Санкт-Петербургский государственный университет

Направление Математическое обеспечение и администрирование информационных систем
Кафедра Информационно-аналитических систем

Шавкунова Дарья Дмитриевна

Разработка библиотеки тестирования облачного сервиса для баз данных

Бакалаврская работа

Научный руководитель:
канд. физ-мат наук Михайлова Е. Г.

Рецензент:
Егоров П. Б.

Санкт-Петербург
2016

SAINT-PETERSBURG STATE UNIVERSITY

Software and Administration of Information Systems
Sub-Department of Analytical Information Systems

Daria Shavkunova

Development of cloud service test library for databases

Graduation Thesis

Scientific supervisor:
PhD Elena Mikhailova

Reviewer:
Pavel Egorov

Saint-Petersburg
2016

Оглавление

Введение

В настоящее время набирают популярность облачные платформы. Cloud Foundry - одна из таких платформ, помогающая разработчикам различного программного обеспечения (ПО). Но прежде чем облачные платформы использовать, нужно написать ПО для этих платформ.

Cloud Foundry - платформа с открытым кодом, поэтому разработчиков много. Хочется, чтобы ПО разных разработчиков вместе работало правильно и без сбоев. Для этого у платформы Cloud Foundry есть общие правила, которые обязаны учитывать все разработчики. Для проверки корректности ПО используют тестирование.

Разные разработчики проверяют свою работу по разному. Некоторые пытаются писать тесты на все возможные случаи. Другие вручную вводят тестовые данные и сверяют результат. Иногда разработчики смотрят только, чтобы запускалось ПО и считают, что все хорошо. Но у нас есть общие правила, которые обязаны выполняться. Значит можно сделать библиотеку с тестами на проверку этих правил, чтобы разработчики вносили параметры своего ПО и смотрели все ли работает. Это сократит время и силы на тестирование.

Формированию такой библиотеки и посвящена данная работа.

1. Постановка задачи

Конечная цель работы - это создание библиотеки тестов на проверку общих правил

1.1. Пути решения

- Изучить облачную платформу Cloud Foundry
- Изучить методы тестирования
- Выбрать инструменты разработки
- Разработать библиотеку

2. Обзор

2.1. Платформа Cloud Foundary

Cloud Foundary - это облачная платформа с открытым исходным кодом, которая позволяет разрабатывать приложения без сложности настройки инфраструктуры для них. Часть инфраструктуры платформа предоставляет(сеть, управления памятью и т.д.), а другие нужно подключать отдельно: базы данных, хранилища памяти.

Чтобы пользоваться внешними ресурсами, нужен специальный сервис, который состоит из ПО, предоставляющего этот ресурс, и Service Broker‘а, приложения обеспечивающего интеграцию сервиса внешнего ресурса в Cloud Foundary[4]. Service Broker должен содержать API[5]. API состоит из 5 методов:

- Список предоставляемых сервисов
- Создание Service Instance
- Удаление Service Instance
- Создание Service Binding
- Удаление Service Binding

Service Instance - это экземпляр ресурса, предоставляемого сервисом, а Service Binding отражает связь между приложением Cloud Foundary и Service Instance.

Эти методы и состовляют общие правила, которые обязаны учитывать разработчики при создании сервисов.

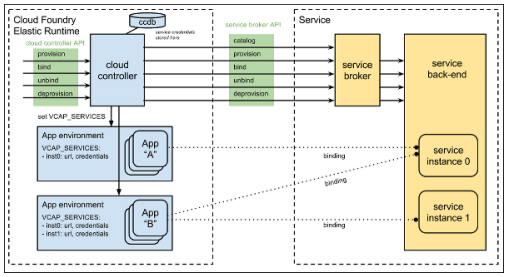

Рис. 1: Схема интеграции сервиса в Cloud Foundry.

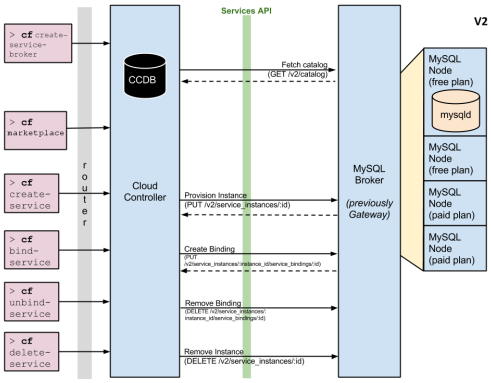

Рис. 2: Пример работы.

2.2. Что такое тестирование

Прежде чем тестировать, нужно определить что это такое с научной точки зрения. Определений существует несколько:

- Тестирование программного обеспечения - проверка соответствия между реальным и ожидаемым поведением программы, осуществляемая на конечном наборе тестов,выбранном определенным образом. [2]

- Тестирование - это одна из техник контроля качества, включающая в себя активности по планированию работ, проектированию тестов, выполнению тестирования и анализу полученных результатов. [1]

- Тестирование — это проверка соответствия программы требованиям, осуществляемая путем наблюдения за ее работой в специальных, искусственно созданных ситуациях, выбранных определенным образом. [11]

Интуитивно понятно, что они похожи между собой.

В тестировании выделяют и комбинируют между собой уровни и виды тестирования.

Уровни тестирования:

- Модульное - тестирование отдельных операций, методов, функций.
- Интеграционное - тестирование предназначено для проверки связи между компонентами, а также взаимодействия с различными частями системы.
- Системное - тестирование на уровне пользовательского интерфейса.

Иногда еще выделяют приемочное тестирование, которое определяет соответствует ли программное обеспечение указанным требованиям.

Виды тестирования:

- Функциональности
- Надежности
- Эффективности
- Практичности
- Сопровождаемости
- Мобильности

Так же есть еще одна классификация - разделение тестирования на два больших класса: тестирование методом черного ящика и тестирование методом белого ящика.

- При тестировании методом черного ящика мы не видим, что внутри ящика, мы не принимаем во внимание внутреннее устройство программы. При данном тестировании должны быть полностью покрыты все входные данные, комбинации входных данных и последовательности комбинаций входных данных.

- При тестировании методом белого ящика мы смотрим, как программа устроена внутри, и эту информацию используем при выполнении и, особенно, при проектировании тестов. При данном тестировании должны быть полностью покрыты все строки кода программы, ветви в коде программы и пути в коде программы.

В работе, в основном, будет рассмотрено модульное тестирование функциональности методом черного ящика.

2.3. Mock-объекты

В объектно-ориентированном программировании мы оперируем понятием класс - это описание объектов определенного типа.[13] Одни классы очень часто используют другие классы в своей работе. Иногда вызов одного метода может повлечь за собой цепочку классов вплоть до базы данных. Если в таких случаях произошла ошибка, то понять где именно - нетривиальная задача. В этом случае используются mock-объекты, предназначенные для симуляции поведения реальных объектов во время тестирования. Термин mock-объект может использоваться в двух смыслах: обозначает любые тест-дублеры (Test Doubles) или конкретный вид этих дублеров – mock-объекты. В работе термин используется во втором смысле. Тест-дублеры делятся на 4 группы:

- Dummy - пустые объекты, которые передаются в вызываемые внутренние методы, но не используются. Предназначены лишь для заполнения параметров методов.

- Fake - объекты, имеющие работающие реализации, но в таком виде, который делает их неподходящими для разработки в крупных компаниях.

- Stub-объекты, которые предоставляют заранее заготовленные ответы на вызовы во время выполнения теста и обычно не отвечающие ни на какие другие вызовы, которые не требуются в тесте. Также могут запоминать какую-то дополнительную информацию о количестве вызовов, параметрах и возвращать их потом тесту для проверки.

- Mock-объекты, которые заменяют реальный объект в условиях теста и позволяют проверять вызовы своих членов как часть системы или unit-теста. Содержат заранее запрограммированные ожидания вызовов, которые они ожидают получить. Применяются в основном для т.н. interaction (behavioral) testing.[10]

В данной работе использовались 2 вида тест-дублеров: stub-объекты и mock-объекты.

2.4. Класс MockMvc

Данная работа в большей степени ориентирована на проверку контроллеров, поэтому основной инструмент, использующийся при тестировании, это один из классов Spring Framework‘а MockMvc[12]. Работа с ним состоит из трех этапов:

- Построение mock - объекта
- Отправка HTTP-запроса контроллеру
- Анализ результатов

Чтобы сформировать MockMvc есть два пути:

- MockMvcBuilders.webAppContextSetup(webApplicationContext) .build() - автоматически вставляет конфигурацию Spring и вставляет WebApplicationContext в тест.
- MockMvcBuilders.standaloneSetup(controller).build() - не подгружает конфигурацию Spring[7].

Второй путь подходит для тестирования с помощью mock‘ов, в первом их использовать не стоит, так как Spring автоматически подгружает все классы. А если заменять их на дублеры, то нужно следить, чтобы он не обращался к уже загруженным, что сильно усложняет тесты. Поэтому в данной работе использовался второй подход.

Для отправки запроса требуется метод perform. У этого метода есть только один параметр - RequestBuilder, который и строит HTTP-запрос[3].

Анализ результата осуществляется с помощью метода andExpect, который предоставляет большие возможность для проверки полученного результата: проверка статуса(status.isOk()), корректность возвращаемого запроса в виде Json(jsonPath(”$.person.name”).equalTo(”Jason”)) и т.д.

2.5. Что используется сейчас

На данный момент существует уже несколько Service Broker‘ов для работы с определенной базой данных. Все тесты были написаны для каждого сервиса с нуля. При этом некоторая часть тестов совпадает. Например, корректно ли создан Service Instance или корректно ли он удален. Поэтому решено было создать библиотеку, в которой часть тестов будет уже реализована, а для их запуска требуется ввести только параметры спроектированного сервиса.

3. Выбор инструментов

Прежде чем создавать библиотеку, нужно выбрать инструменты. Так как проводилось тестирование проекта, в котором основными средствами были выбраны язык java и фреймворк (каркас) Spring Framework[9], то и в работе будут использоваться они же.

3.1. Выбор библиотеки тестирования

Самые популярные библиотеки mock-объектов - это Mockito[8] и EasyMock.

EasyMock

```
import static org.easymock.classextension.EasyMock.*;

List mock = createNiceMock(List.class);

expect(mock.get(0)).andStubReturn("one");
expect(mock.get(1)).andStubReturn("two");
mock.clear();

replay(mock);

someCodeThatInteractsWithMock();

verify(mock);
```

Mockito

```
import static org.mockito.Mockito.*;

List mock = mock(List.class);

when(mock.get(0)).thenReturn("one");
when(mock.get(1)).thenReturn("two");

someCodeThatInteractsWithMock();

verify(mock).clear();
```

Рис. 3: Пример работы.

EasyMock

```
Control control = createStrictControl();

List one = control.createMock(List.class);
List two = control.createMock(List.class);

expect(one.add("one")).andReturn(true);
expect(two.add("two")).andReturn(true);

control.replay();

someCodeThatInteractsWithMocks();

control.verify();
```

Mockito

```
List one = mock(List.class);
List two = mock(List.class);

someCodeThatInteractsWithMocks();

InOrder inOrder = inOrder(one, two);

inOrder.verify(one).add("one");
inOrder.verify(two).add("two");
```

Рис. 4: Число вызовов и проверка аргументов.

EasyMock

```
List mock = createNiceMock(List.class);

mock.clear();
expectLastCall().andThrow(new RuntimeException());

replay(mock);
```

Mockito

```
List mock = mock(List.class);

doThrow(new RuntimeException()).when(mock).clear();
```

Рис. 5: Порядок вызова.

EasyMock

```
List mock = createNiceMock(List.class);

mock.clear();
expectLastCall().times(3);

expect(mock.add(anyObject())).andReturn(true).atLeastOnce();

replay(mock);

someCodeThatInteractsWithMock();

verify(mock);
```

Mockito

```
List mock = mock(List.class);

someCodeThatInteractsWithMock();

verify(mock, times(3)).clear();
verify(mock, atLeastOnce()).add(anyObject());
```

Рис. 6: Void метод.

Общих библиотек тестирования тоже достаточно много. Одни из самых популярных: JUnit[6] и TestNG.

```
public class LocaleUtilsOldStyleTest extends Assert {
  private final Map<String, Locale> parseLocaleData = new HashMap<String, Locale>();

  @BeforeClass
  private void setUp() {
    parseLocaleData.put(null, null);
    parseLocaleData.put("", LocaleUtils.ROOT_LOCALE);
    parseLocaleData.put("en", Locale.ENGLISH);
    parseLocaleData.put("en_US", Locale.US);
    parseLocaleData.put("en_GB", Locale.UK);
    parseLocaleData.put("ru", new Locale("ru"));
    parseLocaleData.put("ru_RU_xxx", new Locale("ru", "RU", "xxx"));
  }

  @AfterTest
  void tearDown() {
    parseLocaleData.clear();
  }

  @Test
  public void testParseLocale() {
    for (Map.Entry<String, Locale> entry : parseLocaleData.entrySet()) {
      final Locale actual = LocaleUtils.parseLocale(entry.getKey());
      final Locale expected = entry.getValue();
      assertEquals(actual, expected);
    }
  }
}
```

Рис. 7: TestNG в стиле JUnit.

```
public class LocaleUtilsTest extends Assert {

  @DataProvider
  public Object[][] parseLocaleData() {
    return new Object[][]{
      {null, null},
      {"", LocaleUtils.ROOT_LOCALE},
      {"en", Locale.ENGLISH},
      {"en_US", Locale.US},
      {"en_GB", Locale.UK},
      {"ru", new Locale("ru")},
      {"ru_RU_some_variant", new Locale("ru", "RU", "some_variant")},
    };
  }

  @Test(dataProvider = "parseLocaleData")
  public void testParseLocale(String locale, Locale expected) {
    final Locale actual = LocaleUtils.parseLocale(locale);
    assertEquals(actual, expected);
  }
}
```

Рис. 8: TestNG.

В результате для проведения тестирования была выбрана библиотека Mockito (библиотека mock-объектов) и JUnit-4 (общая библиотека тестирования).

4. Реализация

4.1. Реализованные тесты для Catalog Controller

- catalogIsRetrievedCorrectly - правильно ли извлекается каталог

4.2. Реализованные тесты для Service Instance Controller

- serviceInstanceIsCreatedCorrectly - Service Instance создается корректно
- unknownServiceDefinitionInstanceCreationFails - неизвестен Service Definition
- duplicateServiceInstanceCreationFails - создание еще одного Service Instance
- badJsonServiceInstanceCreationFails - неправильная структура Json
- badJsonServiceInstanceCreationFailsMissingFields - в Json пропущены поля, обязательные к заполнению
- serviceInstanceIsDeletedSuccessfully - Service Instance удаляется корректно
- deleteUnknownServiceInstanceFailsWithA410 - удаление неизвестного Service Instance
- serviceInstanceIsUpdatedSuccessfully - Service Instance изменился успешно
- updateUnsupportedPlanFailsWithA422 - изменение неподдерживаемого плана

4.3. Реализованные тесты для Service Binding Controlle

- serviceInstanceBindingIsCreatedCorrectly - Service Binding создается корректно
- unknownServiceInstanceFailsBinding - неизвестен Service Instance
- duplicateBindingRequestFailsBinding - создание еще одного Service Binding
- invalidBindingRequestJson - неправильная структура Json
- invalidBindingRequestMissingFields - в Json пропущены поля, обязательные к заполнению
- serviceInstanceBindingIsDeletedSuccessfully - Service Binding удаляется корректно
- unknownServiceInstanceBindingNotDeletedAndA410IsReturned - удаление неизвестного Service Binding
- whenAnUnknownServiceInstanceIsProvidedOnABindingDeleteAnHttp-422IsReturned - удаление связи с неизвестным Service Instance

4.4. Реализованные тесты для Broker API Version

- noHeaderSent - нет начала запроса
- incorrectHeaderSent - неправильное начало запроса
- correctHeaderSent - корректное начало запроса
- BrokerApiVersionIsCorrectly - версия Broker API корректна

4.5. Тестирование

Тестирование библиотеки проводилось на готовом проекте. При несоответствии шаблона и работы сервиса, выдает невыполнение теста, в каком месте произошла ошибка и несовпадающие значения.

5. Заключение

- Разработана библиотека тестов
- Проведено тестирование на готовом проекте
- В библиотеке собраны не все возможные тесты, поэтому оставшуюся часть нужно писать разработчику для конкретного проекта

Список литературы

[1] URL: http://www.protesting.ru/testing/.

[2] Abran Alain. Guide to the software engineering body of knowledge. — IEEE Computer Society, 2004. — Google Books : https://books.google.ru/books?id=IKZQAAAAMAAJ.

[3] Class MockMvc. Документация. — URL: http://docs.spring.io/spring/docs/3.2.0.RC2/javadoc-api/org/springframework/test/web/servlet/MockMvc.html.

[4] Cloud Foundry. Overview. — URL: https://docs.cloudfoundry.org/services/overview.html.

[5] Cloud Foundry. Service Broker API. — URL: https://docs.cloudfoundry.org/services/api.html.

[6] JUnit. Официальный сайт. — URL: http://junit.org/junit4/.

[7] MockMvc + Mockito = Epic Tests. — 2014. — URL: https://myshittycode.com/2014/01/16/mockmvc-mockito-epic-tests/.

[8] Mockito. Официальный сайт. — URL: http://site.mockito.org/.

[9] Spring Framework. Официальный сайт. — URL: https://spring.io/projects.

[10] А. Кондуфоров. Введение в mock-объекты. Классификация. — 2008. — URL: http://merle-amber.blogspot.ru/2008/09/mock.html.

[11] Лупан Алексей. Основные положения тестирования. — 2010. — URL: https://testitquickly.com/2010/03/09/testing-basics-by-barancev/.

[12] Тестирование контроллеров с помощью MockMvc. — 2013. — URL: http://www.pvsm.ru/java/29182.

[13] Ш. Хабибуллин И. Самоучитель Java. — БХВ-Петербург, 2001.