Topics

1

1. What is Data-Driven Testing?
2. Why Data-Driven Testing?
3. Data parameterization
4. Java Database Connectivity(JDBC)
5. Adding data to the Database
6. Adding JDBC Connector dependency
7. Configuration of PostgreSQL JDBC Connector
8. Test SQL Connector to read data
9. Use SQL Connector in mortgage calculator
10. Data Provider in TestNG
11. Use TestNG Data Provider in mortgage calculator

Data-Driven Testing

2

What is Data-Driven Testing?

Data Driven Testing is a software testing method in which test data is stored in table or spreadsheet format. Data driven testing allows testers to input a single test script that can execute tests for all test data from a table and expect the test output in the same table. It is also called table-driven testing or parameterized testing.

In Data Driven automation testing input values are read from data files and stored into variables in test scripts. It enables testers to build both positive and negative test cases into a single test. Input data in data driven framework can be stored in single or multiple data sources like .xls, .xml, .csv and databases.

Why Data Driven Testing?

Data Driven Testing is important because testers frequently have multiple data sets for a single test and creating individual tests for each data set can be time-consuming. Data driven testing helps keeping data separate from test scripts and the same test scripts can be executed for different combinations of input test data and test results can be generated efficiently.

Data Parameterization

3

Parameterization in Selenium:

Parameterization in Selenium is a process to parameterize the test scripts in order to pass multiple data to the application at runtime. It is a strategy of execution which automatically runs test cases multiple times using different values. The concept achieved by parameterizing the test scripts is called Data Driven Testing.

Data Provider in TestNG:

Data Provider in TestNG is a method used when a user needs to pass complex parameters. Complex Parameters need to be created from Java such as complex objects, objects from property files or from a database can be passed by the data provider method. The method is annotated by @DataProvider and it returns an array of objects.

Java Database Connectivity (JDBC)

4

Introduction to JDBC

The JDBC(Java Database Connectivity) API is a Java API that can access any kind of tabular data, especially data stored in a Relational Database.

JDBC helps you to write Java applications that manage these three programming activities:

- Connect to a data source, like a database
- Send queries and update statements to the database
- Retrieve and process the results received from the database in answer to your query

Adding Data to the Database

5

Prerequisite: Install PostgreSql DBMS into your computer. Follow the lecture notes for installation process.
Database Software: PostgreSQL
Download Link: https://www.enterprisedb.com/downloads/postgres-postgresql-downloads download the following installer "Windows (x86, 64-bit), Version 13 and install it.

- From pgAdmin create a database called "mortgage_calculator"
- Now connect to the database by clicking on the ">" to expand the database
- Now open the query tool from Tools > Query Tool
- Create a table called "monthly_mortgage" using the following query -
- Insert the following data inside the table using the following query -

```
create table monthly_mortgage(
    id integer primary key,
    homevalue integer,
    downpayment integer,
    downpaymentoption varchar(5),
    loanamount integer,
    interestrate varchar(5),
    loanterm integer,
    propertytax integer,
    pmi varchar(5),
    homeownerinsurance integer,
    monthlyhoa integer,
    loantype varchar(20),
    buyorrefi varchar(5),
    totalmonthlypayment
varchar(20)
)
```

```
insert into monthly_mortgage values
(1,300000,60000,'$',240000,'3',30,5000,'0.5',1000,100,'FHA',
'Buy', '1,611.85');
```

Adding JDBC Connector dependency

6

For Maven projects: Add the latest dependency of PostgreSQL JDBC connector to your project's pom.xml file copied from the following website https://mvnrepository.com/artifact/org.postgresql/postgresql

```
<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <version>42.3.1</version>
</dependency>
```

- ❑ Now Save pom.xml file and reload the pom.xml file
- ☑ After reloading expand external libraries from Project and check that postgresql is listed.

Configuration of PostgreSQL JDBC connector

7

Now you need to prepare the following:

- The address of the PostgreSQL <database server or host> e.g., **localhost**
- The <port> is optional. Default port number is **5432**
- The database name e.g., **mortgage_calculator**
- The username and password of the account that you will use to connect to the database.

For this information, you can construct the PostgreSQL JDBC connection string by using the following format:

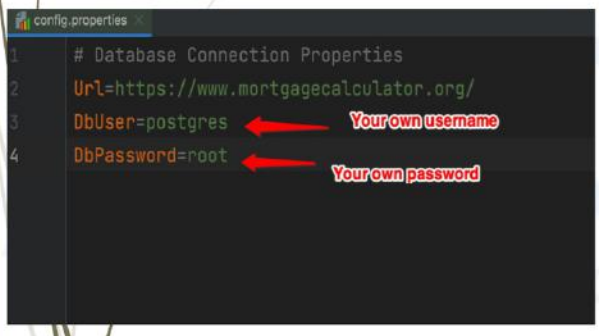
```
jdbc:postgresql://<database_host>:<port>/<database_name>
```

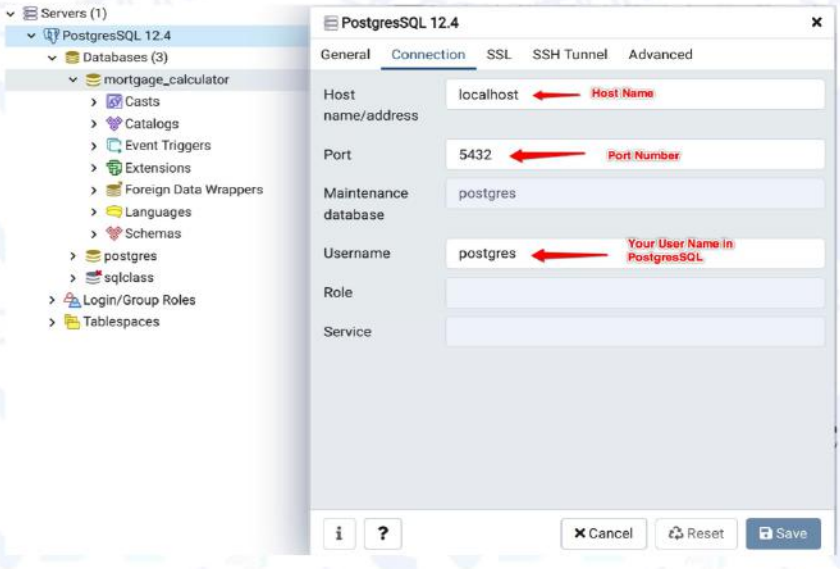
```
private static final String Url = "jdbc:postgresql://localhost:5432/mortgage_calculator";
private static final String User = "postgres";
private static final String Password = "<add your password>";
```

Update config.properties File

8

- Add the following information into your config.properties file which is under resources
- Also in PostgresSQL you can find the connection properties





Configuration of PostgreSQL JDBC connector

9

- Create a class called “SqlConnector” under utilities package.
- Declare Logger and DB Connection variables in global scope
- Create a method called connect and it’s return type is Connection Object
- Create a method called readData and it’s return type is ResultSet Object;

```
package utilities;

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import java.sql.*;
```

```
public class SqlConnector {
    private static final String Url = "jdbc:postgresql://localhost:5432/mortgage_calculator";
    private static final String User = ReadConfigFiles.getPropertyValues( propName: "DbUser");
    private static final String Password = ReadConfigFiles.getPropertyValues( propName: "DbPassword");

    private static final Logger LOGGER = LogManager.getLogger(SqlConnector.class);

    /**
     * Connect to the postgreSQL database
     * @return a Connection Object
     */

    private static Connection connect() {
        Connection conn = null;
        try {
            conn = DriverManager.getConnection(Url,User,Password);
        } catch (SQLException e) {
            LOGGER.error("SQL Connection Exception" + e.getMessage());
        }
        return conn;
    }
```

```
/**
 * Reading Data from the database
 * @param SQL is the method parameter to accept the SQL query
 * @return a ResultSet object
 * @throws SQLException
 */
public static ResultSet readData(String SQL) throws SQLException {
    ResultSet rs = null;
    Connection conn = null;
    try {
        conn = connect();
        LOGGER.debug("Connection object value: " + conn);
        Statement stmt = conn.createStatement();
        rs = stmt.executeQuery(SQL);

    } catch (SQLException e) {
        LOGGER.error("SQL ResultSet Exception" + e.getMessage());
    } finally {
        conn.close();
    }
    return rs;
}
```

Test SqlConnector to read data

10

- Create a package called “parameters” under src > test > java
- Create a class called “parameters” under this package.
- Create a method called “testDatabaseValues” to test that we can successfully read value from the database table through sql query

Note: Make sure you start the postgresql server before running the automation test.

```
The home price is: 300000
The down payment is: 60000
The loan amount is: 240000
```

```
package parameters;

import org.testng.annotations.Test;
import utilities.SqlConnector;

import java.sql.ResultSet;
import java.sql.SQLException;

public class TestSqlParameters {
    @Test
    public void testDatabaseValues() {
        ResultSet resultSet = SqlConnector.readData( SQL: "select * from monthly_mortgage");
        try {
            while (resultSet.next()) {
                System.out.println("The home price is: " + resultSet.getString( columnLabel: "homevalue"));
                System.out.println("The down payment is: " + resultSet.getString( columnLabel: "downpayment"));
                System.out.println("The loan amount is: " + resultSet.getString( columnLabel: "loanamount"));
            }
        } catch (SQLException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Use SqlConnector in mortgage calculator

11

- Create a package called “mortgage_calculator_parameterized” under src > test > java > automation_test
- Under this package create a class called “CalculateMonthlyPaymentParameterized” and now add the following code.

```
package automation_test.mortgage_calculator_parameterized;

import command_providers.ActOn;
import io.github.bonigarcia.wdm.WebDriverManager;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.annotations.AfterMethod;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;
import page_objects.Home;
import utilities.ReadConfigFiles;
import utilities.SqlConnector;

import java.io.IOException;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Properties;
```

```
public class CalculateMortgageRateParameterized {
    private static final Logger LOGGER = LogManager.getLogger(CalculateMortgageRateParameterized.class);
    WebDriver driver;

    @BeforeMethod
    public void browserInitialization() {
        WebDriverManager.chromedriver().setup();
        driver = new ChromeDriver();
        LOGGER.info("--------------------Test Name: Calculate Monthly Payment----------");

        String browserUrl = ReadConfigFiles.getPropertyValues( propName: "Url");
        ActOn.browser(driver).openBrowser(browserUrl);
    }
```


Use SqlConnector in mortgage calculator

12

```
@Test
public void calculateMonthlyPayment(){
    String date = DateUtils.returnNextMonth();
    String[] dates = date.split( regex: "-");
    String month = dates[0];
    String year = dates[1];

    try {
        ResultSet rs = SqlConnector.readData( SQL: "select * from monthly_mortgage");
        while (rs.next()) {
            new Home(driver)
                    .typeHomePrice(rs.getString( columnLabel: "homevalue"))
                    .typeDownPayment(rs.getString( columnLabel: "downpayment"))
                    .clickOnDownPaymentInDollar()
                    .typeLoanAmount(rs.getString( columnLabel: "loanamount"))
                    .typeInterestRate(rs.getString( columnLabel: "interestrate"))
                    .typeLoanTermYears(rs.getString( columnLabel: "loanterm"))
                    .selectMonth(month)
                    .typeYear(year)
                    .typePropertyTax(rs.getString( columnLabel: "propertytax"))
                    .typePmi(rs.getString( columnLabel: "pmi"))
                    .typeHoi(rs.getString( columnLabel: "homeownersinsurance"))
                    .typeHoa(rs.getString( columnLabel: "monthlyhoa"))
                    .selectLoanType(rs.getString( columnLabel: "loantype"))
                    .selectBuyOrRefiOption(rs.getString( columnLabel: "buyorrefi"))
                    .clickOnCalculateButton()
                    .validateTotalMonthlyPayment(rs.getString( columnLabel: "totalmonthlypayment"));
        }
    } catch (SQLException e) {
        LOGGER.error(e.getMessage());
    }
}
```

```
    @AfterMethod
    public void quitBrowser() {
        LOGGER.info("----End Test Case:Calculate Monthly Payment----");
        ActOn.browser(driver).close();
    }
}
```

Data Provider in TestNG

13

We also can achieve Data Parameterization in TestNG with the help of @DataProvider annotation. For simple data storage we can easily utilize the @DataProvider annotation functionalities. Let's take a look at the following examples-

Data Provider with single value in same class:

- Create a class "DataProviderClass" under "parameters" package
- Create a method called "storeSingleValue" and this method will return a two dimensional array of Object.
- Add value inside the method within the `{ {"value"} }`
- Add @DataProvider annotation tag and provide a name of the data provider
- Create a test method called "readSingleValue" to read value from Data Provider method. The method should have a single parameter since we store single value individually..
- Add @Test annotation tag and provide data provider name like (dataProvider = "<data_provider_name>"

```
import org.testng.annotations.DataProvider;
import org.testng.annotations.Test;

public class DataProviderClass {
    @DataProvider(name = "SingleValue")
    public Object[][] storeSingleValue() {
        return new Object[][]{
                {"Rifat"},
                {"Mohammad"},
                {"Ashraf"}};
    }

    @Test(dataProvider = "SingleValue")
    public void readSingleValue(String name) {
        System.out.println("[Single Value] Name is: " + name);
    }
}
```

```
[Single Value] Name is: Rifat

[Single Value] Name is: Mohammad

[Single Value] Name is: Ashraf
```

Data Provider in TestNG

14

Data Provider with multiple values in same class:

- Under the class "DataProviderClass" create a method called "storeMultipleValues" and this method will return a two dimensional array of Object.
- Add value inside the method within the `{ {"value","value","value"} }`
- Add @DataProvider annotation tag and provide a name of the data provider
- Create a test method called "readMultipleValues" to read value from Data Provider method. The method should have a three parameter since we store multiple values.
- Add @Test annotation tag and provide data provider name like (dataProvider = "<data_provider_name>"

```
@DataProvider (name = "MultipleValues")
public Object[][] storeMultipleValues() {
    return new Object[][] {
            {"Rifat","Florida", 33018},
            {"Farid", "New York", 12457},
            {"Ashraf", "Bangladesh", 1207}
    };
}

@Test(dataProvider = "MultipleValues")
public void readMultipleValues(String name, String state, int zipCode) {
    System.out.println("[Multiple Value] Name is: " + name);
    System.out.println("[Multiple Value] State is: " + state);
    System.out.println("[Multiple Value] Zip Code is: " + zipCode);
}
```

```
[Multiple Value] Name is: Rifat
[Multiple Value] State is: Florida
[Multiple Value] Zip Code is: 33018


[Multiple Value] Name is: Farid
[Multiple Value] State is: New York
[Multiple Value] Zip Code is: 12457


[Multiple Value] Name is: Ashraf
[Multiple Value] State is: Bangladesh
[Multiple Value] Zip Code is: 1207
```

Data Provider in TestNG

15

Data Provider with multiple values in different class:

- Move the “DataProviderClass” under “utilities” package to make it reusable by the other class
- Under “parameters” package create a class called “TestDataProvider”
- Create a test method called “readMultipleValues” to read value from Data Provider method. The method should have a three parameter since we store multiple values.
- Add @Test annotation tag and provide data provider name and data provider class name like (dataProvider = “<data_provider_name>” and dataProviderClass = “<data_provider_class_name”>)

TestDataProvider.java

```java
package parameters;

import org.testng.annotations.Test;
import utilities.DataProviderClass;


public class TestDataProvider {
    @Test (dataProvider = "MultipleValues", dataProviderClass = DataProviderClass.class)
    public void readMultipleValues(String name, String state, int zipCode) {
        System.out.println("Name is: " + name);
        System.out.println("State is: " + state);
        System.out.println("Zip Code is: " + zipCode);
    }
}
```

```
Name is: Rifat
State is: Florida
Zip Code is: 33018


Name is: Farid
State is: New York
Zip Code is: 12457


Name is: Ashraf
State is: Bangladesh
Zip Code is: 1207
```

Use TestNG Data Provider in Mortgage Calculator

16

- Under the class “DataProviderClass” create a method called “storeRealAprRates” and this method will return a two dimensional array of Object.
- Add value inside the method within the `{ {“value”,“value”,“value”, “value”} }`
- Add @DataProvider annotation tag and provide a name of the data provider

```java
@DataProvider (name = "RealAprRates")
public Object[][] storeRealAprRates() {
    return new Object[][] {
            {"200000","15000","3","3.130%"}
    };
}
```

Use TestNG Data Provider in Mortgage Calculator

17

- Under the package called “mortgage_calculator_parameterized”create a class called “CalculateRatesParameterized” and now add the following code.
- For the test method Add @Test annotation tag and provide data provider name and data provider class name like (dataProvider = “<data_provider_name>” and dataProviderClass = “<data_provider_class_name”>)
- The method should have a four parameter since we store four values in single row.

```
public class CalculateRealAprRateParameterized {
    private static final Logger LOGGER = LogManager.getLogger(CalculateRealAprRateParameterized.class);
    WebDriver driver;

    @BeforeMethod
    public void openBrowser() {
        WebDriverManager.chromedriver().setup();
        driver = new ChromeDriver();
        LOGGER.info("--------------------Test Name: Calculate Real APR----------");

        String browserUrl = ReadConfigFiles.getPropertyValues( propName: "Url");
        ActOn.browser(driver).openBrowser(browserUrl);
    }

    @Test(dataProvider = "RealAprRates", dataProviderClass = DataProviderClass.class)
    public void calculateRealApr(String homePrice, String downPayment, String interestRate, String expectedApr) {
        new Home(driver)
                .mouseHoverToRates() NavigationBar
                .navigateToRealApr() RealApr
                .waitForPageToLoad()
                .typeHomePrice(homePrice)
                .clickDownPaymentInDollar()
                .typeDownPayment(downPayment)
                .typeInterestRate(interestRate)
                .clickOnCalculateButton()
                .validateRealAprRate(expectedApr);
    }

    @AfterMethod
    public void closeBrowser() {
        LOGGER.info("----End Test Case:Calculate Real APR----");
        ActOn.browser(driver).close();
    }
```

18

References

Learning materials is prepared with the help of following sources

- https://www.tutorialspoint.com/mongodb/mongodb_java.htm
- https://www.guru99.com/data-driven-testing.html
- https://www.guru99.com/parameterization-using-xml-and-dataproviders-selenium.html#:~:text=Parameterization%20in%20Selenium%20is%20a,multiple%20times%20using%20different%20values.
- https://www.postgresqltutorial.com/postgresql-jdbc/connecting-to-postgresql-database/
- https://www.postgresqltutorial.com/postgresql-jdbc/
- https://www.toolsqa.com/testng/testng-dataproviders/