электронный журнал
МОЛОДЕЖНЫЙ НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК
Издатель ФГБОУ ВПО "МГТУ им. Н.Э. Баумана". Эл No. ФС77-51038.

 # 05, май 2016

УДК 004.432

Рассмотрение и решение проблем проверки корректности пользовательских данных на стороне java-сервера

Семенченко И. И., *студент*
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана,
кафедра «Системы обработки информации и управления»

Научный руководитель: Гапанюк Ю.Е., к.т.н., доцент,
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана,
кафедра «Системы обработки информации и управления»
gapyu@bmstu.ru

Введение

В любом клиент-серверном приложении, в котором пользователь может ввести что-нибудь свое и отправить эти данные на сервер, невозможно обойтись без валидации пользовательских данных. Валидацией данных далее будет называться проверка пользовательских данных на соответствие ожидаемым типам, диапазонам значений либо другим параметрам, которые были заданы разработчиком сервера.

С одной стороны, пользователь может случайно или в силу своей неопытности ввести некорректные данные. В этом случае, они могут привести к некорректному поведению на сервере, к его аварийной остановке, потере данных и недовольству самого пользователя.

С другой стороны, пользователь, являющийся злоумышленником, может намеренно отправить некорректные данные. Такая ситуация еще опаснее, ведь такой пользователь четко понимает, что делает. И при неправильной валидации он может украсть закрытые данные, получить доступ к исходному коду и многое другое, на данный момент существует множество различных атак, связанных с намеренным вводом некорректных данных. Например, SQL-инъекции и XSS-атаки.

Таким образом, очень важно, чтобы обрабатывались только запросы с корректными данными, а на все остальные нужно отвечать соответствующими ошибками. С этой целью автором был написан свой класс на языке java. В нем, в первую очередь, реализуется защита от непреднамеренного ввода некорректных данных, потому что на сервер, для которого разрабатывался данный класс, данные идут с интерфейса

администратора, который не станет специально «ломать» сервер, к тому же для защиты от атак злоумышленников есть более эффективные и проверенные средства защиты.

В статье рассматриваются проблемы, которые возникли при попытке осуществить полное отсеивание неправильных данных без использования дополнительных библиотек и с их использованием. И решение этих проблем, после написания своего класса, который основан на библиотеках apache GenericValidator и apache StringUtils и дополняет их.

Далее будут рассмотрены различные методы валидации, указаны их преимущества и недостатки. Будут рассмотрены библиотеки: apache GenericValidator, apache StringUtils и собственный класс SimpleValidator. Рассмотрим все методы валидации на трех примерах: валидация целого числа, бинарного значения и даты. На этих трех примерах можно четко увидеть преимущества и недостатки каждого подхода.

Валидация без использования сторонних библиотек

Стоит отметить, что, изначально, все данные, приходящие с сервера, являются строками, и в нашем частном случае, данные будут считаться корректными, только если их можно преобразовать в соответствующий им тип, либо если эти строки удовлетворяют определенными условиями.

Самый простой способ валидации данных на сервере - с помощью встроенных инструментов языка java. Когда перед разработчиком появляется проблема валидации, он пытается решить её, в первую очередь, этим способом. Далее рассмотрим подробнее этот тип валидации, его достоинства и недостатки.

Пример 1. Валидация целого числа.

```
try {
   int intValue = Integer.parseInt(value);
   //корректное значение
} catch (NumberFormatException e) {
   //ошибка валидации
}
```

В этом примере мы пытаемся преобразовать пришедший параметр (строку) в тип Integer, если преобразование успешно, значит наш параметр действительно является целым числом. Иначе мы обрабатываем ошибку. Аналогично проверяются другие числовые типы.

Пример 2. Валидация бинарного значения.

Бинарные значения Java: true - правда, false - ложь. Нам нужно, чтобы строковые значения “true” и “false” проходили проверку на правильность, а остальные не проходили с учетом регистра. Сразу возникают два способа, которыми можно это сделать.

Первый способ, то что сразу приходит на ум, обернуть функцию parseBoolean класса Boolean в блок try catch, как в первом примере.

```
Boolean boolValue;
String value = "Not binary";
try {
   boolValue = Boolean.parseBoolean(value);
   //значение корректно
} catch (Exception e) {
   //ошибка валидации
   e.printStackTrace();
}
```

Однако, при передаче не бинарного значения в строке, в отличие от предыдущего примера, исключение не возникает.

Функция Boolean.parseBoolean(String) возвращает true для строки “true”, причем регистр не важен, и false для любой другой строки. Такое поведение функции очень часто может быть неудобным, Например, значение может вставляться в sql запрос и содержать в себе sql-инъекцию.

Исходя из вышесказанного, понятно, что для проверки корректности пришедших значений этого типа нужно использовать другой метод. Так как значений такого типа только 2, оптимальный вариант - сравнить пришедшую строку с каждым из них:

```
if (!value.equals(Boolean.TRUE.toString()) &&
    !value.equals(Boolean.FALSE.toString())) {
      //значение корректно
} else {
      //ошибка валидации
}
```

Пример 3. Валидация даты.

Помимо простых типов данных часто приходиться валидироваться более сложные типы, например дату. Для этого нам понадобится встроенный класс DateFormat. Проверка, является ли строка датой заданного формата, будет выглядеть следующим образом:

```
String value = "2015-02-11";
DateFormat df = new SimpleDateFormat("yyyy-MM-dd", Locale.ENGLISH);
try {
```

```
        Date date = df.parse(value);
        //значение корректно
    } catch (ParseException e) {
        //ошибка валидации
    }
```

На основе примеров можно сделать выводы о достоинствах и недостатках такого типа валидации.

Достоинства:

- Все делается встроенными средствами языка, не нужно подключать дополнительные библиотеки.
- Самый быстрый способ проверить один или несколько параметров.

Недостатки:

- Не все типы можно проверить встроенными средствами
- Легко упустить какой-либо вариант данных или не обработать одно из возможных исключений, из-за чего могут возникнуть непредвиденные ошибки на этапе выполнения.
- При большом количестве проверяемых параметров получается громоздкий и трудный для понимания код, так как различные типы проверяются по-разному.

Валидация с использованием сторонних библиотек

На данный момент существует множество сторонних средств для различных проверок данных. Здесь будут рассматриваться две библиотеки: apache StringUtils и apache GenericValidator (ссылки на них будут даны в списке источников в конце статьи). Выбор был обусловлен следующими факторами:

1. Эти библиотеки активно поддерживаются разработчиками, недостатки в них оперативно исправляются. Например, самая ранняя версия библиотеки apache Generic Validator в maven репозитории датируется 2002 годом, а последнее обновление (на момент написания статьи) вышло в ноябре 2015 года.
2. Набор методов, предоставляемых данными библиотеками, в большей мере подходит для выполнения поставленных задач.
3. Наличие библиотек в maven репозитории.

Приведем примеры валидации рассмотренных выше типов данных (целое, бинарное значение, дата), используя библиотеки.

Пример 1. Валидация целого числа.

Функция isNumeric библиотеки StringUtils проверяет, все ли символы в строке являются числовыми, также большим плюсом является то, что внутри встроена проверка на null и на пустую строку. Эта функция работает быстрее, чем parseInt, так как выброс исключения - дорогая операция.

```
System.out.println(StringUtils.isNumeric("1000"));//true
System.out.println(StringUtils.isNumeric("true"));//false
System.out.println(StringUtils.isNumeric(null));//false
System.out.println(StringUtils.isNumeric(""));//false
```

В итоге проверка становится компактней, а код интуитивно понятнее:

```
String value = "100";
if (StringUtils.isNumeric(value)) {
   //корректное значение
} else {
   //ошибка валидации
}
```

Однако при использовании данной функции есть проблема: она не проверяет число на величину. Поэтому, даже если число прошло проверку этой функцией, если число больше, чем максимальное возможное целое, может появиться исключение, которое также нужно обрабатывать:

```
String value = "999999999999999999";
if (StringUtils.isNumeric(value)) {
   try {
      int intValue = Integer.parseInt(value);
      //корректное значение
   } catch (NumberFormatException e) {
      //ошибка валидации
   }
} else {
   //ошибка валидации
}
```

Еще, стоит отметить, что, в различных версиях одни и те же функции при одних и тех же параметрах могут принимать разные значения. Например:

В версии 2.6

```
StringUtils.isNumeric("");//true
```

В версии 3.0

```
StringUtils.isNumeric("");//false
```

Пример 2. Валидация бинарного значения.

Ни в одной из библиотек не предусмотрена валидация бинарного значения, видимо, в силу своей простоты. Поэтому проверять бинарные значения придется вручную, как было показано в в примере выше.

Пример 3. Валидация даты.

Для проверки того, является ли строка датой в библиотеке GenericValidator есть функция, которая проверяет, соответствует ли дата определенному формату или нет. Также можно задавать, является ли соответствие строгим. Например:

```
if (GenericValidator.isDate("10.1.2015","dd.MM.yyyy", false)) {
   //корректное значение
} else {
   //ошибка валидации
}
```

Здесь первый параметр - сама дата, второй - формат, на соответствие которому ее нужно проверять, третий - является ли проверка строгой. В нашем случае значение «10.1.2015» будет считаться корректным, несмотря на то, что в формате указывается, что в месяце должно быть 2 цифры, а у нас одна. Если сделать проверку строгой, то значение будет считаться некорректным:

```
GenericValidator.isDate("10.1.2015","dd.MM.yyyy", true);//false
```

Стоит отметить, что, помимо рассмотренных примеров, в библиотеках GenericValidator и StringUtils есть множество других методов, которые позволяют валидировать самые разные типы данных. Например:
GenericValidator.isUrl - проверяет, является ли строка корректной ссылкой, GenericValidator.isBlankOrNull - проверяет строку на то, является ли она пустой либо неопределенной(null), StringUtils.contains - проверяет наличие подстроки в строке и многие другие.

На основе всего вышесказанного омтетим достоинства и недостатки такого подхода валидации данных.

Достоинства:

- Есть множество методов, которые позволяют проверять сложные типы данных, такие как: дата, почта и т. д.
- Библиотеки проверены временем и активно поддерживатся, в связи с этим, они очень стабильны.

- Различные типы данных валидируются по одному шаблону, поэтому, при их использовании, часто код получается более читаемым, чем без них.

Недостатки:

- Предусмотрены не все типы данных, уделено мало внимания проверке простых типов.
- При переходе от версии к версии методы могут меняться, поэтому при каждом обновлении нужно тщательно проверять работоспособность системы.
- Некоторым методам могут поставить статус deprecated и впоследствии удалить их из библиотеки. После обновления до новой версии библиотеки, код, в котором использовались удаленные методы, придется переписать.

Класс SimpleValidator

Как было показано выше, у каждого метода валидации есть свои преимущества и недостатки. Для того, чтобы скрыть недостатки каждого из методов, при этом, сохранив его преимущества, автором был разработан свой класс - SimpleValidator.

Исходный код данного класса доступен по ссылке: https://github.com/SiiTP/SimpleValidator

На данный момент класс поддерживает 6 различных проверок:

- isBool - является ли строка значением типа Boolean
- isInteger - является ли строка целым числом
- isDate - является ли строка датой заданного формата
- isEmail является ли строка адресом электронной почты
- isInRange - входит ли строка в диапазон допустимых значений
- isString - не превосходит ли строка допустимую длину, также проверяет строку на пустоту.

Были реализованы именно эти методы, потому что при разработке сервера автор статьи столкнулся с задачей валидации соответствующих типов данных, однако класс написан так, что его можно легко расширить, добавив новые проверки.

Некоторые методы работают на основе библиотек apache GenericValidator и apache StringUtils и дополняют их, остальные реализованы без использования сторонних библиотек.

Разберемся, как же работает этот класс. Рассмотрим, как можно валидировать те же типы данных, которые были рассмотрены выше, но на этот раз воспользуемся классом SimpleValidator:

```
SimpleValidator validator = new SimpleValidator(true);
validator.isBool("true")
        .isInteger("1000")
        .isDate("17.11.1995", "dd.MM.yyyy", true);
if (validator.isValid()) {
   //все значения корректны
} else {
   //ошибка валидации
}
```

Итак, мы создаем объект типа SimpleValidator и у этого объекта вызываем функции, которые проверяют, являются ли правильными входные данные. Каждая проверка возвращает объект validator, при этом, если проверка была не успешной, то поле isValid этого объекта становится равным false. Таким образом, если хотя бы одна проверка не пройдет, то функция `validator.isValid()`будет выдавать значение false, если же все параметры корректны, то будет выдаваться значение true. При валидации таким способом решается проблема громоздкости кода, ведь на проверку одного параметра уходит ровно одна строка.

К тому же, во всех методах встроена проверка на пустоту строки, и, если строка пустая(null, “” или состоит только из пробелов), значение также считается некорректным, и валидация становится не успешной. Для реализации такой проверки был использован метод библиотеки StringUtils - isBlankOrNull. Благодаря такой проверке, размер кода, который должен написать пользователь, существенно сокращается.

В процессе валидации часто бывает нужно запоминать возникающие ошибки. В классе SimpleValidator все возникающие ошибки можно сохранять, как список строк. Для этого нужно при создании объекта передать в конструктор параметр true. От этого параметра зависит, будут ли запоминаться все ошибки, произошедшие в процессе валидации. Пример:

```
SimpleValidator validator = new SimpleValidator(true);
validator.isBool("True")
        .isInteger("not int")
        .isString("too large", 5)
        .isEmail("ne email")
        .isEmail("")
        .isDate("17.11.95", "dd.MM.yyyy", true);
```

```
LinkedList<String> errors = validator.getErrors();
errors.forEach(System.out::println);
System.out.println("is valid : " + validator.isValid());
```

Результат выполнения кода представлен на рисунке 1.

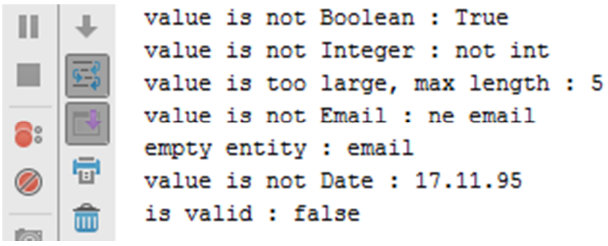

Рис. 1. Результат выполнения кода с учетом ошибок

В случае если мы проверяем множество данных, проверка их всех может быть долгой. Если нам необязательно проверять все значения, а нужно лишь знать, все ли данные корректны, учет ошибок можно отключить. В этом случае, после нахождения первой ошибки, последующие данные обрабатываться не будут. В списке ошибок будет присутствовать только первая ошибка. Чтобы это сделать, необходимо создать объект validator пустым конструктором, или передать в конструктор значение false. Тот же пример, но без учета всех ошибок, результат приведен на рисунке 2:

```
SimpleValidator validator = new SimpleValidator(false);
validator.isBool("True")
        .isInteger("not int")
        .isString("too large", 5)
        .isEmail("ne email")
        .isEmail("")
        .isDate("17.11.95", "dd.MM.yyyy", true);
LinkedList<String> errors = validator.getErrors();
errors.forEach(System.out::println);
System.out.println("is valid : " + validator.isValid());
```

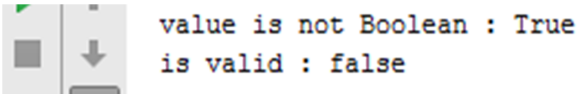

Рис. 2. Результат выполнения кода без учета ошибок

На данный момент поддерживается 4 типа ошибок, для простоты, они задаются в конструкторе, в реальном проекте их можно загружать, например, из файла ресурсов. Список ошибок приведен в таблице ниже.

Код ошибки	Сообщение
1	`"value is not "`
2	`"value is too large, max length : "`
3	`"value is not in range : "`
4	`"empty entity : "`

В каждом методе прослеживается структура, отображенная на рис. 3.

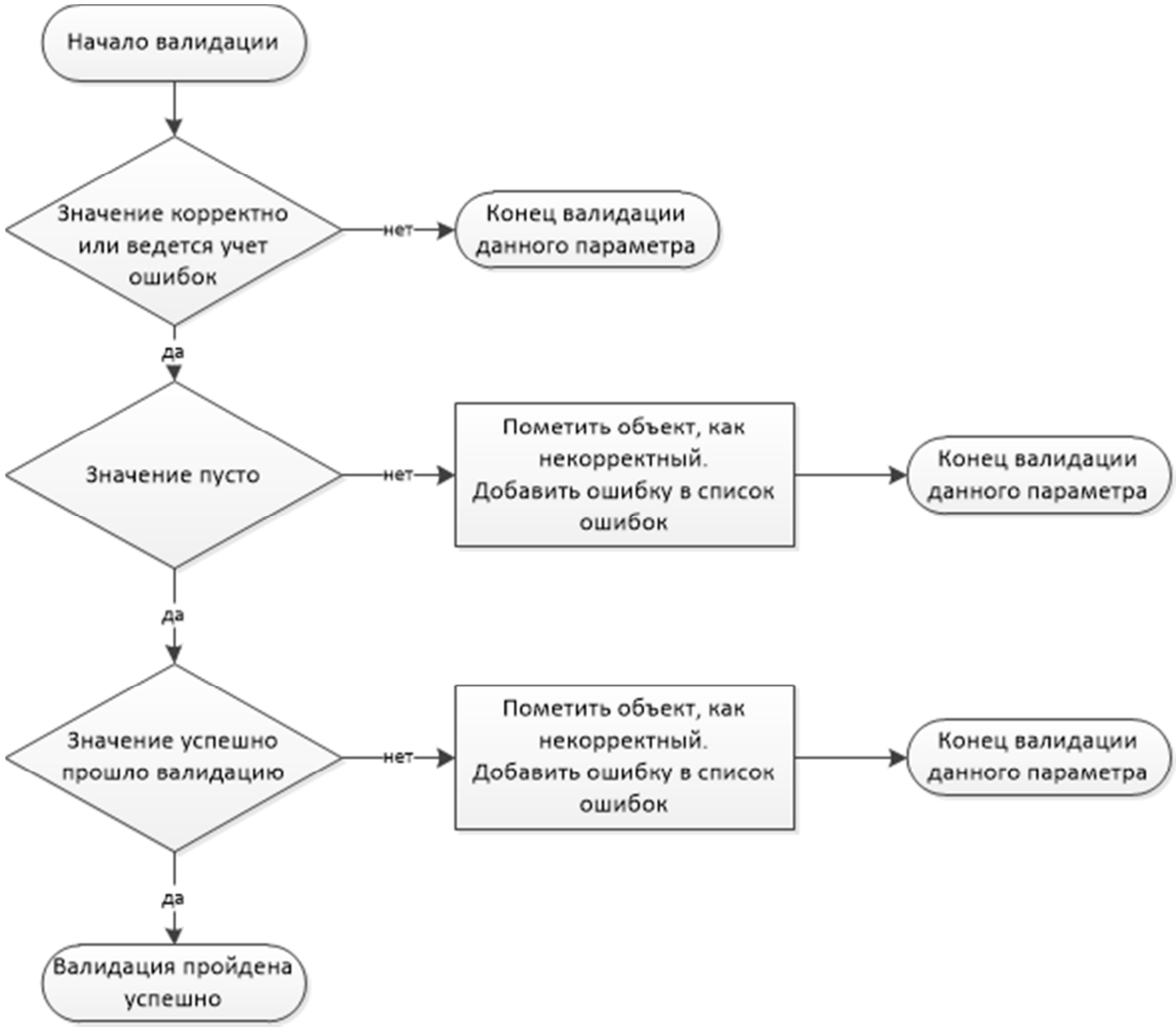

Рис. 3. Типичная структура метода проверки

Таким образом, не совершаются ненужные проверки, которые требуют много ресурсов, если предыдущие более легкие проверки не были пройдены. К тому же,

архитектура построена так, что, при окончании валидации одного параметра сразу начнется валидация следующего.

В дополнение, стоит отметить, что для всех проверочных функций были написаны unit-тесты, они так же есть в репозитории, ссылка на который была указана выше. Благодаря тестам, шанс того, что найдутся данные, которые будут проверяться неправильно, очень мал.

Заключение

В данной статье были рассмотрены два разных подхода к валидации данных. Как оказалось, у обоих подходов есть свои преимущества и недостатки. Чтобы объединить преимущества обоих подходов и скрыть их недостатки автором был разработан собственный класс. Благодаря подходу к валидации данных, предложенному автором, удалось существенно уменьшить код, требуемый для валидации данных, сделать его понятнее. Также при таком подходе код легко поддерживать и дополнять новыми проверками. Сам класс также является легко расширяемым, можно без труда дописать собственный метод, который будет проверять любой другой тип данных, которого на данный момент нет в классе. Кроме всего прочего, в классе присутствует гибкая система учета ошибок, аналогов которой нет ни в одной из рассмотренных библиотек.

Список литературы

[1]. Официальный сайт с документацией по java. Режим доступа: http://www.oracle.com/technetwork/java/javaee/documentation/index.html (дата обращения 27.02.2016).

[2]. Официальный сайт библиотеки apache GenericValidator. Режим доступа: https://commons.apache.org/proper/commons-validator/apidocs/org/apache/commons/validator/GenericValidator.html (дата обращения 27.02.2016).

[3]. Официальный сайт библиотеки apache StringUtils. Режим доступа: https://commons.apache.org/proper/commons-lang/javadocs/api-2.6/org/apache/commons/lang/StringUtils.html#isNumeric(java.lang.String) (дата обращения 27.02.2016).

[4]. Шилдт Г. Java 8. Полное руководство. М.: Издательский дом «Вильямс», 2015. С. 1109-1115.

[5]. Джеймс Гослинг, Билл Джой, Гай Л. Стил, Гилад Брача, Алекс Бакли. Язык программирования Java SE 8. Подробное описание. М.: Издательский дом «Вильямс», 2015. 672 с.