PDSL Custom Grammars

Table of Contents

Sometimes the best way to represent how your application behaves is through some format that no one has natively supported yet. PDSL natively supports gherkin. See the gherkin quickstart if you think gherkin can meet your use case.

A custom grammar

Suppose you made a website. Certain pages can only be accessed by priviledged user. Assume it is your job to make sure that it is possible to navigate from one page to all others a user has permission to visit.

Technically you can do something like this in gherkin:

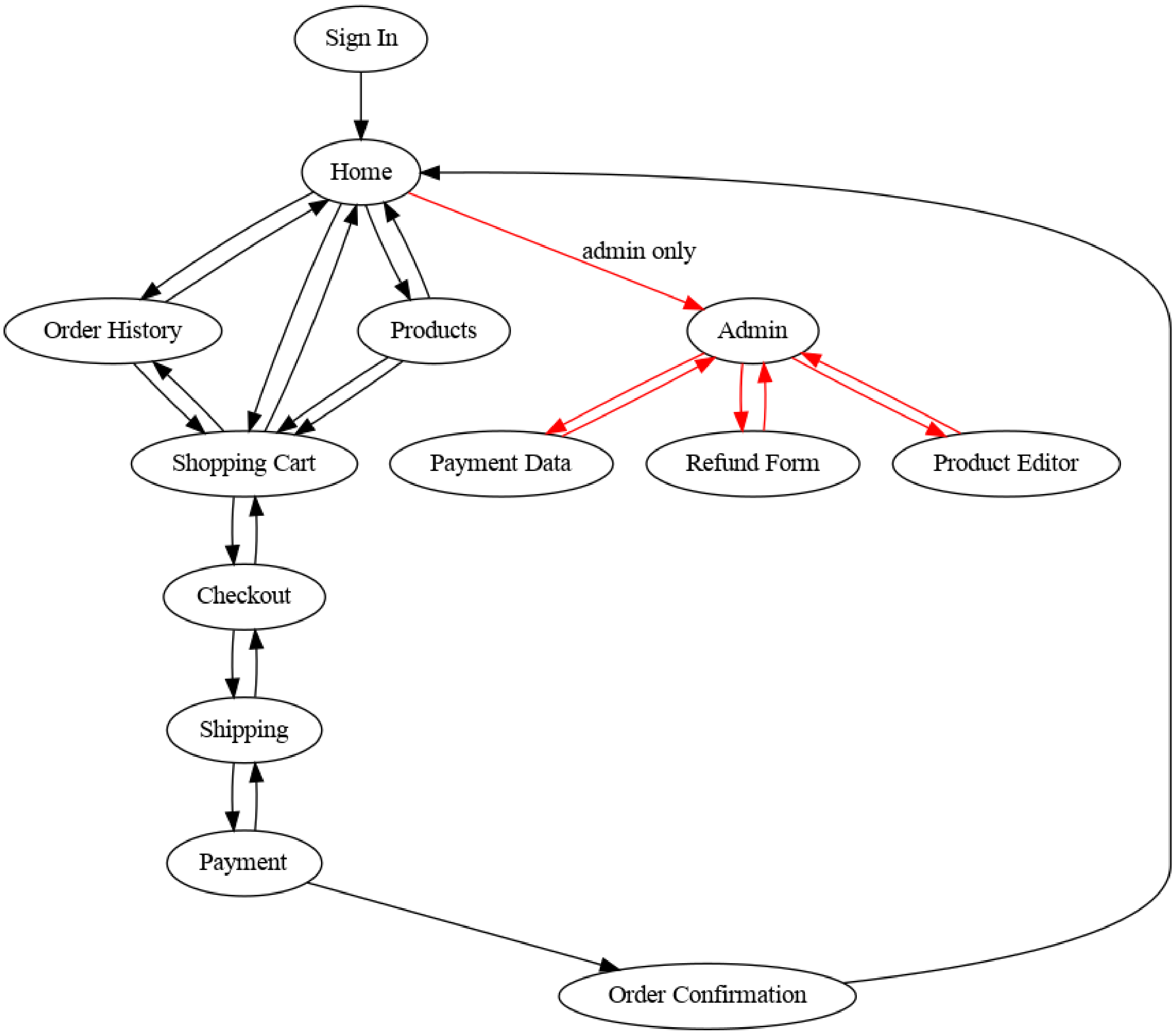
```
Feature: A Website

    Scenario: Users can navigate to all legal pages from any page they're on

        Given a user is on the sign in page
        Then the user can navigate to the home page after they sign in
```

But it can quickly become huge and difficult to reason with.

```
    Given a user is on the sign in page
    Then the user can navigate to the home page after they sign in

    #Going the other way
    Given a user is on the home page
    Then they can sign out and navigate to the sign in page

    Given a user is on the sign in page
    When the user navigates to the home page after they sign in
    Then the user can navigate to their order history

    # Going the other way
    Given the user is on the order history page
    Then they can navigate to the home page

    Given the user is on the order history page
    Then they can navigate to the home page
    Then they can sign out and navigate to the sign in page

    # Dozens or hundreds of similar tests
```

Gherkin is great, but it was designed for user acceptance tests and intended to be simple and easy to understand. While the requirements here are in natural language, it is still difficult to visualize what is going on as it gets bigger.

A better way to visualize these kinds of requirements would be with a graph:



At a glance this is far easier to understand. It is far more concise and the permutation of navigation paths is implied by the image itself. It also indicates that only admin users are able to access priviledged pages. Finally, we can tell that the user is unable to navigate back to previous pages in special areas, such as after they have provided payment.

Creating a Custom Grammar

PDSL requires you to do the following:

1. Define some test specification written in a domain specific language you make
2. Generate an interpreter for it using ANTLR
3. Convert the input into a generic *Test Specification* from 1 or more files.
4. Explain how many *Test Cases* should be created from each specification
5. Tell it where to find the test files you made (likely as files on a hard drive)

Some of that probably doesn't make sense to you right now, but it will as we go.

Use or create a DSL

For our example we're going to use a graph as a test case. The graph above was made using DOT. We're going to use a very simple subset of DOT as our DSL.

Create a Test with the DSL

The image above was made using the following DOT code:

digraph website {

```
    "Sign In" -> Home -> "Order History" -> "Shopping Cart"
    Home -> "Shopping Cart"
    "Shopping Cart" -> Home
    Home -> Products -> "Shopping Cart"
    Products -> Home
    Products -> "Shopping Cart"
    "Shopping Cart" -> "Order History"
    "Order History" -> Home
    "Shopping Cart" -> Checkout -> Shipping -> Payment -> "Order Confirmation" -> Home
    Home -> Admin [label="admin only" color="red"]
    Shipping -> Checkout
    Payment -> Shipping
    Checkout -> "Shopping Cart"
    Admin -> "Payment Data" [color="red"]
    "Payment Data" -> Admin [color="red"]
    Admin -> "Refund Form" [color="red"]
    "Refund Form" -> Admin [color="red"]
    Admin -> "Product Editor" [color="red"]
    "Product Editor" -> Admin [color="red"]
}
```

There really isn't much we need here. We define the links between web pages using `→` tokens, and we indicate a page is admin only using `[color="red"]`.

The above code will be our actual test case.

Create an Interpreter

Our DSL is understandable, but we need to be able to map some Java code to execute when it sees certain phrases in our DSL.

This is where ANTLR comes in. ANTLR is a parser generator we can use to rapidly create something that reads our DOT file and then fire off a Java method when we need it to. It will do a lot of work for us, all we need to do is tell it how to read the DOT file we made.

First we create a lexer (which is essentially a dictionary that specifies what all the words in our language can be).

```
lexer grammar DotNavigationLexer;

fragment WS: [\n\r\t ]+;
HEADER: 'digraph website {';
PAGE: (WS? '"' [a-zA-Z ]+ '"')  //In quoted text that allows a space
    | (WS? [a-zA-Z]+); //Otherwise a single word
LINK: WS '->';
ADMIN: WS ('[color="red"]' | '[label="admin only" color="red"]');
END_GRAPH: WS '}' WS? EOF;
```

After that we make a parser (which specifies how we’re allowed to combine the words).

```
parser grammar DotNavigationParser;

options{tokenVocab=DotNavigationLexer;}

dotFile: HEADER linkedPages+ END_GRAPH;
admin: ADMIN;
page: PAGE admin?;
link: LINK page;
linkedPages: page link+;
```

The most important rule in our parser is *dotFile*, which says we have a *HEADER* token, followed by 1 or more *linkedPages*, followed by the *END_GRAPH*.

This is going to be our *start rule* and *syntax check* rule that the PDSL framework will use. It is the entry point we use to recognize the file and phrases we’ll see.

Create a Listener

This is the part where we’ll fire off Java code when we hit the sentences we care about. ANTLR will generate code we can use for our listener.

Because we named our parser *DotNavigationParser* there will be a base class named

DotNavigationParserBaseListener with an empty, no-op implementation for each parser rule we made. We can just override the method(s) we care about. In our case we only care about one, the *linkedPages* rule which specifies all the possible navigation paths.

```
package com.pdsl.custom;

import com.pdsl.runners.PdslGherkinApplication;
import com.pdsl.runners.PdslTest;
import com.pdsl.runners.PdslConfiguration;
import com.pdsl.runners.RecognizedBy;
import com.pdsl.runners.junit.PdslGherkinJUnit4Runner;
import org.antlr.v4.runtime.tree.ParseTreeListener;
import org.junit.AfterClass;
import org.junit.runner.RunWith;
import com.example.DotNavigationLexer;
import com.example.DotNavigationParser;
import com.example.DotNavigationParserBaseListener;
import javax.inject.Provider;
import com.pdsl.specifications.FileDelimitedTestSpecificationFactory;
import com.pdsl.testcases.SingleTestOutputPreorderTestCaseFactory;
import com.pdsl.runners.junit.PdslJUnit4ConfigurableRunner;

import java.util.*;

/**
 * A listener that fires off Java code every time we encounter a phrase we care about
in the DOT file we are parsing.
 * In this case all we really care about is figuring out what pages link to other
pages, so we simply pay attention
 * to those phrases and ignore all the others.
 */
 public class CustomDotNavigationListener extends DotNavigationParserBaseListener {

    public Map<String, List<String>> getNavigationPath() {
        return navigationPath;
    }

    public Map<String, List<String>> getAdminOnly() {
        return adminOnly;
    }

    private final Map<String, List<String>> navigationPath = new HashMap<>();
    private final Map<String, List<String>> adminOnly = new HashMap<>();



        /**
         * Determines what pages another page can navigate to.
         *
         * In a real test we would probably use something like webdriver to actually
attempt to navigate to these
         * pages, or after we define the graph internally (which we do with the
variables navigationPath and adminOnly)
         * use an algorithm to traverse all possible paths.
         *
         * However since this is just a proof of concept we'll just find out what the
navigation paths are and spot
         * check whether it identified them.
         * @param ctx the parse tree
         */
        @Override
        public void enterLinkedPages(DotNavigationParser.LinkedPagesContext ctx) {
            if (ctx.page().admin() != null || ctx.link().get(0).page().admin() !=
null) {
                List<String> paths =
adminOnly.computeIfAbsent(ctx.page().PAGE().getText().trim(), (p -> new
ArrayList<>()));
                String pageToNavigateToText =
ctx.link().get(0).page().PAGE().getText().trim();
                paths.add(pageToNavigateToText);
            } else {
                List<String> paths =
navigationPath.computeIfAbsent(ctx.page().PAGE().getText().trim(), (p -> new
ArrayList<>()));
                paths.add(ctx.link().get(0).page().PAGE().getText().trim());
            }
            for (int i=1; i < ctx.link().size(); i++) {
                DotNavigationParser.PageContext page = ctx.link().get(i - 1).page();
                DotNavigationParser.PageContext navigateTo = ctx.link().get(i).page();
                String pageText = page.getText().trim();
                List<String> paths = page.admin() != null || navigateTo.admin() !=
null
                        ? adminOnly.computeIfAbsent(pageText, (p -> new
ArrayList<>()))
                        : navigationPath.computeIfAbsent(pageText, (p -> new
ArrayList<>()));
                paths.add(navigateTo.getText().trim());
            }
        }
 }
```

Interpret the Specifications

PDSL was designed to run tests on arbitrary grammars like the one we just made. However, some grammars might have things like loops or something (like the *Examples* tables in gherkin). We need to tell PDSL how to organize information in our test as we read it.

In our case we just have a single DOT file that can be read top to bottom. It has no nested tests within it or any fancy features. Because of that we can just use a built in feature of PDSL: the *FileDelimitedTestSpecificationFactory*.

What this does is produce one specification for each file PDSL reads (as opposed to many). To learn about specification factories in general or the other factories PDSL provides out of the box, see Specification Factories.

Determine how to produce Test Cases from the Specification

Our needs are simple: we read one DOT file and want to create one test case from it.

PDSL has a standard factory for this called *SingleTestOutputPreorderTestCaseFactory* . It is capable of doing some fancy things (such as running some sentences before a group of other sentences). We don't actually need any of that, but it will work just fine for our purposes.

To lear more about Test Case Factories in general see Test Case Factories.

How to find the Test Resources

PDSL needs to know where and how to look for our DOT file. Unless you tell it otherwise, PDSL will look in the file system for the test data. Since we have a DOT file on a hard drive this is perfect.

All we need to do is be able to tell PDSL *where* the files are.

Create the Test Runner

Since we now know everything we need, we can make the PDSL test runner:

```
package com.pdsl.custom;

import com.pdsl.runners.PdslGherkinApplication;
import com.pdsl.runners.PdslTest;
import com.pdsl.runners.PdslConfiguration;
import com.pdsl.runners.RecognizedBy;
import com.pdsl.runners.junit.PdslGherkinJUnit4Runner;
import org.antlr.v4.runtime.tree.ParseTreeListener;
import org.junit.AfterClass;
import org.junit.runner.RunWith;
import com.example.DotNavigationLexer;
import com.example.DotNavigationParser;
import com.example.DotNavigationParserBaseListener;
import javax.inject.Provider;
import com.pdsl.specifications.FileDelimitedTestSpecificationFactory;
import com.pdsl.testcases.SingleTestOutputPreorderTestCaseFactory;
import com.pdsl.runners.junit.PdslJUnit4ConfigurableRunner;

import java.util.*;

/**
 * A proof of concept demonstrating how to use a custom grammar to test a problem that
```

```
is better visualized through
 * an image produced from a DOT file.
 */
@RunWith(PdslJUnit4ConfigurableRunner.class)
@PdslConfiguration(
        testCaseFactoryProvider =
SingleTestOutputPreorderTestCaseFactory.DefaultProvider.class,
        specificationFactoryProvider =
FileDelimitedTestSpecificationFactory.DefaultProvider.class,
        dslRecognizerParser = DotNavigationParser.class,
        dslRecognizerLexer = DotNavigationLexer.class,
        resourceRoot = "../../documentation/images/graphviz",
        recognizerRule = "dotFile")
public class CustomGrammarTest {


    @PdslTest(
            includesResources = "website.dot",
            parser = DotNavigationParser.class,
            lexer = DotNavigationLexer.class,
            startRule = "dotFile",
            listener = CustomGrammarTest.MyCustomPdslProvider.class)
    public void myFirstCustomGrammarPdslTest() {}

    // 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 its
    // own file so that the person running the test can understand if there are
    // special inputs or not (dependency injection, etc).
    public static class MyCustomPdslProvider implements
Provider<CustomDotNavigationListener> {
        private static final CustomDotNavigationListener listener = new
CustomDotNavigationListener();
        @Override
        public CustomDotNavigationListener get() {
            return listener;
        }
    }

    /**
     * Do all the assertions after finding out what all the page links are.
     *
     * This is not what we would do in real life, but for the sake of demonstrating
that we can at least tell what some
     * page links would be, we'll verify we can find some expected navigation paths
after parsing everything.
     */
    @AfterClass
    public static void afterAll() {
        // After the parser interprets the DOT file we provided it stores the
information it finds internally.
        // We retrieve that information and check to see if it is doing what we'd
expect
        Map<String, List<String>> navigationPath =
MyCustomPdslProvider.listener.getNavigationPath();
        Map<String, List<String>> adminOnly =
MyCustomPdslProvider.listener.getAdminOnly();
        assert navigationPath.get("\"Sign In\"").size() == 1;
        assert navigationPath.get("Home").containsAll(Set.of("\"Shopping Cart\"",
"\"Order History\""));
        assert adminOnly.get("Admin").containsAll(Set.of("\"Payment Data\"", "\"Refund
Form\"", "\"Product Editor\""));
        assert adminOnly.get("Admin").size() == 3;
    }
}
```

At this point we're done! We've made a custom grammar out of thin air and gave PDSL enough information it was able to extract the information we care about. While this is just a simple demo, converting this to a practical, real world framework doesn't require much additional effort.