Design a Dynamic Frontend with React

Frontend Testing Using Jest

Engage and Think

Jim, a developer at XYZ Company, ensures the web application runs smoothly and meets quality standards. QA feedback highlights the need for better frontend testing. He now leads efforts to enhance testing strategies. This helps catch issues early in development.

Why do you think frontend testing is crucial for a web application? How can it impact the user experience and business success?

Learning Objectives

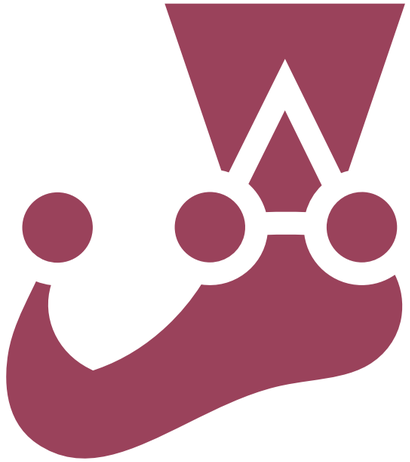
By the end of this lesson, you will be able to:

- Identify the workflow of Jest and its significance in testing the Document Object Model (DOM) during frontend development
- Utilize Jest-DOM to test DOM elements efficiently within web applications
- Apply the features of DOM testing libraries to validate frontend behaviors and functionality
- Construct reliable tests using the React Testing Library for various web development scenarios
- Select the appropriate tools for DOM testing based on project requirements and test coverage goals

Frontend DOM Testing Using Jest

What Is Jest?

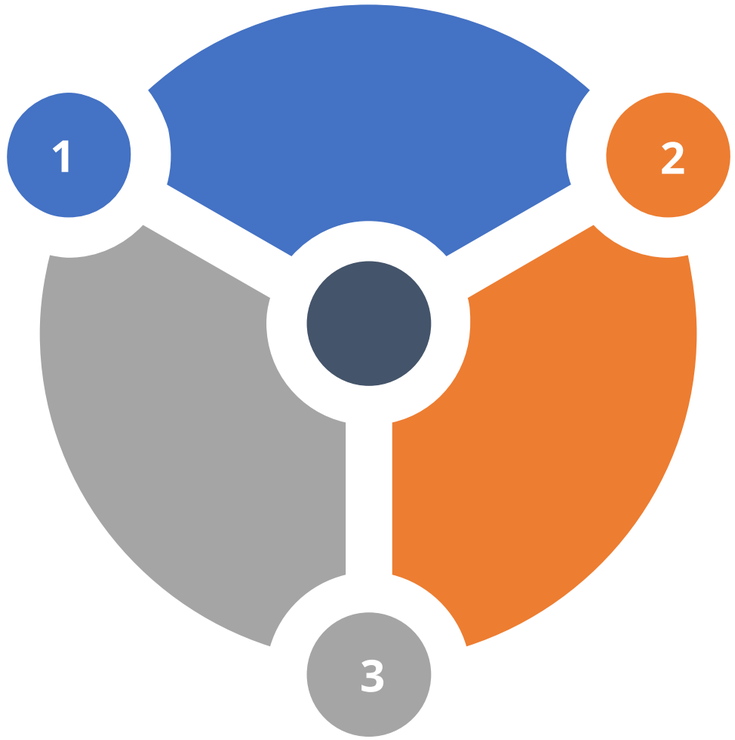
It is an open-source JavaScript testing framework that is automated, easy to use, and fast, providing a seamless developer experience for testing code.



Developed by Facebook, Jest primarily tests JavaScript, especially React, and includes DOM support to ensure proper UI interactions.

Benefits of Jest

Popularity
It is widely adopted in the JavaScript community.

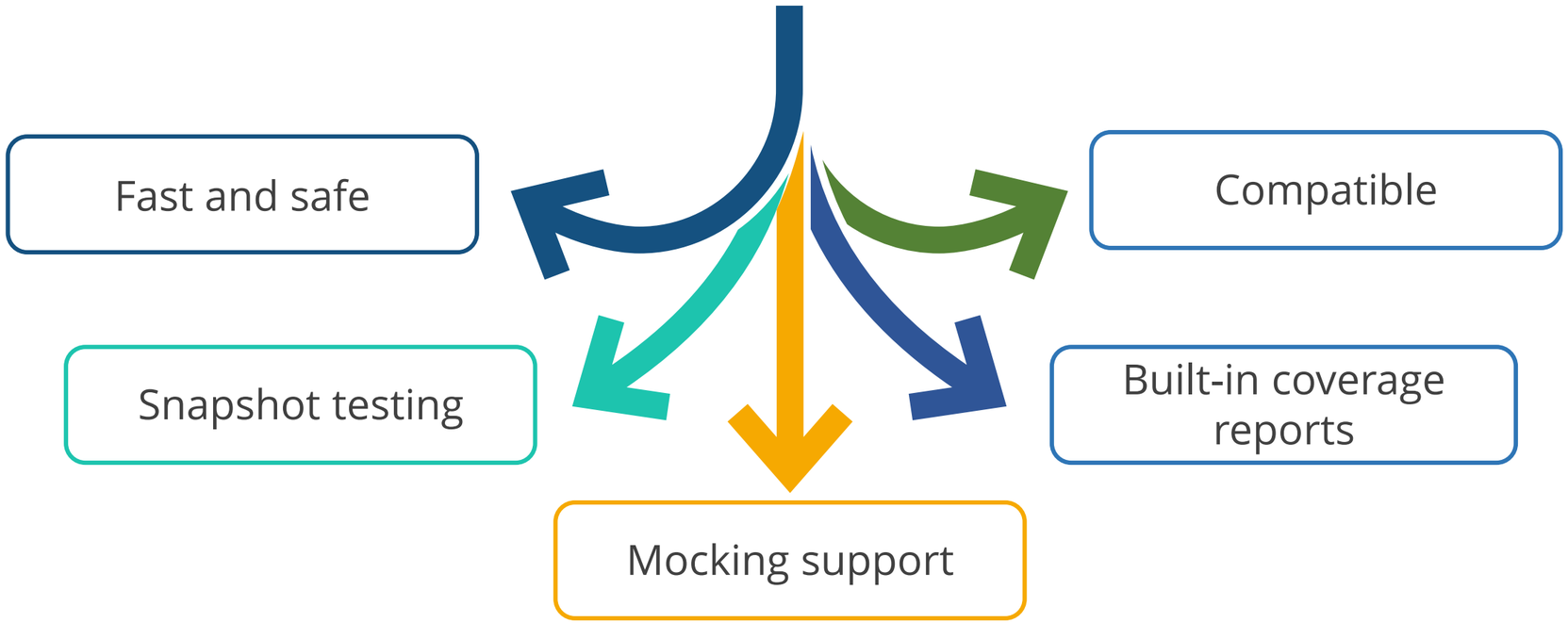


Zero configuration
It is ready to use with a minimal setup.

Developer experience
It has an interactive watch mode that streamlines the testing process.

Features of Jest

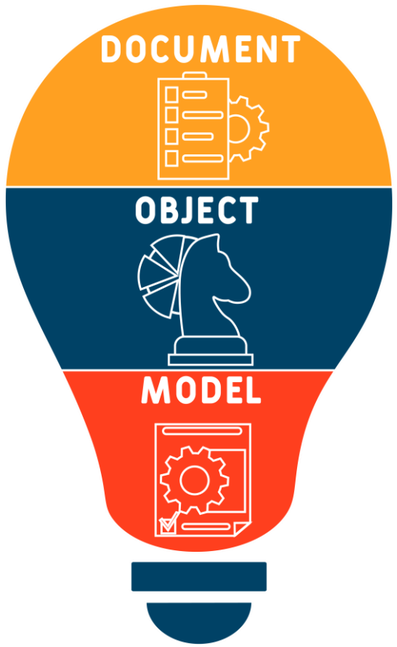
The following are the key features of Jest:



Jest integrates seamlessly with DOM testing libraries. It uses features like mocking, snapshots, and coverage to help verify how the UI behaves from the user’s perspective.

What Is DOM Testing?

It refers to the process of testing the DOM of a web application, where a DOM represents the document as a tree of objects.



It is essential to ensure the proper functionality and behavior of web applications by testing how the application interacts with the DOM.

Why DOM Testing?

It verifies the correctness and functionality of the user interface as rendered in the browser. Its importance lies in:

- Performance optimization
- Automated testing
- Event handling
- Cross-browser compatibility

Importance of Jest in DOM Testing

Integrating Jest into your frontend project enhances testing capabilities by adding a robust layer of functionality. Some key points to note are:

- Ensuring correctness
- Detecting regression
- Enhancing user experience
- Ensuring cross-browser consistency

Jest provides a wide range of tools to ensure code strength and quality.

Setting Up Jest for DOM Testing

The steps for setting up Jest for frontend testing include:

Step 1: Initialize the project

- Create a new directory for your project and run **npm init** to generate a **package.json file**

Step 2: Install Jest

- Run **npm install --save-dev** Jest to install Jest as a development dependency

Setting Up Jest for DOM Testing

Step 3: Configure test script

- Add a test script to your **package.json** file under the **scripts** section with the following content: **test**: **Jest**

Step 4: Babel integration

- Install Babel along with Jest: **npm install --save-dev babel-Jest @babel/core @babel/preset-env**

Setting Up Jest for DOM Testing

Step 5: Write your first test

- Create a test file and name it **example.test.js**

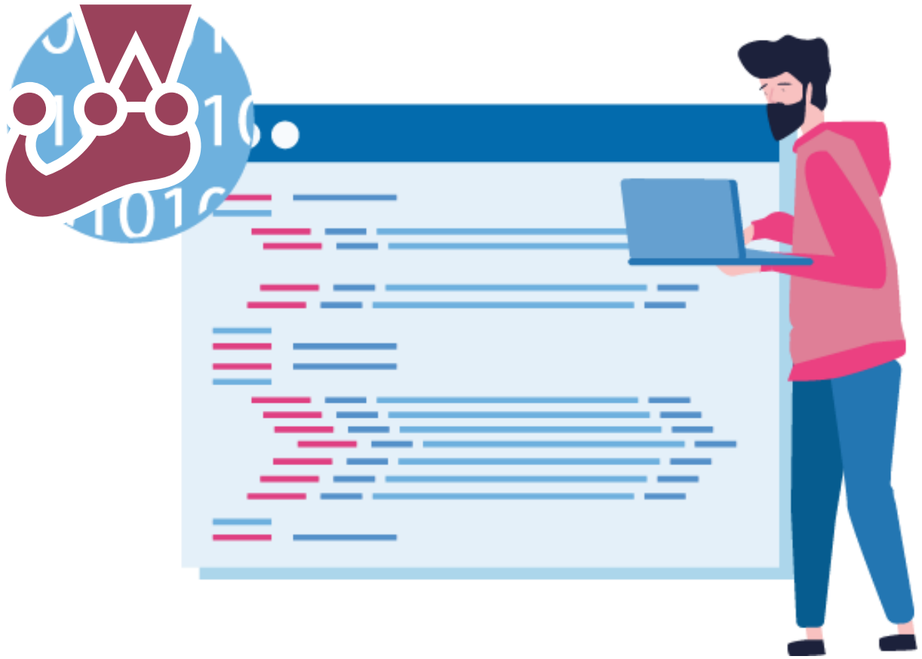
Step 6: Run the test

- Execute the test by running **npm test** in the terminal

After the initial Jest setup and test creation, some components may depend on external sources like APIs or databases. In such cases, mocking becomes essential.

What Is Mocking in Jest?

It is an essential concept in modern testing methodologies. It refers to replacing untestable parts of a system with simplified and controllable replacements.



Its purpose is to test a component's functionality in isolation without relying on external systems.

Why Mocking?

It is used in software development for several purposes, primarily related to testing and maintaining code quality. Here are some key reasons why developers use mocks:

1. Isolation
2. Simplicity
3. Speed
4. Control

Mocking in DOM Testing

In DOM testing, mocking is commonly used to isolate the code being tested from the actual DOM and its interactions. Here are some use cases of mocking in DOM testing:

Simulating user interactions with mock functions

Testing asynchronous behavior with mock APIs

Quick Check

You are working on a web application that displays a list of users fetched from an API, and you want to ensure that the list renders correctly in the browser before deployment. Which tool would be most suitable for testing whether the user list appears correctly in the DOM?

A. A CSS framework like Bootstrap

B. A JavaScript testing framework like Jest

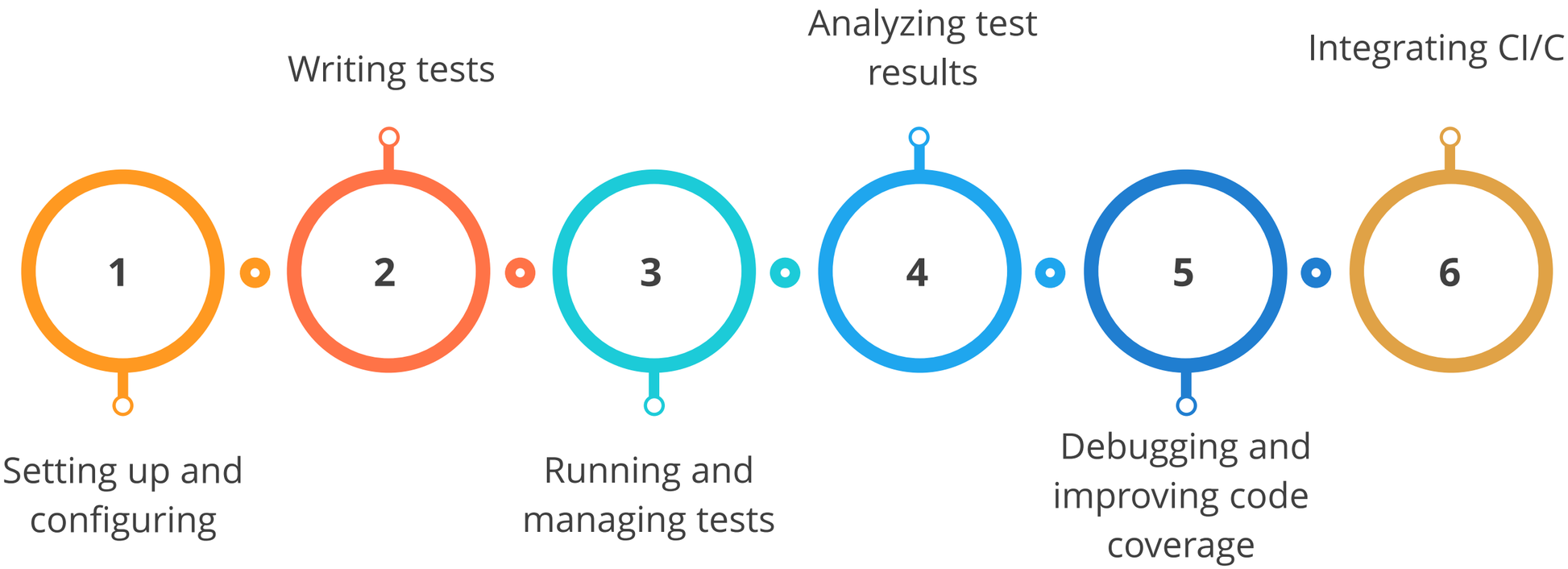
C. A database management system like MySQL

D. A graphic design tool like Figma

Exploring Jest Workflow

Workflow of Jest

Working with Jest involves the following steps:



Tools for Jest Workflow

The tools used in different stages of the Jest testing workflow are:

Jest also provides features like watch mode and Jest reporting for monitoring the test runs, analyzing the bugs, and generating a report.

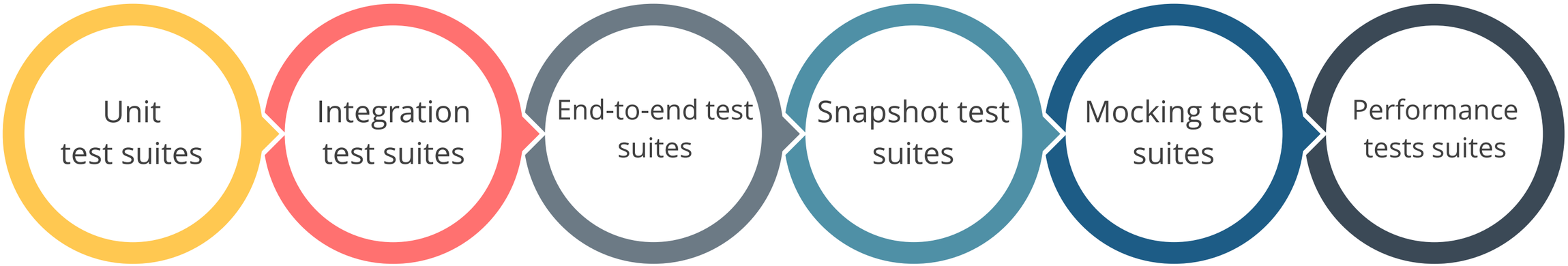
What Is a Test Suite?

It is a collection of related test cases that collectively test a specific unit of functionality.

It organizes tests logically and manageably, making understanding and maintaining the test code easier.

Types of Jest Test Suites

Jest supports different types of test suites, each with its own purpose and execution behavior. Here's an overview of the common types:



These test suites work together to provide complete coverage, identifying issues at various stages of development and deployment.

Why Test Suite?

They are important as they provide the following benefits while testing different parts of software:

Organized structure

Efficient execution

Improved maintainability

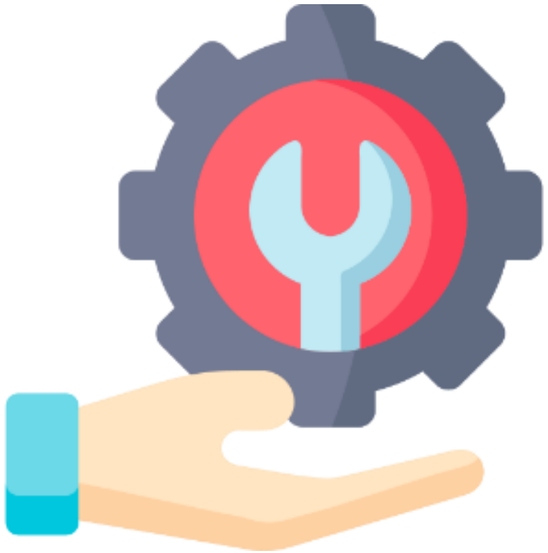
Contextual setup and teardown

Enhanced readability and navigation

Comprehensive coverage

What Is a Test Case?

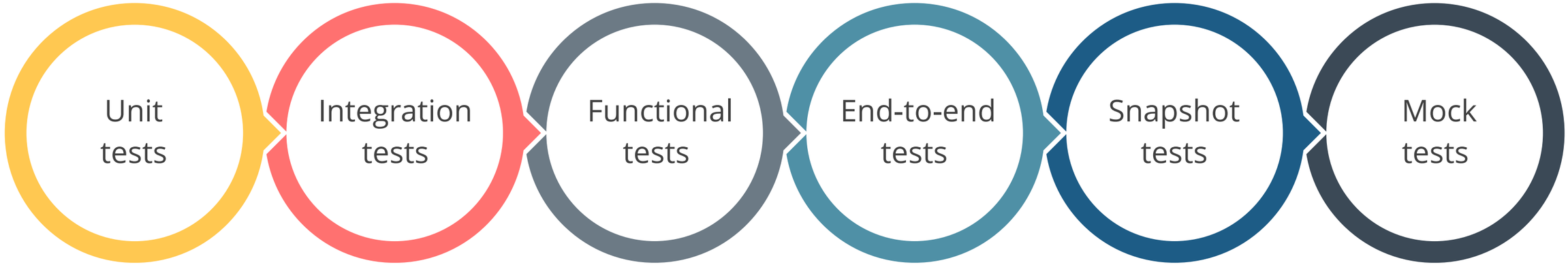
It is an individual unit of testing, typically used for testing one specific aspect or behavior of the code.



A test case employs some special functions within a test suite to validate and verify specific functionalities or behaviors of the software.

Jest Test Case Types

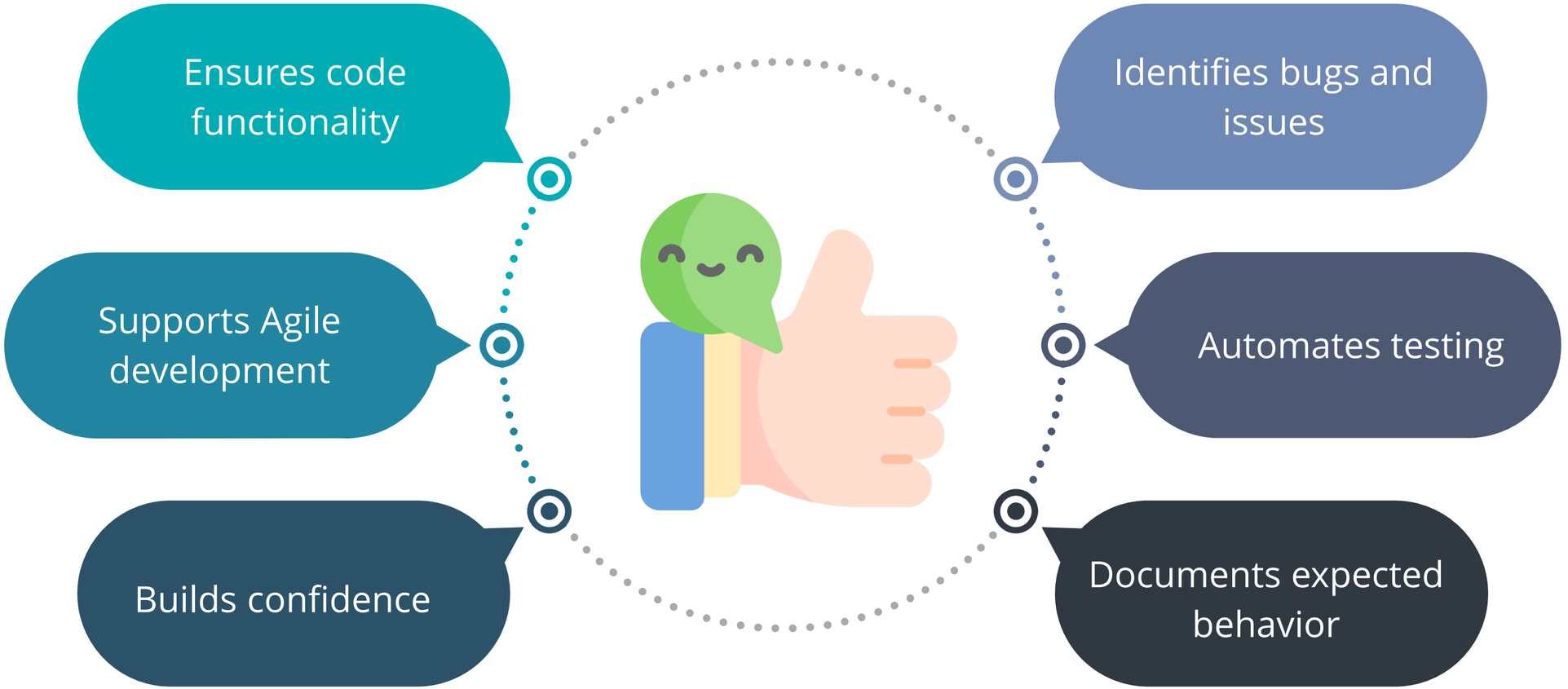
JEST supports different types of test cases, each with its own purpose and execution behavior, such as:



These test types help teams prioritize what to test, streamline debugging, and maintain reliable code throughout development.

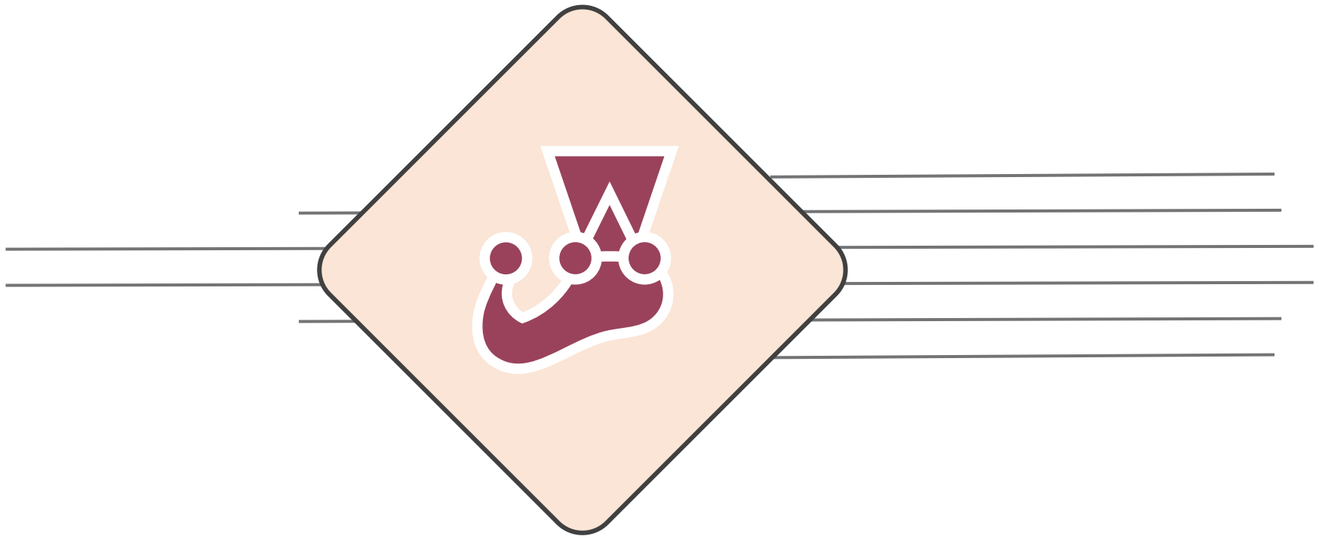
Importance of Test Case

Here is a breakdown of why a test case is important for testing one specific aspect or behavior of the code:



What Is a Test Runner?

It is a tool used to run tests and export results.



It scans the source directory, selects the test files, and runs them to check for bugs and errors.

Test Runner Types

Jest supports different types of test runners, each with its purpose and behavior. Here is an overview of the most common types:

Why Test Runner?

Here is a breakdown of why a test runner is crucial for selecting and running tests to verify bugs.

Consistent test environment

Reports and metrics

Continuous integration

Test-driven development

Parallel execution

Dependencies and lifecycle

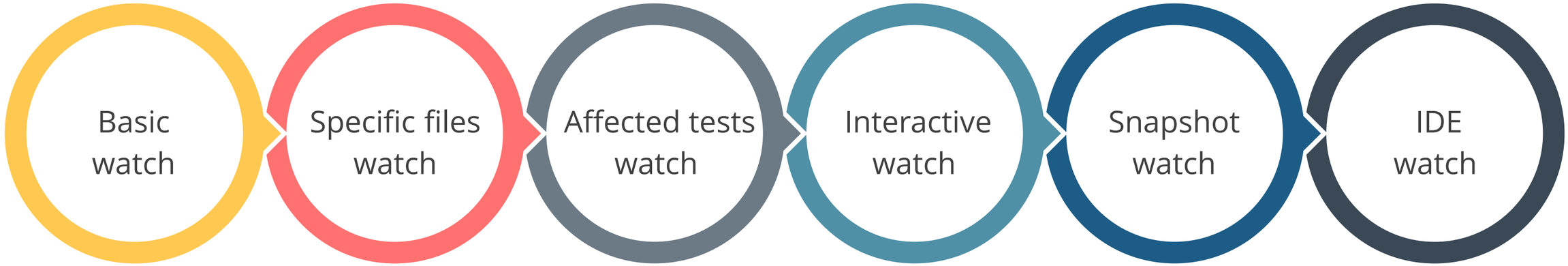
What Is Jest Watch Mode?

It is a feature that automatically reruns tests when it detects any change in the codebase.

It helps developers get instant feedback by rerunning only the affected tests, speeding up the development process.

Jest Watch Mode Types

Jest supports different types of watch modes, each with its own purpose and execution behavior. Here's an overview of the most common types:



Each mode optimizes test execution for different workflows, helping developers work faster.

Why Watch Mode?

Here is a breakdown of why watch mode is important for rerunning the tests when it detects changes in the codebase:

Working of Watch Mode

Watch mode operates by monitoring specific files and their dependencies, running only the relevant tests upon detecting changes. This is achieved through two key processes:

- **File Monitoring**: It watches the changes in files related to your tests, including test files and the files they are testing on.
- **Incremental Testing**: When changes are detected, Jest reruns tests for the modified files, ensuring efficiency in the process.

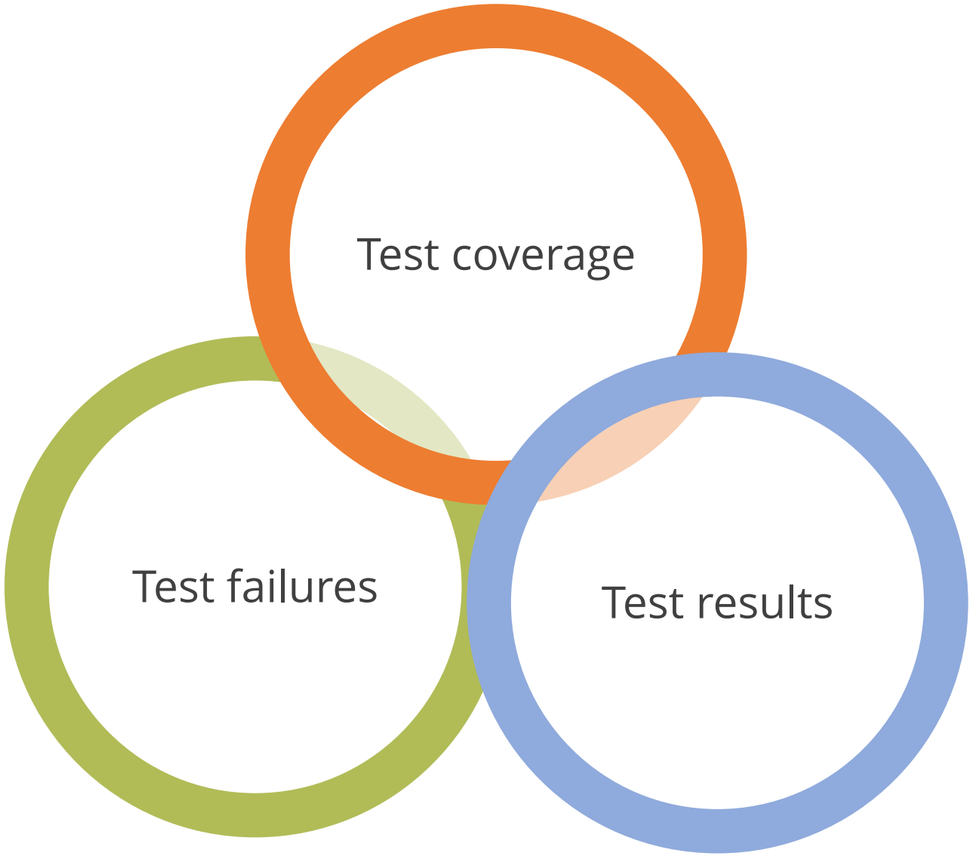
What Is Jest Reporting?

It delivers comprehensive summaries of test executions, indicating which tests succeeded, failed, or were omitted.

Jest provides a text-based report in the console that shows which tests passed, which tests failed, and how long each test took to run.

Jest Reporting Types

In Jest, there are different types of reporting. The most common types include:



Why Reporting?

Reporting is important for comprehensive summaries of test executions as it includes:

- Quality assurance
- Decision making
- Continuous improvement
- Compliance documentation

Example of Using Jest

Here is an example of how to use Jest to test a simple function:

Step 1: Build the function

```
function add(a, b) {
  return a + b;
}
```

This function is designed to add two numbers.

Example of Using Jest

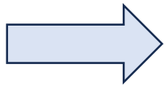
Step 2: Make a test file

```
const add = require('./add');

describe('add function', () => {
  test('should add two numbers', () => {
    expect(add(1, 2)).toBe(3);
  });

  test('should handle negative numbers', () => {
    expect(add(-1, 2)).toBe(1);
  });

  test('should handle zero', () => {
    expect(add(0, 1)).toBe(1);
    expect(add(1, 0)).toBe(1);
  });
});
```

Example of Using Jest

Step 3: Run the test

```
npm test
```



Output:

```
PASS ./add.test.js
  ✓ should add two numbers
  ✓ should handle negative numbers
  ✓ should handle zero

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 passed, 0 total
Time:        3.034s
```

Quick Check

You have just executed a set of test cases using Jest for your web application and need to review the test results. You want to generate a detailed report to analyze passed, failed, and skipped tests. Which Jest feature would help you generate this report?

A. Jest Watch Mode

B. Jest Reporters

C. Jest Snapshot Testing

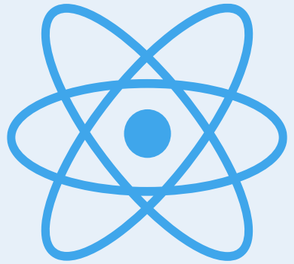
D. Jest Mock Functions

Introduction to Jest-DOM

What Is Jest-DOM?

Jest-DOM assists in testing JavaScript or React code by simplifying and enhancing the validation and interaction with webpage elements like buttons, text, and links during tests.

Jest-DOM
and
React



By providing custom matchers, it makes test assertions more readable and closely aligned with how users interact with the UI.

Matchers in Jest-DOM

Jest-DOM library provides these specialized functions or methods. Some of the common matchers available in Jest-DOM are:

01 **toBeInTheDocument():** Checks if an element is present in the DOM

02 **toHaveStyle():** Verifies if an element has a specific inline CSS style

03 **toHaveAttribute():** Checks if an element has a specific attribute with a given value

Matchers in Jest-DOM

04 **toHaveTextContent():** Asserts that an element contains the specified text content

05 **toHaveClass():** Asserts that an element has a specific CSS class

06 **toContainElement():** Verifies if an element contains its descendant or not

Applications of Jest-DOM

Jest-DOM enhances testing workflows in areas such as:

- React component testing
- End-to-end testing
- Debugging
- User interface testing

Examples of Jest-DOM

Practical examples of using Jest-DOM matchers and utilities include:

1. To verify whether a DOM element exists in a document
2. To check whether an HTML form element, like a button, is in a disabled state
3. To check whether an element's content matches your expectations
4. To verify whether a DOM element has specific CSS classes applied to it

Examples of Jest-DOM

Practical examples of using Jest-DOM matchers to verify element presence and disabled state:

Verifying the element’s presence

```
expect(document.querySelector('.my-element')).toBeInTheDocument();
```

Disabling an element

```
expect(document.querySelector('button')).toBeDisabled();
```

Examples of Jest-DOM

Practical examples of using Jest-DOM matchers to test content and verify CSS classes:

Testing content

```
expect(document.querySelector('.message')).toHaveTextContent('Hello, world!');
```

Checking CSS class

```
expect(document.querySelector('.btn')).toHaveClass('active');
```

Quick Check

You are writing tests for a React component that displays a success message when a form is submitted. You need to verify that the message "Form submitted successfully!" appears in the DOM after submission. Which Jest matcher would be most appropriate for this test?

A. expect(element).toHaveTextContent("Form submitted successfully!")

B. expect(element).toBeVisible("Form submitted successfully!")

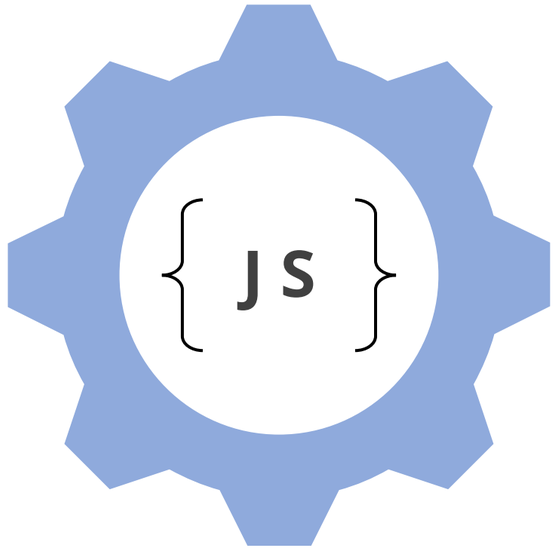
C. expect(element).toExist("Form submitted successfully!")

D. expect(element).toContain("Form submitted successfully!")

Overview of DOM Testing Library

What Is DOM Testing Library?

It is a JavaScript testing library that provides utilities and tools for testing web applications by interacting with the DOM.



It is primarily used for writing tests that mimic user interactions with a web page.

Purpose of DOM Testing Library

DOM testing serves multiple purposes in modern development, such as:

1. User centric testing
2. Accessibility testing
3. Explicit waiting
4. Simplifying testing
5. Helper tool testing
6. Reliable tests

What Is Querying Elements in DOM Testing?

Querying elements in the DOM testing library is a fundamental aspect of writing tests to interact with and make assertions about elements in a web page. Common querying methods include:

Methods	Explanation
getBy	Selects a single element that is expected to exist in the DOM
queryBy	Selects a single element and returns null if not found instead of throwing an error
getAllBy	Selects multiple elements that match the criteria and returns them as an array
findAllBy	Selects multiple elements and returns a promise that resolves to an array of matching elements

Querying Methods: Examples

Some querying methods are:

getBy method

queryBy method

Example:

```
import { getByText, getByRole } from
'@testing-library/dom';

const elementByText =
getByText(document.body, 'Hello,
World!');
const buttonByRole =
getByRole(document.body, 'button');
```

Example:

```
import { queryByText, queryByRole } from
'@testing-library/dom';

const elementByText =
queryByText(document.body, 'Hello,
World!');
const buttonByRole =
queryByRole(document.body, 'button');
```

Querying Methods: Examples

Some querying methods are:

getAll method

findAllBy method

Example:

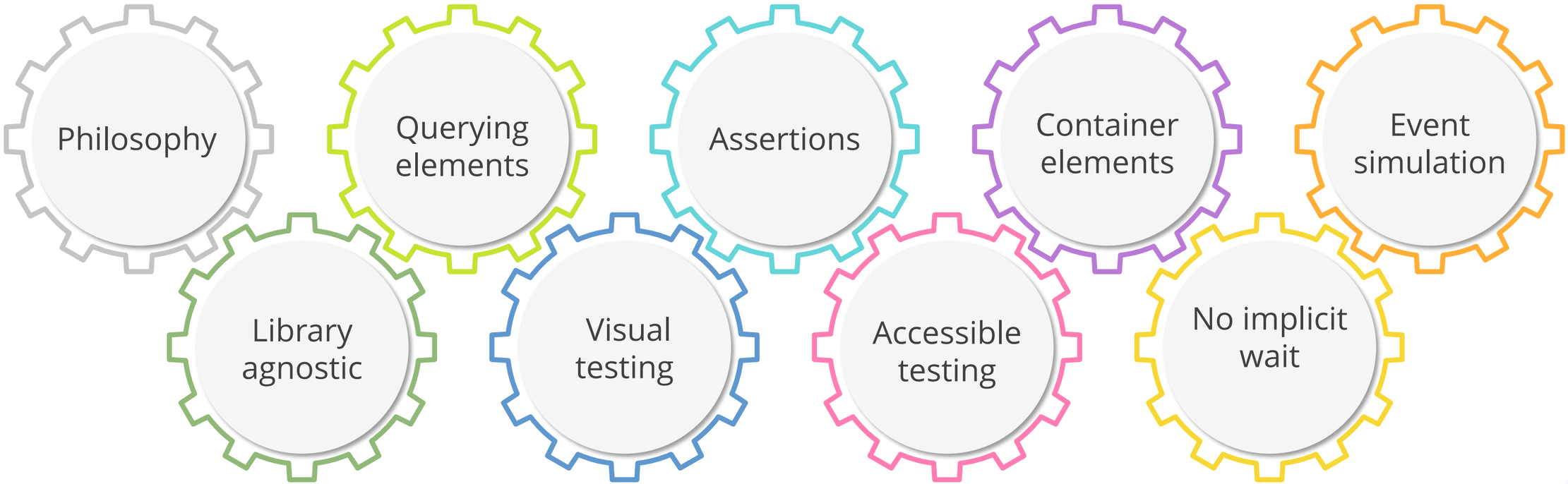
```
import { getAllByText, getAllByRole }
from '@testing-library/dom';

const elementsByText =
getAllByText(document.body, 'Hello');
const buttonsByRole =
getAllByRole(document.body, 'button');
```

Example:

```
import { findAllByText, findAllByRole }
from '@testing-library/dom';

async function fetchDataAndTest() {
  const elementsByText = await
findAllByText(document.body, 'Hello');
  const buttonsByRole = await
findAllByRole(document.body, 'button');
}
```

Features of DOM Testing Library

It offers the following key features:



Assisted Practice

Testing a Document Object Model Using Jest **Duration: 20 Min.**

Problem Statement:

You need to test DOM manipulation logic in a Node.js environment. The task involves writing Jest tests that simulate a user interaction and verify changes in the DOM using jsdom.

Outcome:
By the end of this demo, you will be able to set up a simulated DOM using jsdom and validate DOM updates using Jest test cases.

Note: Refer to the demo document for detailed steps:
01_Testing_a_Document_Object_Model__Using_Jest

Assisted Practice: Guidelines

Steps to be followed:

1. Create a Node project
2. Create an HTML file
3. Create a JavaScript file
4. Write a Jest test
5. Run the Jest test

ASSISTED PRACTICE

Quick Check

You are testing a web application where a button labeled "Submit" should be present in the DOM. You need to select this button in your Jest test to verify its functionality. Which DOM querying method would be the best choice?

A. document.querySelector("Submit")

B. screen.getByText("Submit")

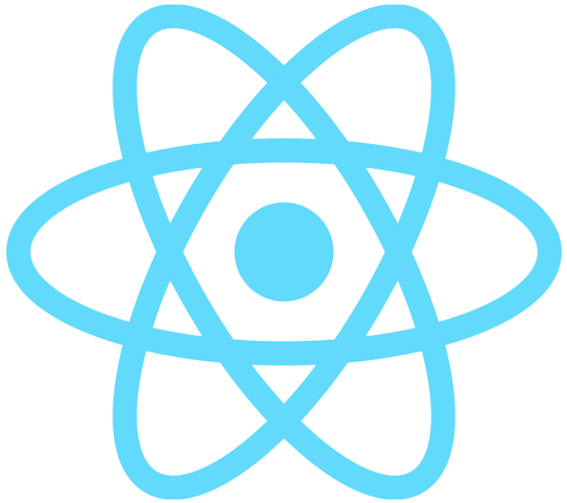
C. screen.findByClass("Submit")

D. document.getElementById("Submit")

Exploring the React Testing Library

What Is React Testing Library?

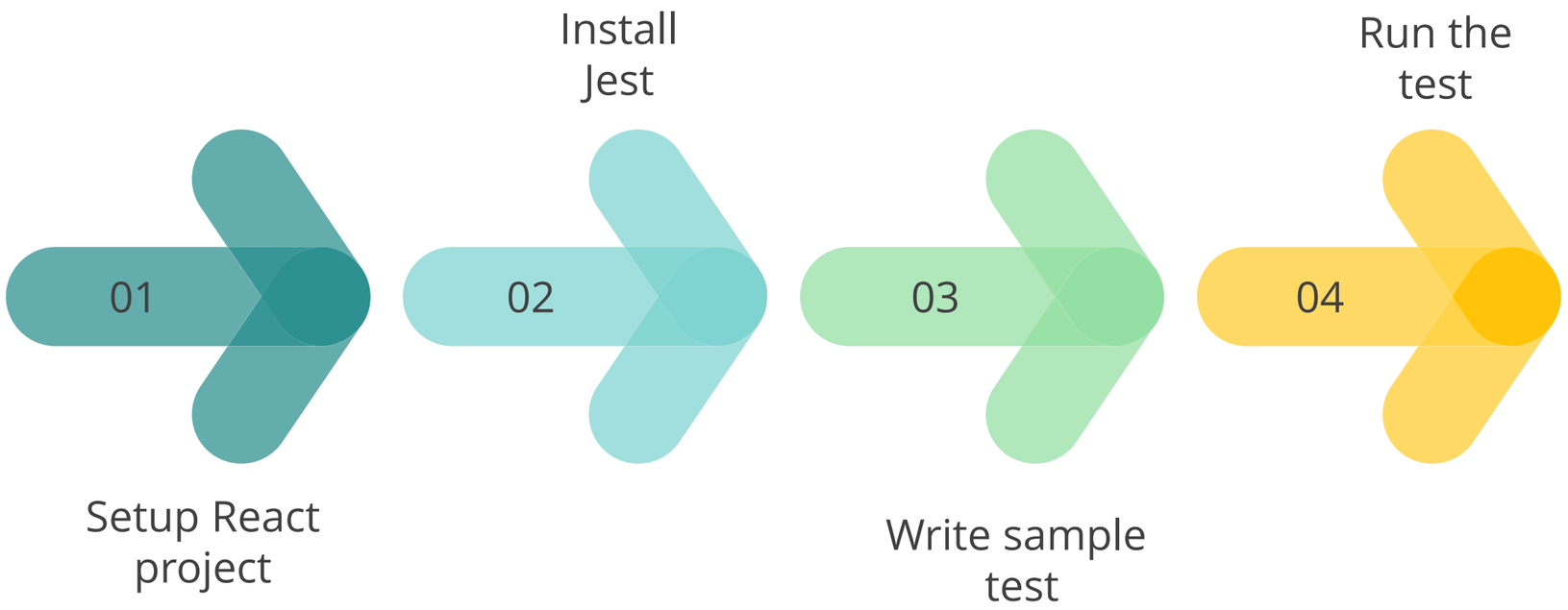
It is a DOM testing utility that helps developers write tests that are focused on user behavior, rather than on the internal implementation details of components.



It enhances testing reliability by avoiding direct access to internal component logic and focusing on observable outcomes.

React Testing Using Jest

Testing React applications using Jest is a common practice in JavaScript development. It involves the following steps:



Jest is a popular JavaScript testing framework developed by Facebook, and it is often used in combination with tools like the React testing library.

Features of React Testing Library

User-centric testing

DOM testing

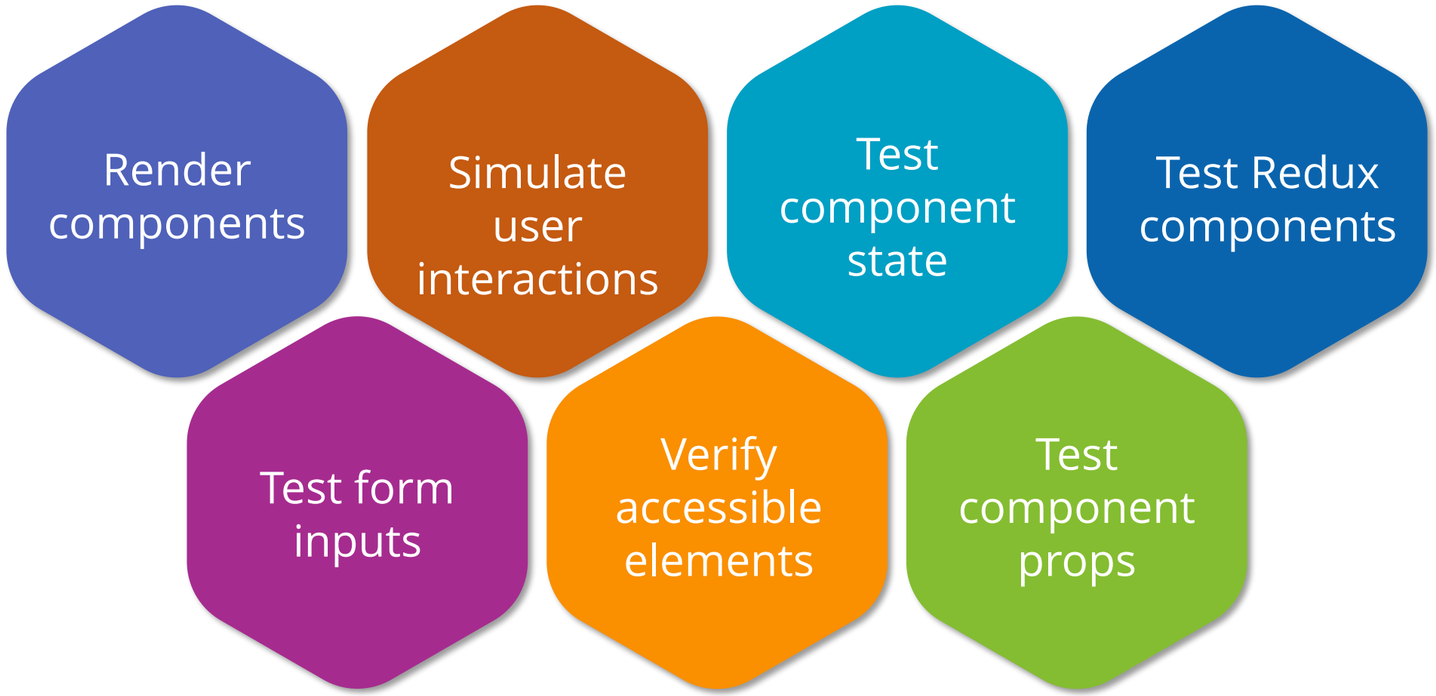
Accessibility testing

Asynchronous testing

Simulating user interactions

Examples of React Testing Library

The following examples demonstrate how React Testing Library supports testing various component behaviors:



Among these examples, one of the most critical is testing user interaction, which involves simulating how users interact with the user interface using React Testing Library.

What Is Testing User Interaction?

It refers to the process of evaluating and verifying that the interface elements of a software application or a website function correctly from the user's perspective.

Why Is Testing User Interaction Important?

It is crucial for several reasons, primarily centered around ensuring a positive and efficient user experience.

Here are some of the reasons why it is important:

- Enhances user experience
- Identifies usability issues
- Facilitates user engagement
- Supports brand reputation

Concepts of Testing User Interaction

Testing user interaction involves several key concepts. Following are the three basic concepts for testing user interaction:

Querying and selecting elements

Accessibility testing

Simulating user actions

Testing User Interaction: Example

Here is an example of testing UI components with Jest:

UI component:

```
function ToggleComponent() {
  const [showMessage, setShowMessage] = useState(false);

  return (
    <div>
      <button onClick={() =>
setShowMessage(!showMessage)}>
        Toggle Message
      </button>
      {showMessage && <p>Hello World</p>}
    </div>
  );
}
```

Testing User Interaction: Example

The following code ensures that the interaction of the mentioned components functions properly:

Test code:

```
import { render, fireEvent, screen } from '@testing-
library/react’;

import ToggleComponent from './ToggleComponent';

test('shows the message when the button is clicked', ()
=> {
  render(<ToggleComponent />);

  const button = screen.getByText('Toggle Message');
  fireEvent.click(button);

  expect(screen.getByText('Hello
World')).toBeInTheDocument();
});
```

What Is Snapshot Testing?

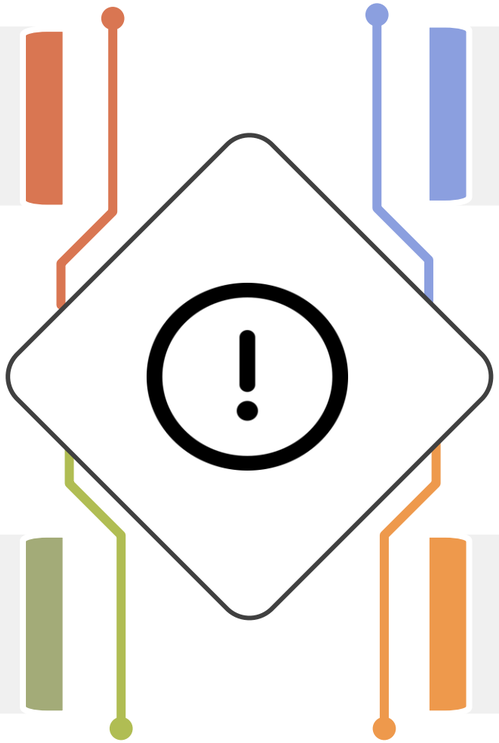
It captures the rendered output of a component and compares it against a saved snapshot. This is useful for ensuring UI consistency and detecting unintended changes.

Why Is Snapshot Testing Important?

It works by capturing the output of your UI component as a snapshot and comparing it to a reference snapshot stored alongside the test. If the two snapshots differ, the test fails.

Here are some of the reasons why it is important:

- Detecting changes
- Facilitating refactoring



- Integrating version control
- Testing efficiency

Concepts of Snapshot Testing

Snapshot testing helps ensure intentional code changes without unexpected UI differences by following these core concepts:

Snapshot Testing: Example

The following example demonstrates how to test a UI component using Jest:

UI component:

```
function UserProfile({ user }) {
    return (
        <div>
            <h1>{user.name}</h1>
            <p>Email: {user.email}</p>
        </div>
    );
}
```

Snapshot Testing: Example

The following code ensures that the component’s snapshot feature works correctly:

Test code:

```
import React from 'react';
import { render } from '@testing-library/react';
import UserProfile from './UserProfile';

test('UserProfile renders correctly', () => {
  const user = { name: 'John Doe', email:
'john@example.com' };
  const { asFragment } = render(<UserProfile user={user}
/>);

  expect(asFragment()).toMatchSnapshot();
});
```

Assisted Practice

Testing a React Component Using Jest **Duration: 20 Min.**

Problem Statement:

You are developing a React app and want to ensure that a component functions correctly. Your task is to create a simple Counter component and write Jest test cases using the React Testing Library to validate its rendering and interactions.

Outcome:
By the end of this demo, you will be able to create a functional React component and write Jest test cases to verify its initial state and user-triggered events.

Note: Refer to the demo document for detailed steps:
02_Testing_the_React_Component_Using_Jest

Assisted Practice: Guidelines

Steps to be followed:

1. Set up a new React project and install Jest
2. Create a simple React component
3. Write a Jest test for the Counter component
4. Configure the Jest
5. Run the Jest test

Quick Check

You have built a React component that displays a user profile, and you want to ensure that its UI does not change unexpectedly. Which Jest feature would you use to compare the current UI output with a previously saved version?

A. Unit Testing

B. Snapshot Testing

C. End-to-End Testing

D. Performance Testing

Developing and Testing an E-commerce React Application

Project agenda: To develop a simple and efficient e-commerce application using React and Vite. The app will fetch and display product data from an external API, demonstrate modular component design, and include unit testing using Jest to ensure component reliability and application stability.

Description: You are tasked with building a React-based e-commerce application to enhance performance and maintainability. This involves using Vite for project setup, fetching product data from an external API, and implementing a reusable product listing component. The project also includes unit testing with Jest to validate component behavior and ensure reliability. The application delivers a smooth user experience with dynamic data rendering and a clean, responsive interface.

LESSON-END PROJECT

Developing and Testing an E-commerce React Application

Steps to be followed:

1. Set up a new React project using Vite
2. Create a ProductList component
3. Modify App.jsx to render the ProductList
4. Create a test file for ProductList
5. Configure Jest
6. Update package.json scripts
7. Test the application

Expected deliverables: A fully functional React-based e-commerce application with core features including dynamic product listing using data fetched from an external API. The application will support modular component structure, responsive UI design, and real-time loading and error handling feedback. It will be integrated with Jest for unit testing of components, structured for scalability using Vite, and optimized for fast development and build performance.

LESSON-END PROJECT

Key Takeaways

- Frontend DOM testing involves validating the Document Object Model (DOM) elements and their behavior in the web application.
- The DOM testing library is used for testing web applications by interacting with them as a user would.
- Jest-DOM is an extension of Jest that provides additional matchers and utilities for DOM testing.
- The React testing library is specifically designed for testing React components.

Thank You