PDSL Gherkin Quickstart

Table of Contents

If you want to make your own grammar instead of using gherkin, follow this guide instead.

Gherkin is a protocol developed by the Cucumber framework. It can be very powerful for clearly and concisely specifying tests in natural language. PDSL supports Gherkin natively.

This tutorial assumes you are using maven as your build system.

NOTE	You can view the source code for this quickstart at https://github.com/google/polymorphicDSL/tree/main/examples/quickstart

Add PDSL and ANTLR to your POM.xml file

```
<dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.13.2</version> <!-- NOTE: Only JUnit4 is currently supported -->
      <scope>test</scope>
</dependency>
<dependency>
    <groupId>com.google.pdsl</groupId>
    <artifactId>pdsl</artifactId>
    <version>1.2.1</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.antlr</groupId>
    <artifactId>antlr4-runtime</artifactId>
    <version>4.9.1</version>
</dependency>
```

You will probably also want to use the ANTLR plugin which will generate java source code for you so you won't need to do that manually. The rest of this quickstart assumes that you use this helpful tool:

```
<plugin>
    <groupId>org.antlr</groupId>
    <artifactId>antlr4-maven-plugin</artifactId>
    <version>4.9.1</version>
    <configuration>
        <arguments>
            <argumenent>-visitor</argumenent>
        </arguments>
        <sourceDirectory>${basedir}/src/test/antlr4</sourceDirectory>
    </configuration>
    <executions>
        <execution>
            <goals>
                <goal>antlr4</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

NOTE	We configured the grammars to be under the src/test/antlr4 directory. By default this plugin will set it to src/main/antlr4 instead.

Write a feature file

Save this file to src/test/resources/features as *PdslQuickstart.feature*

```
Feature: My first PDSL test

    Scenario: Simple test
        Given I have "1" pickle
        Then I have more than "0" pickles
```

Write a Lexer

The lexer is written in ANTLR. It's nice because we are able to share regex across projects and even across different programming languages if we want to. You can think of the lexer as a dictionary with all the words you're allowed to use.

```
lexer grammar MyFirstLexer;

GIVEN_I_HAVE: WS* 'Given I have "';
THEN_I_HAVE_MORE_THAN: WS* 'Then I have more than "';
PICKLES: '" pickle' 's'? WS*;
NUMBER: [0-9]+;
fragment WS: [\r\n\t ]+;
```

It is important that every *token* (such as *PICKLES*) start with an uppercase letter in the lexer. You can kind of think of the tokens as variable names for your regular expressions.

You'll notice the *WS* (whitespace) token has the word *fragment* in front of it. That means you aren't allowed to use that as a word by itself, but you can combine it with other words. For example, in English the suffixes "-ing" or "-er" are not words, but are important parts of words such as "painter" or "painting".

The practical value of fragments is that you can avoid ambiguity with generic tokens (which in this case is any form of whitespace). Another consequence of this is that you cannot directly reference fragment tokens in the parser (which sometimes is helpful).

Save this as src/test/antlr4/com/pdsl/quickstart/MyFirstLexer.g4

The ANTLR4 plugin will generate classes that start with *com.pdsl.quickstart* because that is the directory structure we saved it under.

Write a Parser

The parser is where you specify how you can combine the words or phrases you've defined together.

This allows you to make sure the phrases written make sense. Just like natural language, there are rules you make to prevent them being combined in nonsensical ways: Written you this to phrases sense make sure allows the make.

```
parser grammar MyFirstParser;

options{tokenVocab=MyFirstLexer;}

givenUserHasSpecifiedPickles: GIVEN_I_HAVE NUMBER PICKLES;
thenUserHasMoreThanSpecifiedPickles: THEN_I_HAVE_MORE_THAN NUMBER PICKLES;

polymorphicDslAllRules: (givenUserHasSpecifiedPickles |
thenUserHasMoreThanSpecifiedPickles)+;
```

It is important that every *rule* (such as *givenUserHasSpecifiedPickles*) needs to start with a lowercase letter. Every *rule* that use add will be used to generate a test method.

When you put the tokens next to each other it is like you are appending them. The "words" (or lexemes) in our language are:

1. Given I have "
2. " pickle
3. " pickles
4. Then I have more than "
5. Integers such as 0, 1, etc.

In the parser we've made "production rules" that say how we're allowed to combine these words. For example, a NUMBER in our language can only come after either *GIVEN_I_HAVE* or *THEN_I_HAVE_MORE_THAN*, but nowhere else.

You are also allowed to use logical operators, such as OR. We see this in *polymorphicDslAllRules* which says we can use either of the two rules we specified.

The *polymorphicDslAllRules* is a way for you to turn certain rules on or off. This becomes very important when you start sharing your grammar with other teams because it becomes easy to control which methods you care about executing in your particular context. This is the default *start rule* which can be configured in the PDSL framework to something else.

Save this as src/main/antlr4/com/example/MyFirstParser.g4

Generate the parser

To generate the source code run `mvn antlr4:antlr4`. This will create Java classes in the target directory. There are plugins for IDEs like IntelliJ and Eclipse that will do this for you.

After this the source files will exist under `target/generated-sources/antlr4/com/example` if you are curious and want to look at them.

You probably will want to mark target/generated-sources/antlr4 as a Generated Sources Root using your IDE. This will allow your intellisense to direct the generated code.

Implement a listener

This is analogous to creating step definitions in Cucumber, only you don't have to match the methods with the regexes (the parser file you wrote earlier did that for you).

```
package com.pdsl.quickstart;

import com.example.MyFirstParser;
import com.example.MyFirstParserBaseListener;

public class MyFirstPdslListener extends MyFirstParserBaseListener {
    private int numberOfPickles = -1;
    @Override
    public void
enterGivenUserHasSpecifiedPickles(MyFirstParser.GivenUserHasSpecifiedPicklesContext
ctx) {
        numberOfPickles = Integer.parseInt(ctx.NUMBER().getText());
    }
    @Override
    public void
enterThenUserHasMoreThanSpecifiedPickles(MyFirstParser.ThenUserHasMoreThanSpecifiedPic
klesContext ctx) {
        assert numberOfPickles > Integer.parseInt(ctx.NUMBER().getText());
    }

}
```

Implement the runner

The runner is where you execute the tests. You specify the interpreter to use and which files to read with it. Finally you specify the listener to use that will fire off methods every time a matching phrase is found.

```
package com.pdsl.quickstart;

import com.pdsl.runners.PdslGherkinApplication;
import com.pdsl.runners.PdslTest;
import com.pdsl.runners.junit.PdslGherkinJUnit4Runner;
import org.antlr.v4.runtime.tree.ParseTreeListener;
import org.junit.runner.RunWith;
import com.example.MyFirstLexer;
import com.example.MyFirstParser;

import javax.inject.Provider;

@RunWith(PdslGherkinJUnit4Runner.class)
@PdslGherkinApplication(
        resourceRoot = "src/test/resources/features"
)
public class PdslQuickstartTest {

    @PdslTest(
            parser = MyFirstParser.class,
            lexer = MyFirstLexer.class,
            listener = PdslQuickstartTest.MyFirstPdslProvider.class)
    public void myFirstPdslTest() {}

    // The provider builds the listener for the PDSL framework.
    // It must be static so that it can be seen by the underlying framework.
    // It also must have a default constructor (no parameters).
    // It can be convenient to make it a static class rather than put it in it's
    // own file so that the person running the test can understand if there are
    // special inputs or not (dependency injection, etc).
    public static class MyFirstPdslProvider implements Provider<ParseTreeListener> {

        @Override
        public ParseTreeListener get() {
            return new MyFirstPdslListener();
        }
    }
}
```

After this, simply run your tests using `mvn clean test`.

NOTE	If you noticed you made a mistake in your parser, make sure to run mvn antlr4:antlr4 to regenerate the parser, otherwise you might continue to get the same errors because you're using the old one! Some IDEs such as IntelliJ also have plugins you can download from their marketplace to create the "recognizer" by right-clicking the grammar files.

Next Steps

PDSL was designed to solve problems for more complicated than we've seen in this tutorial. If you want to create your own language for testing instead of using Gherkin like we did here, see this guide. Otherwise proceed to the gherkin scalability tutorial.