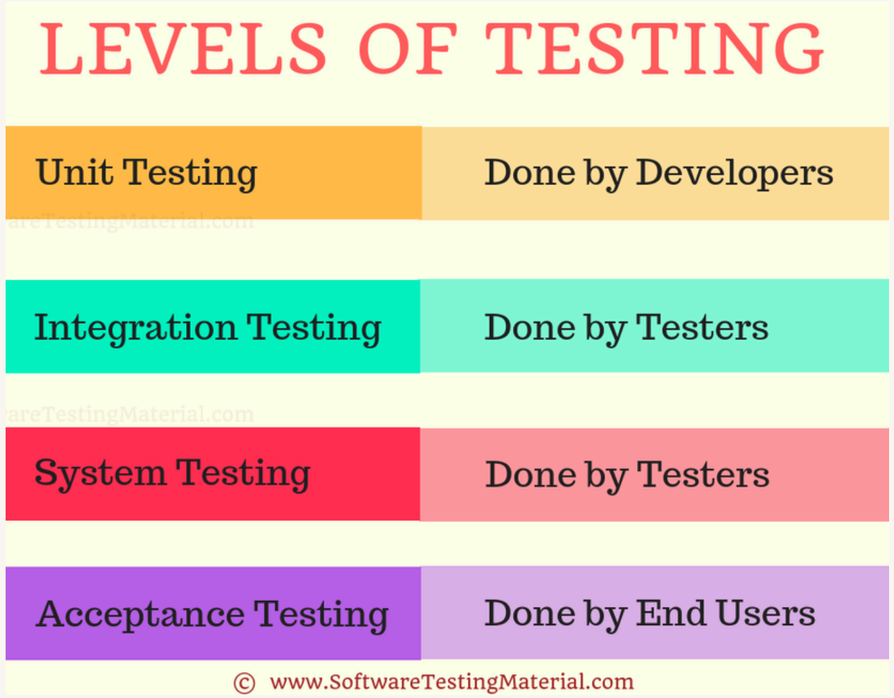
Each query is shaded in terms of the system’s functionality:

- Is each component working as expected?
- Are components performing in proper coordination with one another?
- Is the software poised to support a smooth user interface and deliver the desired user experience (UX)?

Acceptance Testing

Acceptance testing is the final check in software development to ensure the product meets business goals and user expectations before release.

It typically involves validating functionality, performance, and usability to confirm that the software is ready for deployment and will satisfy end-users.



Acceptance Testing

Key Features

- Performed after system testing.
- Focused on real-world usage scenarios.
- Usually involves end users, clients, or QA teams.
- Can include functional and non-functional checks like usability, performance, and reliability.

Examples

- A user performing all tasks in an e-commerce app to verify checkout, payment, and order confirmation work as expected.
- A client validating that a reporting system generates accurate business reports according to requirements.

Type of Acceptance Testing

User Acceptance Testing (UAT) - End-User perspective: Validates if the software meets end-user needs and defined specifications.
Business Acceptance Testing (BAT) - Business workflows and finances: Checks alignment with core business goals and financial purposes.
Contract Acceptance Testing (CAT) - Legal and contractual terms: Ensures all requirements stipulated in the contract (SLA) are fulfilled before payment.
Regulation Acceptance Testing (RAT) - Compliance: Verifies the product does not violate regulations and rules set by governing authorities (e.g., HIPAA compliance).
Operational Acceptance Testing (OAT) - System Readiness: Non-functional testing confirming stability, including recovery, compatibility, and reliability.
Alpha Testing - Internal environment: Is conducted by specialized developer/tester teams to catch early bugs.
Beta Testing - Real-world environment: Is evaluated by external end-users (beta testers) before official release to gather real-world feedback.

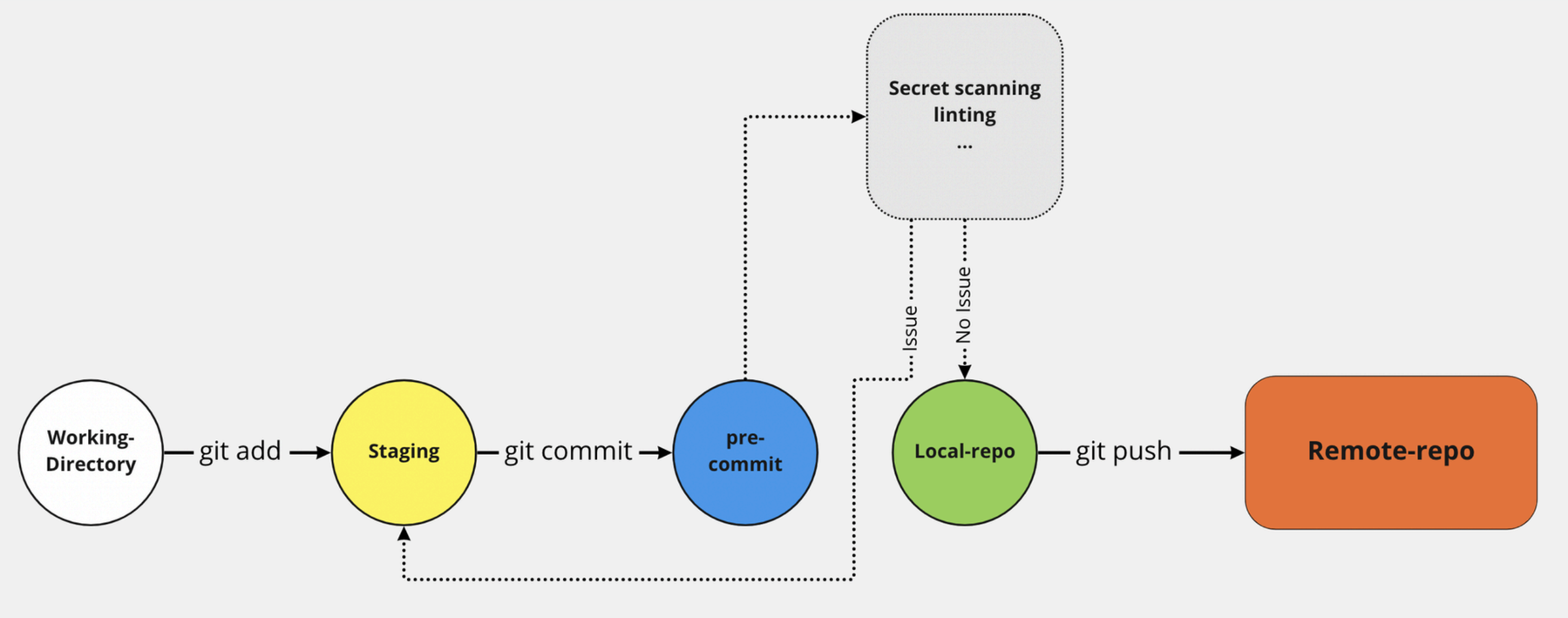
Linting

Linting is the automated checking of your source code for programmatic and stylistic errors. This is done by using a lint tool (otherwise known as linter). A lint tool is a basic static code analyzer.

What can Linting do?

- detect errors in a code and errors that can lead to a security vulnerabilities.
- detect formatting or styling issues and makes the code more readable for more secure code.
- suggest best practices.
- increases overall quality of the code.
- makes maintenance of code easier since everybody follows the same linting rules

Linting



- Before committing code (pre-commit hook)
- In CI/CD pipeline

Code Coverage

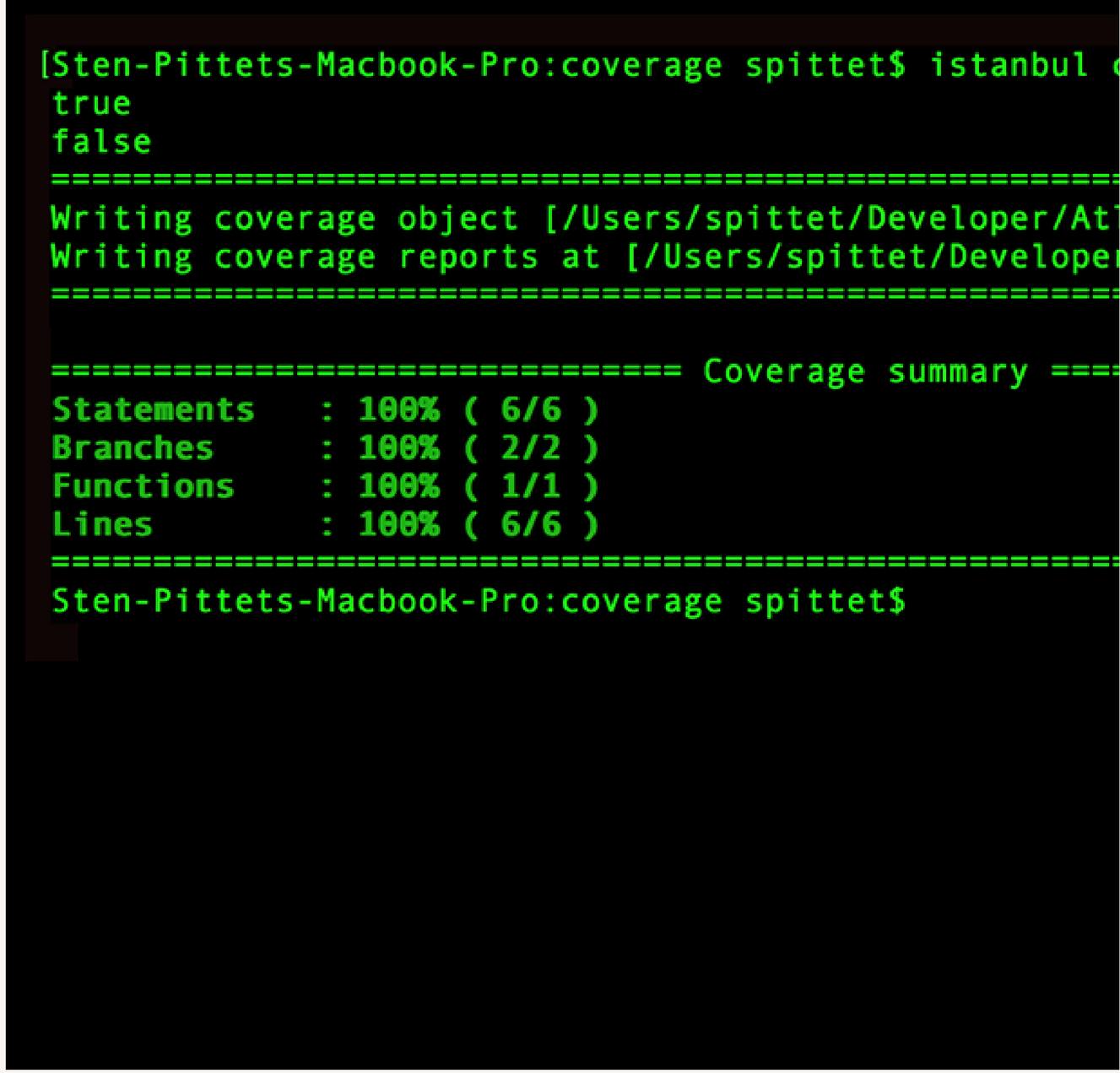
Code coverage is a metric that can help you understand how much of your source is tested. It's a very useful metric that can help you assess the quality of your test suite, and we will see here how you can get started with your projects.

Code coverage tools will use one or more criteria to determine how your code was exercised or not during the execution of your test suite.

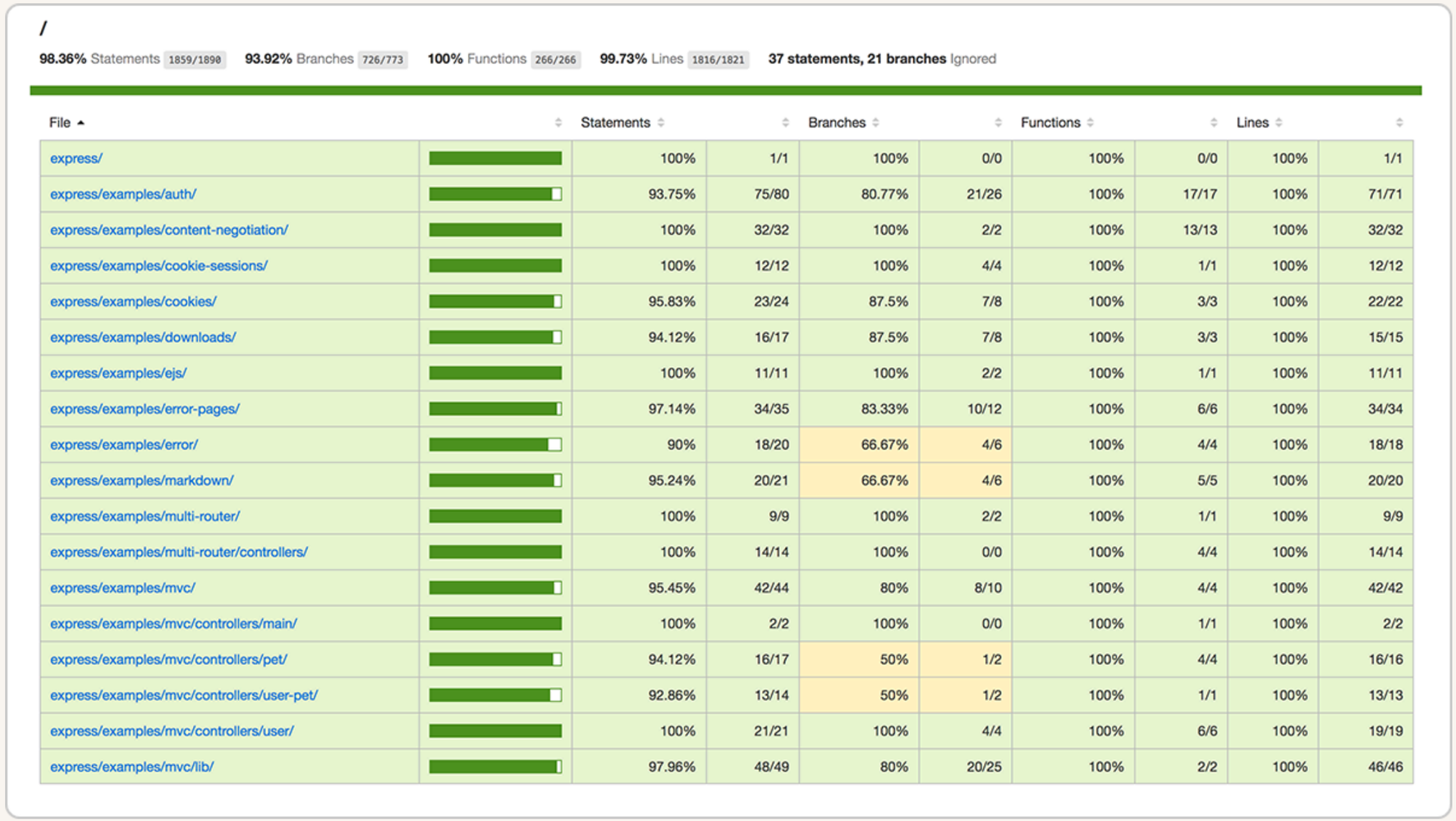
- Function coverage: Percentage of functions/methods called
- Statement coverage: how many of the statements in the program have been executed.
- Branches coverage: Percentage of possible branches (if/else) executed
- Condition coverage: Boolean expressions tested both true and false
- Line coverage: Percentage of code lines executed

Code Coverage

- **Function coverage:** % of functions/methods called
- **Statement coverage:** % of code statements executed by tests
- **Branches coverage**: % of possible branches (if/else) executed
- **Condition coverage:** % of boolean expressions tested both true and false
- **Line coverage:** % of code lines executed



Code Coverage



/

98.36% Statements `1859/1890` **93.92%** Branches `726/773` **100%** Functions `266/266` **99.73%** Lines `1816/1821` **37 statements, 21 branches** Ignored

File	Statements		Branches		Functions		Lines	
express/	100%	1/1	100%	0/0	100%	0/0	100%	1/1
express/examples/auth/	93.75%	75/80	80.77%	21/26	100%	17/17	100%	71/71
express/examples/content-negotiation/	100%	32/32	100%	2/2	100%	13/13	100%	32/32
express/examples/cookie-sessions/	100%	12/12	100%	4/4	100%	1/1	100%	12/12
express/examples/cookies/	95.83%	23/24	87.5%	7/8	100%	3/3	100%	22/22
express/examples/downloads/	94.12%	16/17	87.5%	7/8	100%	3/3	100%	15/15
express/examples/ejs/	100%	11/11	100%	2/2	100%	1/1	100%	11/11
express/examples/error-pages/	97.14%	34/35	83.33%	10/12	100%	6/6	100%	34/34
express/examples/error/	90%	18/20	66.67%	4/6	100%	4/4	100%	18/18
express/examples/markdown/	95.24%	20/21	66.67%	4/6	100%	5/5	100%	20/20
express/examples/multi-router/	100%	9/9	100%	2/2	100%	1/1	100%	9/9
express/examples/multi-router/controllers/	100%	14/14	100%	0/0	100%	4/4	100%	14/14
express/examples/mvc/	95.45%	42/44	80%	8/10	100%	4/4	100%	42/42
express/examples/mvc/controllers/main/	100%	2/2	100%	0/0	100%	1/1	100%	2/2
express/examples/mvc/controllers/pet/	94.12%	16/17	50%	1/2	100%	4/4	100%	16/16
express/examples/mvc/controllers/user-pet/	92.86%	13/14	50%	1/2	100%	1/1	100%	13/13
express/examples/mvc/controllers/user/	100%	21/21	100%	4/4	100%	6/6	100%	19/19
express/examples/mvc/lib/	97.96%	48/49	80%	20/25	100%	2/2	100%	46/46

The Future of Software Testing

- **Low-code and no-code testing**
  - Tools for nontechnical users
  - Faster test creation and execution
- **IoT and edge testing**
  - Test connectivity, security, and performance
  - Simulate diverse real-world network conditions
- **5G and ultralow latency testing**
  - Validate performance for high-speed, low-latency apps
  - Crucial for IoT, autonomous, and remote healthcare systems
- **AI-driven predictive and self-healing systems**
  - AI detects and fixes minor issues automatically
  - Predictive testing prevents failures before they occur
- **Generative AI in testing**
  - AI creates dynamic test cases automatically
  - Expands coverage and improves reliability



References

https://www.ibm.com/think/topics/software-testing
https://www.ibm.com/think/topics/functional-testing
https://testgrid.io/blog/tdd-vs-bdd-which-is-better/