|  |  |
| --- | --- |
| FRAMEWORK | |
|  | |
| May 14, 2025 | RAVI SHANKARTP19 |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  | INTRODUCTIONIn automation testing, a **framework** is a structured set of guidelines and tools that help create, organize, and execute automated test cases efficiently. It provides a consistent foundation by defining coding standards, folder structure, reusable components, test data management, reporting mechanisms, and integration with tools like TestNG. The main goal of a framework is to make test automation scalable, maintainable, and reliable while reducing redundancy and improving code reusability. By following a framework, teams can ensure better collaboration, faster test development, and easier debugging and maintenance of test scripts. | |  |
|  | Use case | |  |
|  | We use a **framework in automation testing** to bring **structure, efficiency, and maintainability** to the testing process. Without a framework, test scripts can become messy, repetitive, hard to manage, and difficult to scale. A framework provides **standardization** by enforcing best practices, **reusability** by allowing shared code and components, and **separation of concerns** by organizing test logic, test data, and configurations separately. It also helps with **reporting, logging, error handling, and integration** with tools like Jenkins for CI/CD pipelines. Ultimately, a well-designed framework saves time, reduces manual effort, makes debugging easier, and improves test coverage and quality making it essential for any serious or long-term automation effort. | * You use a **modular Page Object Model (POM)** to reuse code across tests. * **Test data** is stored in separate Excel or JSON files (Data-Driven Testing). * **Logs** and **HTML reports** are automatically generated after execution. * You execute tests across multiple browsers using **configuration files**. * Tests are triggered automatically via Jenkins on every code push. * You can easily add, update, or remove tests without breaking others. |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  | Components Framework is collection of reuseable components that makes automation development execution and modification easier and faster. List of components  * **Test Scripts** * **Test Data Management** * **Configuration data** * **Object Repository** * **Environment Configuration** * **Generic Libraries/Utilities** * **HTML Reports** * **Screen Shots** * **Test Runner/Execution Engine** | The working flow of an automation testing framework begins when tests are triggered either manually or automatically through CI/CD tools like Jenkins. The framework first loads configuration settings such as browser type, application URL, and environment details from external configuration files. Based on these settings, it initializes the WebDriver and launches the specified browser. As the test execution starts, scripts interact with the application using web elements and methods defined in Page Object classes, Generic Utility and test data is fetched from external sources. During the test run, assertions are made to validate expected outcomes. Logging captures each step, and screenshots are taken for failures. Once the tests complete, detailed reports are generated showing the results, including pass/fail status, execution time, and captured screenshots. Finally, the browser is closed, and cleanup tasks are performed. |  |
|  |  | |  |

### Framework Types

Choosing the right type of automation testing framework depends on several key factors such as the nature of the application, team skills, testing goals, and long-term maintenance needs. For instance, if you're testing a web application and your team has good coding skills, a Page Object Model (POM) combined with a Data-Driven or Hybrid framework is a solid choice. If you need to run the same test logic with multiple sets of data, a Data-Driven framework is ideal. For teams that include non-technical members or business analysts, a Keyword-Driven or Behaviour-Driven Development (BDD) framework like Cucumber allows test cases to be written in plain English. In most real-world scenarios, a Hybrid framework is preferred because it blends the strengths of multiple approaches such as POM for maintainability, Data-Driven for flexibility, and BDD for readability. Additionally, the framework should align with your tools (e.g., Selenium, TestNG, Jenkins) and support features like CI/CD integration, reusable utilities, and externalized configurations for scalability and ease of maintenance.

### Types of Framework

* **Data-Driven Framework**
  + Test data is separated from test scripts.
  + Data is stored in external files (Excel, CSV, JSON, DB, etc.).
  + One script runs with multiple sets of data.
* **Modular Testing Framework**
  + Application is divided into small, independent modules.
  + Each module is tested with separate scripts and later combined.
  + Promotes reusability and maintainability.
* **Keyword-Driven Framework**
  + Actions are defined by keywords (like Click, Enter, Select) in an external file.
  + Testers with little or no programming knowledge can write tests.
  + Often combined with Data-Driven for flexibility.
* **Hybrid Framework**
  + Combines two or more types (POM, Data-Driven, Keyword-Driven).
  + Most practical and scalable approach in real-world projects.
* **Behaviour-Driven Development (BDD) Framework**
  + Tests are written in plain English using tools like Cucumber or SpecFlow.
  + Makes test cases readable and understandable by non-technical stakeholders.
  + Encourages collaboration between QA, developers, and business teams.

### Data Driven Framework

A **Data-Driven Framework** is a type of automation testing framework where test data is separated from the test scripts and stored in external data sources like Excel, CSV, XML, JSON, or databases. Instead of hardcoding input values and expected results inside the test scripts, the framework reads the data from these external files and runs the same test logic with multiple data sets. This helps in testing the application under various conditions without duplicating the test code.

### Use case

**Multiple Sets of Input Are Needed**

* When the same test logic (e.g., login, form submission, search) needs to be tested with many different input values. Example: Testing a registration form with 20 sets of user data.

**Avoid Duplicating Test Scripts**

* Instead of writing separate test cases for each data combination, one script can run with all data sets, making the code clean and maintainable.

**Maintain Data Separately from Code**

* If your testing requires frequent changes in test data, keeping it in external files (Excel, JSON, DB) allows updates without modifying the test scripts.

**Performing Negative and Positive Testing**

* It’s ideal when you want to run the same test with both valid and invalid data to verify all possible outcomes.

**Testing Business Logic Heavily Dependent on Input Variations**

* For example, in financial, insurance, or healthcare systems, where small changes in input produce different outcomes, data-driven testing becomes essential.

**Improve Coverage and Reduce Manual Effort**

* Automating a large range of data combinations ensures better coverage with less manual effort.

### Modular Driven Framework

A **Modular Driven Framework** is an automation testing framework where the application under test is divided into **multiple small, independent modules**, and each module has its own test script. These individual test modules are later combined to create larger, end-to-end test scenarios.

The core idea is **reusability** you write reusable scripts for each part (module) of the application and call them as needed in different test cases. This approach makes the framework **organized, maintainable, and scalable**.

A **Modular-Driven Framework** is an automation testing framework where the application under test is divided into **multiple small, independent modules**, and each module has its own test script. These individual test modules are later combined to create larger, end-to-end test scenarios.

The core idea is **reusability** you write reusable scripts for each part (module) of the application and call them as needed in different test cases. This approach makes the framework **organized, maintainable, and scalable**.

### Use case

**Application Has Repetitive Functional Blocks**

* If certain actions (like login, search, checkout) are used across multiple test cases, creating separate modules for each allows you to reuse them instead of rewriting code.

**High Maintainability**

* When application changes affect only one part of the flow, you only need to update the corresponding module not every test case that uses it.

**Scalable Framework**

* Modular design makes it easier to scale test coverage. As new features are added to the application, you can create new modules without disrupting the existing ones.

**Large or Complex Application**

* In enterprise applications with multiple functionalities, modular testing keeps the test framework organized and easier to manage.

**Team Collaboration**

* Different team members can work on different modules simultaneously, promoting parallel development and faster delivery.

### Keyword Driven Framework

A **Keyword-Driven Framework** is an automation testing framework where test cases are written using **keywords** that represent actions to be performed on the application. These keywords are mapped to functions or methods in the automation code, and the actual test logic is separated from the test data and test steps.

The idea is to make test scripts more **readable**, **reusable**, and **accessible**—especially for **non-programmers**.

### Use case

**Testers Have Limited Programming Knowledge**

* If your team includes manual testers or business analysts who aren’t comfortable with code, keyword-driven testing allows them to write and execute tests using simple, predefined keywords in tools like Excel.

**Clear Separation Between Test Logic and Code**

* This framework separates:
  + Test data
  + Test steps
  + Test code

This improves maintainability and allows changes in one area without affecting others.

**Reusability Across Test Cases Is a Priority**

* Keywords like Login, ClickButton, EnterText, etc., are reusable across many test cases, reducing duplication.

**Tests Need to Be Easily Understood by Stakeholders**

* Keyword-driven test cases are written in a tabular format (often Excel) and can be easily read or reviewed by non-technical stakeholders.

**Medium to Large Projects With Repetitive Actions**

* If the application has many common user actions across modules, using keywords ensures consistency and reduces the time to create new test cases.

### Hybrid Driven Framework

A **Hybrid Testing Framework** is a combination of multiple testing frameworks to leverage the strengths of each. It integrates different types of frameworks like **Data-Driven**, **Keyword-Driven**, and **Modular** frameworks, enabling you to build a flexible, maintainable, and scalable automation test suite.

In a Hybrid framework, you typically combine the **modular structure** of **Modular-Driven Frameworks**, the **data separation** of **Data-Driven Frameworks**, and the **readability** and **reuse** of **Keyword-Driven Frameworks**.

### Use case

**Application is Large and Complex**

* If the application you're testing has many modules, functionalities, and test scenarios, a Hybrid framework helps keep the tests organized and scalable. By breaking the test cases into reusable modules and leveraging data-driven testing, you can efficiently handle large, complex systems.

**Flexibility in Test Design**

* A Hybrid framework allows you to mix and match different testing approaches based on the needs of each test case. For example, you can use:
  + Modular testing for core functionality (like login, checkout, etc.).
  + Data-Driven testing for validating different sets of inputs.
  + Keyword-Driven testing for creating tests that non-technical users can understand and write.
* This flexibility is valuable when the testing requirements change frequently.

**Large Test Suite**

* If you have many repetitive test cases that need to be executed with different data sets or on different configurations, a Hybrid framework’s reusability and data separation features make it easier to manage and execute these tests without rewriting code.

**Maintenance and Scalability**

* Hybrid frameworks offer ease of maintenance because:
  + Changes to data (in data-driven tests) or test steps (in modular tests) are isolated.
  + It’s easier to update test cases by simply modifying modules or keywords, without affecting the entire test suite.

**Cross-Functional Testing**

* If your application spans multiple platforms (web, mobile, API) or requires different testing methodologies (functional, performance, regression), a Hybrid framework allows you to combine tests for all these aspects within a single framework.

### Behavior Driven Framework

A Behavior-Driven Development (BDD) Framework is an automation testing framework that focuses on testing the behavior of an application from the end-user’s perspective, using natural language to describe test scenarios. It bridges the gap between technical teams (developers/testers) and non-technical stakeholders (business analysts, product owners) by allowing them to collaborate through human-readable test cases.

### Use case

**Requirements Need to Be Clear for All Stakeholders**

* If product owners, QA, and developers all need to understand and agree on how a feature should behave, BDD helps by using plain English (Gherkin) to describe expected behaviour. This avoids misunderstandings and misinterpretation of requirements.

**Align Development With Business Goals**

* BDD encourages writing test cases based on real business scenarios, ensuring the application behaves the way end-users expect. This keeps development focused on delivering real value.

**Shared Test Language**

* In Agile teams with cross-functional members, BDD provides a common language so everyone from testers to clients can understand and review test cases.

**Test-First or Test-Driven Development**

* If your team follows TDD, BDD is a natural extension that makes tests more expressive and behaviour-focused. It helps define expected outcomes before writing code.

**Complex User Journeys or Flows**

* + BDD is great for defining and testing end-to-end scenarios like:
    - User logins
    - Checkout flows
    - Form submissions
    - Role-based access controls
  + These behaviours are easier to describe and maintain in Gherkin syntax.

**Manual Test Cases Already Exist in Descriptive Format**

* If your manual test cases are written in step-by-step format, converting them to BDD scenarios is straightforward. This can help automate them while preserving readability.

### Framework Architecture

"Framework architecture" refers to the **structured design** and **organization** of the components within a software testing or development framework.

### Components

**Base Layer**

* **DriverManager**: Handles WebDriver initialization (Chrome, Firefox, etc.).
* **BaseTest**: Setup and teardown of WebDriver, reusable test config.
* **ConfigManager**: Reads from properties or JSON files.

**Page Object Layer POM**

* **Page Classes**: One class per page/component. Contains:
* WebElement locators (@FindBy||@FindBys||@FindAll )
* Page actions/methods (login(),logout(), clickSubmit())

**Utilities layer**

* **ExcelUtils**: Read/write Excel for data-driven testing.
* **WaitUtils**: Fluent, Explicit, and Implicit wait abstractions.
* **WebDriverUtils**: To Perform webdriver actions.
* **LoggerUtils**: Log4j2 or SLF4J-based logging utilities.
* **JsonUtils / PropertyUtils**: Config and test data reading.

**Test Data Layer**

* **testdata/** folder for JSON & Excel.
* Could also integrate with DBs, APIs, or Faker libraries.

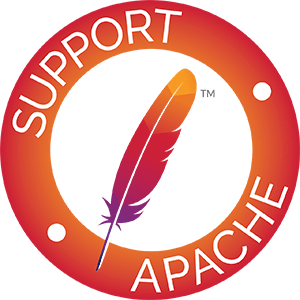
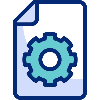
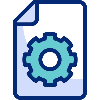
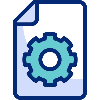
**Test Layer**

* **Test Classes**: Actual test cases written using TestNG.
* **Tags/Groups**: Organize tests by smoke, regression, end to end in xml suite runner.
* **Parallel/Cross Browser**: Xml suites coded for parallel and cross browser tests.
* **DataProviders**: For parameterized testing using Excel.

**Assertions & Reporting layer**

* **AssertionUtils**: Soft/hard assertion wrappers.
* **Listener Implementation** class.
* **Reporting**:
  + **Extent Reports** or **Allure Reports** integration.
  + Screenshot attachments.
  + Test step logging.

### Framework Architecture Diagram



Test Out

Advanced Reports

Html Reports

Screen Shots

Configuration Data

Test Script data

Business utility

Business Actions

Base Class

Listener Implement

Contant Interface

ListenerUtil

JavaUtil

DatabaseUtil

WebDriverUtil

FileUtil

Generic utility

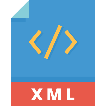
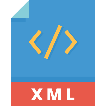
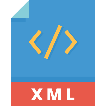
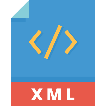
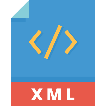
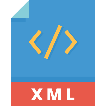


POM Repo

Pom.xml

Dependencies

Xml Test Suite



Test Scripts

### Framework Advantages

An **automation framework** offers numerous advantages that make test development, execution, and maintenance more **efficient, scalable, and reliable**. Here is a breakdown of the key benefits:

**Advantages of Using a Test Automation Framework**

* **Code Reusability**
* Common logic is written once and reused across tests.
* Utility classes, data readers, and page objects promote DRY (Don't Repeat Yourself) principles.

**Improved Test Structure & Maintenance**

* Clear separation of concerns:
  + Pages → Webelements are maintained seperately
  + Tests → Test Cases are separated from test data
  + Utils → Utilities Optimize the code
* Easier to debug, extend, and maintain tests over time.

**Consistent Reporting**

* Centralized reporting using tools like **Extent Reports** or **Allure**.
* Automatically capture **screenshots on failure**, **logs**, and **step-wise status**.

**Scalability**

* Supports adding more test cases, environments, or data sets without major rework.
* Easily integrated with CI/CD pipelines and parallel execution tools.

**CI/CD and DevOps Friendly**

* Frameworks can be triggered from Jenkins, GitHub Actions, or GitLab CI.
* Allows **regression suites**, and **smoke tests** to run automatically.

**Reduced Execution Time**

* **Parallel execution** support using TestNG.
* Reduces total run time of large test suites.

**Supports Multiple Test Data Sources**

* Easily integrates with Excel, JSON, databases, or APIs for **data-driven testing**.

**Team Collaboration**

* Standardized structure allows any team member to contribute, review, or onboard quickly.
* Better version control and teamwork using Git.

**Environment Config Management**

* Centralized configuration for browser, environment, base URL, credentials, etc.
* Promotes flexibility and environment switching (QA → UAT → PROD).

**Better Debugging and Logs**

* Logs via **Log4j**, **SLF4J**, or custom logger help in understanding failures.
* Combined with screenshots, it becomes easy to trace what went wrong.

**Extensibility**

* Easily integrate with:
* Selenium Grid for distributed testing.
* BrowserStack or Sauce Labs for cloud/browser coverage.
* REST-assured for API testing.
* Cucumber for BDD-style tests.

### Framework Implementation

### In our project, we implemented a custom automation framework to improve reusability, scalability, and maintainability of our test cases.

### Design

**Base Layer (Java Concept : Inheritance)**

* **Base Class** which is used to configure all the Test Scripts by inheriting the base class.
* **DriverManager** which creates a thread safe WebDriver variable on the call of different configuration inputs like browser type and login credentials.
* **ConfigarationManager** reads the data provided in the property file and gives input to the **DriverManager.**

**Page Object Layer POM (Java Concept : Encapsulation)**

* **Page Classes**: Designed for each webpage. Contains:
* WebElements which are encapsulated for safety and utilised by getters.
* All the Pages receive WebDriver instance in the constructor and initialize through PageFactory.inItElements(driver,this).
* This pattern is done in order to achieve lazy initialization.

**Utilities layer (Java Concept : Abstraction)**

* **ExcelUtils**: excel utility is designed to read data from excel and provide to the test scripts, receives path of the excel file and sheet name in order to initialize the utility.
* **WaitUtils**: Fluent, Explicit, and Implicit wait abstractions.
* **WebDriverUtils**: used to perform webdriver actions which on the application this class receives the WebDriver instance and initializes and provides functional methods for repeated actions.
* **LoggerUtils**: For logging the Status and Informations.
* **JsonUtils / PropertyUtils**: Helps in Reading the data from Property Files and JSON files.

**Test Data Layer**

* **Congiguration Data:** Stored in the Property files.
* **Test Script Data :** Stored in Excel Files

**Test Layer**

* **Test Classes**: Test Scripts extends Base Class and All the test cases are developed within and stored in Module Wise. – Modular Driven.
* **Tags/Groups**: TestNG.XML are developed for Smoke & Regression Testing.
* **Parallel/Cross Browser**: Xml suites coded for parallel and cross browser tests.
* **DataProviders**: For parameterized testing using Excel.

**Assertions & Reporting layer**

* **AssertionUtils**: for validation we used Assert class for main features we used Hard Assert and for minor features we used Soft Assert.
* **Listener Implementation** class : Listeners we configured to capture Important runtime events like onTestSuccess, onTestFailure, and onTestSkipped.
* **Reporting**:
  + We used Extent Report Advance reporting tool which configured in Listeners.
  + Screenshot attachments were attached to the Advanced Report.
  + Test step logging we done in order to get information on the executions helped in debugging.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Framework is a set of instructions followed by every organization that makes automation test engineers life easy.  An automation framework provides a structured and reusable foundation for creating, executing, and maintaining automated tests efficiently and consistently. |  |  |