Why automation?

- ➔ Manual testing is time consuming, in some case its not possible
- ➔ Lot of cost
- ➔ Accuracy issue

Why selenium?

- ➔ Cost
- ➔ Support
- ➔ Enough talent

Why selenium with java?

- ➔ Widely used
- ➔ Framework
- ➔ Integration
- ➔ Community support

What type of framework?

- ➔ Data -driven framework
- ➔ Keyword driven
- ➔ Hybrid
- ➔ POM

Components

- ➔ Ide – outdated
- ➔ Webdriver- 2.0/3.x/4.x
- ➔ Grid- parrlel testing , compatibility testing

Automation with selenium

Login test case:

1. Type username in username field

What	**where(95%)**	**data**
Type	username field	valid username
Type	password field	valid password
Click	Login button	not required
Verify exist	Home page	not required

Identifier/locator

ID , Name , combination of properties , xpath

Xpath rules(Shiv)- http://tutorialsninja.com/demo/

1. if element has an ID or name , use complete id , name or partial to make element unique

   //div[@id='search']

2. if no ID, name found. then use any other attribute to make element unique. full or partial

   //a[@href='http://tutorialsninja.com/demo/index.php?route=common/home']

3. if not able to find element unique by one attribute , use more then one to make element unique

//button[@data-toggle='dropdown' and @type='button']

4. if not able to identify element based on above 3 rule , move to parent element , identify parent element unique and navigate to child element
   //div[@id='search']//button
   //div[@id='search']/span/button

5. sibling
   //div[@id='search']/parent::div/following-sibling::div
   //div[@id='cart']/parent::div/preceding-sibling::div[1]

6. use contains to identify element if no above rule satisfies
   //a[contains(text(),'Your Store')]

7. if no above rule working out , get the path starting from HTML , keep on cutting and verifying . till the place u get unique, use thaat using double '/' . if the last one with index more then 3,4 then replace that with appropriate attribute to identify the element by any above rule.

   //div[2]/div/div[1]/a

Complex example
//a[contains(text(),'Apple Cinema 30')]/parent::h4/parent::div/following-sibling::div//span

Implicit wait

- ➔ Instructs the webdriver to wait for maximum of certain time before it throws “No such element exception”
- ➔ Applicable for page level

Explicit wait:

- ➔ Instructs the webdriver to wait for certain condition till the maximum time exceeds
- ➔ Applicable for element level

TestNG

Installation:

Help-> install new software

Click add

Enter name as testNg

Enter URL- https://testng.org/testng-eclipse-update-site

Click on next , next and finish at the end

Restart eclipse

testNG should be available now

why TestNG:

- ➔ Simple verification &validation – aasert statements
- ➔ Need of automatic report- default testing html report
- ➔ Ability to group my test into category – regression/smoke … grouping test case and enable group run from testing.xml
- ➔ Ability to prioritize my test case- enable priority in testing tests
- ➔ Ability to run in parallel- set parallel from testing xml file
- ➔ Easy data handling- using dataprovider

After installing and adding dependency:

Convert your project into testNG-> it will generate xml file – testing.xml

Testing.xml

This can be configured for group running by adding group attribute

API

Why API?

Amazon – HDFC bank
Python -> Java

Two different system has to have a common technology or common understanding

- Language known to every technology

- Used by all technology
- Something that is universal
- Generally used for data passing

XML, JSON

➔ SOAP (xml based, fixed format, much secured)
➔ Rest (JSON based, format is not fixed, not as secured as soap, better performance)
    - It works on http protocol

Why wsdl?

To let the client know the structure of SOAP api

➔ Method name
➔ Method signature
➔ Argument list
➔ Arugument name
➔ Argument type
➔ Return type
➔ Response variable

BDD/Gherkins

Why BDD?

➔ To bring business, developer, qe into single goal/commitment/understanding
➔ It has to be in simple English like language
➔ That language is Gherkins

Feature file:

Scenario: Login functionality should work as expected

➔ Given: User navigate to login page
➔ When: user enters username
➔ And: user enters password
➔ And: user click on login button
➔ Then: login should be successful
➔ And: Home page should be displayed

Scenario: Login functionality should throw error for invalid login

➔ Given: User navigate to login page
➔ When: user enters wrong username
➔ And: user enters password
➔ And: user click on login button
➔ Then: login should not be successful
➔ But: user should not be able to navigate into home page

Gherkin uses a set of special keywords to give structure and meaning to executable specifications. Each keyword is translated to many spoken languages; in this reference we'll use English.

Most lines in a Gherkin document start with one of the keywords.

Comments are only permitted at the start of a new line, anywhere in the feature file. They begin with zero or more spaces, followed by a hash sign (`#`) and some text.

Block comments are currently not supported by Gherkin.

Either spaces or tabs may be used for indentation. The recommended indentation level is two spaces. Here is an example:

```
Feature: Guess the word

  # The first example has two steps
  Scenario: Maker starts a game
    When the Maker starts a game
    Then the Maker waits for a Breaker to join

  # The second example has three steps
  Scenario: Breaker joins a game
    Given the Maker has started a game with the word "silky"
    When the Breaker joins the Maker's game
    Then the Breaker must guess a word with 5 characters
```

The trailing portion (after the keyword) of each step is matched to a code block, called a step definition.

Please note that some keywords *are* followed by a colon (`:`) and some *are not*. If you add a colon after a keyword that should not be followed by one, your test(s) will be ignored.

Keywords

Each line that isn't a blank line has to start with a Gherkin *keyword*, followed by any text you like. The only exceptions are the free-form descriptions placed underneath `Example`/`Scenario`, `Background`, `Scenario Outline` and `Rule` lines.

The primary keywords are:

- `Feature`

- `Rule` (as of Gherkin 6)
- `Example` (or `Scenario`)
- `Given`, `When`, `Then`, `And`, `But` for steps (or `*`)
- `Background`
- `Scenario Outline` (or `Scenario Template`)
- `Examples` (or `Scenarios`)

There are a few secondary keywords as well:

- `"""` (Doc Strings)
- `|` (Data Tables)
- `@` (Tags)
- `#` (Comments)

Localisation

Gherkin is localised for many spoken languages; each has their own localised equivalent of these keywords.

Feature

The purpose of the `Feature` keyword is to provide a high-level description of a software feature, and to group related scenarios.

The first primary keyword in a Gherkin document must always be `Feature`, followed by a `:` and a short text that describes the feature.

You can add free-form text underneath `Feature` to add more description.

These description lines are ignored by Cucumber at runtime, but are available for reporting (they are included by reporting tools like the official HTML formatter).

```
Feature: Guess the word

  The word guess game is a turn-based game for two players.
  The Maker makes a word for the Breaker to guess. The game
  is over when the Breaker guesses the Maker's word.

  Example: Maker starts a game
```

The name and the optional description have no special meaning to Cucumber. Their purpose is to provide a place for you to document important aspects of the feature, such as a brief explanation and a list of business rules (general acceptance criteria).

The free format description for `Feature` ends when you start a line with the keyword `Background`, `Rule`, `Example` or `Scenario Outline` (or their alias keywords).

You can place tags above `Feature` to group related features, independent of your file and directory structure.

You can only have a single `Feature` in a `.feature` file.

Descriptions

Free-form descriptions (as described above for `Feature`) can also be placed underneath `Example`/`Scenario`, `Background`, `Scenario Outline` and `Rule`.

You can write anything you like, as long as no line starts with a keyword.

Descriptions can be in the form of Markdown - formatters including the official HTML formatter support this.

Example

This is a *concrete example* that *illustrates* a business rule. It consists of a list of steps.

The keyword `Scenario` is a synonym of the keyword `Example`.

You can have as many steps as you like, but we recommend 3-5 steps per example. Having too many steps will cause the example to lose its expressive power as a specification and documentation.

In addition to being a specification and documentation, an example is also a *test*. As a whole, your examples are an *executable specification* of the system.

Examples follow this same pattern:

- Describe an initial context (`Given` steps)
- Describe an event (`When` steps)
- Describe an expected outcome (`Then` steps)

Steps

Each step starts with `Given`, `When`, `Then`, `And`, or `But`.

Cucumber executes each step in a scenario one at a time, in the sequence you've written them in. When Cucumber tries to execute a step, it looks for a matching step definition to execute.

Keywords are not taken into account when looking for a step definition. This means you cannot have a `Given`, `When`, `Then`, `And` or `But` step with the same text as another step.

Cucumber considers the following steps duplicates:

```
Given there is money in my account
Then there is money in my account
```

This might seem like a limitation, but it forces you to come up with a less ambiguous, more clear domain language:

```
Given my account has a balance of £430
Then my account should have a balance of £430
```

Given

`Given` steps are used to describe the initial context of the system - the *scene* of the scenario. It is typically something that happened in the *past*.

When Cucumber executes a `Given` step, it will configure the system to be in a well-defined state, such as creating and configuring objects or adding data to a test database.

The purpose of `Given` steps is to **put the system in a known state** before the user (or external system) starts interacting with the system (in the `When` steps). Avoid talking about user interaction in `Given`'s. If you were creating use cases, `Given`'s would be your preconditions.

It's okay to have several `Given` steps (use `And` or `But` for number 2 and upwards to make it more readable).

Examples:

- Mickey and Minnie have started a game
- I am logged in
- Joe has a balance of £42

When

`When` steps are used to describe an event, or an *action*. This can be a person interacting with the system, or it can be an event triggered by another system.

It's strongly recommended you only have a single `When` step per Scenario. If you feel compelled to add more, it's usually a sign that you should split the scenario up into multiple scenarios.

Examples:

- Guess a word
- Invite a friend
- Withdraw money

Imagine it's 1922

Most software does something people could do manually (just not as efficiently).

Try hard to come up with examples that don't make any assumptions about technology or user interface. Imagine it's 1922, when there were no computers.

Implementation details should be hidden in the step definitions.

Then

`Then` steps are used to describe an *expected* outcome, or result.

The step definition of a `Then` step should use an *assertion* to compare the *actual* outcome (what the system actually does) to the *expected* outcome (what the step says the system is supposed to do).

An outcome *should* be on an **observable** output. That is, something that comes *out* of the system (report, user interface, message), and not a behaviour deeply buried inside the system (like a record in a database).

Examples:

- See that the guessed word was wrong
- Receive an invitation
- Card should be swallowed

While it might be tempting to implement `Then` steps to look in the database - resist that temptation!

You should only verify an outcome that is observable for the user (or external system), and changes to a database are usually not.

And, But

If you have successive `Given`'s, `When`'s, or `Then`'s, you *could* write:

```
Example: Multiple Givens
  Given one thing
  Given another thing
  Given yet another thing
  When I open my eyes
  Then I should see something
  Then I shouldn't see something else
```

Or, you could make the example more fluidly structured by replacing the successive `Given`'s, `When`'s, or `Then`'s with `And`'s and `But`'s:

```
Example: Multiple Givens
  Given one thing
  And another thing
  And yet another thing
  When I open my eyes
  Then I should see something
  But I shouldn't see something else
```

*

Gherkin also supports using an asterisk (`*`) in place of any of the normal step keywords. This can be helpful when you have some steps that are effectively a *list of things*, so you can express it more like bullet points where otherwise the natural language of `And` etc might not read so elegantly.

For example:

```
Scenario: All done
  Given I am out shopping
  And I have eggs
  And I have milk
  And I have butter
  When I check my list
  Then I don't need anything
```

Could be expressed as:

```
Scenario: All done
  Given I am out shopping
  * I have eggs
  * I have milk
  * I have butter
  When I check my list
  Then I don't need anything
```

Background

Occasionally you'll find yourself repeating the same `Given` steps in all of the scenarios in a `Feature`.

Since it is repeated in every scenario, this is an indication that those steps are not *essential* to describe the scenarios; they are *incidental details*. You can literally move such `Given` steps to the background, by grouping them under a `Background` section.

A `Background` allows you to add some context to the scenarios that follow it. It can contain one or more `Given` steps, which are run before *each* scenario, but after any Before hooks.

A `Background` is placed before the first `Scenario`/`Example`, at the same level of indentation.

For example:

```
Feature: Multiple site support
  Only blog owners can post to a blog, except administrators,
  who can post to all blogs.

  Background:
    Given a global administrator named "Greg"
    And a blog named "Greg's anti-tax rants"
    And a customer named "Dr. Bill"
    And a blog named "Expensive Therapy" owned by "Dr. Bill"

  Scenario: Dr. Bill posts to his own blog
    Given I am logged in as Dr. Bill
    When I try to post to "Expensive Therapy"
    Then I should see "Your article was published."

  Scenario: Dr. Bill tries to post to somebody else's blog, and fails
    Given I am logged in as Dr. Bill
    When I try to post to "Greg's anti-tax rants"
    Then I should see "Hey! That's not your blog!"

  Scenario: Greg posts to a client's blog
    Given I am logged in as Greg
    When I try to post to "Expensive Therapy"
    Then I should see "Your article was published."
```

`Background` is also supported at the `Rule` level, for example:

```
Feature: Overdue tasks
  Let users know when tasks are overdue, even when using other
  features of the app

  Rule: Users are notified about overdue tasks on first use of the day
    Background:
      Given I have overdue tasks

    Example: First use of the day
      Given I last used the app yesterday
      When I use the app
      Then I am notified about overdue tasks

    Example: Already used today
      Given I last used the app earlier today
      When I use the app
```

```
      Then I am not notified about overdue tasks
  ...
```

You can only have one set of `Background` steps per `Feature` or `Rule`. If you need different `Background` steps for different scenarios, consider breaking up your set of scenarios into more `Rule`s or more `Feature`s.

For a less explicit alternative to `Background`, check out conditional hooks.

Tips for using Background

- Don't use `Background` to set up **complicated states**, unless that state is actually something the client needs to know.
    - For example, if the user and site names don't matter to the client, use a higher-level step such as `Given I am logged in as a site owner`.
- Keep your `Background` section **short**.
    - The client needs to actually remember this stuff when reading the scenarios. If the `Background` is more than 4 lines long, consider moving some of the irrelevant details into higher-level steps.
- Make your `Background` section **vivid**.
    - Use colourful names, and try to tell a story. The human brain keeps track of stories much better than it keeps track of names like `"User A"`, `"User B"`, `"Site 1"`, and so on.
- Keep your scenarios **short**, and don't have too many.
    - If the `Background` section has scrolled off the screen, the reader no longer has a full overview of what's happening. Think about using higher-level steps, or splitting the `*.feature` file.

Scenario Outline

The `Scenario Outline` keyword can be used to run the same `Scenario` multiple times, with different combinations of values.

The keyword `Scenario Template` is a synonym of the keyword `Scenario Outline`.

Copying and pasting scenarios to use different values quickly becomes tedious and repetitive:

```
Scenario: eat 5 out of 12
  Given there are 12 cucumbers
  When I eat 5 cucumbers
  Then I should have 7 cucumbers

Scenario: eat 5 out of 20
```

```
  Given there are 20 cucumbers
  When I eat 5 cucumbers
  Then I should have 15 cucumbers
```

We can collapse these two similar scenarios into a `Scenario Outline`.

Scenario outlines allow us to more concisely express these scenarios through the use of a template with `< >`-delimited parameters:

```
Scenario Outline: eating
  Given there are <start> cucumbers
  When I eat <eat> cucumbers
  Then I should have <left> cucumbers

  Examples:
    | start | eat | left |
    |    12 |   5 |    7 |
    |    20 |   5 |   15 |
```

Examples

A `Scenario Outline` must contain one or more `Examples` (or `Scenarios`) section(s). Its steps are interpreted as a template which is never directly run. Instead, the `Scenario Outline` is run *once for each row* in the `Examples` section beneath it (not counting the first header row).

The steps can use `<>` delimited *parameters* that reference headers in the examples table. Cucumber will replace these parameters with values from the table *before* it tries to match the step against a step definition.

You can also use parameters in multiline step arguments.