Contents

[Suggested project structure 1](#_Toc5373792)

[Configuration.properties file 1](#_Toc5373793)

[Tests Development 1](#_Toc5373794)

[Pages / Panels 1](#_Toc5373795)

[Overview 1](#_Toc5373796)

[Page and Panel Example 1](#_Toc5373797)

[How to create a Page class 2](#_Toc5373798)

[How to create a Panel class 3](#_Toc5373799)

[How to use your new Page 3](#_Toc5373800)

[Steps 4](#_Toc5373801)

[Overview 4](#_Toc5373802)

Guide to Test Automation Framework

## Suggested project structure

The suggested folder structure for the project is provided below. Feel free to adjust it to your needs.

1. **core** – a folder for base classes such as BasePanel (with default setup/teardown actions), BasePage, etc.
2. **steps** – a folder to store classes which describe certain flow, like Login flow, Search flow etc.
3. **pageobject** – a folder to store page and panels classes.

## Configuration.properties file

Framework can be configured using configuration.properties file.

# Tests Development

## Pages / Panels

### Overview

**Page** is a high-level container for all elements displayed on the screen, excluding those we are not interested in. Page can contain standalone elements and panels.

**Panel**, or Complex Control, is a class, which describes some group of elements on the page. It can contain elements and the other panels.

Panels are very useful for:

* Repeatable element groups
* Element groups located under common parent element

### Page and Panel Example

The page on screenshot below contains multiple settings widgets. We can declare a SearchResultsPanel panel class to use it multiple times on the page.

**public class** SearchResultsPanel **extends** BasePanel {  
  
 **public** SearchResultsPanel(WebElement element) {  
 **super**(element);  
 }  
  
 @FindBy(how = How.***CLASS\_NAME***, using = **"sr\_item\_content"**)  
 **private** List<WebElement> **resultItems**;  
  
 **public** List<ResultItem> getResults() {  
 …  
 }  
 **return** items;  
 }

The corresponding Page class:

**public class** ResultsPage **extends** BasePage {  
  
 @FindBy(how = How.***ID***, using = **"search\_results\_table"**)  
 **private** WebElement **searchResultPanel**;  
 **public** SearchResultsPanel **searchResults** = **new** SearchResultsPanel(**searchResultPanel**);  
}

### How to create a Page class

1. Create a class, which extends BasePage class.

**public class** HomePage **extends** BasePage

Add some elements to your page.

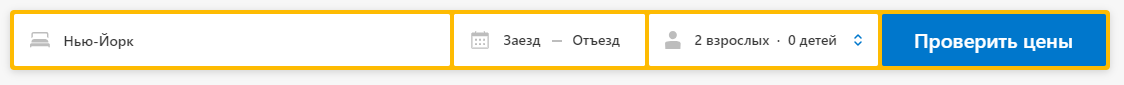
**public** WebElement **destinationBox**;

**public** WebElement **checkInDatesButton**;

**public** WebElement **searchButton**;

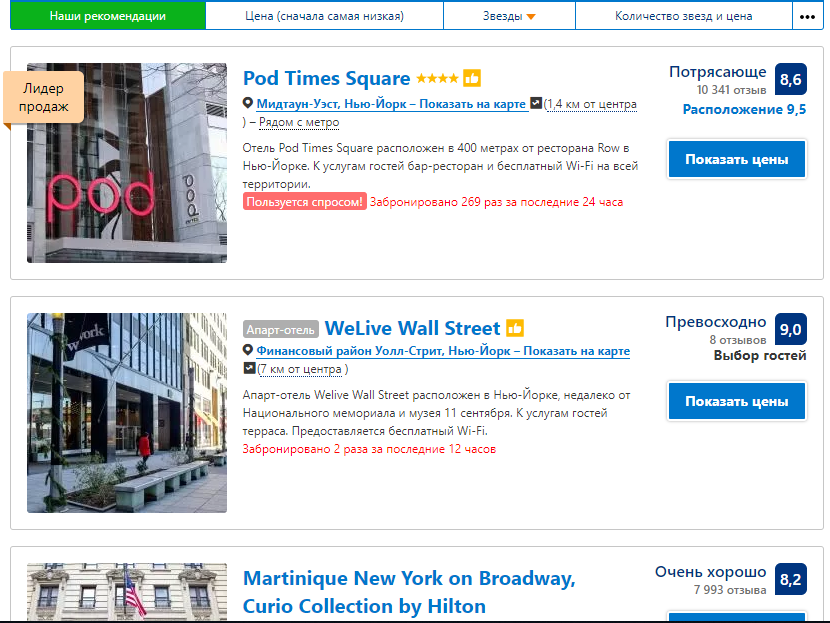
1. Add locator attributes to determine unique element position.

@FindBy(how = How.***CLASS\_NAME***, using = **"sb-destination\_\_input"**)  
**public** WebElement **destinationBox**;  
  
@FindBy(how = How.***CLASS\_NAME***, using = **"xp\_\_dates"**)  
**public** WebElement **checkInDatesButton**;  
  
@FindBy(how = How.***CLASS\_NAME***, using = **"sb-searchbox\_\_button"**)  
**public** WebElement **searchButton** ;



1. Add collections of elements (you will be able to iterate through it and access specific elements by index).

@FindBy(how = How.***CLASS\_NAME***, using = **"sr\_item\_content"**)  
**private** List<WebElement> **resultItems**;



1. Add panels.

**private** WebElement **searchResultPanel**;  
**public** SearchResultsPanel **searchResults** = **new** SearchResultsPanel(**searchResultPanel**);

1. Add locator attributes to panel field if all its elements are located under the same parent element.

@FindBy(how = How.***ID***, using = **"search\_results\_table"**)  
**private** WebElement **searchResultPanel**;  
**public** SearchResultsPanel **searchResults** = **new** SearchResultsPanel(**searchResultPanel**);

### How to create a Panel class

1. Create a class, which extends BasePanel class.

**public class** SearchResultsPanel **extends** BasePanel

1. Add elements / collections of elements / other panels

FindBy(how = How.***CLASS\_NAME***, using = **"sr\_item\_content"**)  
**private** List<WebElement> **resultItems**;

1. (Optional) Add simple methods involving only panel elements – no other panel or pages should be used.

**public** List<ResultItem> getResults() {  
 ArrayList<ResultItem> items = **new** ArrayList<ResultItem>();  
 **for** (WebElement item : **resultItems**)  
 {  
 String[] coord = item.findElement(By.*cssSelector*(**".address a"**)).getAttribute(**"data-coords"**).split(**","**);  
 String name = item.findElement(By.*cssSelector*(**".sr-hotel\_\_name"**)).getText();  
 ResultItem product = **new** ResultItem();  
 product.setCoordinates(**new** Location(Double.*parseDouble*(coord[1]), Double.*parseDouble*(coord[0])));  
 product.setName(name);  
 items.add(product);  
 }  
 **return** items;  
}

1. Place Panel onto the Page with a proper parent element locator (if needed).

@FindBy(how = How.***ID***, using = **"search\_results\_table"**)  
**private** WebElement **searchResultPanel**;  
**public** SearchResultsPanel **searchResults** = **new** SearchResultsPanel(**searchResultPanel**);

### How to use your new Page

1. Declare the Page in your Test or Steps class.

**public class** ResultsPageSteps {  
  
 **private** ResultsPage **resultsPage**;

}

1. Create Page instance in steps constructor or test configuration method:

**public class** ResultsPageSteps {  
  
 **private** ResultsPage **resultsPage**;  
  
 **public** ResultsPageSteps()  
 {  
 **resultsPage** = **new** ResultsPage();  
 }

1. Access all public fields, properties and methods of the page as with a regular class.

## Steps

### Overview

**Step** is a class, which contains mini-scenarios related to certain flow. The pages, which are involved in the flow, are the fields of this class. The scenarios are its public methods.

Example: ResultsPageSteps can contain multiple methods related to login procedure:

**public class** ResultsPageSteps {  
  
 **private** ResultsPage **resultsPage**;  
  
 **public** ResultsPageSteps()  
 {  
 **resultsPage** = **new** ResultsPage();  
 }  
  
 @Step(**"Verify results have correct location with acceptable deviation."**)  
 **public void** verifyResultsHaveCorrectLocation(**double** deviation, Location expectedLocation) {  
  
 List<Location> location = **resultsPage**.**searchResults**.getResults()  
 .stream().map(x->x.getCoordinates()).collect(Collectors.*toList*());  
  
 Double expectedLatitude = expectedLocation.getLatitude();  
 List<Double> actualLatitudes = location.stream().map(x -> x.getLatitude()).collect(Collectors.*toList*());  
 actualLatitudes.forEach(v ->  
 Assert.*assertTrue*(expectedLatitude - deviation <= v && v <= expectedLatitude + deviation,  
 **"Locations has correct Latitude"**));  
  
 Double expectedLongitude = expectedLocation.getLongitude();  
 List<Double> actualLongitudes = location.stream().map(x -> x.getLongitude()).collect(Collectors.*toList*());  
 actualLongitudes.forEach(v ->  
 Assert.*assertTrue*(expectedLongitude - deviation <= v && v <= expectedLongitude + deviation,  
 **"Locations has correct Longitude "**));  
 }