Data formatting JSON, XML, YAML, GeoJSON, BSON, Google Protobuf

JSON (JavaScript Object Notation)

Пример:

{

"glossary": {

"title": "example glossary",

"GlossDiv": {

"title": "S",

"GlossList": {

"GlossEntry": {

"ID": "SGML",

"SortAs": "SGML",

"GlossTerm": "Standard Generalized Markup Language",

"Acronym": "SGML",

"Abbrev": "ISO 8879:1986",

"GlossDef": {

"para": "A meta-markup language, used to create markup languages",

"GlossSeeAlso": ["GML", "XML"]

},

"GlossSee": "markup"

}

}

}

}

}

***Что такое JSON?***

JSON - простой, основанный на использовании текста, способ хранить и передавать структурированные данные.

JSON имеет следующие преимущества:

* Он компактен.
* Его предложения легко читаются и составляются как человеком, так и компьютером.
* Его легко преобразовать в структуру данных для большинства языков программирования (числа, строки, логические переменные, массивы и так далее)
* Многие языки программирования имеют функции и библиотеки для чтения и создания структур JSON.

Данные в формате JSON (RFC 4627) представляют собой:

- JavaScript-объекты { ... } или

- Массивы [ ... ] или

- Значения одного из типов:

- строки в двойных кавычках,

- число,

- логическое значение true/false,

- null.

Почти все языки программирования имеют библиотеки для преобразования объектов в формат JSON.

***Для чего используется JSON?***

Наиболее частое распространенное использование JSON - пересылка данных от сервера к браузеру. Обычно данные JSON доставляются с помощью AJAX, который позволяет обмениваться данными браузеру и серверу без необходимости перезагружать страницу.

Также можно использовать JSON для отправки данных от браузера на сервер, передавая строку JSON в качестве параметра запросов GET или POST. Но данный метод имеет меньшее распространение, так как передача данных через запросы AJAX может быть упрощена. Например, ID продукта может быть включен в адрес URL как часть запроса GET.

***Как работать с JSON?***

Библиотека jQuery имеет несколько методов, например, getJSON() и parseJSON(), которые упрощают получение данных с помощью JSON через запросы AJAX.

Основные методы для работы с JSON в JavaScript — это:

* JSON.parse — читает объекты из строки в формате JSON.
* JSON.stringify — превращает объекты в строку в формате JSON, используется, когда нужно из JavaScript передать данные по сети.

[***Метод JSON.parse***](https://learn.javascript.ru/json#метод-json-parse)

Вызов JSON.parse(str) превратит строку с данными в формате JSON в JavaScript-объект/массив/значение.

***Например:***

var numbers = "[0, 1, 2, 3]";

numbers = JSON.parse(numbers);

alert( numbers[1] ); // 1

var user = '{ "name": "Вася", "age": 35, "isAdmin": false, "friends": [0,1,2,3] }';

user = JSON.parse(user);

alert( user.friends[1] ); // 1

***JSON-объекты ≠ JavaScript-объекты***

Объекты в формате JSON похожи на обычные JavaScript-объекты, но отличаются от них более строгими требованиями к строкам — они должны быть именно в двойных кавычках.

В частности, первые два свойства объекта ниже — некорректны:

{

name: "Вася", // ошибка: ключ name без кавычек!

"surname": 'Петров',// ошибка: одинарные кавычки у значения 'Петров'!

"age": 35 // .. а тут всё в порядке.

"isAdmin": false // и тут тоже всё ок

}

В формате JSON не поддерживаются комментарии. Он предназначен только для передачи данных. Есть нестандартное расширение формата JSON, которое называется JSON5 и как раз разрешает ключи без кавычек, комментарии и т.п, как в обычном JavaScript. На данном этапе, это отдельная библиотека.

Для интеллектуального восстановления из строки у ***JSON.parse(str, reviver)*** есть второй параметр ***reviver***, который является функцией function(key, value).

Если она указана, то в процессе чтения объекта из строки JSON.parse передаёт ей по очереди все создаваемые пары ключ-значение и может возвратить либо преобразованное значение, либо undefined, если его нужно пропустить.

В данном случае мы можем создать правило, что ключ date всегда означает дату:

// дата в строке - в формате UTC

var str = '{"title":"Конференция","date":"2014-11-30T12:00:00.000Z"}';

var event = JSON.parse(str, function(key, value) {

if (key == 'date') return new Date(value);

return value;

});

alert( event.date.getDate() ); // теперь сработает!

***Сериализация, метод JSON.stringify***

Метод JSON.stringify(value, replacer, space) преобразует («сериализует») значение в JSON-строку.

var event = {

title: "Конференция",

date: "сегодня"

};

var str = JSON.stringify(event);

alert( str ); // {"title":"Конференция","date":"сегодня"}

// Обратное преобразование.

event = JSON.parse(str);

Во втором параметре JSON.stringify(value, replacer) ***можно указать массив свойств, которые подлежат сериализации.***

var user = {

name: "Вася",

age: 25,

window: window

};

alert( JSON.stringify(user, ["name", "age"]) );

// {"name":"Вася","age":25}

***Красивое форматирование***

В методе JSON.stringify(value, replacer, space) есть ещё третий параметр space. Если он является числом — то уровни вложенности в JSON оформляются указанным количеством пробелов, если строкой — вставляется эта строка.

var user = {

name: "Вася",

age: 25,

roles: {

isAdmin: false,

isEditor: true

}

};

var str = JSON.stringify(user, "", 4);

alert( str );

/\* Результат -- красиво сериализованный объект:

{

"name": "Вася",

"age": 25,

"roles": {

"isAdmin": false,

"isEditor": true

}

}

\*/

XML( *e****X****tensible****M****arkup****L****anguage* — расширяемый язык разметки)

Этот язык получил в название «расширяемость» за возможность создавать собственную разметку, которая будет приспособлена разработчиками к тем или иным особенностям сайта, будучи ограниченным лишь синтаксическими правилами языка.

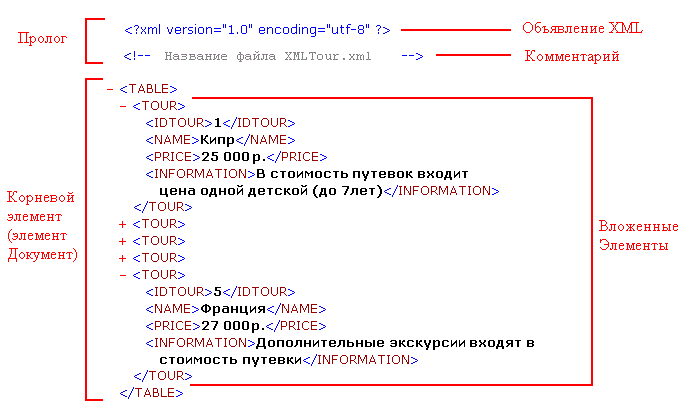
XML решает ряд проблем, которые не решает HTML, например:

- Представление документов любого (не только текстового) типа, например, музыки, математических уравнений и т.д.

- Сортировка, фильтрация и поиск информации.

- Представление информации в структурированном (иерархическом) виде.

Документ XML разбивается на**пролог**и**корневой** элемент.

[](http://testing.biz.ua/wp-content/uploads/2015/05/xml-1.png)

Символы разметки в XML используются такие же, как и для HTML:

разметка всегда начинается символом **«<«** и заканчивается символом **«>».**Также важную роль играет апмперсанд **«&»**, который позволяет выполнить замену при помощи сущностей (например символ **«<»**следует писать в коде как **«&lt»,**а символ**«&»**как **«&amp»** и т.д.). Для употребления апострофа и кавычек также нужно использовать соответствующие сущности. Правила замены символов на сущности, которыми они обозначаются, выполняется во всем документе, кроме секций «CDATA».

***Простейший XML-документ состоит из двух частей***: пролога и корневого элемента. Пролог содержит объявление XML, указывающее на то, что это XML-документ, и содержит номер версии XML.

Пролог может также содержать необязательные компоненты:

* Объявление типа документа.
* Одну или несколько инструкций по обработке.

***Обязательным в объявлении*** является указание версии языка и может указываться информация о кодировке.

<?xml version="1.1" encoding="UTF-8" ?>

В прологе объявляется и **тип документа**. Для этого используется специальная инструкция !DOCTYPE, которая при помощи языка DTD позволяет задать какие элементы входят в документ, их атрибуты, сущности, которые могут использоваться. Также при помощи объявления типа документа можно описывать его содержание и логическую структуру.

<?xml version="1.0"?>

<!DOCTYPE greeting SYSTEM "hello.dtd">

<greeting>Hello, world!</greeting>

###### ***Корневой элемент XML.***

**Корневой элемент** является частью логической структуры документа и включает в себя вложенные элементы, границы которых представляются начальным(<..>) и конечным тегами(</..>). Элемент также может быть представлен пустым тегом(<…/>) и не включать в себя другие элементы или символьные данные. Как и в HTML, в XML атрибуты могут быть указаны только в начальном теге или теге пустого элемента.

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

<!DOCTYPE recipe>

<recipe name="хлеб" preptime="5min" cooktime="180min">

<title>

Простой хлеб

</title>

<composition>

<ingredient amount="3" unit="стакан">Мука</ingredient>

<ingredient amount="0.25" unit="грамм">Дрожжи</ingredient>

<ingredient amount="1.5" unit="стакан">Тёплая вода</ingredient>

<ingredient amount="1" unit="чайная ложка">Соль</ingredient>

</composition>

<instructions>

<step>

Смешать все ингредиенты и тщательно замесить.

</step>

<step>

Закрыть тканью и оставить на один час в тёплом помещении.

</step>

<!--

<step>

Почитать вчерашнюю газету.

</step>

- это сомнительный шаг...

-->

<step>

Замесить ещё раз, положить на противень и поставить в духовку.

</step>

</instructions>

</recipe>

В зависимости от уровня соответствия стандартам документ может быть "верно сформированным" ("well-formed"), либо "валидным" ("valid").

***Основные правила создания верно сформированного ("well-formed") документа:***

- Документы XML должны соответствовать всем правилам синтаксиса языка*,* иначе это не будет считаться документом XML.

- Все XML элементы должны иметь закрывающийся тег (в HTML есть некоторые теги, которые не требуют закрытия).

- Теги XML чувствительны к регистру.

- Все элементы обязаны соблюдать корректную вложенность.

- XML документ должен обязательно иметь один корневой элемент

- Значение всех атрибутов обязательно заключается в кавычки.

- Помнить о замене специальных знаков(<,>,&,’,») сущностями XML.

- В XML- документе сохраняются все пробельные символы (в отличии от HTML).

XML-документ можно набрать в любом текстовом редакторе, сохранив документ как текстовый файл с расширением .xml. В дальнейшем такой документ будет открываться двойным щелчком в Internet Explorer.

***Создание валидных ("valid") XML-документов***

Валидным (valid) называется корректно сформированный (well-formed) документ, отвечающий двум дополнительным требованиям:

* Пролог документа должен содержать определение типа документа (DTD - Document Type Definition), задающее структуру документа.
* Оставшаяся часть документа должна отвечать структуре, заданной в DTD.

Любое отклонение от требований корректности формирования (well-formed) считается фатальной ошибкой (fatal error). Если XML-процессор сталкивается с фатальной ошибкой, он останавливает обработку документа и не пытается её возобновить. Отклонение от требований валидности (valid) считается лишь ошибкой (error). Если XML-процессор сталкивается с ошибкой, он может просто выдать сообщение о ней и продолжить обработку.

Объявление типа документа (DTD) представляет собой блок разметки, который вы должны добавить в пролог XML-документа, и имеет следующую форму записи:

<!DOCTYPE Имя DTD>

DTD может содержать следующие типы объявлений разметки:

* Объявления типов элементов, которые может содержать документ, их содержимое и порядок следования.
* Объявления списков атрибутов, которые могут быть использованы с определёнными типами элементов, типы данных атрибутов и значения атрибутов по умолчанию.
* Объявления примитивов для хранения часто используемых фрагментов текста или для встраивания не относящихся к XML данных в ваш документ.
* Объявления нотаций, которые описывают форматы данных или идентифицируют программу, используемую для обработки определённого формата.
* Инструкции по обработке.
* Комментарии.
* Ссылки на параметрические примитивы. Любой из приведённых выше компонентов может содержаться внутри параметрического примитива и добавляться путём ссылки на параметрический примитив.

YAML

***YAML («YAML — Не язык разметки»)*** — «дружественный» формат сериализации данных, концептуально близкий к языкам разметки, но ориентированный на удобство ввода-вывода типичных структур данных многих языков программирования.

YAML – это обычный текстовый файл, с древовидной разметкой для хранения данных. Вложенность веток “регулируется” некоторым количеством пробелов перед наименованием параметра или данными (если они не умещаются в одну строку).

# Это комментарий. Он вообще игнорируется и нужен лишь человеку, а не машине))

Item-1:

  Sub-Item-1: "обычная строка/текст"

  Sub-Item-2: #массив

      - array-item-1

      - array-item-2

Item-2:

  Sub-Item-1:

      Sub-sub-Item-1: false

  Sub-Item-2: 5678

Item-3:

  Sub-Item-1: true

  Sub-Item-2: |

     Много

            строчный

      текст

  Sub-Item-3: [array-item-1, array-item-2, array-item-3] #массив

Item-4: Hello world

Item-5: Free public

Как можно заметить, в этом примере приведены следующие типы данных: строка(String), массив(Array), число(Integer) и логическое значение(Boolean).

Каждый новый подпункт должен иметь перед собой хотя бы на 1 пробел больше, чем у предыдущего пункта (принято ставить 2-4 пробела).

Массивы, как видно, можно записывать двумя способами: по пунктам или через запятую. Разницы никакой нет. Вы сами выбираете, какой вариант удобнее. Многострочный текст объявляется вертикальной чертой “|” и записывается как новая ветка, с пробелами в начале.

YAML более человекочитаем чем JSON, но из-за этого он теряет в производительности.

GeoJSON

***GeoJSON*** - формат представления различных структур географических данных.

GeoJSON поддерживает следующие ***геометрические типы***:

* Point (точка),
* LineString (ломаная),
* Polygon (полигон),
* MultiPoint (мультиточка),
* MultiLineString (мультиломаная),
* MultiPolygon (мультиполигон)
* GeometryCollection (коллекция геометрий).

Объект (feature) в GeoJSON состоит из геометрии и дополнительных свойств, коллекция объектов (feature collection) – из набора объектов (feature).

в GeoJSON существует иерархия объектов вида GeoJSON

Object => feature collection => feature => geometry.

условимся называть GeoJSON Object – объект GeoJSON, feature collection – коллекция элементарных объектов, feature – элементарный объект, geometry – геометрия.

В GeoJSON объект состоит из набора пар ключ/значение, также называемых свойствами. Имя каждого свойства – строка. Значение свойства может представлять собой строку, число, объект, массив или один из литералов: «true», «false» и «null». Массив состоит из элементов, где каждый элемент может принимать одно из значений, описанных выше.

Коллекция элементарных объектов GeoJSON:

{ "type": "FeatureCollection",

"features": [

{ "type": "Feature",

"geometry": {"type": "Point", "coordinates": [102.0, 0.5]},

"properties": {"prop0": "value0"}

},

{ "type": "Feature",

"geometry": {

"type": "LineString",

"coordinates": [

[102.0, 0.0], [103.0, 1.0], [104.0, 0.0], [105.0, 1.0]

]

},

"properties": {

"prop0": "value0",

"prop1": 0.0

}

},

{ "type": "Feature",

"geometry": {

"type": "Polygon",

"coordinates": [

[ [100.0, 0.0], [101.0, 0.0], [101.0, 1.0],

[100.0, 1.0], [100.0, 0.0] ]

]

},

"properties": {

"prop0": "value0",

"prop1": {"this": "that"}

}

}

]

}

## *Объекты GeoJSON*

* GeoJSON всегда представляет собой единственную сущность: геометрию, элементарный объект или коллекцию элементарных объектов.
* Объект GeoJSON может иметь произвольное количество свойств (пар ключ/значение).
* Объект GeoJSON должен иметь свойство «type». Значение этого свойства – строка, содержащая тип объекта GeoJSON.
* Значение свойства «type» должно принимать одно из следующих значений: «Point», «MultiPoint», «LineString», «MultiLineString», «Polygon», «MultiPolygon», «GeometryCollection», «Feature» или «FeatureCollection». Регистр символов значения поля «type» имеет значение.
* Объект GeoJSON может иметь необязательное свойство «crs», значение которого должно содержать объект системы координат Объект GeoJSON может иметь свойство «bbox», значение которого представляет массив координат вершин ограничивающего прямоугольника

***Геометрия***

Геометрия – это объект GeoJSON, для которого в качестве значения свойства «type» используется одна из строк: «Point», «MultiPoint», «LineString», «MultiLineString», «Polygon», «MultiPolygon» или «GeometryCollection».

Все геометрии, тип которых отличен от «Geometry Collection», должны иметь свойство «coordinates». Значение данного свойства всегда представляет собой массив. Структура элементов массива определяется типом геометрии.

#### **Координаты**

Координаты – фундаментальная геометрическая концепция. Свойство «coordinates» объекта геометрия состоит из пары/триплета координат (в случае геометрии типа «Point»), массива координат (объекты типа «LineString» или «MultiPoint»), массива массивов координат (объекты типа «Polygons», «MultiLineStrings») или многомерного массива координат (объекты типа «MultiPolygon»).

Координаты определяются массивом чисел. Этот массив должен содержать минимум два элемента, но их может быть больше. Порядок элементов должен быть следующим: x, y, z (для данных, находящихся в прямоугольной системе координат - смещение на восток, смещение на север, высота, для данных, находящихся в географической системе координат – долгота, широта, высота). Допускается введение дополнительных элементов, однако их интерпретация выходит за рамки данной спецификации.

#### Point

Для объектов типа «Point» свойство «coordinates» должно содержать одну пару/триплет координат.

#### MultiPoint

Для объектов типа «MultiPoint» свойство «coordinates» должно содержать массив пар/триплетов координат.

#### LineString

Для объектов типа «LineString» свойство «coordinates» должно содержать массив из двух и более пар/триплетов. Тип «LinearRing» - это замкнутый «LineString», содержащий 4 и более пар/триплетов координат. Первая и последня пара/триплет эквивалентны (представлены одинаковыми точками). Хотя тип «LinearRing» явно не входит в список типов геометрий, он используется при описании типа «Polygon».

#### MultiLineString

Для объектов типа «MultiLineString» свойство «coordinates» должно содержать массив массивов пар/триплетов координат «LineString».

#### Polygon

Для объектов типа «Polygon» свойство «coordinates» должно содержать массив массивов пар/триплетов координат «LinearRing». Для полигонов с несколькими кольцами первым должно идти описание внешнего кольца и только затем внутренних, или дырок.

#### MultiPolygon

Для объектов типа «MultiPolygon» свойство «coordinates» должно содержать массив массивов пар/триплетов координат «Polygon».

#### GeometryCollection

Объект типа «GeometryCollection» - это геометрия, представляющая коллекцию других геометрий. Такая коллекция должна содержать свойство с именем «geometries». Значение данного свойства - массив. Каждый элемент этого массива представляет собой геометрию.

## Примеры геометрий

Каждый из приведенных далее примеров представляет собой законченный GeoJSON объект. Отметим, что не обрамленные в кавычки пробелы в JSON не принципиальны. Пробелы использованы в примерах для наглядности представления структуры данных и не являются обязательными.

### Point

Координаты объекта Point идут в порядке x, y (смещение на восток, смещение на север для прямоугольных систем координат и долгота, широта для географических системы координат):

{ "type": "Point", "coordinates": [100.0, 0.0] }

### LineString

Координаты LineString – массив пар/триплетов координат

{ "type": "LineString",

"coordinates": [ [100.0, 0.0], [101.0, 1.0] ]

}

Polygon

Координаты Polygon – массив массивов пар/триплетов координат LinearRing. Первый элемент массива описывает внешнее кольцо. Последующие элементы описывают внутренние кольца (дырки).

Без дырок:

{ "type": "Polygon",

"coordinates": [

[ [100.0, 0.0], [101.0, 0.0], [101.0, 1.0], [100.0, 1.0], [100.0, 0.0] ]

]

}

С дыркой:

{ "type": "Polygon",

"coordinates": [

[ [100.0, 0.0], [101.0, 0.0], [101.0, 1.0], [100.0, 1.0], [100.0, 0.0] ],

[ [100.2, 0.2], [100.8, 0.2], [100.8, 0.8], [100.2, 0.8], [100.2, 0.2] ]

]

}

GeometryCollection

Каждый элемент в массиве GeometryCollection – геометрия:

{ "type": "GeometryCollection",

"geometries": [

{ "type": "Point",

"coordinates": [100.0, 0.0]

},

{ "type": "LineString",

"coordinates": [ [101.0, 0.0], [102.0, 1.0] ]

}

]

}

BSON

BSON - бинарный JSON.

BSON это расширение JSON которое позволяет хранить данные в различных форматах: дата и время, бинарные данные и т.п.

В основном BSON используется в проекте MongoDB (документо ориентированная база данных) для передачи и хранения документов.

Преимущества BSON:

- Маловесный (важно т.к. используется для передачи данных по сети)

- Удобен для травестинга (необходимо т.к. он используется для представления данных в МонгоДБ)

- Эффективный (Удобно конвертировать в/из него)

BSON объект представляет из себя упорядоченный список элементов, каждый элемент содержит название поля, типа и значения. Название поля – строка.

Поле может быть таких типов:

- string;

- integer (32- or 64-bit);

- double (64-bit IEEE 754 floating point number);

- date (integer number of milliseconds since the Unix epoch);

- byte array (binary data);

- boolean (true and false);

- null;

- BSON object;

- BSON array.

Документ hello="world" будет представлен в виде:

Bson:

\x16\x00\x00\x00 // ***total document size***

\x02 // 0x02 = ***type*** String

hello\x00 // ***field name***

\x06\x00\x00\x00world\x00 // ***field value***

\x00 // 0x00 = ***type EOO*** ('end of object')

BSON можно сравнить с бинарным форматом передачи данных Proto­col Buf­fers.

BSON более "schema-less" чем Proto­col Buf­fers, что дает ему преимущество в гибкости, но недостаток в эффективности распределения места.

Google Protobuf

Protocol buffers это гибкий, эффективный, автоматизированный механизм для сериализации структурированных данных – как XML, только меньше, быстрее и проще.

Структура данных определяется единожды, после чего можно пользоваться сгенерированными классами для легкого доступа к чтению/записи структур данных из различных потоков, используя различные языки программирования. Так же можно изменять структуру данных без нарушения развертывания программ которые собраны со “старой” структурой данных.

Технология Protocol buffers была спроектирована для таких требований:

- Новые поля можно легко добавить, и промежуточным серверам не требуется проверять данные, они могут просто разбирать их и отправлять дальше, без необходимости знать все поля;

- Формат более самоописуем, и может использоваться для разных языков (C++, Java и т.д.).Тем не менее, пользователям по-прежнему проходилось в ручную писать код разбора данных. Однако, так как система развивалась, она приобрела ряд других особенностей и областей применения:

- Автоматически генерируемый код для сериализации/десериализации для отказа от ручного разбора данных;

- Помимо того, что основное использование это короткоживущие RPC (Remote Procedure Call) запросы, люди начали использовать protocol buffer как удобный самоописуемый формат для постоянного хранения данных (например в Bigtable);

- Серверные RPC интерфейсы начали объявлять как часть файла протокола, благодаря сгенерированным компилятором классам-заглушкам пользователи могут переопределять их фактической реализацией интерфейса сервера.

Protocol buffers сейчас – это гугловская lingua franca для данных – на время написания, было определено 48 162 различных типов сообщений, а в Google code содержится 12 183 .proto файлов. Они используются как в RPC системах, так и в различных системах хранения данных.

***Как это работает?***

Вы определяете структуру информации для сериализации посредством описания фомата сообщения protocol buffer в .proto файле. Каждое сообщение protocol buffer это небольшая логическая запись информации, содержащая серию пар имя-значение. Ниже представлен простой пример .proto файла, определяющего сообщение содержащее информацию о человеке:

message Person {   
required string name = 1;   
required int32 id = 2;   
optional string email = 3;   
  
enum PhoneType {   
MOBILE = 0;   
HOME = 1;   
WORK = 2;   
}   
  
message PhoneNumber {   
required string number = 1;   
optional PhoneType type = 2 [default = HOME];   
}   
  
repeated PhoneNumber phone = 4;   
} 

Как можно увидеть, формат сообщения прост – каждое сообщение имеет одно или несколько уникальных нумерованных полей, а каждое поле имеет имя и тип сообщения, где тип сообщения может быть численным (целочисленным или числом с плавающей точкой), логическим, строковым, бинарным или даже (как в вышеприведенном примере) типом описанным в другом .proto файле, что позволяет создавать иерархическую структуру данных. Вы можете определять необязательные, необходимые и повторяющиеся поля. Вы можете найти больше информации о написании .proto файлов в руководстве по языку Protocol Buffer.   
После определения сообщений, необходимо запустить предназначенный для вашего языка программирования компилятор protocol buffer, который на основе .proto файлов создаст классы доступа к данным. Они предоставляют простой доступ к каждому полю (например query() и set\_query()), а так же методы сериализации/разбора всей структуры в/из бинарных данных.

Вы можете добавить новые поля в ваше сообщение без нарушения обратной совместимости; старые бинарники просто игнорируют поля при парсинге. Так что если у вас есть протокол коммуникации который использует protocol buffers как формат передачи данных, вы можете расширять ваш протокол без боязни сломать, что либо в существующем коде. 

***Почему бы просто не использовать XML?*** 

Protocol buffers имеет множество преимуществ перед XML как способа сериализации данных. Protocol buffers: 

* проще;  
  - имеет от 3 до 10 раз меньший размер;
* от 20 до 100 раз быстрее;
* более однозначный;

- генерируемые классы доступа к данным проще для использования в коде.  
  
Например, вы хотите создать модель “человек” с полями name и email. В XML, вам пришлось бы написать нечто подобное:

*<person>   
<name>John Doe</name>   
<email>jdoe@example.com</email>   
</person>* 

в то же время соответствующее сообщение protocol buffer (в текстовом формате protocol buffer) выглядит так:

# Текстовое представление сообщения protocol buffer.   
# Это \*не\* бинарный формат использующийся при передаче.   
person {   
name: "John Doe"   
email: "jdoe@example.com"   
} 

Когда данное сообщение кодируется в бинарном формате protocol buffer (текстовый формат приведенный выше, он является простой и понятой для человека формой представления данных для отладки и редактирования), которое будет занимать около 28 байт, а разбор будет производиться примерно за 100-200 наносекунд. XML версия данного сообщения будет занимать 69 байт(если удалить пробелы), а разбор будет проходить примерно за 5 000 – 10 000 наносекунд.

### When Is JSON A Better Fit?

- You need or want data to be human readable

- Data from the service is directly consumed by a web browser

- Your server side application is written in JavaScript

- You aren’t prepared to tie the data model to a schema

- You don’t have the bandwidth to add another tool to your arsenal

- The operational burden of running a different kind of network service is too great

Однако, protocol buffers не всегда лучшее решение чем XML – для модели текстового документа с разметкой (например HTML) protocol buffers будет проигрывать XML, так как у вас не выйдет легко чередовать структуры данных с текстом. В дополнение к этому XML – это понятная для человека форма просмотра и редактирования данных, в то время как protocol buffers, в своем исходном (бинарном) формате – нет. XML – так же в некоторой степени самоописуемый, в то время как protocol buffer имеет смысл использовать, только если есть определенный формат сообщения (.proto файл).   
  
Официальный protobuf поддерживает только only Java, C++, and Python

Для того чтоб использовать их с джаваскриптом, нужно использовать

Protobuf.js: <https://github.com/dcodeIO/ProtoBuf.js> (!)

protobuf-js: <http://code.google.com/p/protobuf-js/>

protojs: <http://github.com/sirikata/protojs>

Links

https://learn.javascript.ru/json