Как устроен язык XML.
XML элементы: их атрибуты и значение; тэги.

XML (*e**X**tensible **M**arkup **L**anguage)* - расширяемый язык разметки, использующийся для хранения и передачи данных. Увидеть его можно не только в API, но и в коде.
XML — это не исполняемый код, а язык описания данных.

Теги XML идентифицируют данные и используются для хранения и организации данных, а не для указания способа их отображения, как теги HTML, которые используются для отображения данных. XML, также, не может считаться языком программирования, поскольку он не выполняет никаких вычислений или алгоритмов. Обычно он хранится в простом текстовом файле и обрабатывается специальным программным обеспечением, способным интерпретировать XML.

Устройство XML документа

Синтаксически в XML, по сравнению с HTML, нет ничего нового. Это такой же текст, размеченный тэгами, но с той лишь разницей, что в HTML существует ограниченный набор тэгов, которые можно использовать в документах в то время, как XML позволяет создавать и использовать любую разметку, которая может понадобиться для разметки данных.

Несомненным достоинством XML является и то, что это достаточно простой язык. Основных конструкций в XML мало, но, несмотря на это, с их помощью можно создавать разметку документов практически любой сложности.

Кодировкой по умолчанию для XML является **unicode**. Далее находится открывающий тэг корневого (главного) элемента <**корневой_элемент**>, содержащий элемент <**элемент**>, который, в свою очередь, содержит элемент <**еще_элемент атрибут**="значение" /> с атрибутом **атрибут**. Правила записи элементов, атрибутов и их значений в XML ничем не отличаются от правил записи элементов атрибутов и их значений в HTML (также есть открывающие и закрывающие тэги элементов, элементы с содержимым и без и т.д.), только набор элементов несколько расширен, благодаря чему мы и можем "нагрузить" разметку семантикой.

Несколько правил построения XML документа.

- любой XML документ должен начинаться строкой <?**xml version**="1.0" ?>
- любой XML документ должен иметь единственный (не более, не менее!) корневой элемент; например, в HTML для этих целей использовался элемент **<html>**, в примере выше - это <**корневой_элемент**>.
- кодировкой по умолчанию для символов XML документа является **Unicode** кодировка **UTF-8**, поэтому XML файлы должны быть сохранены в соответствующей кодировкой или в 1-й строке документа должна быть задана кодировка документа, например **encoding**="Windows-1251" (при работе только с латиницей это никак себя не проявляет, так как кодировка этих символов в **ASCII**совпадает с **UTF-8**).
- правила записи большинства конструкций языка совпадает с правилами **XHTML**, изучавшемся вами ранее (более подробно речь об основных конструкциях языка пойдет далее в уроке).

XML документ представляет собой обыкновенный текстовый файл с расширением **.xml**. Единственная особенность их заключается в том, что для символов файла рекомендуется использовать кодировку **Unicode**.

Три важные характеристики XML, которые делают его полезным в различных системах и решениях:

- XML является расширяемым − XML позволяет создавать собственные самоописательные теги или язык, который подходит для вашего приложения.
- XML переносит данные, но не представляет их − XML позволяет хранить данные независимо от того, как они будут представлены.

- XML является общедоступным стандартом. XML был разработан организацией под названием World Wide Web Consortium (W3C) и доступен в качестве открытого стандарта.

Использование XML

- XML может работать "за сценой", упрощая создание HTML-документов для больших веб-сайтов.
- XML может использоваться для обмена информацией между организациями и системами.
- XML можно использовать для разгрузки и перезагрузки баз данных.
- XML можно использовать для хранения и упорядочивания данных, что позволяет настроить ваши потребности в обработке данных.
- XML можно легко объединить с таблицами стилей для создания практически любого желаемого результата.
- Практически любой тип данных может быть представлен в виде XML-документа.

Разметка

XML — это язык разметки, который определяет набор правил для кодирования документов в формате, понятном как для человека, так и для машины. Разметка — это информация, добавляемая в документ, которая определенным образом улучшает его значение, поскольку она идентифицирует части и то, как они связаны друг с другом. Более конкретно, язык разметки — это набор символов, которые могут быть размещены в тексте документа для разграничения и обозначения частей этого документа.

Теги и элементы

XML-файл структурирован несколькими XML-элементами, также называемыми XML-узлами или XML-тегами. Теги бывают открывающими и закрывающими. У закрывающего есть дополнительный символ — “/”. Имена XML-элементов заключены в треугольные скобки < >:

Синтаксис элемента − Каждый XML-элемент должен быть закрыт либо начальными, либо конечными элементами:

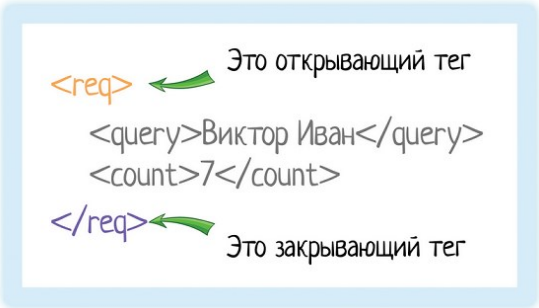
Вложенность элементов − XML-элемент может содержать несколько XML-элементов в качестве своих дочерних элементов, но дочерние элементы не должны перекрываться, то есть, конечный тег элемента должен иметь то же имя, что и у самого последнего несогласованного начального тега.

В XML каждый элемент должен быть заключен в теги. Тег — это некий текст, обернутый в угловые скобки:

<tag>
Текст внутри угловых скобок — название тега.
Тега всегда два:

Открывающий — текст внутри угловых скобок **<tag>**

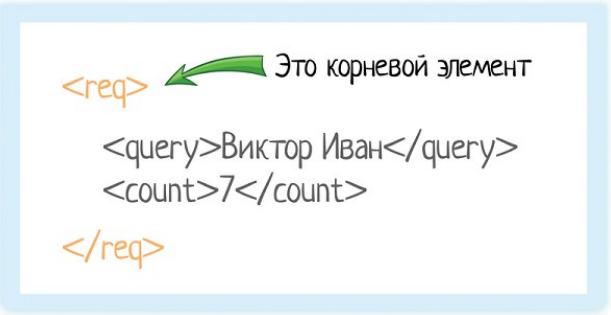
Закрывающий — тот же текст (это важно!), но добавляется символ «/» **</tag>**



Каждому открывающему тегу должен соответствовать закрывающий. Они показывают, где начинается и где заканчивается описание каждого элемента в файле. Теги могут быть вложенными и это не ограничивается одним уровнем: у вложенных тегов могут быть свои вложенные теги, и т. д. Такая конструкция называется деревом тегов.

Корневой элемент

В любом XML-документе есть корневой элемент. **Д**окумент XML может иметь один корневой элемент. Это тег, с которого документ начинается, и которым заканчивается. В случае REST API документ — это запрос, который отправляет система. Или ответ, который получает. Чтобы обозначить этот запрос, нужен корневой элемент.

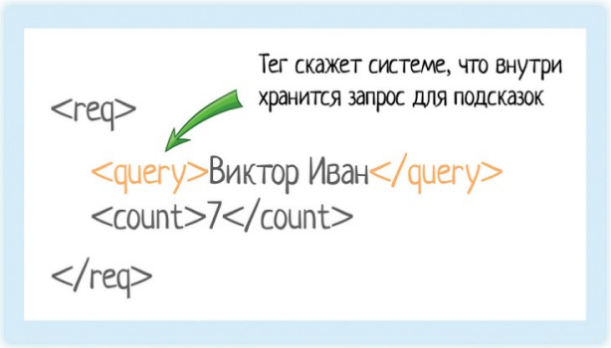


только

она нам

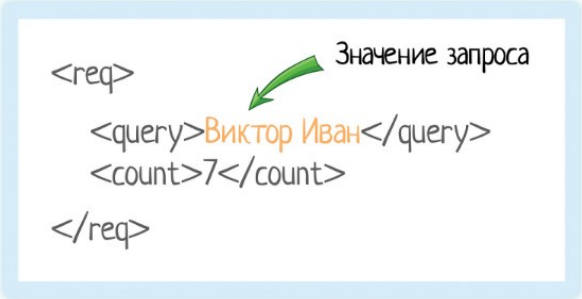
Значение элемента

Значение элемента хранится между открывающим и закрывающим тегами. Это быть число, строка, или даже вложенные

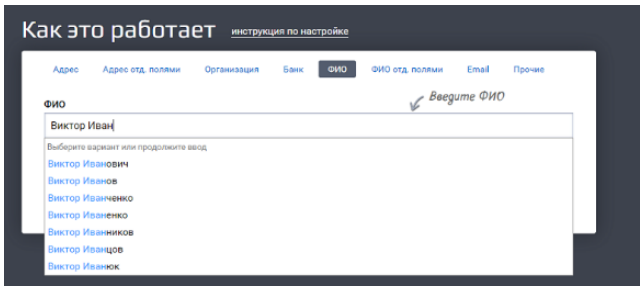


может теги! Вот тег мы

«query». Он обозначает запрос, который отправляем в подсказки.
Внутри — значение запроса.



Это как если бы мы вбили строку «Виктор GUI (графическом интерфейсе пользователя):
Пользователю лишняя обвязка не нужна, нужна красивая форма. А системе надо передать, что «пользователь ввел именно Чтобы показать ей, где начинается и заканчивается переданное используются



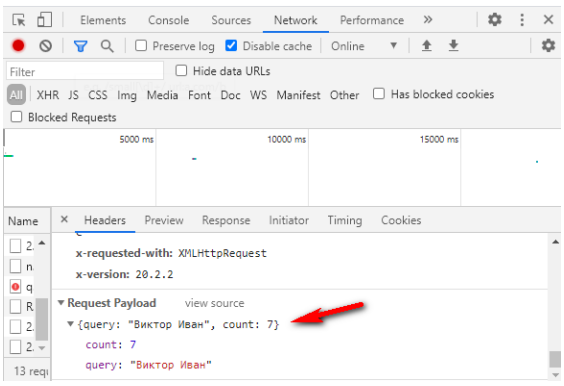
Иван» в

ему как-то это».

теги.

Система видит тег «query» и понимает, что внутри него «строка, по которой нужно вернуть подсказки».

Параметр *count = 7* обозначает, сколько подсказок вернуть в ответе. Если обратиться к документации метода, count можно выбрать от 1 до 20. Откройте консоль разработчика через f12, вкладку Network, и посмотрите, какой запрос отправляется на сервер. Там будет значение count = 7.



Ссылки на XML

Ссылки обычно позволяют добавлять или включать дополнительный текст или разметку в XML-документ. Ссылки всегда начинаются с символа **"&"**, который является зарезервированным символом, и заканчиваются символом **";".**XML имеет два типа ссылок:

- **Ссылки на сущности − ссылка** на сущность содержит имя между начальным и конечным разделителями. Например, **&** где *amp* -это *имя*. *Имя* относится к предопределенной строке текста и /или разметки.
- **Ссылки на символы** − они содержат ссылки, такие как **A**, содержит хэш-метку (“#”), за которой следует число. Число всегда относится к коду символа в Юникоде. В этом случае 65 относится к алфавиту "A".

Текст XML

Имена XML-элементов и XML-атрибутов чувствительны к регистру, что означает, что имена начальных и конечных элементов должны быть записаны в одном регистре. Чтобы избежать проблем с кодировкой символов, все XML-файлы должны быть сохранены как файлы Unicode UTF-8 или UTF-16.

Пробельные символы, такие как пробелы, табуляции и разрывы строк между XML-элементами и между XML-атрибутами, будут игнорироваться.

Некоторые символы зарезервированы самим синтаксисом XML. Следовательно, их нельзя использовать напрямую. Для их использования используются некоторые заменяющие объекты, которые перечислены ниже -

Недопустимый символ	Заменяющий объект	Описание символов
<	<	менее
>	>	больше, чем
&	&	амперсанд
'	& apos;	апостроф
"	"	кавычки

Атрибуты элемента

У элемента могут быть атрибуты — один или несколько. Их мы указываем внутри отрывающегося тега после названия тега через пробел в виде

Атрибут задает единственное свойство для элемента, используя пару имя/ значение. XML-элемент может иметь один или несколько атрибутов. Например −

 Учебная точка!

Здесь **href** - это имя атрибута и **http://www.tutorialspoint.com /** - значение атрибута.

Правила синтаксиса для атрибутов XML

- Имена атрибутов в XML (в отличие от HTML) чувствительны к регистру. То есть *HREF* и *href* считаются двумя разными атрибутами XML.
- Один и тот же атрибут не может иметь два значения в синтаксисе.

```
название_атрибута = «значение атрибута»
```

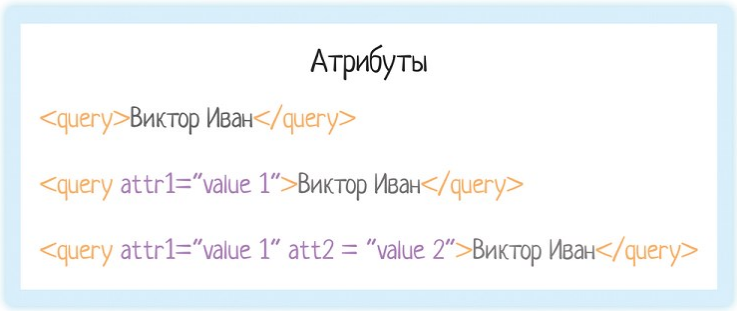
Например:

```
<query attr1=“value 1”>Виктор Иван</query>
```

```
<query attr1=“value 1” attr2=“value 2”>Виктор Иван</query>
```

Из атрибутов принимающая API-запрос система понимает, что ей пришло.

Например, мы делаем поиск по системе, ищем клиентов с именем Олег. Отправляем простой запрос:

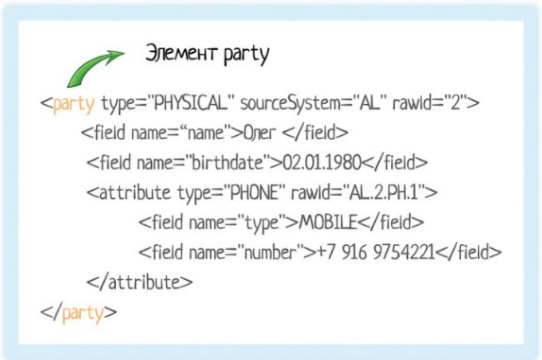



```
<query>Олег</query>
```

А в ответ получаем целую пачку Олегов! С разными датами рождения, номерами телефонов и другими данными. Допустим, что один из результатов поиска выглядит так:

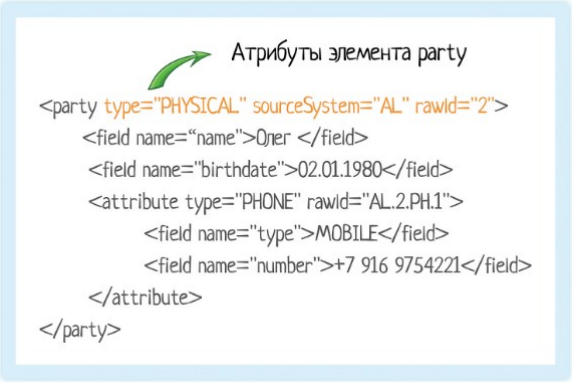
```
<party type="PHYSICAL" sourceSystem="AL" rawId="2">
   <field name=“name”>Олег </field>
   <field name="birthdate">02.01.1980</field>
   <attribute type="PHONE" rawId="AL.2.PH.1">
      <field name="type">MOBILE</field>
      <field name="number">+7 916 1234567</field>
   </attribute>
</party>
```

Разберем эту запись. У нас есть основной элемент **party**.

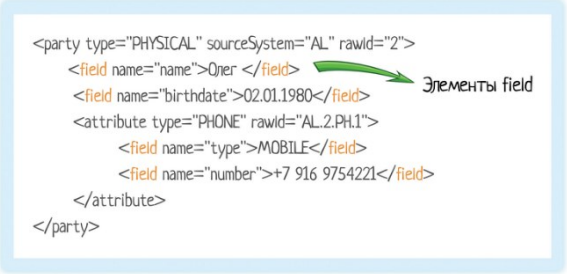


У него есть 3 атрибута:

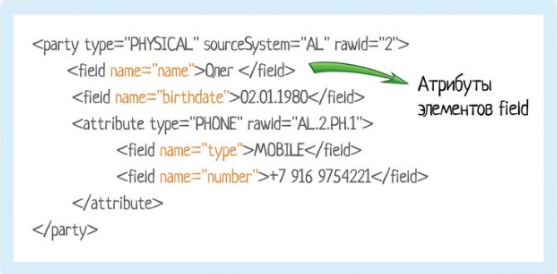
- **type = «PHYSICAL»** — тип возвращаемых данных. Нужен, если система умеет работать с разными типами: ФЛ, ЮЛ, ИП. Тогда благодаря этому атрибуту мы понимаем, с чем именно имеем дело и какие поля у нас будут внутри. А они будут отличаться! У физика это может быть ФИО, дата рождения ИНН, а у юр лица — название компании, ОГРН и КПП
- **sourceSystem = «AL»** — исходная система. Возможно, нас интересуют только физ лица из одной системы, будем делать отсев по этому атрибуту.
- **rawId = «2»** — идентификатор в исходной системе. Он нужен, если мы шлем запрос на обновление клиента, а не на поиск. Как понять, кого обновлять? По связке sourceSystem + rawId!



Внутри **party** есть элементы **field**.



У элементов **field** есть атрибут **name**. Значение атрибута — название поля: имя, дата рождения, тип или номер телефона. Так мы понимаем, что скрывается под конкретным **field**.



Это удобно с точки зрения поддержки, когда в работе есть коробочный продукт и 10+ заказчиков. У каждого заказчика будет свой набор полей: у кого-то в системе есть ИНН, у кого-то нету, одному важна дата рождения, другому нет, и т.д.

Но, несмотря на разницу моделей, у всех заказчиков будет одна XSD-схема (которая описывает запрос и ответ):
— есть элемент party;
— у него есть элементы field;
— у каждого элемента field есть атрибут name, в котором хранится название поля.

Конкретные названия полей можно не описывать в XSD. (Зависит от ТЗ). Когда заказчик один или вы делаете ПО для себя или «вообще для всех», удобнее использовать именованные поля — то

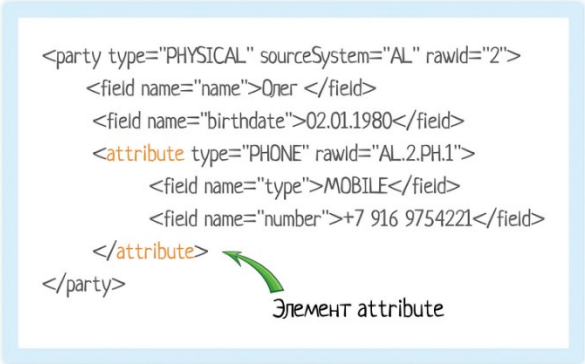
есть «говорящие» теги. Преимущества этого подхода:

— При чтении XSD сразу видны реальные поля. ТЗ может устареть, а код будет актуален.
— Запрос легко дернуть вручную в SOAP Ui — он сразу создаст все нужные поля, нужно только значениями заполнить. Это удобно тестировщику + если иногда тестирует заказчик иногда так тестирует, ему тоже удобно.

Любой подход имеет право на существование. Надо смотреть по проекту, что будет удобнее именно вам.

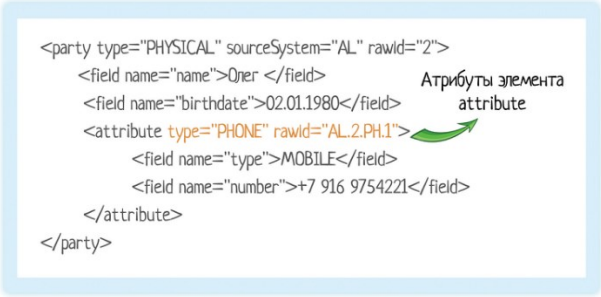
Помимо элементов **field** в party есть элемент **attribute**. Не следует путать xml-нотацию и бизнес-прочтение:

- с точки зрения бизнеса это атрибут физ лица, отсюда и название элемента — *attribute*.
- с точки зрения xml — это элемент (не атрибут!), просто его назвали *attribute*. XML все равно (почти), как вы будете называть элементы, так что это допустимо.



У элемента **attribute** есть атрибуты:

- **type = «PHONE»** — тип атрибута. Они ведь разные могут быть: телефон, адрес, емейл...
- **rawId = «AL.2.PH.1»** — идентификатор в исходной системе. Он нужен для обновления. Ведь у одного клиента может быть несколько телефонов, как без ID понять, какой именно обновляется?



Это упрощенный пример XML. В реальных системах, где хранятся физ лица, данных сильно больше: штук 20 полей самого физ лица, несколько адресов, телефонов, емейл-адресов. Но прочитать даже огромную XML не составит труда, если знать, что где. И если она отформатирована — вложенные элементы сдвинуты вправо, остальные на одном уровне. С форматированием всё просто — у нас есть элементы, заключенные в теги. Внутри тегов — название элемента. Если после названия идет что-то через пробел: это атрибуты элемента.

XML пролог

Иногда вверху XML документа можно увидеть что-то похожее:

<?xml version="1.0" encoding="UTF-8"?>

Эта строка называется XML прологом. Она показывает версию XML, который используется в документе, а также кодировку. Пролог необязателен, если его нет — это ок. Но если он есть, то это должна быть первая строка XML документа. *UTF-8* — кодировка XML документов по умолчанию.

XSD — умный XML

XSD — это язык описания структуры XML документа. Его также называют XML Schema. При использовании XML Schema XML парсер может проверить не только правильность синтаксиса XML документа, но также его структуру, модель содержания и типы данных.
XSD (**X**ML **S**chema **D**efinition) — это описание вашего XML. Как он должен выглядеть, что в нем должно быть. Это ТЗ, написанное на языке машины тоже в формате XML. Получается XML, который описывает другой XML.

Кроме того, XSD расширяем, и позволяет подключать уже готовые словари для описания типовых задач, например веб-сервисов, таких как SOAP.
Стоит также упомянуть о том, что в XSD есть встроенные средства документирования, что позволяет создавать самодостаточные документы, не требующие дополнительного описания.
Удобство в том, что проверку по схеме можно делегировать машине. И разработчику даже не надо расписывать каждую проверку. Достаточно сказать «вот схема, проверяй по ней».

Если мы создаем SOAP-метод, то указываем в схеме:

- какие поля будут в запросе;
- какие поля будут в ответе;
- какие типы данных у каждого поля;
- какие поля обязательны для заполнения, а какие нет;
- есть ли у поля значение по умолчанию, и какое оно;
- есть ли у поля ограничение по длине;
- есть ли у поля другие параметры;
- какая у запроса структура по вложенности элементов;
- ...

Теперь, когда к нам приходит какой-то запрос, он сначала проверяется на корректность по схеме. Если запрос правильный, запускаем метод, отрабатываем бизнес-логику.

Простому пользователю вашего SOAP API схема помогает понять, как составить запрос. Кто такой «простой пользователь»?

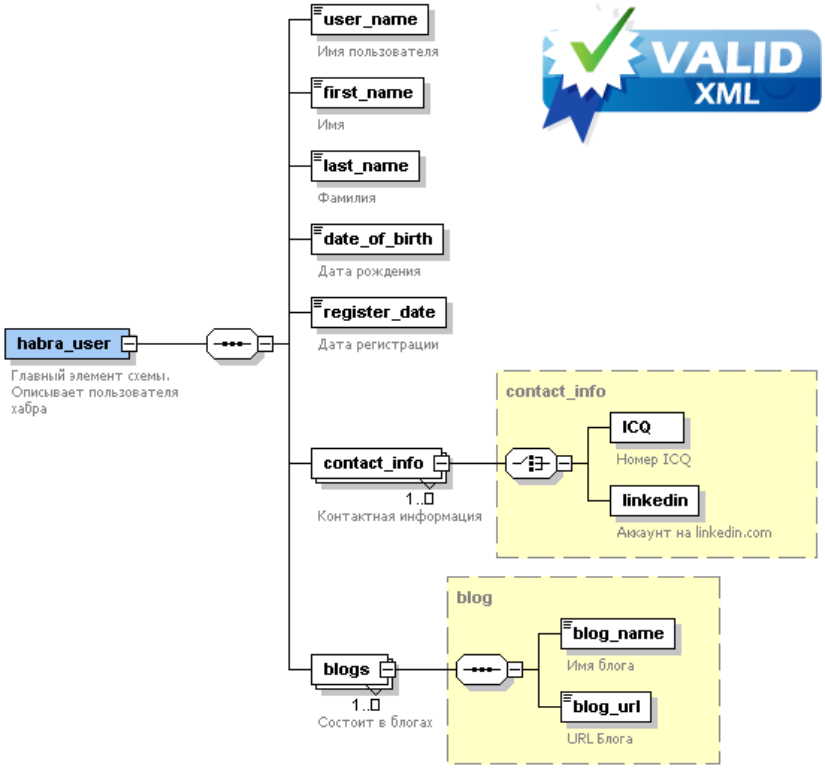
1. Разработчик системы, использующей ваше API — ему надо прописать в коде, что именно отправлять из его системы в вашу.
2. Тестировщик, которому надо это самое API проверить — ему надо понимать, как формируется запрос.

Иногда ТЗ просто нет, а иногда оно устарело. А вот схема не устареет, потому что обновляется при обновлении кода. И она как раз помогает понять, как запрос должен выглядеть.

Итого, как используется схема при разработке SOAP API:

- Наш разработчик пишет XSD-схему для API запроса: нужно передать элемент такой-то, у которого будут такие-то дочерние, с такими-то типами данных. Эти обязательные, те нет.
- Разработчик системы-заказчика, которая интегрируется с нашей, читает эту схему и строит свои запросы по ней.
- Система-заказчик отправляет запросы нам.
- Наша система проверяет запросы по XSD — если что-то не так, сразу отлуп.
- Если по XSD запрос проверку прошел — включаем бизнес-логику!

Рассмотрим в качестве примера XSD документ, описывающий часть структуры аккаунта на хабре.



Первая строчка схемы указывает, что документ является XML документом и использует кодировку UTF-8.

```
<?xml version="1.0" encoding="UTF-8"?>
```

Со следующей строки начинается описания главного элемента документа — *habra_user*.

```
<xs:element name="habra_user">
```

Строки, документирующие элемент:

```
 <xs:annotation>
  <xs:documentation>Главный элемент схемы. Описывает пользователя хабра</xs:documentation>
 </xs:annotation>
```

Тег *<xs:complexType>* описывает «сложный» тип данных *user_name*. При желании его можно вынести как отдельный тип данных, по аналогии с *contact_info*. Для этого, нужно блок *<xs:complexType>* перенести в *<xs:schema>* и указать атрибут *name*, а элементу задать атрибут *type*.

Элементы *user_name*, *first_name*, *last_name* имеют строковый тип и описывают пользователя, имя и фамилию владельца аккаунта.

Элемент *date_of_birth* имеет тип данных «дата» и описывает дату рождения.

Дату регистрации описывает *register_date*, имеющий собственный тип данных *customDateTime*. Значение этого тега будет задаваться с помощью атрибута *value*. На это указывают строки.

```
<xs:attribute name="value" use="required">
```

При этом атрибут — обязательный. Чтобы значение соответствовало требованиям, опишем «проверки»:

```
  <xs:simpleType>
   <xs:restriction base="xs:string">
    <xs:length value="19"/>
    <xs:pattern value="[1-2][0-9][0-9][0-9]-[0-1][0-9]-[0-3][0-9] [0-2][0-9]:[0-5][0-9]:[0-5][0-9]"/>
   </xs:restriction>
  </xs:simpleType>
```

В таком случае длина строки будет всегда 19, это задано тегом <*xs:length*> и само значение будет соответствовать шаблону, указанным в теге <*xs:pattern*>.

Элементы *contact_info* и *blog* — массивы, на это указывает атрибут *maxOccurs=«unbounded»*.

Тег <*xs:choice*> определяет то, что вложенным элементом будет один из элементов ICQ или linkedin.

Тег <*xs:sequence*> указывает на то, что вложенные элементы будут *blog_name* и *blog_url* именно в такой последовательности. Если последовательность не важна, то нужно использовать тег <*xs:all*>.

Ещё один пример, как XSD-схема, как схема может выглядеть.

Возьмем для примера метод doRegister в Users. Чтобы отправить запрос, мы должны передать email, name и password.

<wrap:doRegister> <email>olga@gmail.com</email> <name>Ольга</name> <password>1</password> </wrap:doRegister>	<wrap:doRegister> <email>maxim@gmail.com</email> <name>*(&$%*($</name> <password>Парольчик</password> </wrap:doRegister>

Попробуем написать для него схему. В запросе должны быть 3 элемента (*email, name, password*) с типом *«string»* (строка). Пишем:

```
<xs:element name="doRegister ">
  <xs:complexType>
  <xs:sequence>
   <xs:element name="email" type="xs:string"/>
   <xs:element name="name" type="xs:string"/>
   <xs:element name="password" type="xs:string"/>
  </xs:sequence>
  </xs:complexType>
</xs:element>
```

А в WSDl сервиса она записана еще проще:

```
<message name="doRegisterRequest">
  <part name="email" type="xsd:string"/>
  <part name="name" type="xsd:string"/>
  <part name="password" type="xsd:string"/>
</message>
```

Конечно, в схеме могут быть не только строковые элементы. Это могут быть числа, даты, boolean-

значения и даже какие-то свои типы:

```
<xsd:complexType name="Test">
  <xsd:sequence>
    <xsd:element name="value"   type="xsd:string"/>
    <xsd:element name="include" type="xsd:boolean" minOccurs="0" default="true"/>
    <xsd:element name="count" type="xsd:int" minOccurs="0" length="20"/>
    <xsd:element name="user" type="USER" minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>
```

А еще в схеме можно ссылаться на другую схему, что упрощает написание кода — можно переиспользовать схемы для разных задач.

Ок, теперь мы знаем, как «прочитать» запрос для API-метода в формате XML. Но как его составить по ТЗ? Давайте попробуем. Смотрим в документацию. И вот почему я даю пример из Дадаты — там классная документация!

Параметры

Параметр	Обязательный?	Описание
query	да	Запрос, для которого нужно получить подсказки
count	нет	Количество возвращаемых подсказок (по умолчанию — 10, максимум — 20).
parts	нет	Подсказки по части ФИО
gender	нет	Пол (UNKNOWN / MALE / FEMALE)

Что, если я хочу, чтобы мне вернуть только женские ФИО, начинающиеся на «Ан»? Берем наш исходный пример:

```
<req>
  <query>Виктор Иван</query>
  <count>7</count>
</req>
```

В первую очередь меняем сам запрос. Теперь это уже не «Виктор Иван», а «Ан»:

```
<req>
  <query>Ан</query>
  <count>7</count>
</req>
```

Далее смотрим в ТЗ. Как вернуть только женские подсказки? Есть специальный параметр — *gender*. Название параметра — это название тегов. А внутри уже ставим пол. «Женский» по английски будет *FEMALE*, в документации также. Итого получили:

```
<req>
  <query>Ан</query>
  <count>7</count>
```

```
  <gender>FEMALE</gender>
</req>
```

Ненужное можно удалить. Если нас не волнует количество подсказок, параметр count выкидываем. Ведь, согласно документации, он необязательный. Получили запрос:

```
<req>
  <query>Ан</query>
  <gender>FEMALE</gender>
</req>
```

Вот и все! Взяли за основу пример, поменяли одно значение, один параметр добавили, один удалили. Не так сложно, когда есть подробное ТЗ и пример.

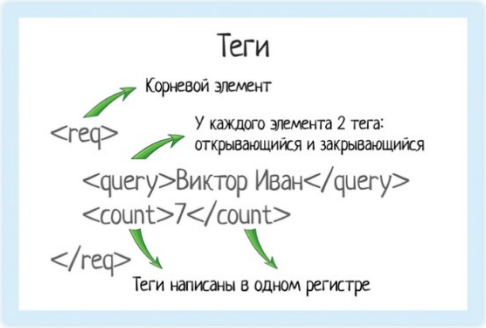
Well Formed XML

Разработчик сам решает, какой XML будет считаться правильным, а какой нет. Но есть общие правила, которые нельзя нарушать. XML должен быть well formed, то есть синтаксически корректный.

Чтобы проверить XML на синтаксис, можно использовать любой XML Validator (так и гуглите). Я рекомендую сайт w3schools. Там есть сам валидатор + описание типичных ошибок с примерами.

В готовый валидатор вы просто вставляете свой XML (например, запрос для сервера) и смотрите, всё ли с ним хорошо. Но можете проверить его и сами. Пройдитесь по правилам синтаксиса и посмотрите, следует ли им ваш запрос.
Правила well formed XML:

1. Есть корневой элемент.
2. У каждого элемента есть закрывающийся тег.
3. Теги регистрозависимы!
4. Соблюдается правильная вложенность элементов.
5. Атрибуты оформлены в кавычках.



Давайте пройдемся по каждому правилу и обсудим, как нам применять их в тестировании. То есть как правильно «ломать» запрос, проверяя его на well-formed xml. Зачем это нужно? Посмотреть на фидбек от системы. Сможете ли вы по тексту ошибки понять, где именно облажались?

1. Есть корневой элемент

Нельзя просто положить рядышком 2 XML и полагать, что «система сама разберется, что это два запроса, а не один». Не разберется. Потому что не должна.

И если у вас будет лежать несколько тегов подряд без общего родителя — это плохой xml, не well formed. Всегда должен быть корневой элемент:

Нет	Да

<test> <user>Тест</user> <pass>123</pass> </test> <dev> <user>Антон</user> <pass>123</pass> </dev>	<credential> <test> <user>Тест</user> <pass>123</pass> </test> <dev> <user>Антон</user> <pass>123</pass> </dev> </credential>
Есть элементы «test» и «dev», но они расположены рядом, а корневого, внутри которого все лежит — нету. Это скорее похоже на 2 XML документа	А вот тут уже есть элемент credential, который является корневым

Что мы делаем для тестирования этого условия? Правильно, удаляем из нашего запроса корневые теги!

2. У каждого элемента есть закрывающийся тег

Тут все просто — если тег где-то открылся, он должен где-то закрыться. Хотите сломать? Удалите закрывающийся тег любого элемента.

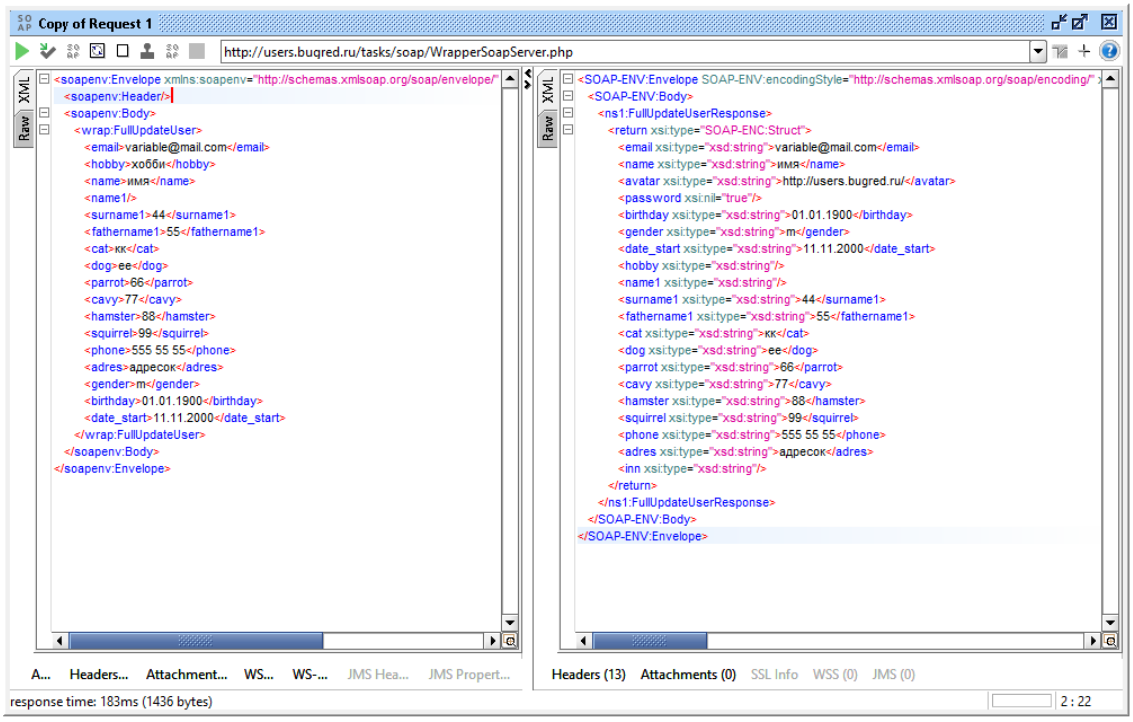
Но тут стоит заметить, что тег может быть один. Если элемент пустой, мы можем обойтись одним тегом, закрыв его в конце:

<name/>

Это тоже самое, что передать в нем пустое значение

<name></name>

Аналогично сервер может вернуть нам пустое значение тега. Можно попробовать послать пустые поля в Users в методе *FullUpdateUser*. И в запросе это допустимо (я отправила пустым поле *name1*), и в ответе SOAP Ui нам именно так и отрисовывает пустые поля.



Итого — если есть открывающийся тег, должен быть закрывающийся. Либо это будет один тег со слешом в конце.

Для тестирования удаляем в запросе любой закрывающийся тег.

Нет	Да
<user>Тест	<user>Тест</user>
Тест</user>	<user/>

3. Теги регистрозависимы

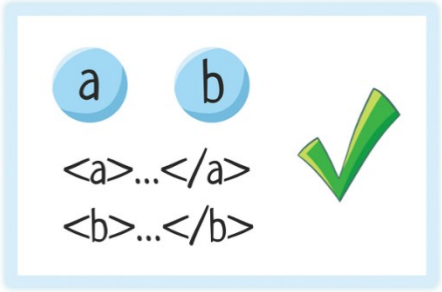
Как написали открывающий — также пишем и закрывающий. ТОЧНО ТАК ЖЕ! А не так, как захотелось.

А вот для тестирования меняем регистр одной из частей. Такой XML будет невалидным

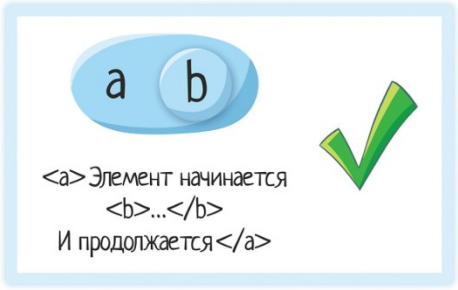
Нет	Да
<Name>Тест</name> <NAME>Иван</name> <NAME>Тест</name>	<name>Тест</name>

4. Правильная вложенность элементов

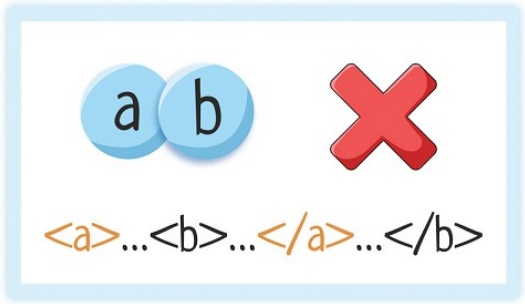
Элементы могут идти друг за другом



Один элемент может быть вложен в другой



Но накладываться друг на друга элементы НЕ могут!



Нет	Да
<fio>Иванов <name>Иван</fio> Иванович </name>	<fio>Иванов Иван Иванович</fio> <name>Иван</name>
<fio>Иванов <name>Иван</name> Иванович</fio>	<fio>Иванов <name>Иван</name> Иванович</fio>

Атрибуты оформлены в кавычках

Даже если вы считаете атрибут числом, он будет в кавычках:

```
<query attr1=“123”>Виктор Иван</query>
<query attr1=“атрибутик” attr2=“123” >Виктор Иван</query>
```

Для тестирования пробуем передать его без кавычек:

<query attr1=123>Виктор Иван</query>

Заключение.

XML (e**X**tensible **M**arkup **L**anguage) используется для хранения и передачи данных.

Передача данных — это запросы и ответы в API-методах. Если вы отправляете SOAP-запрос, вы априори работаете именно с этим форматом. Потому что SOAP передает данные только в XML. Если вы используете REST, то там возможны варианты — или XML, или JSON.

Хранение данных — это когда XML встречается внутри кода. Его легко понимает как машина, так и человек. В формате XML можно описывать какие-то правила, которые будут применяться к данным, или что-то еще.

Если вы тестировщик, то при тестировании запросов в формате XML обязательно попробуйте нарушить каждое правило. Да, система должна уметь обрабатывать такие ошибки и возвращать адекватное сообщение об ошибке. Но далеко не всегда она это делает.

А если система публичная и возвращает пустой ответ на некорректный запрос — это плохо. Потому что разработчик другой системы допустит ошибку в запросе, а по пустому ответу даже не поймет, где именно. Убедитесь, что система выдает понятное сообщение об ошибке.