GeekBrains

Java для тестировщиков, курс 2.

Работа с JSON

Обзор библиотеки Jackson.

Подключение библиотеки

На прошлом занятии мы рассмотрели подключение JAR библиотеки OkHttp в проект через настройки в интерфейсе Intellij IDEA. В случае с Jackson алгоритм абсолютно идентичен. В случае, если преподаватель проговорился и показал (пусть даже и в первом приближении) как пользоваться сборщиком Maven, в pom.xml в раздел dependencies нужно вставить следующий блок:

```
<dependency>
   <groupId>com.fasterxml.jackson.core</groupId>
   <artifactId>jackson-databind</artifactId>
   <version>2.11.2</version>
</dependency>
```

(2.11.2 - актуальная библиотека на момент написания методички)

на чем предварительную настройку можно считать оконченной.

Object Mapper

Рассмотрим класс, отвечающий за сериализацию Java-объектов в JSON и десериализацию объектов из JSON строки в Java-объекты.

В качестве подопытного примера, используем класс **Car:**

```
class Car {
   private String color;
   private String type;

   // геттеры и сеттеры обязательно должны быть определены, но не указаны для
экономии места
}
```

Экземпляры классов в JSON

Давайте посмотрим на первый пример сериализации объектов Java в JSON, используя метод **writeValue** класса **ObjectMapper**:

```
public static void main(String[] args) throws IOException {
   ObjectMapper objectMapper = new ObjectMapper();
   Car car = new Car("red", "BMW");
   objectMapper.writeValue(new File("car.json"), car);
}

car.json >> {"color":"red","type":"BMW"}
```

Методы **writeValueAsString** и **writeValueAsBytes** вернут сгенерированный JSON как строку или как массив байтов соответственно.

```
String jsonString = objectMapper.writeValueAsString(car);
byte[] jsonBytes = objectMapper.writeValueAsBytes(car);
```

JSON в экземпляры классов Java

Для чтения из различных источников используется метод **readValue.** Метод имеет огромное количество реализаций, отличающихся по источнику данных и поддерживает чтение из файлов, строк, URL, массива байтов и уже знакомых нам классов **Reader** и **InputStream**. Вторым аргументом указывается класс, экземпляр которого мы хотим десериализовать - в данном случае Car.class:

```
public static void main(String[] args) throws IOException {
   ObjectMapper objectMapper = new ObjectMapper();
   Car carFromFile = objectMapper.readValue(new File("car.json"), Car.class);
   System.out.println(carFromFile.toString());
}

console output >> Car{color='red', type='BMW'}
```

Важно! Если в классе, экземпляр которого мы хотим получить нет пустого конструктора, то мы получим **InvalidDefinitionException**, так как Jackson сначала создает пустой объект, и только затем наполняет его поля через сеттеры. Это означает что также **важно** создавать сеттеры и геттеры для сериализации/десериализации объектов.

Метод readValue, помимо прочего, может считывать целые массивы данных и преобразовывать их в списки, но конструкция вида

```
List<Car> carList = objectMapper.readValue(carsList, ArrayList<Car>.class);
```

не скомпилируется.

Для того чтобы получить класс обобщенного типа используется абстрактный класс **TypeReference**:

```
public static void main(String[] args) throws IOException {
   String carsList = "[{\"color\":\"red\", \"type\":\"BMW\"}," +
       " {\"color\":\"black\", \"type\":\"lada priora\"}]";
   ObjectMapper objectMapper = new ObjectMapper();
   List<Car> carList =
```

```
            objectMapper.readValue(carsList, new TypeReference<List<Car>>() {});
    System.out.println(carsList);
}


console output >> [{"color":"red", "type":"BMW"}, {"color":"black", "type":"lada
priora"}]
```

Конфигурирование сериализации-десериализации

Игнорирование неизвестных полей

Допустим, мы пишем десктопный клиент для получения погоды от сервера. Само собой, собственной метеостанции у нас нет, а следовательно, данные будем получать из сторонних источников. Работая со сторонними источниками всегда следует помнить одну вещь - формат данных может измениться, совместимость может быть утрачена. Вернемся к нашему примеру и смоделируем эту ситуацию в упрощенном виде: добавим к представлению класса **Car** поле “год”:

```
public static void main(String[] args) throws IOException {
    String jsonString = "{ \"color\" : \"white\", \"type\" : \"Volga\", \"year\"
:\"1970\" }";
    ObjectMapper objectMapper = new ObjectMapper();
    Car car = objectMapper.readValue(jsonString, Car.class);
    System.out.println(car);
}

console output >> Exception in thread "main"
com.fasterxml.jackson.databind.exc.UnrecognizedPropertyException: Unrecognized
field "year" (class ru.geekbrains.qa.lesson7.Car), not marked as ignorable (2
known properties: "type", "color"])
```

ObjectMapper не смог распознать поле “**year**” и бросил исключение - это стандартное поведение. Было бы здорово рассказать библиотечному классу “игнорируй все, что не сможешь понять” - и эта возможность, к нашему счастью, есть - экземпляр **ObjectMapper** может быть настроен через метод **configure()**

```
public static void main(String[] args) throws IOException {
    String jsonString = "{ \"color\" : \"white\", \"type\" : \"Volga\", \"year\"
:\"1970\" }";
    ObjectMapper objectMapper = new ObjectMapper();

   objectMapper.configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES,
false); // <- указание поведения при обнаружении неизвестных полей

    Car car = objectMapper.readValue(jsonString, Car.class);
    System.out.println(car);
}
```

```
console output >> Car{color='white', type='Volga'}
```

Есть и другой способ - с помощью аннотации **@JsonIgnoreProperties** определяем стратегию игнорирования неопознанных полей в json на уровне класса.

```
@JsonIgnoreProperties(ignoreUnknown = true)
class Car {
   private String color;
   private String type;

   ...
```

Игнорирование поля класса при сериализации

Но ситуация может быть и противоположной - в нашем классе может быть поле, полезное с точки зрения бизнес-логики приложения, но абсолютно бесполезное для отправки на сервер (в нашем случае - булево поле isActive). Помечая его аннотацией **@JsonIgnore** мы решаем свою проблему - поле не попадет в сериализованную json-строку.

```
@JsonIgnoreProperties(ignoreUnknown = true)
class Car {
   private String color;
   private String type;

   @JsonIgnore
   private boolean isActive;

   ...
}
```

Изменение названия поля класса

Представим другую ситуацию - на стороне сервера решили что type - не очень удачное название для обозначения модели автомобиля и изменили его на model. Чтобы не менять название этого поля в классе (и всех частях программы, где мы вызывали это поле) существует аннотация **@JsonProperty**, в которой можно указать название поля в json:

```
@JsonProperty(value = "model")
private String type;

console output >> {"color":"green","model":"audi a8"}
```

Чтение полей json строки без десериализации

Иногда нам может потребоваться прочесть какое-нибудь значение одного поля из json строки, но заводить для этих целей целый класс было бы излишне расточительным и трудоемким занятием. Вместо этого, можно воспользоваться методом **reedTree**() класса **ObjectMapper**, который возвращает экземпляр класса **JsonNode**, позволяющий обходить элементы json объекта, считывать и записывать значения.

В качестве примера рассмотрим следующий json объект

```
{
 "name": "Ivan",
 "education": {
   "school": "School #52",
   "university": {
     "degree": "master",
     "name": "SPbGU"
   }
 }
}
```

Название университета вложено в объект “university”, который, в свою очередь, вложен в объект “education”, который находится в корневом объекте. Структура легко описывается в методе **at**:

```
ObjectMapper objectMapper = new ObjectMapper();
JsonNode universityName = objectMapper
    .readTree(jsonString)
    .at("/education/university/name");

System.out.println(universityName.asText());

>> SPbGU
```

Автогенерация Java классов на основе Json

В некоторых ситуациях нам приходится конструировать класс на основе примера ответа в формате JSON. Так как программисты - люди ленивые, в интернете существует довольно большое количество сервисов, аналогичных этому - https://codebeautify.org/json-to-java-converter

Подобные конвертеры генерируют код разной степени замусоренности, но для быстрого прототипирования такой подход имеет право на существование.

Полезные аннотации

Аннотация @JsonRootName

Применяется вместе с включенной опцией оборачивания в корневое значение

```
objectMapper.enable(SerializationFeature.WRAP_ROOT_VALUE);
```

По умолчанию, корневое значение будет совпадать с названием класса, но используя аннотацию можно изменить это поведение:

```
@JsonRootName(value = "car")
class Car {
  ...
}

serialization output>> {"car":{"color":"yellow","type":"Renault Logan"}}
```

Аннотация @JsonCreator

Позволяет помечать конструктор класса для использования его при создании экземпляра при десериализации. Применяется совместно с указанием аннотации **@JsonProperty**, которой помечаются аргументы конструктора с указанием соответствия названиям полей в json:

```
class Person {
   private int age;
   private String name;

   @JsonCreator
   public Person(@JsonProperty("age") int age,
                 @JsonProperty("firstName") String name) {
       this.age = age;
       this.name = name;
   }
}
```

```
String jsonPerson = "{ \"age\" : 30, \"firstName\" : \"Vsevolod\" }";
ObjectMapper objectMapper = new ObjectMapper();
Person p = objectMapper.readValue(jsonPerson, Person.class);
System.out.println(p);
>> Person{age=30, name='Vsevolod'}
```

Аннотация @JsonGetter, @JsonSetter

Применяются в случае, если название геттера или сеттера поля не соответствует конвенции Java, например:

```
class Student {
   private String name;
   private double averageMark;

   @JsonGetter("name")
   public String getStudentName() {
       return name;
   }

   @JsonSetter("name")
   public void setStudentName(String name) {
       this.name = name;
   }

   @JsonGetter("averageMark")
   public double getAvgMark() {
       return averageMark;
   }

   @JsonSetter("averageMark")
   public void setAvgMark(double averageMark) {
       this.averageMark = averageMark;
   }
```

Тем не менее, вывод отрабатывает без ошибок:

```
ObjectMapper objectMapper = new ObjectMapper();
Student student = new Student("Ivan", 4.87);
String jsonStudent = objectMapper.writeValueAsString(student);
System.out.println(jsonStudent);

console output >> {"name":"Ivan","averageMark":4.87}
```

Практическое задание

Задание необходимо сдать через Git. Инструкция

1. Реализовать корректный вывод информации о текущей погоде. Создать класс WeatherResponse и десериализовать ответ сервера. Выводить пользователю только текстовое описание погоды и температуру в градусах Цельсия.
2. Реализовать корректный выход из программы
3. Реализовать вывод погоды на следующие 5 дней в формате

В городе CITY на дату DATE ожидается WEATHER_TEXT, температура - TEMPERATURE

   где CITY, DATE, WEATHER_TEXT и TEMPERATURE - уникальные значения для каждого дня.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. https://www.baeldung.com/jackson
2. http://tutorials.jenkov.com/java-json/jackson-jsonnode.html#get-json-node-at-path
3. https://github.com/FasterXML/jackson