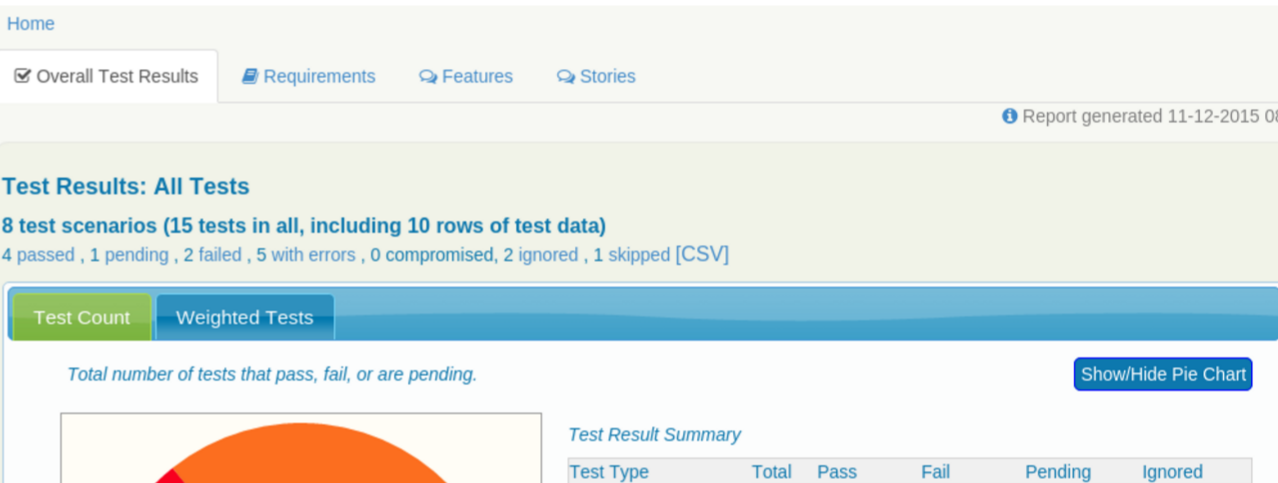
1. Serenity BDD helps to see not only the current state of the tests but also what requirements have been or not tested
2. Serenity BDD is commonly used for both automated acceptance tests and regression tests
3. BDD is a development approach where team members(BA/product owner’s, development team, test team) explore and build a shared understanding of application requirements through conversations around examples)  
   Scenario:  
   Given:  
   When:  
   Then:
4. Serenity BDD performs:  
   a) Makes it easy to write, execute and report automated acceptance tests in a way BA’s, developers and testers relate to

b) Structure automated acceptance tests into steps and sub steps  
c) produce narrative style report

1. Aggregation Reports: Serenity BDD aggregation report can be organized into features stories, steps, scenarios/tests  
   “examples” in jbehave/cucumber have the same meaning as “test data” in junit  
   “scenario” in jbehave/cucumber is same as “test” in junit
2. Sample serenity BDD report will be displayed as below with tabs as “Overall test results”, “requirements”, “features”, “stories”  
   
3. Serenity report supports filtering option based on keyword search
4. Serenity BDD projects can be using build tools as “Maven”,”ant”, or “gradle” build tools, the configuring java or groovy build to use serenity bdd is just the matter of adding right dependencies and a task or plugin to generate aggregate reports
5. Next write a junit test to express the acceptance criteria that you would want to automate:

@RunWith(SerenityRunner.class)  
public class SerenityDemo{

@Managed(driver=”chrome”)  
 WebDriver driver;

@Steps  
 testClass testObject;

@Test  
 public void testfunc(){

//given  
testObject.func1();

//when  
testObject.func2();

//then

testObject.func3();

}

Public class testClassSteps extends testClass{  
 @Step(“Given Statement”)  
 public void func1(){  
 //statements  
 }

@Step(“When Statement”)  
 public void func2(){  
 //statements  
 }

@Step(“Then Statement”)  
 public void func3(){  
 //statements  
 }

}  
  
10) We can automate our acceptance criteria by writing test stories and scenarios using Jbehave BDD notation, sample as below:  
  
//story description  
Meta:  
@tag product:search  
Narrative:  
//give a brief overview about the background in a story  
Scenario: //decription about the scenario  
Given://given statement  
When://when statement  
Then: //Then statement  
  
Scenarios as above will be written in “.story” file

1. Junit test runner: Jbehave tests are run via a unit test runner by extending the “SerenityStories” class,   
    public class acceptanceTests extends SerenityStories{

}

When we run this test, serenity will try to run the Jbehave stories present under the default story location : “src/test/resources/stories”

1. Serenity supports simple directory based structure for organizing the requirements, standard structure uses three levels:  
   capabilities  
   feature  
   stories  
   A story is represented as : .story file:  
     
   +src  
    +test  
    +resources  
    +stories  
    +<high level description> [capability]  
    +< feature description>[feature]  
    +<user story1.story>  
    +<user story <n>.story>

+<second feature descriptipn>[feature]  
 +<userstory 1.story>  
 +userstory 2.story>

1. In case we prefer another hierarchy, we can override the “serenity.requirements.type” file as per the needs, by default the value for “serenity.requirements.type” is a comma separated value as “capability, feature, story”)
2. An optional but a useful feature of a jbehave story format is a narrative context that provides more information about the background and some additional information about each story, this narrative section appears in the serenity report, to help give information about the motivation and background of each user story  
     
   Narrative:  
   In order to <perform some functionality>  
   As a <some role>  
   I <have to perform some steps to get the functionality implemented>
3. It’s a good practice to enter narrative section not only for the user stories but also for the features/capabilities so that the requirements sections will have individual user stories with narrative for better clarity on the acceptance criteria
4. Customizing requirements module:  
    We extend the serenity requirements support by implementing the “RequirementsTagProvider” interface

<implement and paste code here>

Once the implementation is complete we need to create a new directory under “src/main/resources/META-INF/services” named as “net.thucydides.core.statistics.service.Tagprovider” and enter the fully qualified name of the “RequirementsTagProvider” implementation

1. Story Meta Data:  
   We can user Jbehave Meta tag to provide additional info to Serenity about the test  
    Sample: For Junit

@Meta:  
 @driver  
 @Issue  
 @Issues  
 @Title

For “serenity.properties” contains url as :  
 serenity.issue.tracker.url=https://example.server.com/cases/view/{0}

1. Running all tests in a single browser window:  
    All web tests can be run in a single browser window using by either setting the “restart.browser.each.scenario” to either true or false or by  
   “runSerenity.inASingleSession();”  
    public class JbehaveTestCase extends SerenityStories{

public JbehaveTestCase(){  
 runSerenity.inASingleSession();  
}  
}

1. By default serenity will convert the test method names into a readable form in the reports, i.e both “camelCasedMethods()” and “methods\_with\_underscores()” into a form with spaces
2. @Managed  
    WebDriver driver -> By default will launch Firefox Browser  
     
   @Managed(driver=”chrome”) -> will launch chrome browser  
   Other values are “iexplorer”,”phantomjs”,”htmlunit”  
     
   @Managed(uniqueSession=true) -> default value is “false”  
     
   @Managed(clearCookies=BeforeEachTest) -> BeforeEachTest is default value, possible value is “never”
3. Writing Serenity Page Objects:  
   Page Objects are a way of isolating implementation details of a web page inside a class
4. Switching to another Page:  
   @DefaultUrl(“<http://google.com>”)  
   public class LoginPage extends PageObject{

RedirectToPage redirectObject= this.switchToPage(RedirectToPage.class);

redirectObject.open();  
}

1. Uploading a file:  
    public class newPage extends PageObject{

@FindBy(id=”some\_text”)  
 WebELementFacade logoField;  
 public newPage(WebDriver driver){  
 super(driver);  
 }  
// If the file is in the same package as the current class  
 InputStream fileName=getClass().getResourceAsStream(“path\_to\_the\_file”);  
 public void uploadFile(String fileName){  
 upload(filename).to(logoField);

}  
}

1. Using Pages in Step library:  
    If we need to use a pageObject in one of our Steps, we need to create a variable of type PageObject in step library:  
   @Step  
   @At(“[http://.\*.some\_url.com](http://.*.some_url.com)”)  
   public class demo\_class extends PageObject{  
    currentPageAt(verifyPage.class);  
    demo\_class page;  
   .  
   .  
   .  
   }
2. Opening a page:  
    @DefaultUrl(“<http://some_url>”)  
    @NamedUrls(  
    {  
    @NamedUrl(name=”open.issue”,url=”/issues/{1}  
    }  
   class demoPage extends PageObject{  
    demoPage page;

page.open(“open.issue”,withParameters(“ISSUE-1”)); //this will open “<http://some_url/issues/ISSUE-1>” web page  
}

1. Checking whether webElements are visible  
    WebElementFacade is similar to WebElement with some additional features  
     
   public class FindAnElement extends PageObjects{

WebElementFacade searchButton;  
 public boolean searchButtonVisible(){  
 return searchButton.isVisible(); //waits for certain timeinterval due to AjaxCalls  
 return searchButton.isCurrentlyVisible(); //verifies and returns immediately  
 return searchButton.isEnabled(); //verifies if the element is enabled  
 return searchButton.isPresent(); //verifies if element is present but not visible  
 return serachButton.isNotPresent();//verifies if element is not present  
 }

public WebElementState searchButton(){  
 return element(searchButton);  
 }  
}  


1. Select List implementation:  
    <select id=”colour”>  
    <option value=”Red”>Red<\option>  
    <option value=”blue”>Blue<\option>  
    <option value=”Green”>Green<\option>  
   <\select>  
     
   public class selectFromList extends PageObject{  
    @FindBy(id=”colour”)  
    WebElementFacade colourDropDown;  
    public selectValuefromList(){  
    colourDropDown.selectByVisibleText(“Blue”);  
    assertThat(colourDropDown.getSelectedVisibleTextValue(), is(“Blue”));  
     
    colourDropDown.selectByValue(“blue”);  
    assertThat(colourDropDown.getSelectedValue(), is(“blue”));  
    colourDropDown.selectByIndex(2);  
    assertThat(colourDropDown.getSelectedValue(),is(“Green”));  
    }  
   }
2. WebElement Focus, Enable, Visibility:  
    public class demoWebElements{  
    @FindBy(id=”some\_value”)  
    WebElement field;  
    @FindBy(id=”some\_other\_value”)  
    WebElement buttonField;  
    public boolean checkField(){  
    return field.hasFocus();  
    }  
    buttonField.waitUntilEnabled();  
    butonField.waitUntilDisabled();  
    buttonField.waitUntilVisible();  
    buttonField.waitUntilNotVisible();

}

1. Asynchronous Pages:  
    Asynchronous pages are those whose fields or data is not visible / displayed when the page is loaded completely, in such scenarios we need to wait until a stipulated time interval for those to be visible, as such serenity provides some methods in pageObject base class to help with such scenarios  
    Checking whether an element is visible:  
   @FindBy(xpath=”//some\_value”)  
    WebElement field;  
    field.isElementVisible();  
    field.shouldBeVisible();  
   //If the WebElement does not appear immediately  
   @FindBy(xpath=”//some\_value”)  
    WebElement field;  
    field.WaitForRenderedElement();  
    field.WaitForrenderedElementsToBePresent();
2. TimeOut Scenarios:  
     
   Implicit Waits:  
     
     
   Explicit Waits:
3. Fluent matcher expressions:
4. Working with HTMLTables:
5. Analyzing Test Outcomes: