

# Design a Dynamic Frontend with React



# Frontend Testing Using JEST



# A Day in the Life of a MERN Stack Developer

Jim is a software developer at XYZ Company. His team is responsible for developing and maintaining a web application that helps streamline the company's internal processes. One of the critical aspects of their work is ensuring that the application's functionality and user interface are bug-free and meet the company's quality standards.

Recently, the team received feedback from the company's quality assurance (QA) department about the need to improve their testing process, especially regarding frontend testing. Jim takes the lead in exploring and implementing a robust testing strategy to catch potential issues early in the development cycle.

The knowledge of frontend testing using Jest will empower Jim to effectively achieve the task, ensuring a user-friendly web application.



# Learning Objectives

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

- Comprehend the workflow of JEST and its relevance specifically to testing the document object model within frontend development
- Illustrate the essential characteristics and practical uses of JEST-DOM for testing the DOM in web development
- Demonstrate the features of the DOM testing library for testing web applications effectively
- Evaluate the versatility of the React testing library by utilizing it for diverse testing scenarios within a React environment

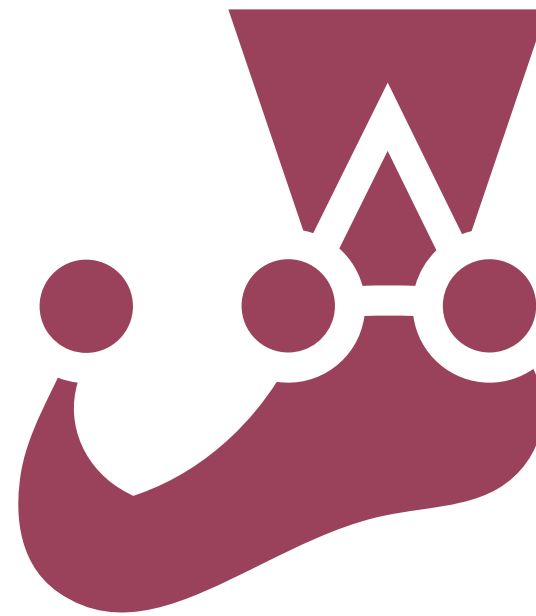




# Frontend DOM Testing Using JEST

# What Is JEST?

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



It was developed by Facebook and is designed primarily for testing JavaScript code, especially React applications.

# Why JEST?

## Popularity

It is widely adopted in the JavaScript community.



## Zero configuration

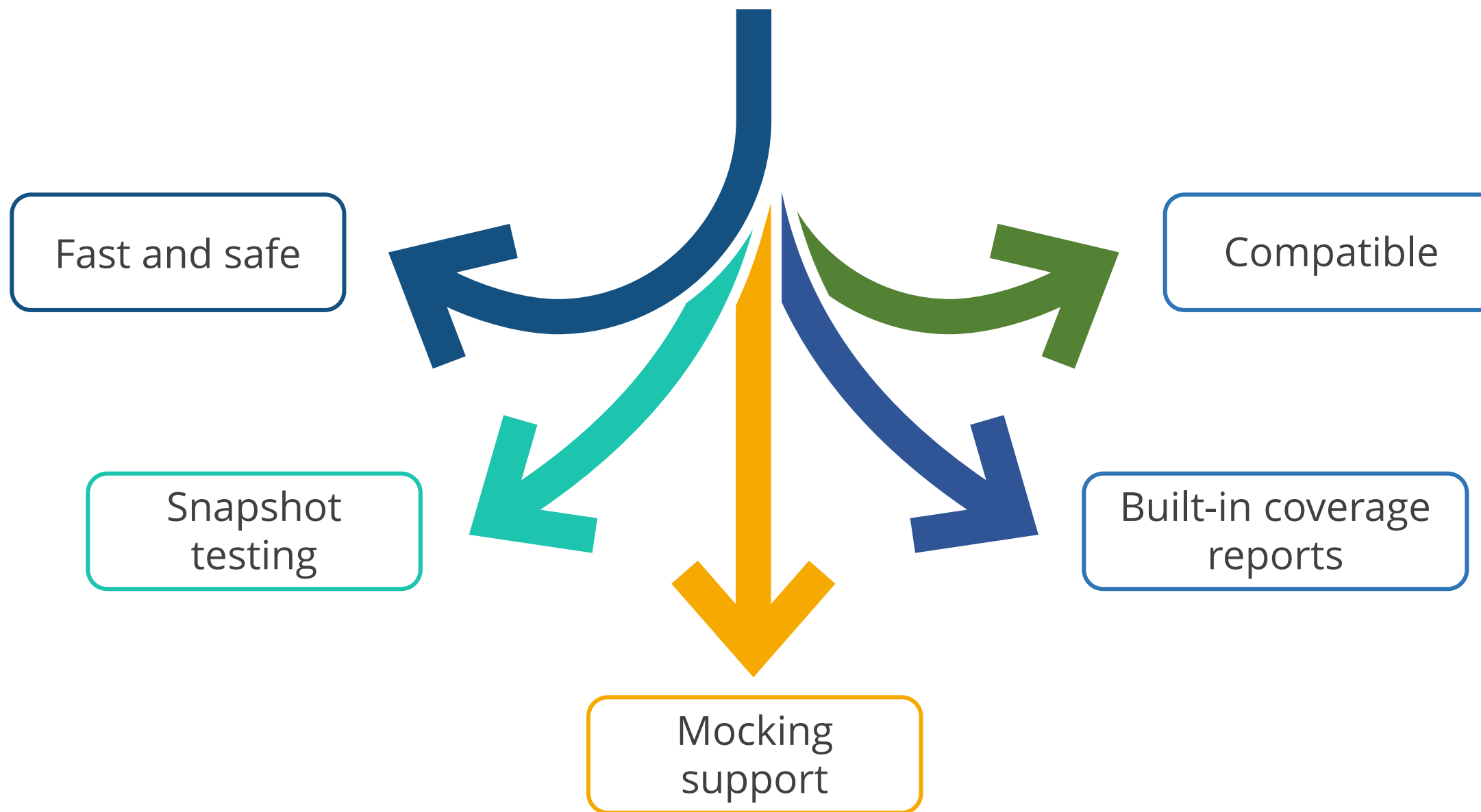
It is ready to use with a minimal setup.

## Developer experience

It has an interactive watch mode which streamlines the testing process.

# Features of JEST

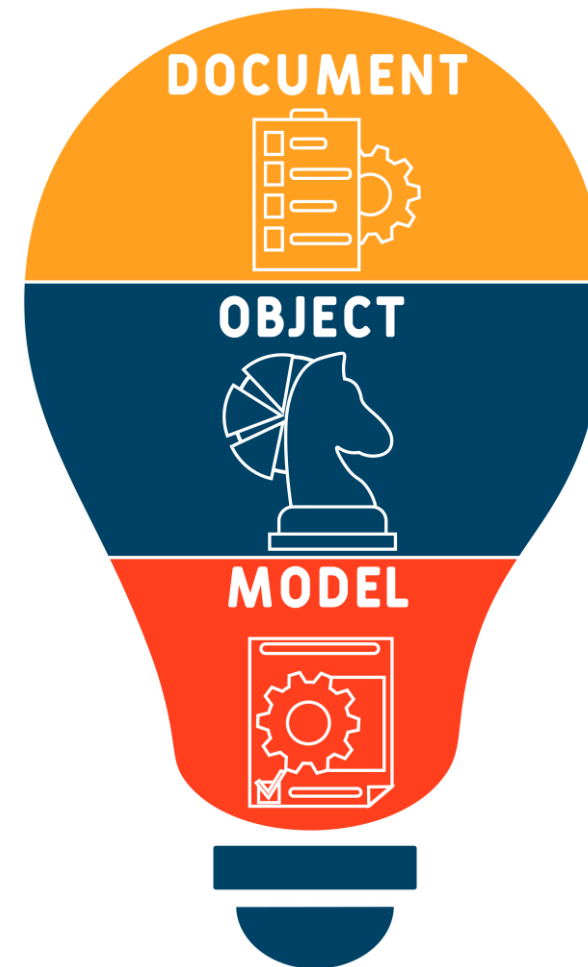
The following are the key features of JEST:





# What Is DOM Testing?

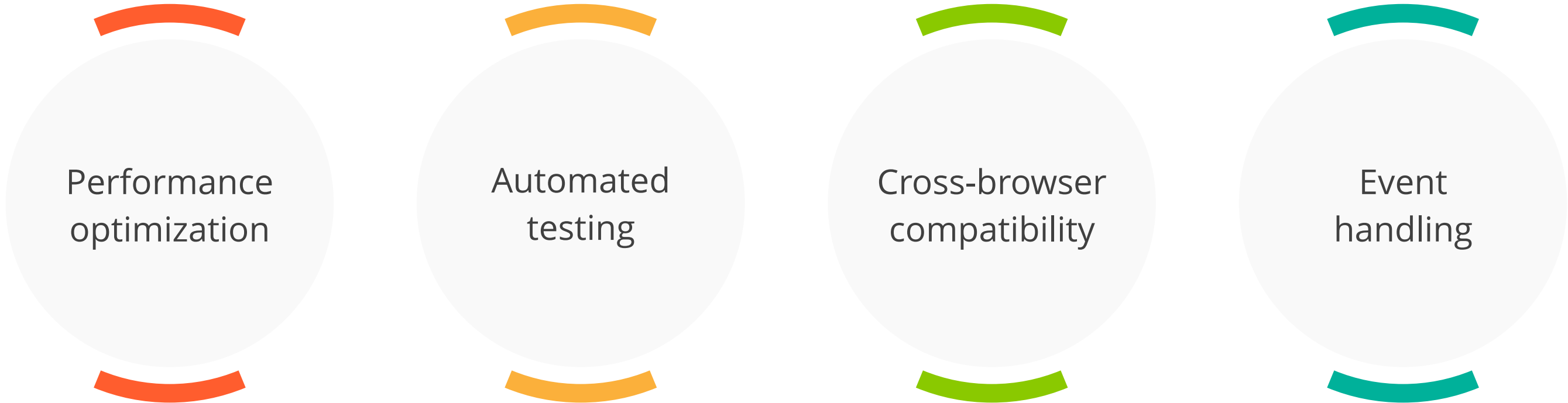
It refers to the process of testing the DOM of a web application, where a DOM represents the document as a tree of objects.



It is essential to ensure 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. Here is why DOM testing is important:



Performance  
optimization


Automated  
testing

Cross-browser  
compatibility

Event  
handling

# Importance of JEST in DOM Testing

Integrating JEST into your frontend project effortlessly enhances testing capabilities with a robust layer of functionality.



Ensuring  
correctness

Detecting  
regression

Enhancing user  
experience

Cross-browser  
consistency

JEST offers a variety of tools to guarantee the strength and quality of the code.

# Setting Up JEST for DOM Testing

Here are the steps for setting up JEST for frontend testing:

## Step 1: Initialize the project

- Make 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

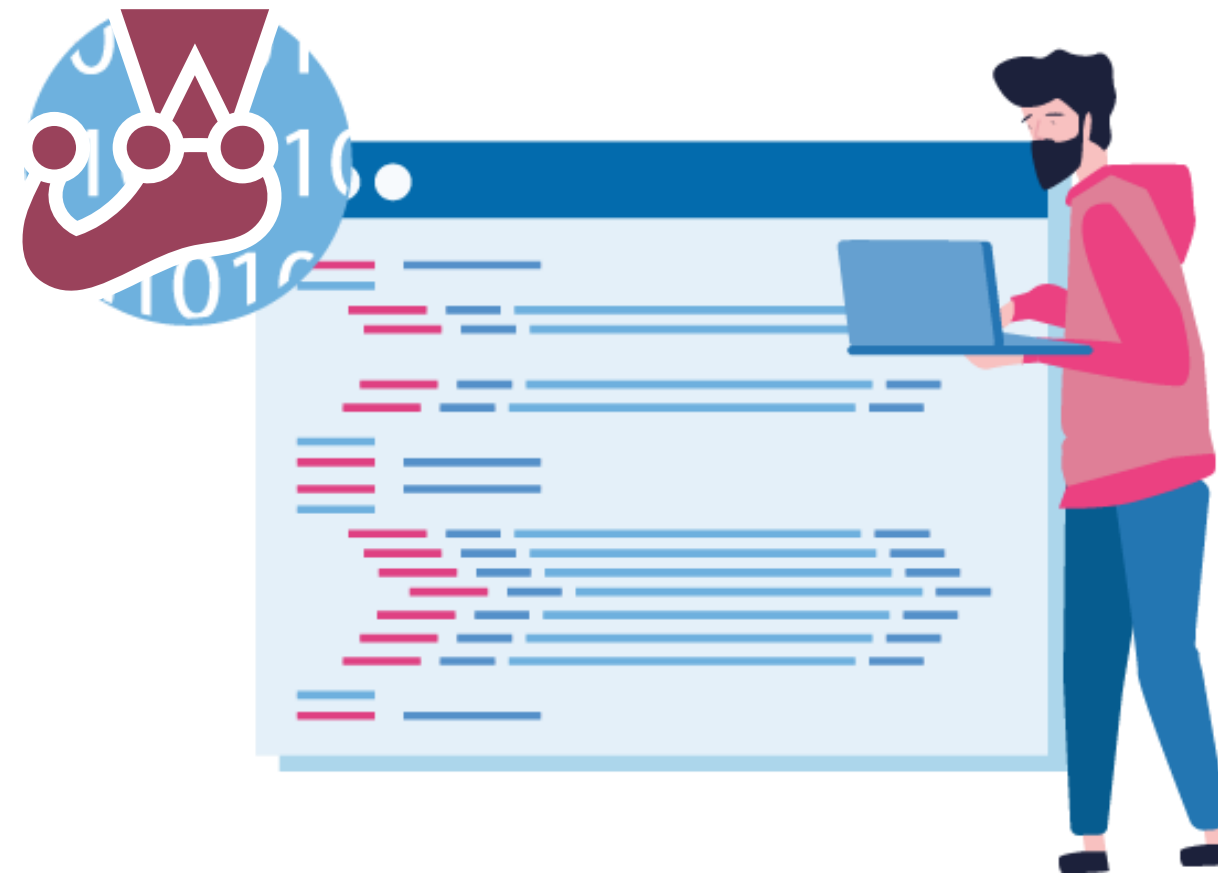
- Make a test file and name it **example.test.js**

Step 6: Run the test

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

# 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:



Isolation

Simplicity

Speed

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

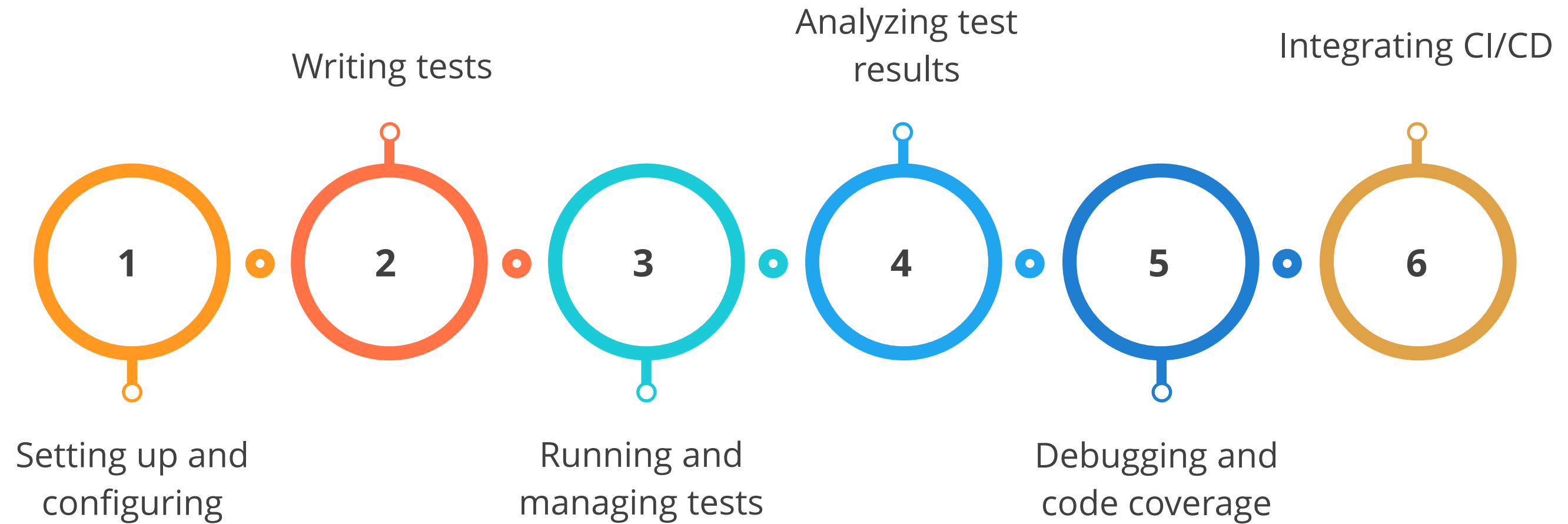
Testing asynchronous behavior with mock APIs



## Workflow of JEST

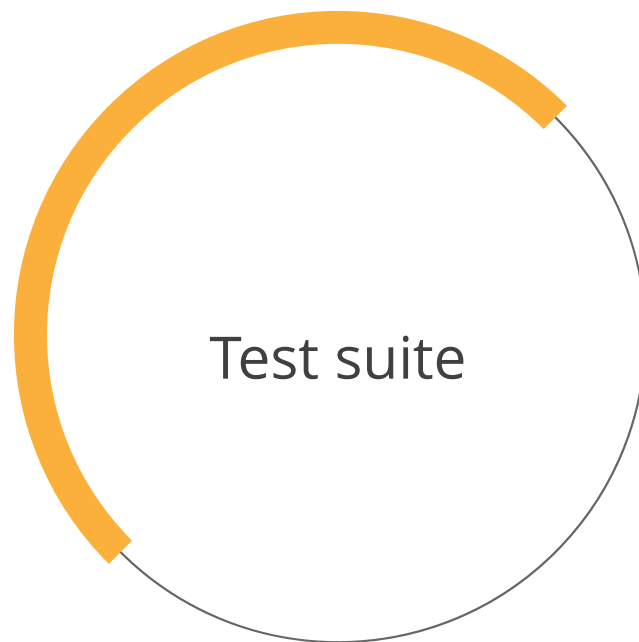
# Workflow of JEST

Working with JEST typically involves the following steps:



# Tools for JEST Workflow

Following is a list of tools used in different stages of the JEST testing workflow:



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

# What Is a Test Suite?

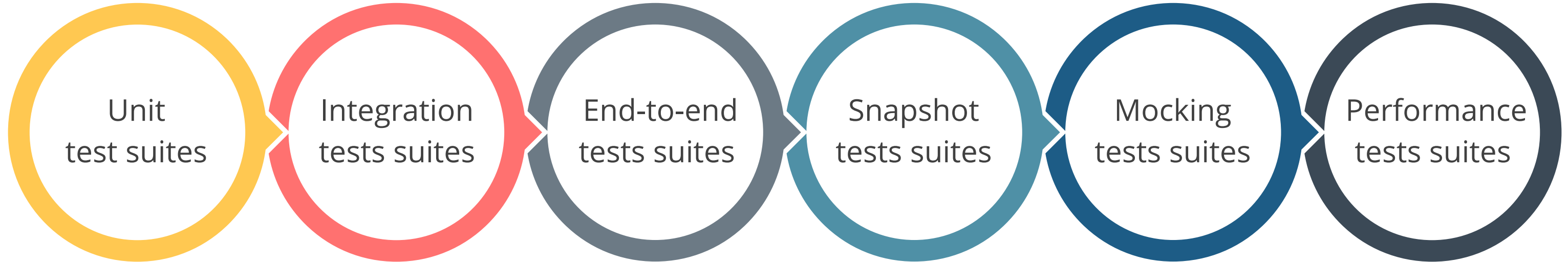
It is a collection of related test cases that collectively test a specific unit of functionality.



It organizes tests logically and manageably, making it easier to understand and maintain the test code.

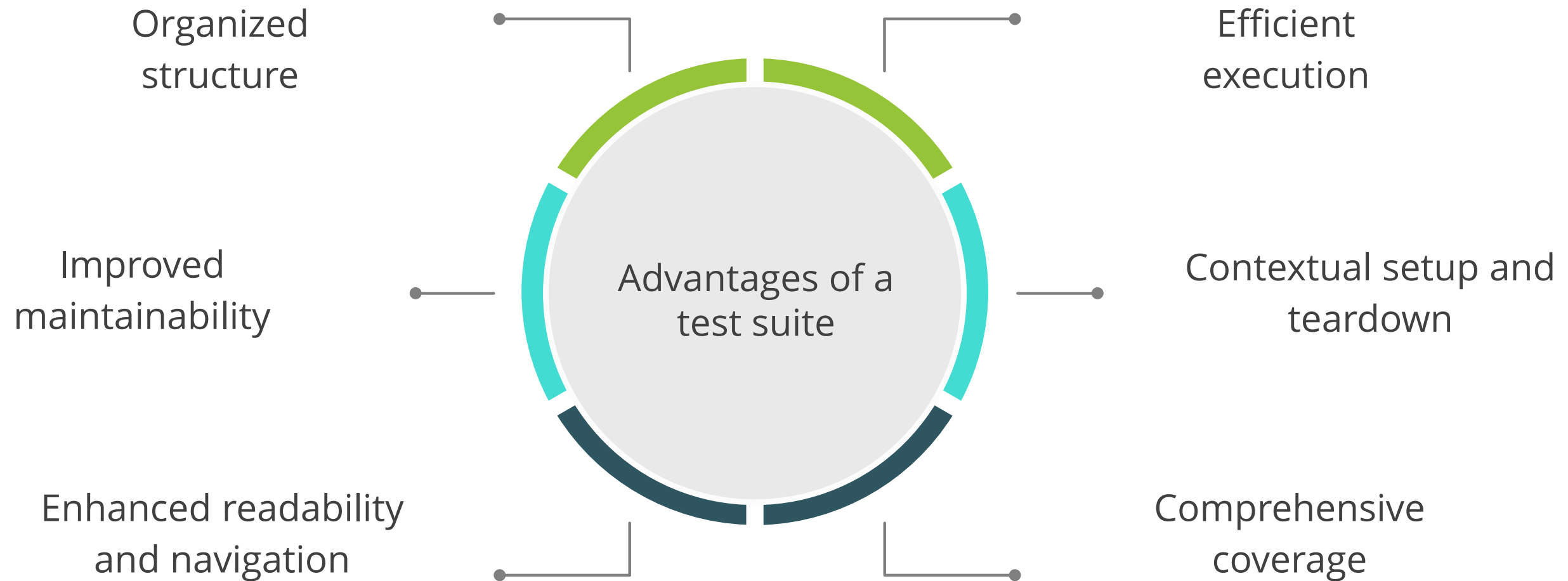
# Types of JEST Test Suites

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



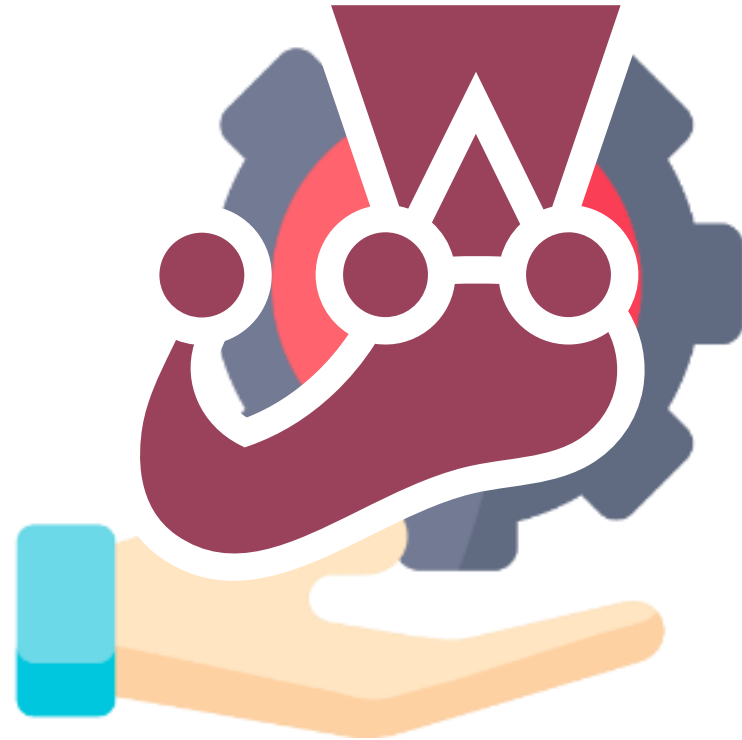
# Why Test Suite?

Test suites are important as they provide the following set of benefits while testing different parts of software:



# What Is a Test Case?

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

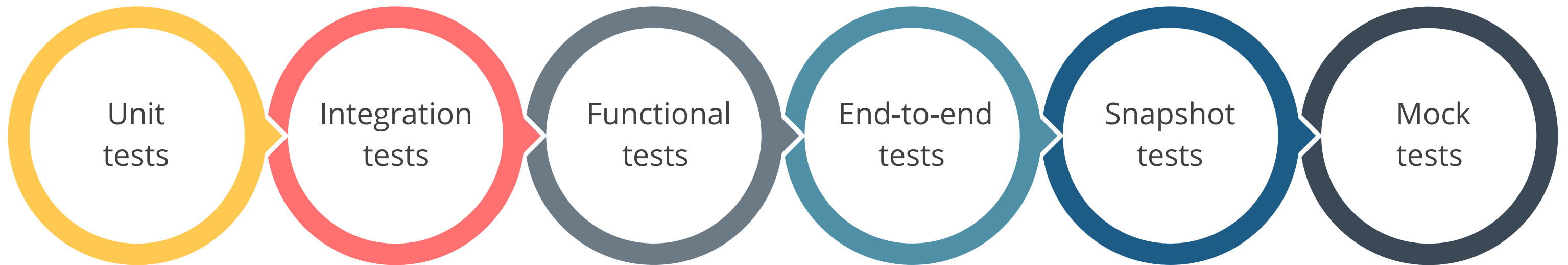


A test case employs some special functions within a test suite to validate and verify specific functionalities or behaviors of the software.



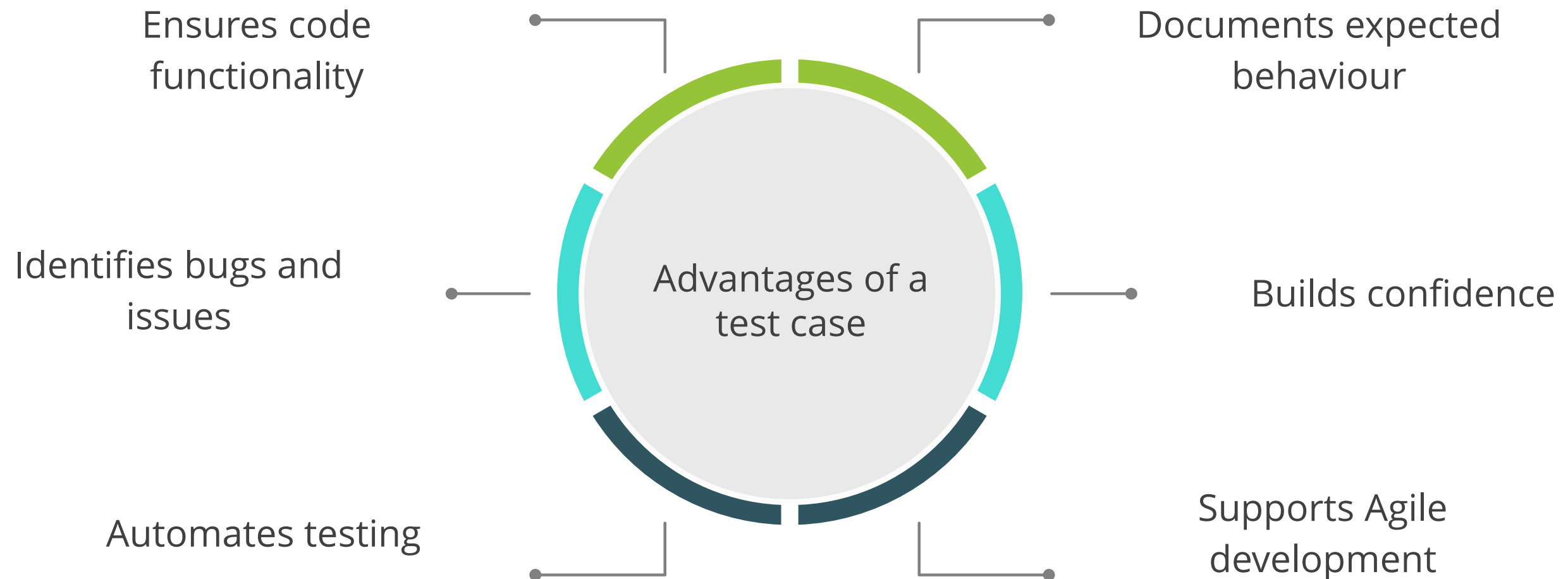
# JEST Test Case Types

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



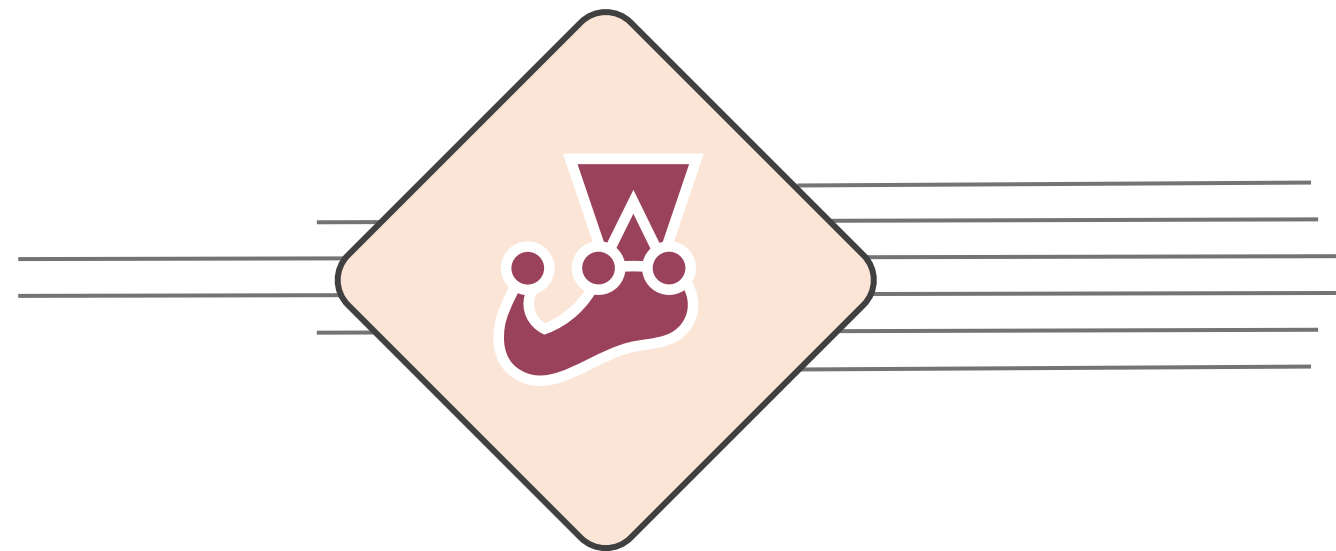
# Why Test Case?

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?

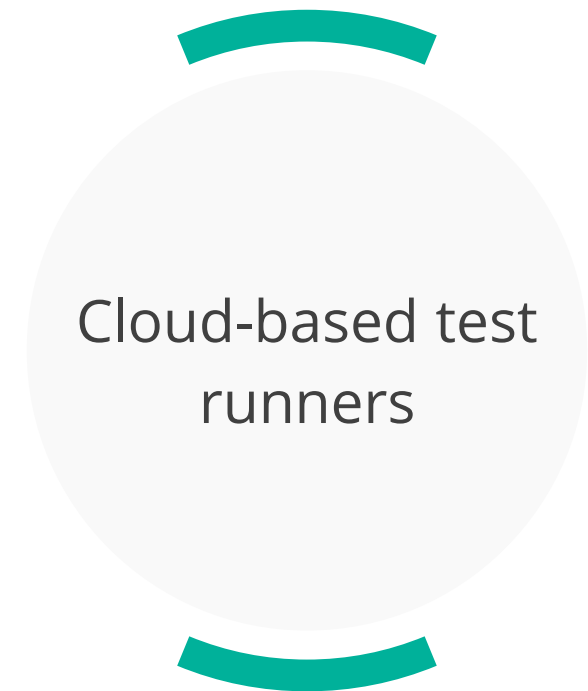
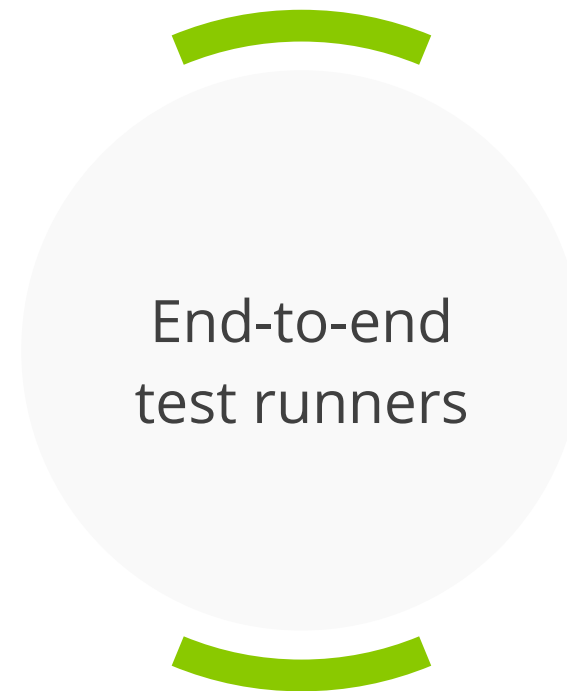
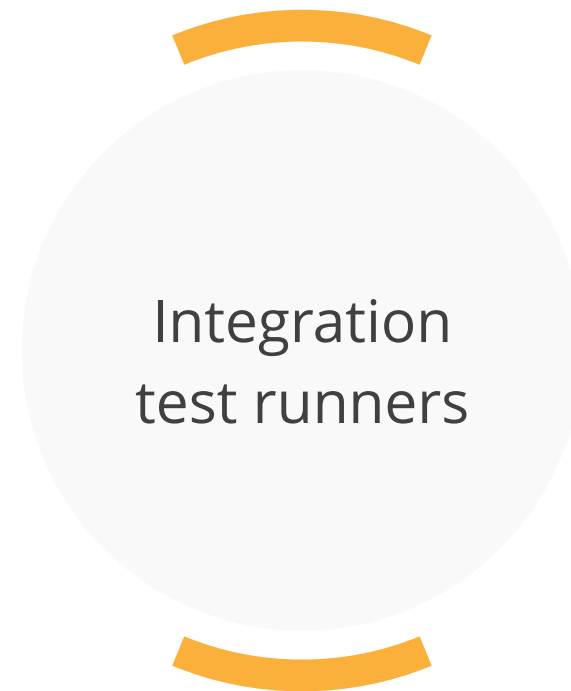
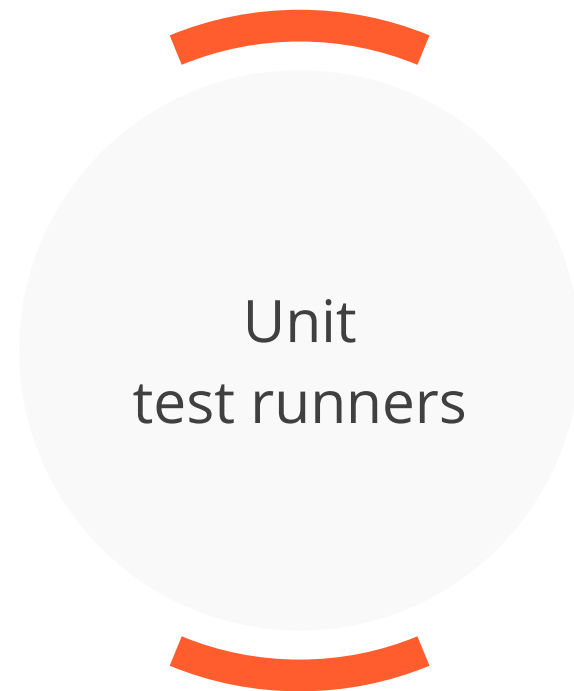
It is a tool used to run or execute tests and export results.



It is a library that selects the source code directory and picks the test files to run them to verify bugs and errors.

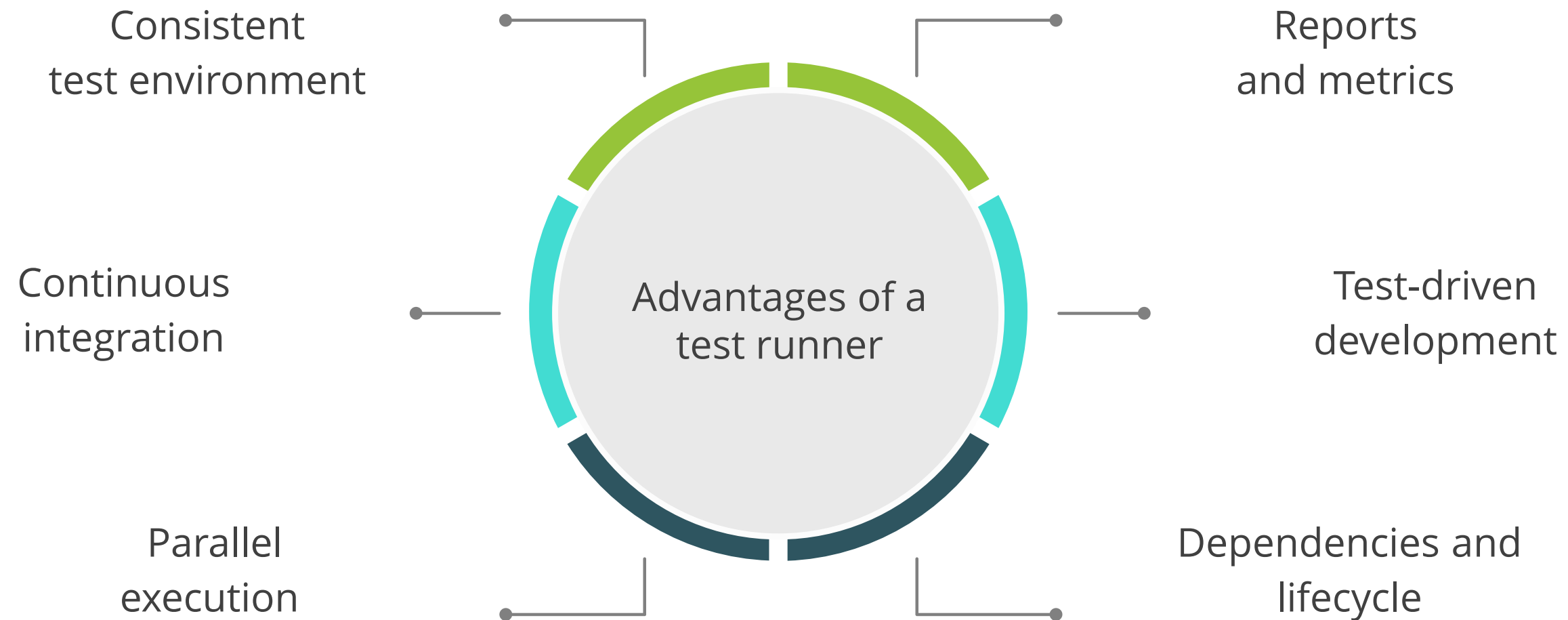
# JEST Test Runner Types

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



# Why Test Runner?

Here is a breakdown of why a test runner is important for picking the test files to run for verifying bugs and errors:



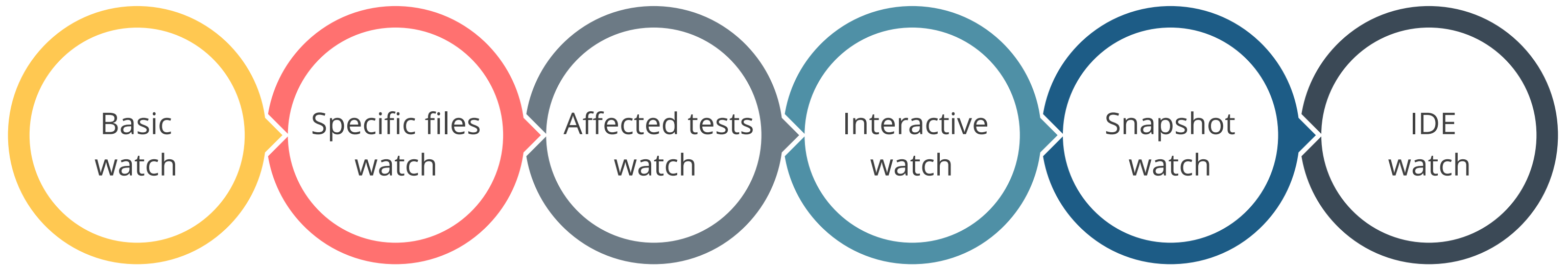
# What Is JEST Watch Mode?

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



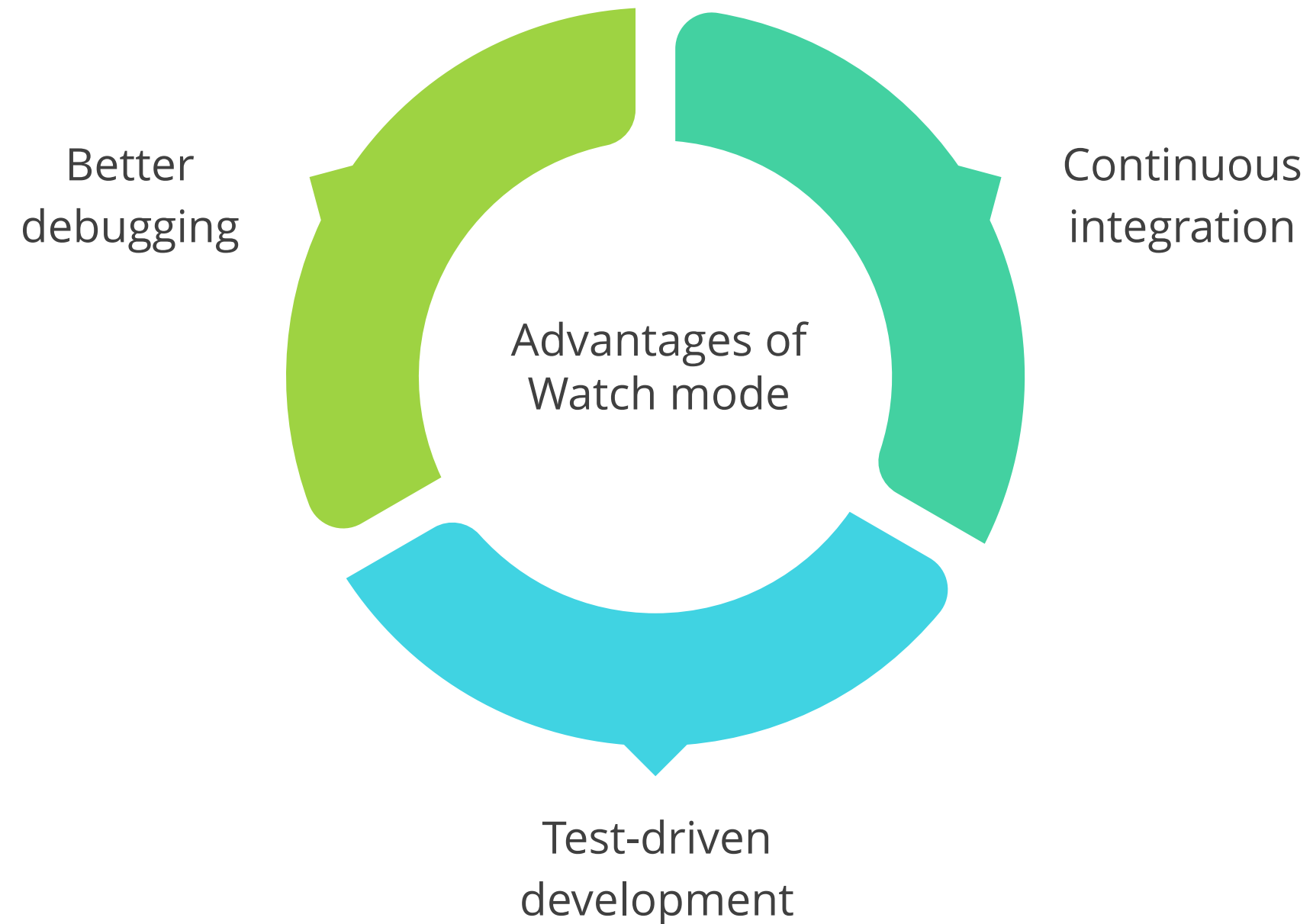
# JEST Watch Mode Types

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



# Why Watch Mode?

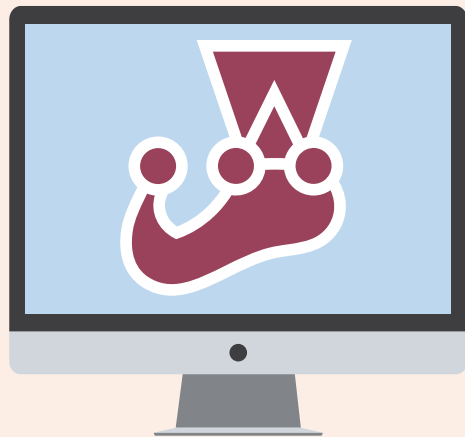
Here is a breakdown of why watch mode is important for rerunning the tests when it detects changes in the codebase:





# Working of Watch Mode

Watch mode operates by monitoring specific files and their dependencies, only running the relevant tests upon detecting changes.



- **File Monitoring:** It watches the changes in files related to your tests, including test files and the files they are testing.
- **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. Here's an overview of the most common types:



Test failures



Test coverage



Test results

# Why Reporting?

Here is a breakdown of why reporting is important for comprehensive summaries of test executions:



# 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
```



# **Introduction to JEST-DOM**

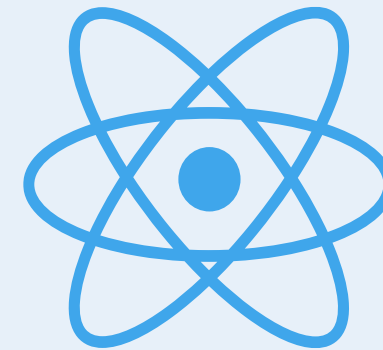


# What Is JEST-DOM?

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*



# Matchers In JEST-DOM

Matchers in JEST-DOM are specialized functions or methods provided by the JEST-DOM library.

Here are some of the common matchers available in JEST-DOM:

toBeInTheDocument()

toHaveStyle()

toHaveAttribute()



toHaveTextContent()

toHaveClass()

toContainElement()

# Matchers In JEST-DOM

Some of the common matchers available in JEST-DOM:

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():** Verify an element contains its descendant or not

# Applications of JEST-DOM

Here are some common applications of JEST-DOM:

React component testing

End-to-end testing

Debugging

User interface testing

# Examples of JEST-DOM

Some practical examples of using JEST-DOM matchers and utilities:

- 1 To verify 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 and utilities in verifying element's presence and to disable elements:

Verifying 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 and utilities in testing content and checking CSS class:

Testing content

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

Checking CSS class

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

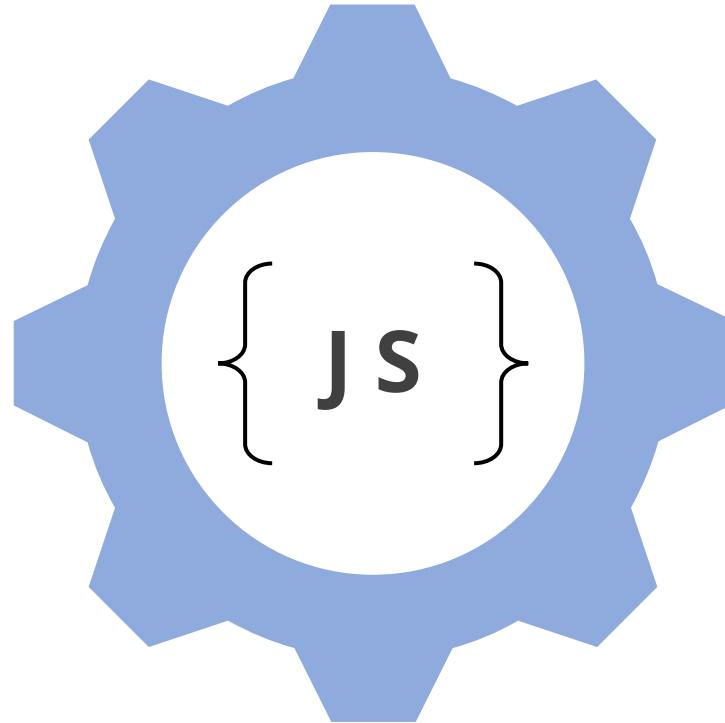




# DOM Testing Library

# What Is DOM Testing Library?

The DOM testing library is a JavaScript testing library that provides a set of 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 is essential for various purposes in web development:

1. User centric testing

2. Accessibility testing

3. Explicit waiting

4. Simplifying testing

5. Helper tool testing

6. Reliable tests

# 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.

Here are some common querying methods:



# Querying Methods

Some querying methods are as follow:

Methods	Explanation
getBy	These methods select elements that are expected to be in the DOM.
queryBy	These methods are like the getBy method but return null if the element is not found instead of throwing an error.
getAllBy	These methods are used to select multiple elements that match the criteria. They return an array of matching elements.
findAllBy	Similar to the getAllBy method, these methods return a promise that resolves to an array of matching elements.

# Querying Methods

Some querying methods are:

getBy method

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

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

queryBy method

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

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

# Querying Methods

Some querying methods are:

getAll method

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

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

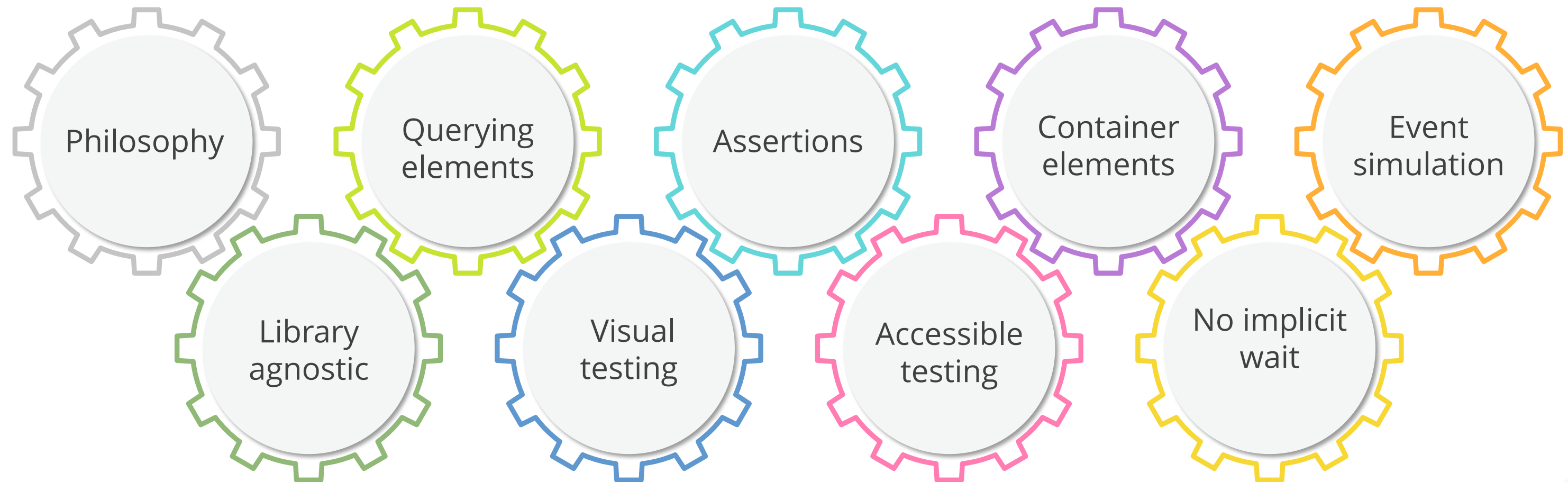
findAllBy method

```
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

Here are some of the key features and aspects of DOM testing:





# Assisted Practice



## DOM Testing

Duration: 20 Min.

### Problem Statement:

You have been assigned a task to demonstrate the implementation of a DOM (document object model) testing example using Jest.

# Assisted Practice: Guidelines



Steps to be followed:

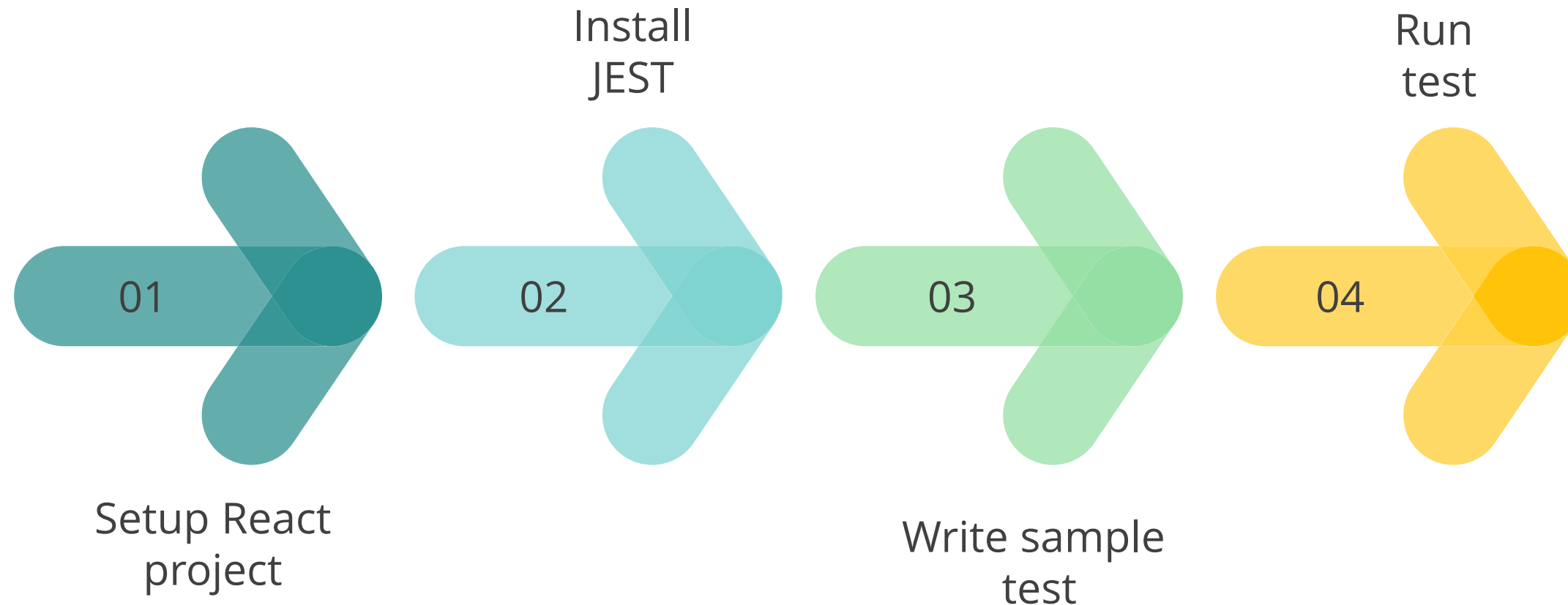
1. Create a Node project
2. Create an HTML file
3. Create a JavaScript file
4. Write a Jest test
5. Run the Jest test



# React Testing Library

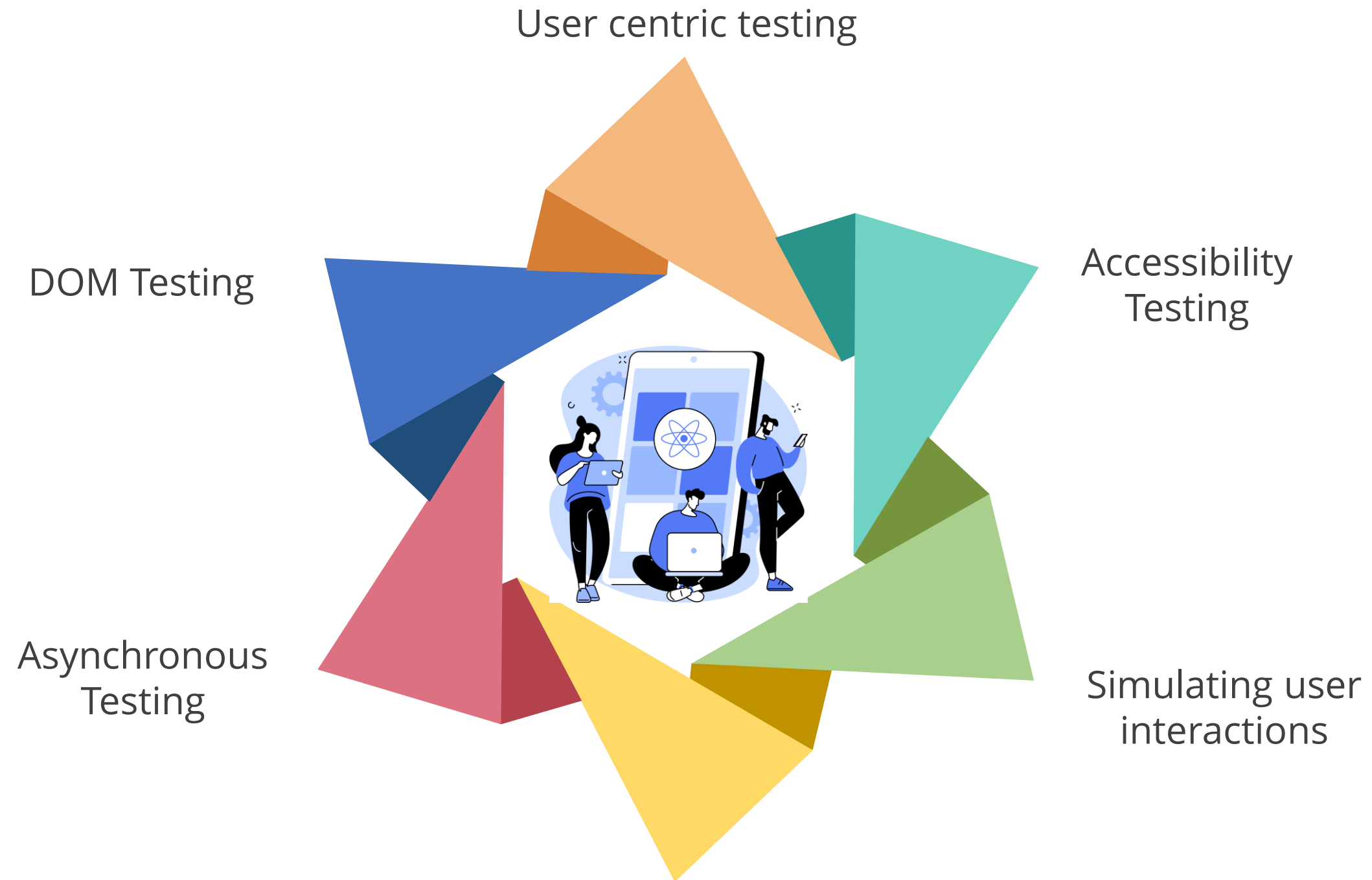
# React Testing Using JEST

Testing React applications using JEST is a common practice in the world of JavaScript development.



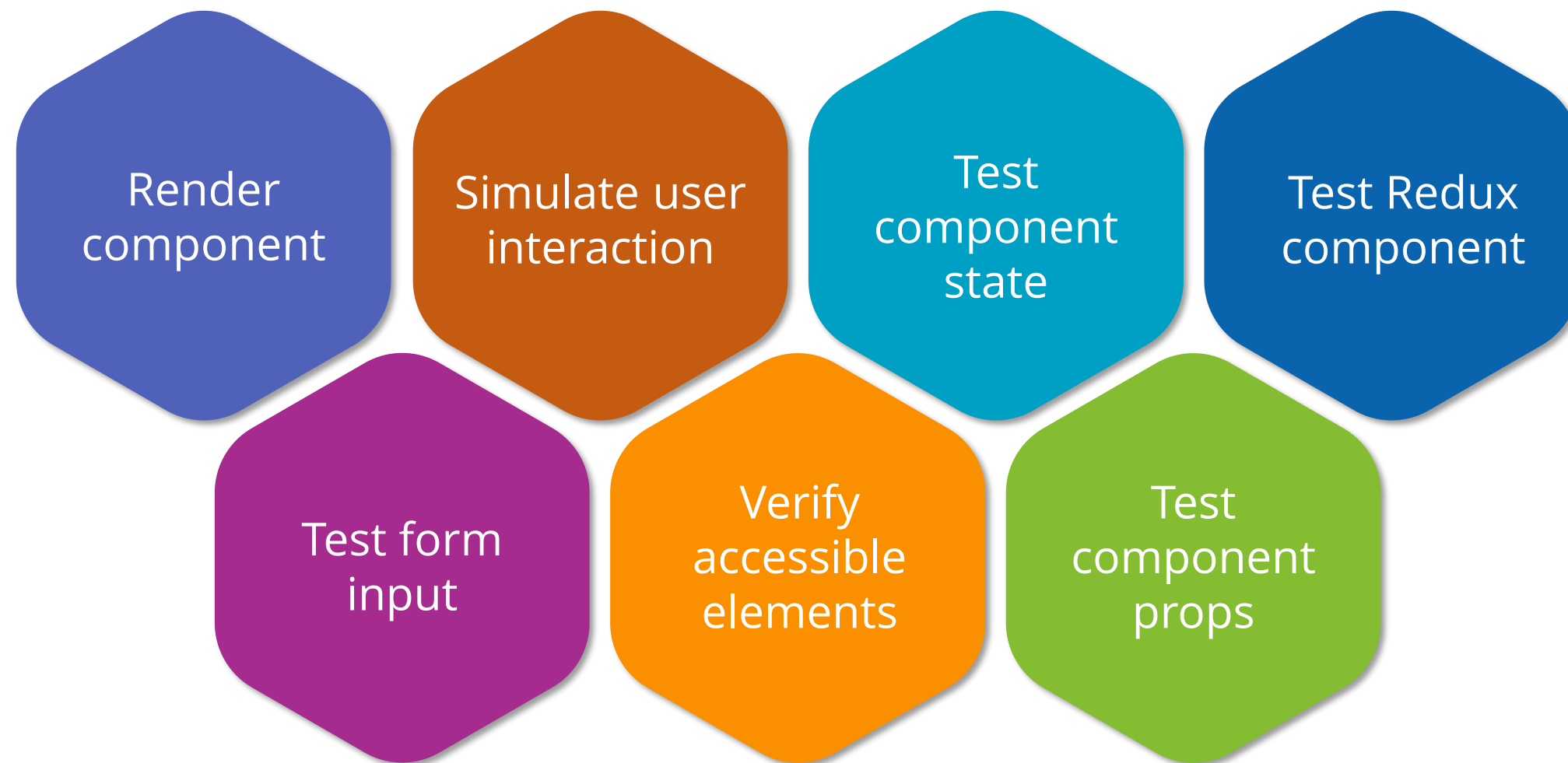
JEST is a popular JavaScript testing framework developed by Facebook, and it's often used in combination with tools like the React testing library.

# Features of React Testing Library



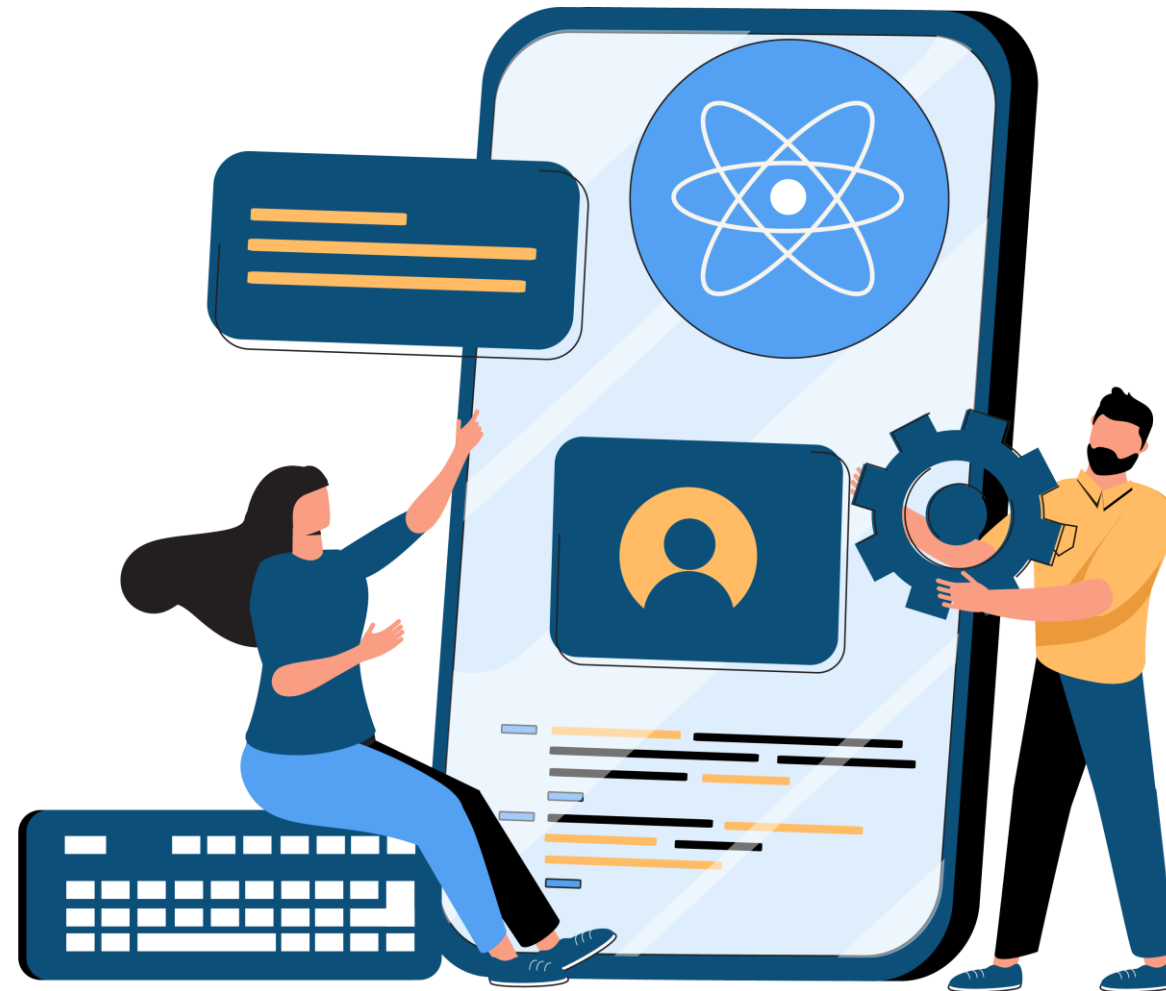
# Examples of React Testing Library

Here are some examples of how the React testing library can be used to test various aspects of React components:



# What Is Testing User Interaction?

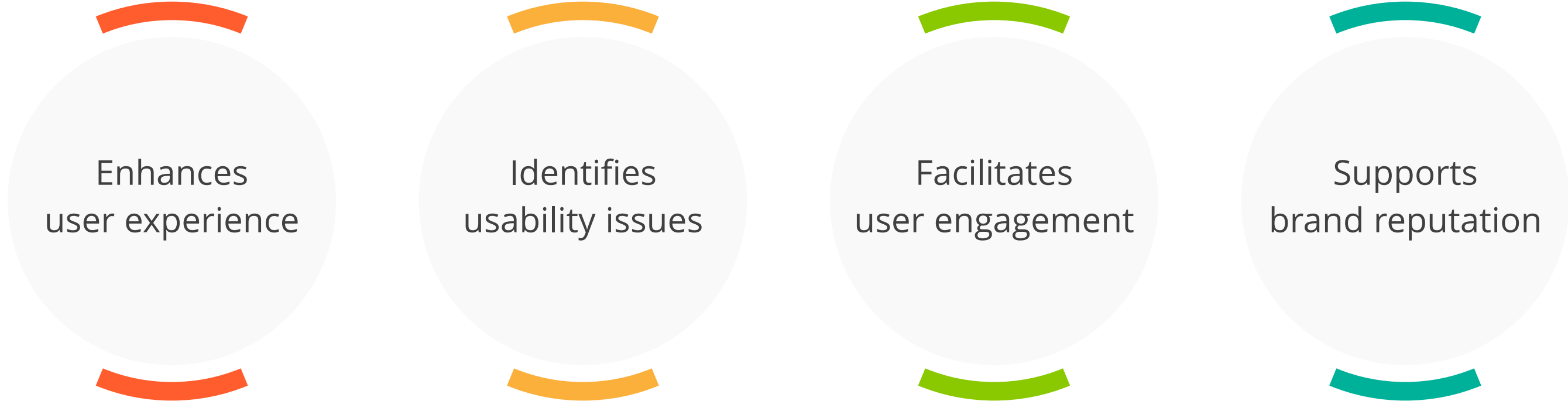
It refers to the process of evaluating and verifying that the interface elements of a software application or a website function correctly from the user's perspective.



# Why Testing User Interaction?

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

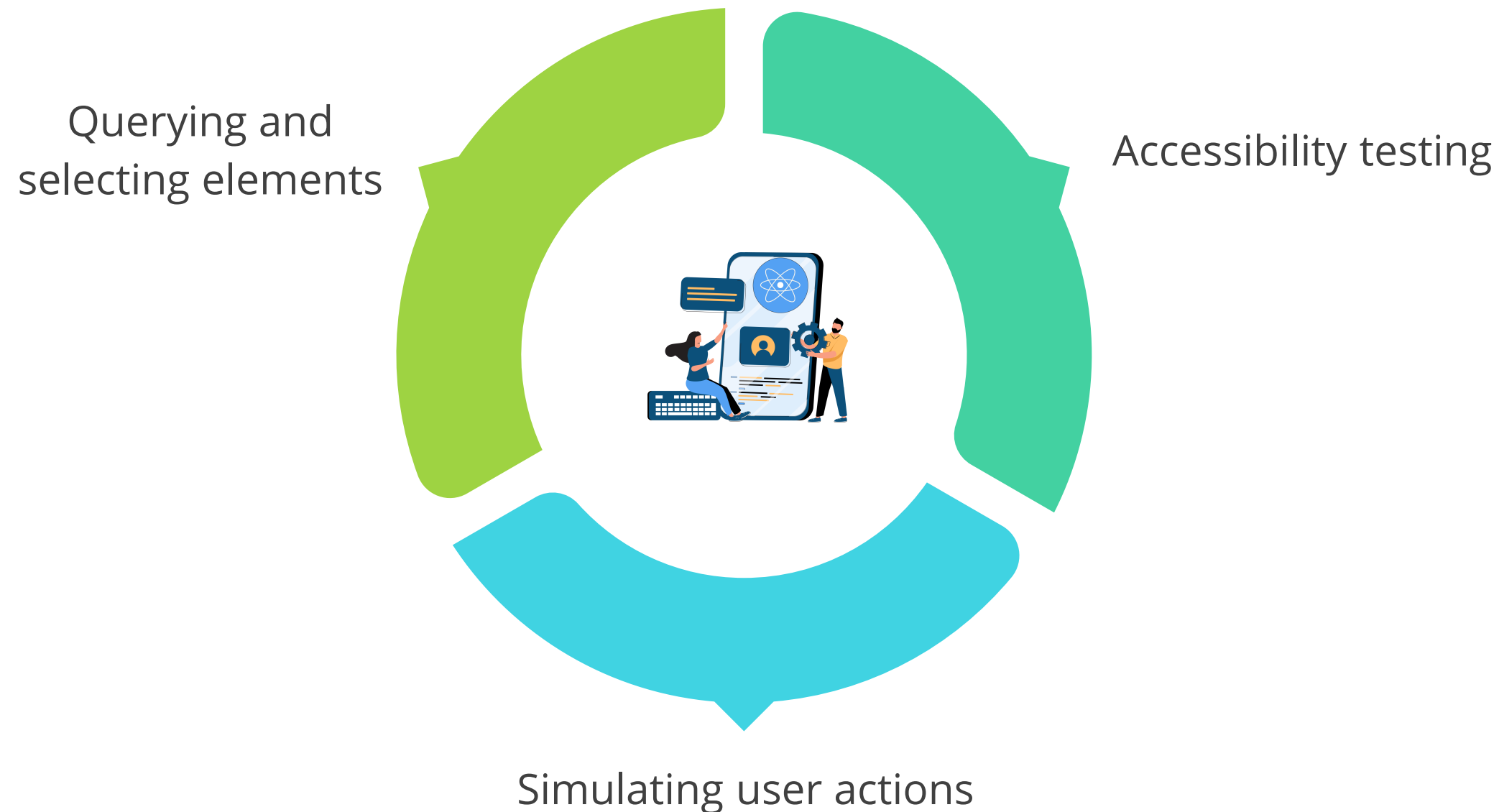
Facilitates  
user engagement

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:



# Example of Testing User Interaction

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>  
  );  
}
```

# Example of Testing User Interaction

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 Snapshot Testing?

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

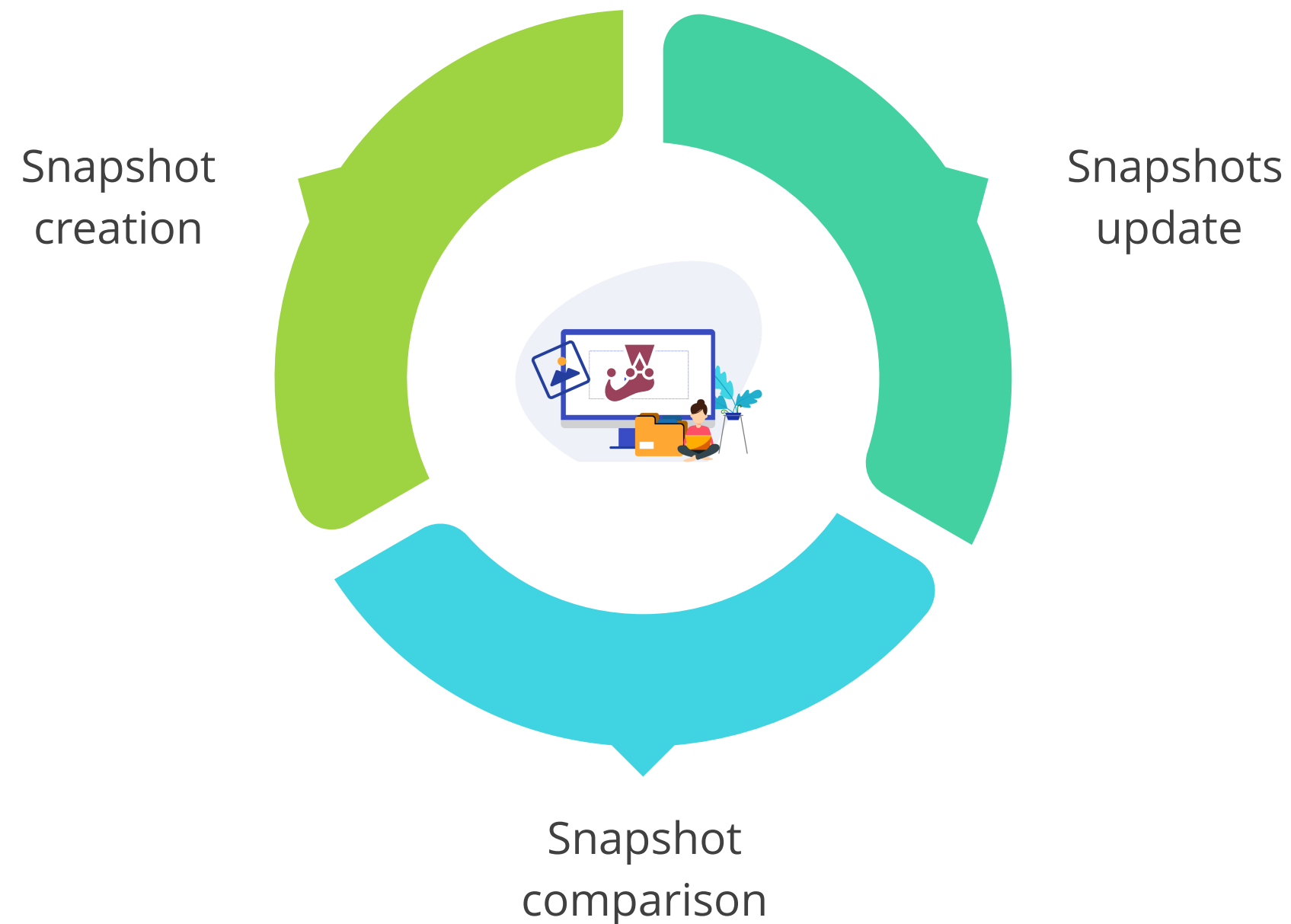
Facilitating refactoring

Integrating version control

Efficiency testing

# Concepts of Snapshot Testing

By leveraging the following concepts, snapshot testing can streamline testing processes, ensuring that changes made to the codebase are intentional and don't introduce unexpected alterations to the output:



# Example of Snapshot Testing

Here is an example of testing UI components with JEST:

## UI component:

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

# Example of Snapshot Testing

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



## Component Testing Using JEST

Duration: 20 Min.

### Problem Statement:

You have been assigned a task to demonstrate the implementation of react component testing 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
4. Configure the Jest
5. Run the Jest test

# 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**