What is a framework?

- Frameworks are basically commonly used, and proven approach of creating file and folder structure, creating smart logic (implementing design patterns, utilities) to be able to easily handle and maintain our project

Why do we create a framework?

- Frameworks are created to apply re-usability, maintainability, scalability, easy to use, easy to understand.

BDD Framework : Maven(build) + Cucumber(behave) + Selenium(automation) + Junit(testing)
POM Design pattern

- Project
  - src
    - test
      - java
        - com.cydeo (company/application name)
          - pages
            - PageClasses (We store webElement)
          - runner
            - CukesRunner (We run our tests)
          - step_definitions
            - StepDefClasses (implementation layer: We do code implementation with using Java+Selenium+Junit)
            - Hooks (We have @Before and @After Scenario annotations coming from cucumber. Especially using for taking screenshot under @After)
          - utilities
            - BrowserUtil (We have common useful methods which I need in my most tests)
            - ConfigurationReader (To stay away from hardcoding I store my mostly using test data in properties file. This class helping me to read this file)
            - Driver (I have singleton design pattern in my Driver class [remember from if(driver==null){setup driver}else {return driver;}]
              Whenever I need driver I am calling it from Driver class and it makes me to use one single driver each time)
      - resources
        - features
          - file.feature (business layer: We write Feature, Scenario, and scenario steps with using gherkin language)

  - target (We get Cucumber Report under target folder)
  - configuration.properties (We store test data)
  - pom.xml (comes from maven and we are getting need tools with using dependencies)

Tell me about your Framework:

- I used java as my programming language in my framework.
- I used Selenium to automate my browsers.
- I used maven as my build automation tool. which has pom.xml file that allows me to manage my dependencies/versions easily.
- I used Page Object Model to simplify managing and maintaing my framework for myself and my team.
- this design pattern allows me to locate web elements only once, in their respective classes.
- so that if there are any problems with any web elements, I know exactly where to go and how to fix it.
- I created Singleton Design Pattern to allow my framework to pass the same instance of my webdriver in one session.
- (one session: when you click run, selenium creates one session. the session will end when the driver stops.)

→ continue…

- I created a configuration.PROPERTIES type of file where I keep the important test data about my framework. I keep Test data that can change the running flow of the whole framework, such as:
  - browser
  - username/password
  - url: to change and run-on different environments
- I created utility class from existing java library to read from properties type of file.
  - ConfigurationReader
- you should be ready to talk about how to read from properties file.
- opening the file and passing the path of the file into FileInputStream
- loading the file to properties class object.
- I implement BDD approach to simplify reading and understanding my framework for the non-technical teammates in my team.
- I write my test cases as if they are scenarios from the end users' perspective in Gherkin language in my feature files.
- Gherkin is very similar to English. Therefore, it is easy to understand for non-technical teammates.

  → continue…

- I implemented the actual coding logic with JAVA-SELENIUM-JUNIT... inside of my step_definitions package and it their own respective/related classes
- I trigger my framework from my runner class.
- Runner class allows me to run different types of testing suites that I created with my tags, such as smoke, regression, mini-regression.
- I have different types of reports. But mainly I use "maven-cucumber-reporting" which is a very detailed reporting tool that has pie-charts, matrixes on which tests are passing and failing.
- Even has the option to show what percentage of which tags are failing and passing.
- Hooks class, where we implement some cucumber annotations such as Before, After, beforestep, afterstep to create outline for my scenarios.
- I also implemented a logic where my framework is taking a screenshot and attaching it to my report if a scenario is failing.
- DDT - SCENARIO OUTLINE
- APACHE POI
- SQL/JBDC
- API

WHAT IS TDD (Test Driven Development)?

- Bug-free development

How do we implement TDD?

1- Write unit tests
2- Run the tests
3- Tests will fail
4- Write just enough code to pass the currently written tests (MVP)
5- Run the tests
6- Tests will PASS
7- REFACTOR & REPEAT: We write more tests and repeat the TDD CYCLE.

How is BDD similar to TDD?

- Instead of writing test, we write SCENARIOS.
- We implement the logic that turns those scenarios into actions using Java+selenium+junit

What are the 2 different sides of BDD?

#1- BUSINESS LAYER : feature files
#2- IMPLEMENTATION LAYER : step definitions

#1- **BUSINESS LAYER** : feature files

- Where we write our features and scenarios in Gherkin language
- Gherkin is basically English.
- It makes it very easy to understand for non-technical member of the team
- We use certain keywords to implement Cucumber logic in feature files.
  - Feature:
  - Scenario:
  - Given, When, Then, And, But

#2- **IMPLEMENTATION LAYER** : step definitions

- We generate snippets and implement them in "step_definitions" package

What is a snippet?

- Unimplemented step definition methods that are automatically generated by Cucumber.

Different ways of generating snippet:

#1- Run the RunnerClass and generate the snippet in console
#2- Hover over the unimplemented step --> Create step definition --> Select class where we want it

--> We can generate snippet for more steps:

- Hover over one line --> More actions --> Create all step definitions

#3- Put your cursor on the unimplemented line:

Windows: ALT + Enter --> Create step definition --> Select class where we want it

Mac: Option + Enter --> Create step definition --> Select class where we want it

**We can re-use the step we generate in feature files, but we will have only 1 snippet and 1 implementation for that specific STEP.

- Every time we use the same step, our code will find the implementation and execute the same method regardless of which feature or sceario we use our step from.

How to create a feature file and implement scenarios STEP BY STEP

1- Create a feature file "something.feature" under our "features" directory
2- Use "Feature:" keyword to provide information about the feature that will be tested, and also (optionally) you can pass here the Agile story as well.
3- Use "Scenario:" keyword to generate a "Scenario:" and describe what this scenario will do
4- Add our "Steps" using assigned keywords such as "Given", "When", "Then", "And", "But"
5- Generate our snippets to implement "Step_definitions"
- We have multiple ways to get our snippets
#1- run the RUnner class and copy the auto-generated snippets from console
#2- hover over the unimplemented step, and IntelliJ will auto-generate the snippet for us
#3- we can run the scenario or feature from the feature file itself to generate snippet as well
6- Copy paste the snippet into the step_definition ".java" class we have
7- Implement our "JAVA+SELENIUM+JUNIT" logic to turn the step into an actual action

BACKGROUND:

- Background is very similar to @BeforeMethod in TestNG.
- @BeforeMethod executes given code/method before each Test in that SPECIFIC CLASS.
- Background runs before each SCENARIO in that SPECIFIC FEATURE FILE.
- The step we pass under the "Background:" will execute only once before each step
- Important thing to keep in mind is that "Background" will be applying to each and every scenario in the same feature file.
- Therefore, we must make sure every scenario is able to pick up and continue where the background is leaving the code.

HOOKS CLASS:

- Hooks class will allow us to pass pre and post conditions for each scenario and each step.
- Hooks class is separated from feature file
- Therefore, it will go into the implementation side (step_definitions)

How does my project knows where to find the Hooks class and execute the annotations?

- Basically, logic is coming from the cucumber annotations and also glue path we provide in the runner class.

Hooks VS Background

@Before in Hooks class will be run before the first step of each scenario. They will run in the same order of which they are registered.

Background allows you to add some context to the scenarios in a single feature. A Background is much like a scenario containing a number of steps.

**Use Background when you provide customer-readable pre-conditions to your scenarios
**Use Before when you have to do some technical setup before your scenarios

How do you take screenshot in framework?

- I use Scenario class to get certain information from current scenario such as name, and condition
- I downcast my driver type to TakesScreenshot interface and use method getScreenshotAs to store my screenshot as array of bytes
- And attach my screenshot into report using scenario class object and attach method.

SYNTAX:

```
if (scenario.isFailed()){
    byte [] screenshot = ((TakesScreenshot)Driver.getDriver()).getScreenshotAs(OutputType.BYTES);
    scenario.attach(screenshot, "image/png", scenario.getName());
}
```

1-> scenario.isFailed() :

- if scenario fails this will return TRUE.

2- byte [] screenshot = We are creating an array of bytes to be able to store our screenshot
3- ((TakesScreenshot)Driver.getDriver()) ---> we are downcasting our driver type to TakesScreenshot
4- .getScreenshotAs(OutputType.BYTES); ---> we are returning the screenshot as byte so we can use
5- scenario.attach() --> this method is able to attach the screenshot into our report

- it accepts 3 arguments

arg1 : array of bytes --> byte [] screenshot :
arg2 : String image type --> "image/png"
arg3 : String scenario name --> scenario.getName()

If we have multiple versions of the same annotation, we can prioritize the running order using the "order" keyword.

- The lower the number passed in the order, earlier it will be executed.
  - The methods will be executed in the order it is specified with numbers.

```
@Before (order = 1)
public void setupScenario(){
    System.out.println("====Setting up browser using cucumber @Before");
}
@Before (value = "@login", order = 2)
public void setupScenarioForLogins(){
    System.out.println("====this will only apply to scenarios with @login tag");
}
```

- We can specify which annotation is running for which scenarios or features using @TAGS.
- If I want some scenario/feature to have pre- /post- conditions, I can use certain @TAG, and pass the same @TAG into the annotation in Hooks class.

@Before

- comes from cucumber-java dependency
- this will change the behavior of the method we use it.
- this method will be running BEFORE each and every SCENARIO in our project (unless we specify with @TAGS)

@After

- comes from cucumber-java dependency
- this will change the behavior of the method we use it.
- this method will be running AFTER each and every SCENARIO in our project (unless we specify with @TAGS)

@BeforeStep

- comes from cucumber-java dependency
- this will change the behavior of the method we use it.
- this method will be running BEFORE each and every STEP in our project (unless we specify with @TAGS)

@AfterStep

- comes from cucumber-java dependency
- this will change the behavior of the method we use it.
- this method will be running AFTER each and every STEP in our project (unless we specify with @TAGS)

Where do we control or trigger our framework?

- Runner class
- TestRunner
- CukesRunner

What aspects of the project we control from Runner class?

- **plugin** : determines what type of report we want to generate with our project, and also where we want to store.
- **features** : we provide the path of the 'features' directory and let our project know where to find all the feature files.
- **glue** : glue gives path to the package of the step_definitions
- **dryrun** : dryRun determines if we want to execute the step_definitions or not
  - if **dryRun is true**: dryRun is turned on, and step definitions are turned off (will not run)
  - if **dryRUn is false**: dryRun is turned off, and step definitions are turned on (will run)
  - this is mostly used for when implementing new steps and step definitions, and we do not want to run the code and open browser etc, just to get the snippets.
  - we turn off the step_definitions and just generate snippets

- tags:

- What are tags and why we use them?
- Tags allows us to create different scenario suites or groups to run
- we can include, or exclude different feature files or scenarios using tags.

- or:

- less specific way of creating group.
- @a or @b
- if a scenario or feature has either @a or @b, it will be executed.
- this is similar to || 'or' in java

- and

- more specific way of creating group
- @a and @b
- the scenario MUST have both @a and @b to be able to get executed.
- this is similar to && 'and' in java

- and not

- if we want to exclude some certain @tag, group of feature files or scenarios, we can put the tag on top of them, and use with 'and not'
- they will not be executed

@a or @b and not @c

DATA DRIVEN TESTING:

- Execute the same test with multiple sets of test data.

- Assume, you need to test login form with 50 different sets of test data
- As a manual tester, you do log in with all the 50 different sets of test data for 50 times
- As an automation tester, you create a test script and run 50 times by changing the test data on each run or you create 50 test scripts to execute the scenario

- We can achieve Data-driven framework using:

  - TestNG's data provider
  - Cucumber scenario outlines
  - Excel apache poi

PARAMETERIZATION:

- this allow us to pass arguments to our steps from feature files
- allows us to do data driven testing

DATATABLES:

- Why do we use the data tables?
- Parameterization allows us to pass multiple arg in one step
- But we cannot pass collection types under one step using parameterization
- Data tables allows us to pass collection types **under one step**
- We can pass List, Map, List of Maps, Maps of Maps etc...

SCENARIO OUTLINES:

- Using scenario outlines, we can create a data table of examples, and **run our scenario against this table.**
- Instead of using "Scenario" keyword, we use "Scenario Outline keyword"
- After this step, it will not compile unless we provide a table of "Examples"
- The arguments we provide in our steps become "header" for the "examples table"
- and we provide rest of the data under headers.
- we can pass more than one examples table under one scenario outline

What is the difference between data table and scenario outline?

- Data tables allow us to pass collection type into a step in a feature file.
- Scenario outline is completely different. It is similar approach to Parameterization. But instead of passing the test data inside of the step, we provide a test data as a table under our Scenario outline.

- We cannot create SCENARIO OUTLINE/TEMPLATE without providing an "Examples" table.
- But we can use "data tables" without having to use "examples”

**Scenario Outline and Scenario Template are the same thing. Just alternative keyword.

RERUN:

- CUCUMBER RERUN IS A PLUGIN COMING FROM CUCUMBER
- This plugin allows us to keep track of the FAILED scenarios.
- All we have to do is use the plugin, and provide where it should be creating the .txt file.

Why do we need RERUN and FailedTestRunner?

- Mostly to save time when tests fails.
- If we have hundreds of tests, and only few of them are failing, we don't want to execute all of the hundreds of the tests again.
- That’s why we can just execute the failed tests that are automatically stored under the rerun.txt using our FailedTestRunner.

- FailedTestRunner cannot change or edit the rerun.txt.
- Therefore, even if the tests are passing, the lines will still in the rerun.txt, which is normal.
- We can delete rerun.txt by going Maven --> clean

DEPENDENCIES VS PLUGINS

-> Dependency: is basically libraries and jar files.

- We add to pom.xml file and manage libraries and version
- IMPORTANT: Dependencies does not take part in maven lifecycle.
- When you kickoff your projects using maven lifecycles, dependencies are not ACTIVELY INVOLVED.

-> Plugins: ARE VERY SIMILAR TO DEPENDENCIES.

- Basically, Plugins are DEPENDENCIES with additional configuration.
- We can change values from plugins and when we kickoff our tests from Maven Lifecycles, it will be immediately applied to our project/tests.

Why do we use parallel testing?

--> To save time.

- 1 test 1 min 1 machine = 1 min
- 10 test 1 min 1 machine = 10 min
- 10 test 1 min 2 machine = 5 min
- 10 test 1 min 3 machine = 3.3 min
- 10 test 1 min 5 machine = 2 min
- 10 test 1 min 10 machine = 1 min

- • https://cucumber.io/docs/guides/parallel-execution/

What is a plugin?

- Plugins are actually jar files just like dependencies.
- They are very similar, but plugins are involved in the maven lifecycles.
- Most of the work in the maven lifecycle are done by plugins.
- Whereas dependencies are only jar files.

Plugins we are using in our framework

- Maven surefire plugin.

Maven compiler plugin.

- Forces the maven compiler level to Java 8 or given version.

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.6.1</version>
    <configuration>
        <source>8</source>
        <target>8</target>
    </configuration>
</plugin>
```

Maven Surefire Plugin

- Maven Surefire Plugin allows us to kickoff our tests from Maven Lifecycles and applies the extra configuration we provide.
  - Running multiple threads
  - Running feature files or scenarios in parallel to each other.
  - Therefore, allows us to do parallel testing.

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>3.0.0-M5</version>
    <configuration>
       <parallel>methods</parallel>
       <threadCount>5</threadCount>
       <perCoreThreadCount>false</perCoreThreadCount>
       <testFailureIgnore>true</testFailureIgnore>
       <includes>
          <include>**/CukesRunner*.java</include>
       </includes>
    </configuration>
 </plugin>
```

What does this piece of code do?

```
<parallel>methods</parallel>
<threadCount>4</threadCount>
<testFailureIgnore>true</testFailureIgnore>
```

- Basically, additional configuration that allows us to execute parallel tests in different threads.

<parallel>methods</parallel>

--> decides if we want to run feature files in parallel

<threadCount>4</threadCount>

--> determines how many threads we want to open at once

<testFailureIgnore>true</testFailureIgnore>

--> Even if some test fails, I don't want to stop execution and continue with rest of tests/feature scenarios

What problem we need to solve after adding plugin?

- Driver utility, singleton design pattern.
- It limits our code so that we cannot run with multi threads.

What is the solution?

- We will use **ThreadLocal** class from Java to create **driverPool**.
- ThreadLocal is a class that creates the object of the given class PER THREAD.
- You can think of it as this class will provide us with a pool of drivers, where our code will be able to go and use as many as needed.

We will adjust our Driver utils class accordingly

- driverPool.remove() → will remove particular object for given thread
  - We will use this instead of driver.quit()
- driverPool.set() → set/add object
- For setting, creating webdriver
- driverPool.get() → return object

Steps to make our driver thread local.

1- We wrap our driver with ThreadLocal:

- **private static InheritableThreadLocal<WebDriver> driverPool = new InheritableThreadLocal<>();**

2- driverPool.get():

-Instead of using if(driver==null){} , We will use **if(driverPool.get()==null){}**

3- driverPool.set():

Instead of using this: driver=new ChromeDriver();

We will use this: **driverPool.set(new ChromeDriver());**

4- return driver:

Instead of returning driver.

We return **driverPool.get();**

We also need to change our closeDriver() method .

```
public static void closeDriver() {
    if (driverPool.get() != null) {
        driverPool.get().quit();
        driverPool.remove();
    }
}
```

Interview question

If you used Singleton in your Driver, how did you handle parallel execution?

- #1- I wrapped my WebDriver object with ThreadLocal that creates copy of driver object per thread.
- #2- Instead of using driver directly I used driverPool.get() method to get a driver instance from a pool of drivers objects
- #3- This will provide me as many drivers as the number of threads I am running

Can you explain how you set up parallel testing?

1. I use different plugins to achieve parallel testing.
   --> **Maven compiler plugin** makes sure my compiler level is java 8
   --> **Maven surefire plugin** allows me to pass different configurations
   that allows to run my feature files in parallel to each other.
   -> I can also pass the number of threads that I want to open in parallel
   to each other.
2. I adjusted my **Driver class** to be able to generate as many driver object as needed, using **ThreadLocal class.**
3. I use "**synchronized**" block to make sure multiple threads are not going for the same driver instance.
4. When I kick off my tests from **mvn verify** command, it will trigger my "maven surefire plugin" and my tests are executed in parallel

Cucumber io parallel execution

- JUNIT can execute Feature files in parallel
- TestNG can execute Scenarios in parallel