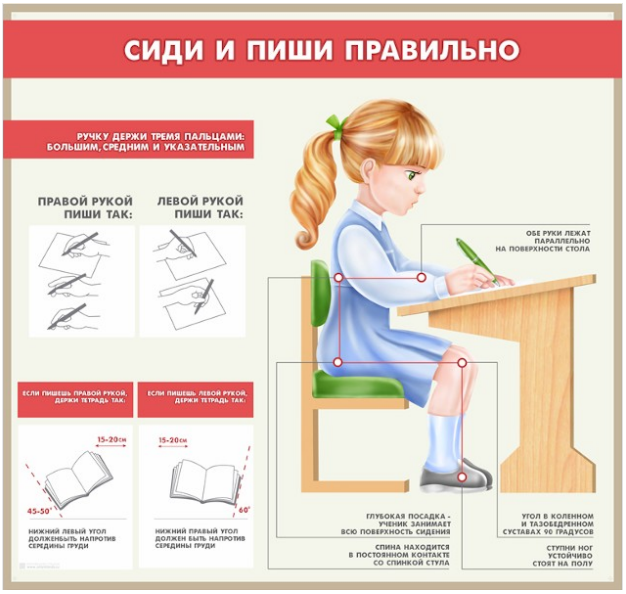
Автоматический контроль качества документации в Asciidoc или DocOps для Хабра

Содержание



Один из шагов выпуска документации — это применение алгоритмов автоматического контроля качества. Часть подходов будет применима только к документации ИТ-продуктов, часть — к любым видам документации.

Для примеров использована сама статья. В репозитории есть ссылки на автоматически публикуемые варианты статьи в различных форматах, в том числе в формате Хабра.

Обратите внимание, в новой версии редактора Хабр некорректно происходит вставка списков. Лучше использовать старую версию.

Точка применения алгоритмов контроля качества документации

Документация — это совокупность данных и документов. Используя для создания документации такие инструменты, как Asciidoc, мы предполагаем, что данные для построения документов хранятся в одном или нескольких репозиториях, точно так же, как обычный программный код.

При любом изменении документации в репозитории обязательна проверка качества документов, которые выпускаются на основе данных репозитория. При ручной проверке документов этот процесс затратен и ограничен. А автоматические тесты можно проводить практически в любом объеме.

Если документация расположена в нескольких репозиториях, должен быть отдельный репозиторий с набором совместимых версий документации. При изменении этих версий необходимо проверить согласованность данных во всех репозиториях.

Если мы говорим о документировании ИТ-системы, программный код является элементом документации, на него распространяется указанное правило. Код и документацию следует на согласованность.

Указанные проверки обычно производят в момент добавления данных в репозитории при помощи систем контроля версий. Мы используем Github и Gitlab и встроенные в эти системы CI/CD-инструменты. В сложных случаях дополнительно используем Jenkins.

Фреймворк тестирования

При тестировании документации основные инструменты проверки обычно запускают вне фреймворка тестирования, например, с помощью интерфейса командной строки (`cli`). Фреймворк тестирования проверяет результаты работы этих инструментов. Поэтому для тестирования документации подходят любые фреймворки, чаще всего определяемые экосистемой документируемой программы (информационной системы). В своих статьях я делаю примеры с использованием инструментов экосистемы Ruby, т.к.

сам Asciidoctor написан на Ruby, поэтому в статье будет использована библиотека minitest.

Проверка оформления исходных файлов в формате Asciidoc

Насколько мне известно, для проверки оформления исходных файлов в формате Asciidoc поддерживаемых проектов нет.

Мы используем простейшие проверки при помощи регулярных выражений.

Ключевое слово `describe` описывает содержание каждой проверки.

```
describe "The source file " do
  before do
    @isxodnyj_fajl = File.read("statqya.adoc")
  end
  it "should not contain more than one line break" do
    assert_nil @isxodnyj_fajl.match('\n\n\n')
  end
  it "should not contain whitespaces" do
    assert_nil @isxodnyj_fajl.match(' \n')
  end
  it "should contain only linux line breaks" do
    assert_nil @isxodnyj_fajl.match('\r\n')
  end
  it "should contain empty lines after headings" do
    assert_nil @isxodnyj_fajl.match('^[=]{2,}.*\n[^\n]')
  end
end
```

Проверка содержания текста (грамматика, орфография и т.п.)

Исходные файлы или выходные документы?

Проверять содержание текста можно как в исходных файлах, так и в выходных. С моей точки зрения, в большинстве случаев проверять имеет смысл именно выходные документы, а не исходные файлы Asciidoc. Например, в Asciidoc активно используют атрибуты и может возникнуть ситуация, при которой ошибку будет пропущена:

```
:document: документ
{document}овация
```

В исходном документе ошибки нет, а вот выходное слово `документовация` ошибку содержит.

Все ли понимают Asciidoc

Существует множество готовых инструментов, которыми можно проверять текстовые документы: например, vale, textlint, Aspell, LanguageTool.

Часть из этих инструментов поддерживают синтаксис Asciidoc. Но степень этой поддержки разная. Asciidoctor — самый богатый язык среди языков текстовой разметки, реализация в перечисленных средствах поддержки синтаксиса Asciidoctor может быть неполной или вообще неверной с точки зрения ваших требований к тексту.

Обычно, подобные проблемы легко преодолеть. Например, для `textlint` есть плагин, представление элементов в объектном дереве `textlint` определено в этом файле. Его можно легко поменять. Но иногда самой модели `textlint` может не хватить для проведения всех необходимых видов тестирования.

Как я уже говорил проверять статическим анализатором лучше выходные документы. В Asciidoctor нет встроенной функции, которая превращает исходники в формате Asciidoc в составной Asciidoc-файл. Но Дэн Аллен сделал специальный скрипт, который справляется с данной задачей.

Использование шаблонов Asciidoctor

Альтернативный способ подключения к Asciidoctor любых статических анализаторов — это превращение документа в текстовый файл. При этом появляется возможность размещать в данный файл дополнительную информацию, которая позволит понять, в каких исходниках произошла ошибка.

Для того, чтобы извлечь текст для проверки, Asciidoc поддерживает механизм шаблонов. Наименование папки с шаблонами передают в ключе `-t`.

Например, в следующем примере показан шаблон `inline_quoted.slim`, который помещает в файл только куски текста, не содержащие роль `no-spell`.

```
- if " #{role} " !~ / no-spell /
  =text
```

Далее в примере показано использование утилиты `aspell` непосредственно для выполнения функции проверки.

```
docker run --rm -v $(pwd):/documents/ curs/asciidoctor-od asciidoctor \
  statqya.adoc -b spell -o statqya.spell -T slim/base -T slim/spell
cat statqya.spell | sed "s/-/ /g" | \
  aspell --master=ru --personal=./dict list > misspelled-list
```

Само тестирование можно выполнить следующим образом:

```
describe "Final document " do
...
  it "has no typos " do
    assert_equal File.read('misspelled-list'), ''
  end
```

```
...
end
```

Тест, написанный таким образом, удобен тем, что в выводе `minitest` будет информация об ошибочно написанных словах:

```
  1) Failure:
Final document #test_0001_has no typos  [test.rb:30]:
--- expected
+++ actual
@@ -1,3 +1 @@
-"Адин
-шогов
-"
+""
```

Аналогичный подход можно использовать для реализации всевозможных самостоятельных проверок — отсутствие запрещенных слов, запрет параграфов, задаваемых несколькими строками и т.п.

Последняя проверка заслуживает отдельного внимания, т.к. её отсутствие — частый источник ошибок. Рассмотрим следующий пример.

```
Я
иду
в магазни
```

Поскольку перенос строки заменяется на пробел, параграф правильно отобразится в конечном документе. Следующий пример, оформленный аналогичным образом, уже приведёт к ошибке.

```
Неправильно оформленный список:
* Первый пункт
* Второй пункт
```

Так как после первого предложения отсутствует пустая строка, на выходе получится:

```
Неправильно оформленный список: * Первый пункт * Второй пункт
```

Запретить такое оформление достаточно просто. В шаблоне `paragraph.slim` необходимо указать, что в выходной файл выводится исходный текст параграфа (`source`):

```
="\n" + source + "\n"
```

В примере к исходному тексту параграфа добавлены два символа переноса строки, чтобы отличать этот (правильный) случай от случая с одним переносом.

И далее в тесте необходимо искать параграфы, в которых есть переносы строк:

```
describe "Final document " do
...
  it "is not based on paragraphs with line breaks " do
```

```
    assert_nil File.read('statqya.break-line').match('[^\n^+][\n][^\n]')
  end
...
end
```

Обратите внимание, после знака + перенос разрешён, т.к. это специальный синтаксис Asciidoctor, который позволяет вставить в абзац мягкие переносы.

Следующий тест выявляет различные несуразности в тексте.

```
describe "Final document " do
...
  it "more or less pretty as a russian text" do
    assert_nil File.read('statqya.spell').match('и т\.п\.'), "и{nbsp}т.п."
    assert_nil File.read('statqya.spell').match('и т\.д\.'), "и{nbsp}т.д."
    assert_nil File.read('statqya.spell').match('[Нн]ужн'), "Нужн... ->
Необходим..."
    assert_nil File.read('statqya.spell').match('[Оо]днако'), "Однако --> ?"
    assert_nil File.read('statqya.spell').match('[ \(](Вы|Вас|Вам)[^а-я]'),
"вы, вас, вам"
    assert_nil File.read('statqya.spell').match('Если[^\.]*, то'),
      "Если.. то, -- не программирование"
  end
...
end
```

Встроенные проверки Asciidoctor

Asciidoctor содержит собственные механизмы проверки. Для этого его необходимо запустить в режиме `Verbose`. Самые типовые выявляемые ошибки — битые ссылки внутри документа, нарушенная иерархия заголовков, отсутствие включаемых файлов и т.п. Для этого в командной строке используется ключ `-v`, как в следующем примере.

```
docker run --rm -v $(pwd):/documents/ curs/asciidoctor-od asciidoctor \
  statqya.adoc -b docbook -v 2> asciidoctor_log
```

Можно также запустить тестирование из библиотеки `minitest`:

```
describe "Final document " do
...
  it "has no Asciidoctor errors " do
    assert_equal File.read('asciidoctor_log'), ''
  end
...
end
```

Проверка структуры документов при помощи Docbook

Поскольку Asciidoctor изначально создавался как средство написания документов в формате Docbook, но в простом текстовом формате, то поддержка экспорта в формат Docbook реализована очень качественно.

Docbook — это вариант XML. Для тестирования структуры xml-файлов обычно используют два подхода.

Проверка при помощи схемы документа

XML поддерживает несколько стандартов схем документов. На сегодня самый распространенный — xsd-схемы.

Учитывая то, что Asciidoc поддерживает очень много элементов синтаксиса и не каждый конвертер корректно работает со всеми элементами (а Хабр вообще мало, что поддерживает), в примере ограничим используемые элементы параграфами, маркированными списками и врезками кода, также разрешим картинку после заголовка:

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://docbook.org/ns/docbook"
           elementFormDefault="qualified"
           attributeFormDefault="unqualified"
           xmlns:db="http://docbook.org/ns/docbook">
    <xs:import namespace="http://www.w3.org/XML/1998/namespace"
               schemaLocation="xml.xsd"/>
    <xs:element name="article">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="info">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element type="xs:string" name="title"/>
                            <xs:element type="xs:date" name="date"/>
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
                <xs:element name="informalfigure"
                            minOccurs="1" maxOccurs="unbounded">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:any minOccurs="0" processContents="skip"
maxOccurs="unbounded"/>
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
                <xs:element name="simpara" type="db:simpara"
                            minOccurs="0" maxOccurs="unbounded"/>
                <xs:element name="section" type="db:section"
                            minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
            <xs:attribute name="version"/>
            <xs:attribute ref="xml:lang"/>
        </xs:complexType>
    </xs:element>
    <xs:complexType name="simpara" mixed="true">
        <xs:choice minOccurs="0" maxOccurs="unbounded">
            <xs:element name="literal"/>
            <xs:element name="phrase"/>
            <xs:element name="link"/>
        </xs:choice>
```

```
    </xs:complexType>
    <xs:complexType name="section">
        <xs:choice maxOccurs="unbounded" minOccurs="0">
            <xs:element type="xs:string" name="title"/>
            <xs:element name="simpara" type="db:simpara"/>
            <xs:element name="screen"/>
            <xs:element name="section" type="db:section"/>
            <xs:element name="itemizedlist">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="listitem"
                                    minOccurs="1" maxOccurs="unbounded">
                            <xs:complexType>
                                <xs:sequence>
                                    <xs:element name="simpara"
                                                type="db:simpara"
                                                minOccurs="1"
                                                maxOccurs="unbounded"/>
                                </xs:sequence>
                            </xs:complexType>
                        </xs:element>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
        </xs:choice>
        <xs:attribute ref="xml:id"/>
    </xs:complexType>
</xs:schema>
```

В тесте проверка выглядит следующим образом:

```
describe "Final document " do
...
  it "has correct structure" do
    xsd = Nokogiri::XML::Schema(File.read("statqya.xsd"))
    doc = Nokogiri::XML(File.read("statqya.xml"))
    assert_equal xsd.validate(doc).join("\n"), ''
  end
...
end
```

Обычно такой подход применяют к кускам документа. В DITA — есть термин `topic` (тема). В зависимости от типа темы мы можем определять её структуру. Все темы определенного типа будут иметь одинаковую структуру.

Это удобно, если в документации активно используются похожие блоки.

Проверка при помощи xpath-выражений

Xpath-выражения — инструмент, который позволяет делать выборки из файлов в формате xml.

Полученную выборку можно проанализировать на соответствие определенным правилам.

Например, в следующем примере мы проверяем, что в элементе списка не может быть более одного абзаца.

Эту задачу можно было бы решить, прописав в предыдущей схеме ограничение на один элемент типа `simpara`, но часто формулировка локальных правил в виде xpath-выражений проще:

```
describe "Final document " do
...
  it "contains only list items with only one paragraph per item" do
    doc = Nokogiri::XML(File.read("statqya.xml"))
    assert_equal doc.xpath("//db:listitem[count(db:simpara) != 1]",
      'db' => 'http://docbook.org/ns/docbook').size, 0
  end
...
end
```

Этот же подход можно использовать для проверки сложных правил, не описываемых xsd-схемой, например, соответствие списка терминов тексту или работоспособность внешних ссылок:

```
describe "Final document " do
...
  it "has no 404 hyperlinks" do
    doc = Nokogiri::XML(File.read("statqya.xml"))
    erroneous_links = ''
    doc.xpath("//db:link/@xl:href",
        'db' => 'http://docbook.org/ns/docbook',
        'xl' => 'http://www.w3.org/1999/xlink').each do |link_href|
      begin
        puts link_href.to_s
        url = URI.parse(link_href.to_s)
        req = Net::HTTP.new(url.host, url.port)
        req.use_ssl = (url.scheme == "https")
        res = req.request_head(url.path)
      rescue  SocketError => e
        erroneous_links += link_href.to_s + "(#{e})\n"
      end
    end
    assert_equal erroneous_links, ''
  end
...
end
```

Проверка соответствия документации коду

В статье Автоматическая генерация технической документации рассмотрены инструменты автоматической генерации текстовых фрагментов из кода. Обычно формирование этих фрагментов происходит не в момент сборки документации, а при её подготовке.

Например, вы используете описание различных методов из спецификации OpenAPI. Предположим, есть шаблон, превращающий эту спецификацию в необходимые фрагменты текста. Если спецификация была изменена, необходимо заново сгенерировать соответствующие фрагменты и проверить, что они корректно легли в существующие документы.

В момент сборки имеет смысл проверить, что сформированные фрагменты текста соответствуют текущей версии спецификации. Для этого достаточно запустить генерацию фрагментов и проверить, что полученные файлы полностью совпадают с версиями, которые находятся в проекте документации.

Проверка выходных файлов

Документация представляется пользователю в удобочитаемых форматах, например, html, pdf, odt, docx и т.п.

Если вы используете стандартные конвертеры Asciidoctor, возможно, выходной файл проверять не надо. Но желательно открыть и сохранить файл в нативном приложении. Например, в моём проекте сделана специальная точка вызова, которая конвертирует файл и автоматически открывает/сохраняет его при помощи LibreOffice Writer. Достаточно проверить, что выходной файл есть.

```
describe "Final document " do
...
  it "has an odt output" do
    assert File.exists?("statqya.odt")
  end
...
end
```

Офисные приложения — Microsoft Word, LibreOffice Writer — иногда портят документы при открытии. Например, Microsoft Word заменяет поля на текст «Ошибка. Закладка не определена». Если такие случаи часты, для исключения целесообразно делать соответствующие проверки.

Выводы

- Предложенная технология универсальна и может быть использована для создания любых документов с высоким уровнем требований по качеству, в том числе, статей на Хабре.
- Asciidoc дает много возможностей по проверке качества документации. В совокупности они позволяют проверить оформление исходных файлов, качество текста, структуру документов и т.п.
- Наличие нативного статического анализатора для Asciidoc могло бы значительно упростить процесс задания правил для проверки документации.
- Результат проверки данной статьи — `12 runs, 17 assertions, 0 failures, 0 errors, 0 skips`, а ошибки всё равно есть. PRs are welcome.