Automation Framework

Introduction

The primary goal of the framework is to provide a standardized, structured, and simplified way to write, manage, and run automated tests. It aims to improve the quality of software by efficiently identifying and addressing issues.

The framework employs **Maven** as its build tool. Maven's project management capabilities offer a comprehensive understanding of the project and encourage a clear organization of project resources and dependencies.

For managing our tests, we've adopted **TestNG**, a testing framework that allows us to have a large amount of control over how tests are run. It provides features such as test categorization, parallel test execution, and prioritization, ensuring a structured and organized testing process.

In the realm of reporting, the framework leverages **Extent Reports**, a flexible tool that generates interactive and detailed reports of our tests. These reports provide critical insights into the status and outcomes of test cases, assisting in effective debugging and analysis when tests fail.

We use the **Log4J** logging utility to capture logs during the testing process. Log4J facilitates efficient debugging and tracking of issues, providing greater visibility into the testing process and ensuring robust and reliable software.

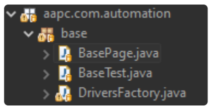
Tools and Libraries (What do we have in Pom.xml)

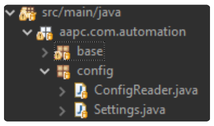
To bolster the framework's capabilities, several libraries and tools are utilized. Here's a brief overview of each and their role within the framework:

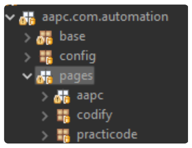
1. **Selenium**: It provides tools for browser automation, allowing us to simulate user interactions with a web application. We use version 4.9.0.
2. **Webdriver Manager (io.github.bonigarcia)**: This library helps to manage WebDriver, the key component in Selenium for driving a browser. It simplifies the setup process, ensuring the WebDriver is always up to date.
3. **Extent Reports (com.aventstack)**: This tool provides beautiful, interactive, and detailed reports for our tests. It allows us to see the status of test cases and helps in debugging when tests fail.
4. **Project Lombok (org.projectlombok)**: Lombok is a Java library that helps to reduce boilerplate code in our project, making the code cleaner and easier to read.
5. **TestNG (org.testng)**: A powerful testing framework that provides features such as test categorization, prioritization, and parallel test execution. It gives us a great deal of control over how tests are run.
6. **Log4J (org.apache.logging.log4j)**: This Java-based logging utility allows us to capture logs during test execution, making it easier to debug and track issues.
7. **JavaFaker (com.github.javafaker)**: This library is used to generate fake data (like names, addresses, etc.) that can be used during testing.
8. **Commons IO (commons-io)**: It provides utility classes for handling IO operations, like reading and writing to files, which is often needed during testing.
9. **JExcelApi (net.sourceforge.jexcelapi)**: This library enables reading and writing data from and to Excel files. It's useful for data-driven testing, where test data is stored in Excel files.
10. **Rest Assured (io.rest-assured)**: It simplifies the testing and validation of REST APIs. This library comes in handy when our tests need to interact with backend services.

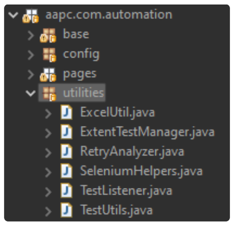
Architecture

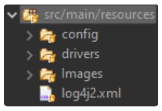
The architecture of the framework is designed to promote modularization, ease of use, and scalability. The structure and organization of the framework are mainly grouped into several key components, each with their specific responsibilities.

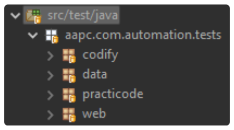
1. **Base Component**: The base component includes three essential classes: BasePage, BaseTest, and DriversFactory. The **BasePage** class contains common web page interactions used across different page classes. **BaseTest** serves as a parent class for all the test classes, and it sets up and tears down test execution. **DriversFactory** is responsible for managing WebDriver instances, ensuring a seamless interaction with the browser.


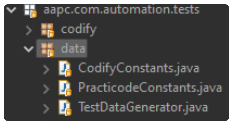
2. **Config Component**: This component has the **ConfigReader** and **Settings** class, which handle the reading of configuration details from the config.properties file and their distribution throughout the framework. It allows us to maintain environment-specific settings in a centralized manner.


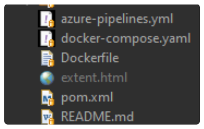
3. **Pages Component**: Here, each class represents a different page of the web application. They contain methods that perform actions on that particular page. This makes tests more readable and allows for code reuse across multiple test cases.


4. **Utilities Component**: This includes various utility classes such as **ExcelUtil**, **ExtentTestManagerClass**, **RetryAnalyzer**, **TestListener**, **TestUtils**, and **SeleniumHelpers**. These classes provide auxiliary functions like reading data from Excel files, managing test reports, implementing retry logic, listening to test events, general-purpose functions, and Selenium-specific helper functions.


5. **Resources**: This is where the config.properties and **log4j2.xml** files are stored. The config.properties file contains configuration details like base URLs for different environments, timeout values, etc. The log4j2.xml file holds the configuration for the logging done by Log4J.


6. **Test Folders (under src/test)**: These contain the actual test classes. They are organized into different folders like **codify**, **practicode**, and **web**, each representing different modules or areas of the web application. These tests use the resources from the pages and utilities components to interact with the application and validate its functionality.


7. **Data Folder (under src/test)**: This includes classes like **CodifyConstants**, **PracticodeConstants**, and **TestDataGenerator**, which provide test data and constants used in test classes.


8. **Project Root Level**: At the root level, we have the azure-pipelines.yml file for Azure Pipelines integration, extent.html for the test report, pom.xml for Maven configuration, and README.md to document the project.


Environment Setup

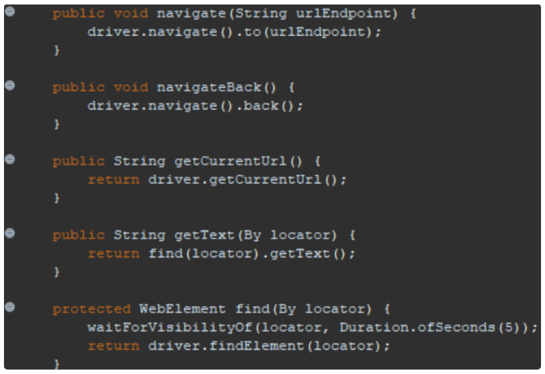
README file is available in project repository.

Folder Structure

Selenium auto… ork.pdf
02 Aug 2023, 04:57 PM

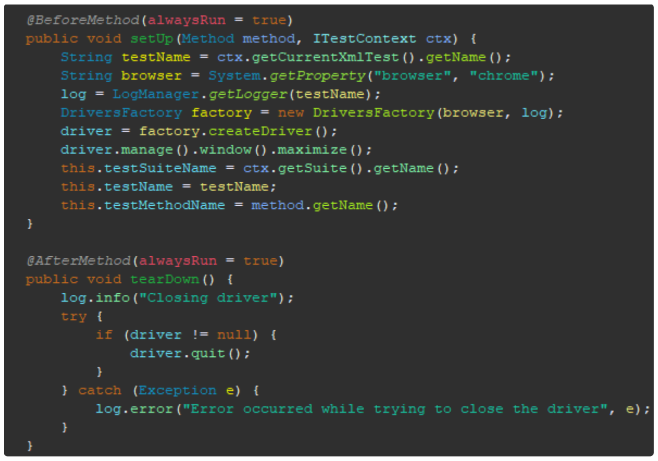
src/main

1. **Base Folder**:
   a. `BasePage`: The BasePage class is fundamental in the Page Object Model, as it includes all the common functions that can be performed on any page. This class includes functions such as navigating to a URL, fetching the current URL, fetching text from a webpage, finding a web element, clicking a web element, typing text, and many more. It also includes functions for handling alerts and dropdowns. Each function in this class is designed to interact with web elements in different ways, or to retrieve information from these elements.



```
    public void navigate(String urlEndpoint) {
        driver.navigate().to(urlEndpoint);
    }

    public void navigateBack() {
        driver.navigate().back();
    }

    public String getCurrentUrl() {
        return driver.getCurrentUrl();
    }

    public String getText(By locator) {
        return find(locator).getText();
    }

    protected WebElement find(By locator) {
        waitForVisibilityOf(locator, Duration.ofSeconds(5));
        return driver.findElement(locator);
    }
```

   b. `BaseTest`: This class acts as the base class for all the tests. The class includes setup and teardown methods that are run before and after each test. The setup method includes creating a driver instance, and the teardown method includes closing the driver. It also sets some basic information about the test such as the suite name, test name, and the method name. This class is the one to extend while creating new test classes.



```
@BeforeMethod(alwaysRun = true)
public void setUp(Method method, ITestContext ctx) {
    String testName = ctx.getCurrentXmlTest().getName();
    String browser = System.getProperty("browser", "chrome");
    log = LogManager.getLogger(testName);
    DriversFactory factory = new DriversFactory(browser, log);
    driver = factory.createDriver();
    driver.manage().window().maximize();
    this.testSuiteName = ctx.getSuite().getName();
    this.testName = testName;
    this.testMethodName = method.getName();
}

@AfterMethod(alwaysRun = true)
public void tearDown() {
    log.info("Closing driver");
    try {
        if (driver != null) {
            driver.quit();
        }
    } catch (Exception e) {
        log.error("Error occurred while trying to close the driver", e);
    }
}
```

   c. `DriversFactory`: This class is responsible for creating WebDriver instances for different browsers like Chrome, Firefox, and Edge. Depending on the browser specified, it creates the corresponding WebDriver instance. This class utilizes the WebDriverManager library to manage the WebDriver binaries for different browsers. The driver is created with or without headless options based on the configuration.

2. **Config Folder**:

   a. The `ConfigReader` class's primary responsibility is to read properties from a configuration file. This is achieved via the `readProperty()` method, which fetches configuration data from a properties file located at "src/main/resources/config/config.properties".

      The method `populateSettings()`, which is statically called in the `Settings` class, invokes `readProperty()`. The `populateSettings()` method initiates the reading of properties and appropriately handles any potential `IOExceptions`.

      The `readProperty()` method also retrieves the "branch" and "testEnv" system properties. Depending on their values, it determines the URL for the website under test. If no system property is provided, the default value "test" is used.

   b. The `Settings` class contains a series of static variables that hold the configuration details read by the `ConfigReader` class.

      These variables include paths to driver executables, screenshot details, URL endpoints, test user credentials, and other settings that might be needed across the test suite.

      The `Settings` class utilizes a static block, executed when the class is loaded, to call `ConfigReader.populateSettings()`. This ensures that configuration properties are loaded into the `Settings` class variables before they're needed by the test suite.

3. **Pages Folder**: `Page classes` in our framework represent individual pages of our web application under test. They're part of the Page Object Model (POM) design pattern implementation and provide a user-friendly interface to the test methods. In essence, they encapsulate the technicalities of interacting with the page, making test methods more readable and maintainable.
   a. **UI Map (Locators)**: They act as a repository for the locators of the significant elements on the page. This way, if an element's attributes change, you only need to update the locator in one place.
   b. **Services (Methods)**: They offer methods to interact with the elements on the page, such as clicking a button or entering text into a field. These methods use the defined locators and the WebDriver instance to perform actions.
4. **Utilities Folder**: The `utilities` folder in our framework contains utility classes that provide various helper methods and functionalities to assist in testing. These utility classes make our tests more robust and easier to maintain.
   a. **ExcelUtil**: This utility is used to read data from Excel files. This can be useful when you want to use data-driven testing, where your test data is stored in an Excel file.
   b. **ExtentTestManagerClass**: This class is responsible for managing the Extent Reports, which is a third-party library that creates beautiful, interactive, and detailed reports for our tests. The `ExtentTestManagerClass` initializes the reports and provides methods to update the status of the tests.
   c. **RetryAnalyzer**: The `RetryAnalyzer` class is used to implement the retry logic for failed test cases in TestNG. It overrides the `retry()` method of the `IRetryAnalyzer` interface. If a test fails, the `RetryAnalyzer` class will determine whether to re-run the test based on the maximum retry count.
   d. **TestListener**: The `TestListener` class implements the `ITestListener` interface and overrides its methods to create specific actions at different stages of the test execution (like test start, test success, test failure, etc.). It is used in conjunction with Extent Reports to provide more detailed reporting.
   e. **TestUtils**: The `TestUtils` class provides a set of utility methods that are used across tests. These include taking screenshots, generating random numbers, fetching browser logs, etc.
   f. **SeleniumHelpers**: This class would contain helper methods that use Selenium WebDriver to perform common actions, such as clicking a button, entering text, selecting from a dropdown, etc. (We are currently using all the selenium helpers in BasePage)
5. **Resources**: The `resources` directory in our framework contains files that are used to configure the behavior of our tests and the logging of test execution details.
   a. The `config.properties` file is a properties file where we can define various configuration options for our tests. In this file, you can define URLs for different environments like production and testing, as well as user credentials for logging into the application under test. You can also specify whether the browser should run in headless mode and the paths to your Chrome and Firefox WebDriver executables. To modify this file, just open it in a text editor and change the values of the properties you're interested in. The key-value pair structure is straightforward: the key is on the left side of the equals sign and the value is on the right.

src/test

1. This is where we write and create our test utilizing the page classes and utility libraries.

- **codify**

- **practicode**
- **web**
- **data**
    - **CodifyConstants**
    - **PracticodeConstant**
    - **TestDataGenerator** - ⭐ to generate various types of fake or randomly generated data. This can be useful in testing scenarios, where we need to simulate input but don't want to use real data (for privacy, legal reasons, or to avoid altering real data). It uses the `Faker` library, a popular Java library for generating fake data, to do this.

testsuites

1.This is where we have out TestNG.xml files

- codify.xml
- practicode.xml
- regression.xml
- smoke.xml

Project Root Level

1. extent.html - this is where we can see our test results after the test run
2. pom.xml - where we manage our dependencies and plugin
3. README.md

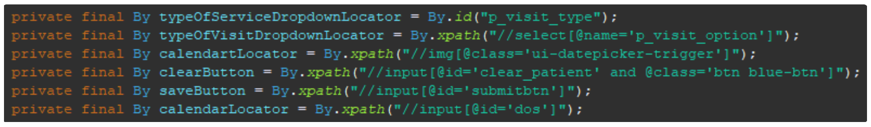
Create a New Page Class:

Components of a Page Class

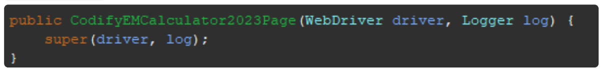
A page class typically consists of:

1. **Locators**: These are identifiers that are used to find and interact with the elements on the page.
2. **Constructor**: This method is used to initialize the WebDriver instance and any other required variables or services.
3. **Methods**: These are used to interact with the elements on the page, such as clicking buttons, entering text, and selecting from dropdowns. These methods are made public so they can be called from your test classes.

Creating a Page Class

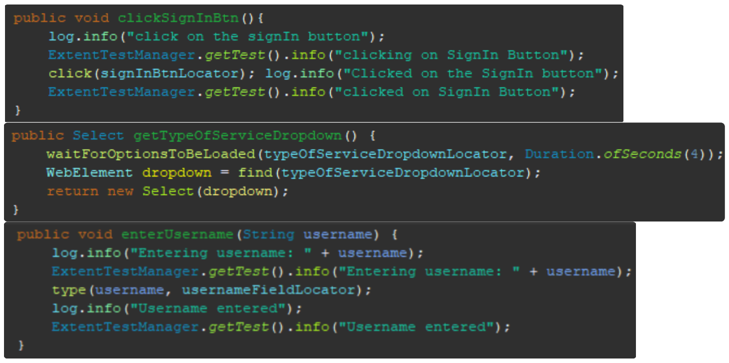
1. **Locators**: The locators are stored as private final By objects at the top of the class.


```
private final By typeOfServiceDropdownLocator = By.id("p_visit_type");
private final By typeOfVisitDropdownLocator = By.xpath("//select[@name='p_visit_option']");
private final By calendartLocator = By.xpath("//img[@class='ui-datepicker-trigger']");
private final By clearButton = By.xpath("//input[@id='clear_patient' and @class='btn blue-btn']");
private final By saveButton = By.xpath("//input[@id='submitbtn']");
private final By calendarLocator = By.xpath("//input[@id='dos']");
```

2. **Constructor**: The constructor takes a WebDriver and a Logger instance as arguments. It calls the constructor of the superclass `BasePage` with these arguments.


```
public CodifyEMCalculator2023Page(WebDriver driver, Logger log) {
    super(driver, log);
}
```

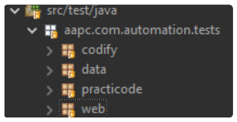
3. **Methods**: class has several methods that interact with the page:



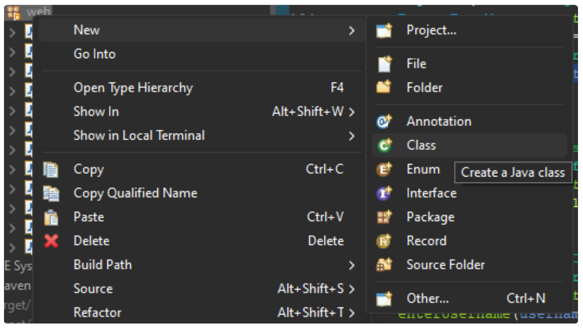
```java
public void clickSignInBtn(){
    log.info("click on the signIn button");
    ExtentTestManager.getTest().info("clicking on SignIn Button");
    click(signInBtnLocator); log.info("Clicked on the SignIn button");
    ExtentTestManager.getTest().info("clicked on SignIn Button");
}
public Select getTypeOfServiceDropdown() {
    waitForOptionsToBeLoaded(typeOfServiceDropdownLocator, Duration.ofSeconds(4));
    WebElement dropdown = find(typeOfServiceDropdownLocator);
    return new Select(dropdown);
}
 public void enterUsername(String username) {
     log.info("Entering username: " + username);
     ExtentTestManager.getTest().info("Entering username: " + username);
     type(username, usernameFieldLocator);
     log.info("Username entered");
     ExtentTestManager.getTest().info("Username entered");
 }
```

Create a New Test Class:

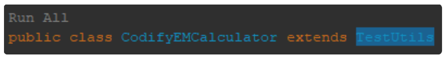
- Inside the 'src/test' directory, you can find different folders representing different modules or areas of the application (e.g., 'codify', 'practicode', 'web'). Navigate to the appropriate folder where you want to add your test.



- Create a new Java class. This will be your test class. Please follow the naming convention 'XYZTest' where 'XYZ' is the name of the feature or functionality you are testing.

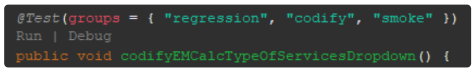


- Make sure your class extends the 'TestUtils' class, so it can utilize the setup and teardown methods.



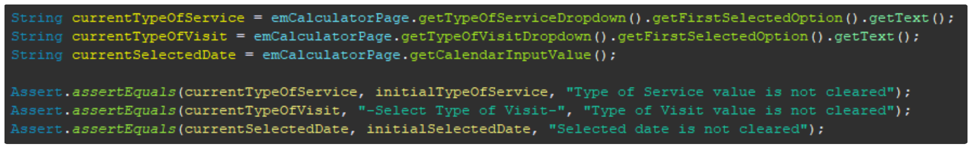
```java
Run All
public class CodifyEMCalculator extends TestUtils
```

1. **Write Test Methods:**
   - Within your test class, create a method for each scenario you want to test. The naming convention should be 'testXYZ' where 'XYZ' is a brief description of the test scenario.



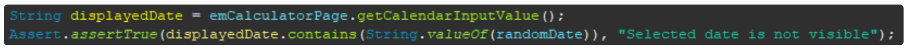
```java
@Test(groups = { "regression", "codify", "smoke" })
Run | Debug
public void codifyEMCalcTypeOfServicesDropdown() {
```

   - Each test method should be annotated with '@Test', a TestNG annotation that denotes a test case.
   - Optionally, you can include other TestNG annotations like '@DataProvider', as per your test requirements.
   - Use assertions to validate your test's outcome. TestNG's 'Assert' class provides a set of assertion methods useful in writing tests.
2. **Write Assertions:**
   - Assertions are used to compare the expected and actual results in automation testing.



```java
String currentTypeOfService = emCalculatorPage.getTypeOfServiceDropdown().getFirstSelectedOption().getText();
String currentTypeOfVisit = emCalculatorPage.getTypeOfVisitDropdown().getFirstSelectedOption().getText();
String currentSelectedDate = emCalculatorPage.getCalendarInputValue();

Assert.assertEquals(currentTypeOfService, initialTypeOfService, "Type of Service value is not cleared");
Assert.assertEquals(currentTypeOfVisit, "-Select Type of Visit-", "Type of Visit value is not cleared");
Assert.assertEquals(currentSelectedDate, initialSelectedDate, "Selected date is not cleared");
```

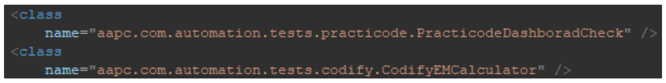
   - TestNG provides different types of assertion methods like assertEquals(), assertTrue(), assertFalse(),



```java
String displayedDate = emCalculatorPage.getCalendarInputValue();
Assert.assertTrue(displayedDate.contains(String.valueOf(randomDate)), "Selected date is not visible");
```

   - assertNotNull() which you can use based on your requirements.
3. **Add Test to Test Suite:**

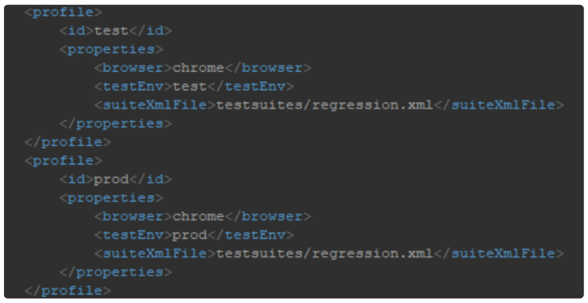
- After you've written your test, you need to add it to a test suite. Test suites are located in the 'testsuites' directory, with each suite represented as a '.xml' file.
  - The test can be run as part of regression, smoke or both.
- To add a test to a suite, simply add a new '<class>' entry inside the '<test>' tag of the desired suite's '.xml' file.



```
<class
    name="aapc.com.automation.tests.practicode.PracticodeDashboradCheck" />
<class
    name="aapc.com.automation.tests.codify.CodifyEMCalculator" />
```

Running Tests

1. **Running Tests from an IDE:**
   - You can directly run individual tests or test suites from your preferred Java IDE (like IntelliJ IDEA or Eclipse).
   - Right-click the test class or TestNG suite file and select 'Run'.
2. **Running Tests from Command Line:**
   - To run tests from the command line, navigate to the project root folder in your terminal or command prompt.
   - Execute the command `mvn clean test -Ptest`. This will run all the tests in the regression suite against test environment (you can modify the test profile in `pom.xml`).
   - If you want to run a specific test suite, use `mvn test -DsuiteXmlFile=testsuites/yourSuiteName.xml`.
3. **Running Tests with Different Configurations:**
   - By utilizing Maven's profile feature, you can run tests in different environments (dev, test, prod) or with different browsers.
   - To do this, execute `mvn test -P yourProfileName`. `yourProfileName` refers to the name of the Maven profile you want to use.
     - For instance, we have different profiles pre-configured for `dev`, `test`, and `prod` environments, each with different sets of properties (like `browser`, `testEnv`, and `suiteXmlFile`). This makes it easy to switch between different configurations depending on the needs of our testing.

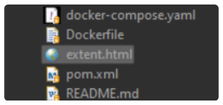


```
<profile>
    <id>test</id>
    <properties>
        <browser>chrome</browser>
        <testEnv>test</testEnv>
        <suiteXmlFile>testsuites/regression.xml</suiteXmlFile>
    </properties>
</profile>
<profile>
    <id>prod</id>
    <properties>
        <browser>chrome</browser>
        <testEnv>prod</testEnv>
        <suiteXmlFile>testsuites/regression.xml</suiteXmlFile>
    </properties>
</profile>
```

     - Most commonly used profile: `mvn clean test -Ptest`

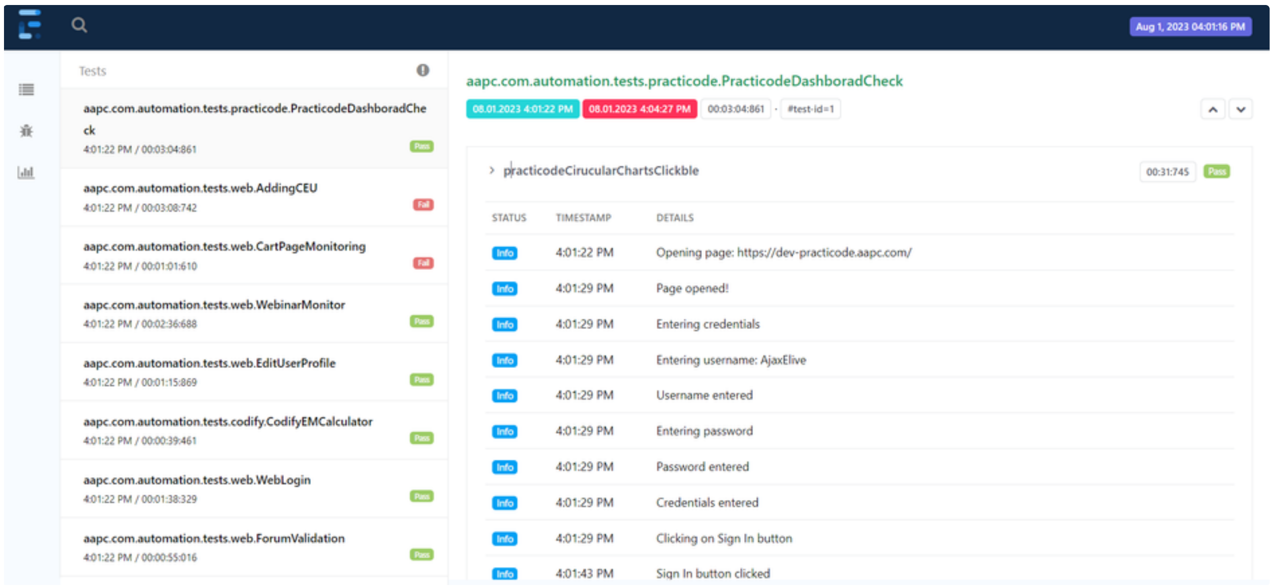
Reporting

1. **Access and Interpret Extent Reports:**
   - After running your tests, an Extent Report will be generated in the root directory of the project, named 'extent.html'.



```
docker-compose.yaml
Dockerfile
extent.html
pom.xml
README.md
```

   - Open this file in a web browser to view the report.



Aug 1, 2023 04:01:16 PM

Tests

aapc.com.automation.tests.practicode.PracticodeDashboradCheck
4:01:22 PM / 00:03:04:861 Pass

aapc.com.automation.tests.web.AddingCEU
4:01:22 PM / 00:03:08:742 Fail

aapc.com.automation.tests.web.CartPageMonitoring
4:01:22 PM / 00:01:01:610 Fail

aapc.com.automation.tests.web.WebinarMonitor
4:01:22 PM / 00:02:36:688 Pass

aapc.com.automation.tests.web.EditUserProfile
4:01:22 PM / 00:01:15:869 Pass

aapc.com.automation.tests.codify.CodifyEMCalculator
4:01:22 PM / 00:00:39:461 Pass

aapc.com.automation.tests.web.WebLogin
4:01:22 PM / 00:01:38:329 Pass

aapc.com.automation.tests.web.ForumValidation
4:01:22 PM / 00:00:55:016 Pass

aapc.com.automation.tests.practicode.PracticodeDashboradCheck

08.01.2023 4:01:22 PM | 08.01.2023 4:04:27 PM | 00:03:04:861 · #test-id=1

practicodeCirucularChartsClickble 00:31:745 Pass

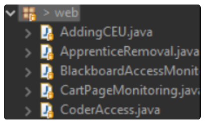
STATUS	TIMESTAMP	DETAILS
Info	4:01:22 PM	Opening page: https://dev-practicode.aapc.com/
Info	4:01:29 PM	Page opened!
Info	4:01:29 PM	Entering credentials
Info	4:01:29 PM	Entering username: AjaxElive
Info	4:01:29 PM	Username entered
Info	4:01:29 PM	Entering password
Info	4:01:29 PM	Password entered
Info	4:01:29 PM	Credentials entered
Info	4:01:29 PM	Clicking on Sign In button
Info	4:01:43 PM	Sign In button clicked

- The report provides detailed information about each test run, including the total number of tests passed, failed, and skipped.
- Each test in the report can be expanded to view the steps executed, the assertions made, and any error messages with screenshot if the test failed.

2. **Log4j Configuration and Usage:**
   - Log4j configuration is done through the 'log4j2.xml' file located in the 'resources' directory.
   - You can modify this file to change the logging level, format, and appenders.
   - To use Log4j in your tests, create a Logger instance in your test class (`private static Logger log = LogManager.getLogger(YourClassName.class);`).

Best Practices

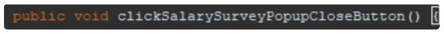
1. **Naming Conventions:**
   - Follow the convention of 'XYZTest' => `PascalCase`

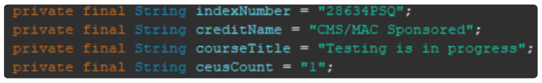
     ```
     > web
       AddingCEU.java
       ApprenticeRemoval.java
       BlackboardAccessMonit
       CartPageMonitoring.jav
       CoderAccess.java
     ```

     for test classes and 'testXYZ' => `camelCase`

     for test methods and variables.

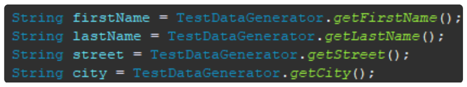
     ```
     public void clickSalarySurveyPopupCloseButton() {
     ```

     ```
     private final String indexNumber = "28634PSQ";
     private final String creditName = "CMS/MAC Sponsored";
     private final String courseTitle = "Testing is in progress";
     private final String ceusCount = "1";
     ```
   - Use meaningful and descriptive names that convey what the test class or method does.
2. **Code Structure:**
   - Each test class should correspond to a particular feature or functionality.
   - Each test method should cover a single test scenario.
   - Keep your test methods small and concise. If a test method becomes too large or complex, consider breaking it down into multiple smaller tests.
3. **Assertions:**
   - Make sure to include assertions in your tests to validate the outcome.
   - Whenever possible, assert against real application data rather than hardcoded expected values. This makes your tests more robust to changes in the application.
4. **Use of Page Objects:**

- Follow the Page Object Model pattern to abstract away the technicalities of interacting with the application. This makes your tests more readable and maintainable.
- ⭐ Ensure that all interactions with the application go through a page object, rather than directly using WebDriver commands in the tests.

5. **Separation of Test Data:**
    - Keep your test data separate from your test logic. This makes your tests easier to maintain and allows for data-driven testing.
    - Consider using the TestDataGenerator class for generating random test data when needed.



```
String firstName = TestDataGenerator.getFirstName();
String lastName = TestDataGenerator.getLastName();
String street = TestDataGenerator.getStreet();
String city = TestDataGenerator.getCity();
```

6. **Logging:**
    - Use `Log4j` and `ExtentTestManager` to log important events and information during your tests.
    - This can help with debugging when tests fail, as you can trace the sequence of events leading up to the failure.

How to?

How to install Java?

- ***Download Java***
    - Download the Latest Java LTS Free - windows x64 Installer
- ***Install Java***
- ***Set Java Environment Path***
    - https://www.java.com/en/download/help/path.html
- ***Verify Java Installation***
    - `java -version`

How to install Maven?

- **Download Maven**
    - Maven – Download Apache Maven -
- **Set Maven Environment Path**
- **Verify Maven Installation**
    - `mvn -version`