Форматы обмена данными и протоколы

XML (eXtensible Markup Language) - развитие и упрощение SGML для представления любых видов информации (включая структуры, списки). Для описания структуры: DTD (Document Type Definition), XSD (XML Schema Definition), RelaxNG

JSON (JavaScript Object Notation) - формат сериализации javascript-объектов. Структура: JSON Schema.

YAML (Yet Another Markup Language) - формат с отступами для описания структуры. Также есть попытка создания YAML Schema

Есть ли альтернативы?

Двоичные протоколы сериализации, обычно специфичны для технологии разработки (например, RMI, PHP serialization)

Есть универсальные способы сериализации, например Google Protobuf (Protocol Buffers), Thrift, Avro (C#). Protobuf и Thrift используют IDL для описания структуры объектов, производительность и синтаксис примерно похожи.

```
message DictionaryIntToDoubleMessage {
   map<int32, double> Values = 1;
}
```

Avro определяет схемы через JSON, существенно медленнее.

XML

Избыточный (из-за повторений тэгов), но привычный (синтаксис во многом аналогичен HTML).

Есть реализация для всех существующих технологий разработки приложений (могут работать как с монолитом - представлением XML в виде дерева, так и с потоком событий)

В Javascript есть, например, jQuery.parseXML(), в браузере можно работать напрямую через XMLHttpRequest, либо через DOMParser. Или можно конвертировать в JS-объекты (xml-js).

JSON

Стандарт де-факто для передачи информации на клиентскую сторону. Не подразумевает обязательной схемы, неоднозначно типизирует данные (различаются только строки, числа и можно определить логический тип).

Фактически основной формат для RESTful (клиент-серверная модель без хранения состояния на сервере между запросами, при этом возможно сохранение связи токена с сеансом с целью аутентификации, унифицированный интерфейс доступа и манипуляцией данными, невозможность определения кэширующих и балансировочных посредников).

Что обычно используют RESTful API?

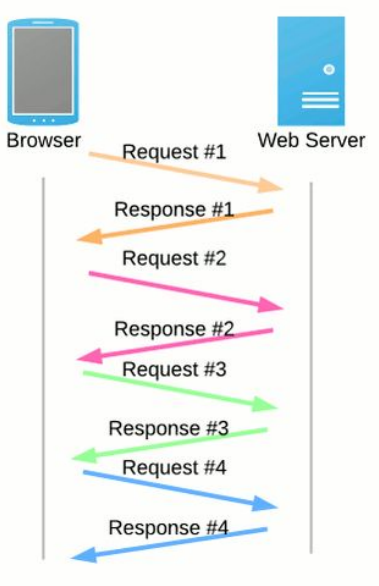
Часто реализуется поверх HTTP:

- **GET** - получение информации о списке объектов или конкретном объекте
- **POST** - отправка изменений объекта или выполнение действия (например, аутентификации)
- **PUT** - добавление нового объекта
- **DELETE** - удаление объекта

Категория и идентификация объекта выполняется через URI (например, /user/<id> указывает на пользователя с указанным идентификатором).

Недостатки RESTful.

Каждый запрос возвращает одиночный объект и, возможно, его связи. Если нужна дополнительная информация - отправляются вспомогательные запросы (а это замедление, если не используется HTTP/2.0… да и в этом случае множество раз инициализируется сессия, например в PHP)



Недостатки RESTful

Запрос не совпадает по структуре с ответом

Невозможно выбрать отдельные атрибуты, объект всегда возвращается в полном соответствии с API

Нет метаинформации о структуре объекта в API и сложно делать версионирование (можно изменять URI)

Первая попытка - JSON API

Общий подход к созданию RESTful API, рекомендации по дополнительным возможностям (получение фрагмента списка, сортировка, изменение объектов)

Создаёт расширенный json с метаданными и указаниями на типы связанных данных (+attributes, relationships)

```
"author": {
    "id": "1",
    "name": "Paul"
  },
```

```
    "relationships": {
        "author": {
          "type": "user",
          "id": "1"
        },
        "comments": {
          "type":  "comment",
          "id": "324"
        }
    }
```

JSON API

Никак не решает проблему изоморфности запроса и ответа (наоборот, добавляет много избыточности, необходимой для каскадных запросов расширенной информации о связанных объектах)

Спецификация с мая 2015 года не обновлялась, не очень хорошо работает с типами данных.

Но при этом представляется как ORM в Python, Java и Node.JS

GraphQL

Спецификация запросов к JSON-ориентированным сервисам, позволяет выбрать конкретные атрибуты и получить расширенную информацию одним запросом.

Атрибуты типизированы и выполняется дополнительная проверка типов на извлечении (query) и изменении (mutation)

GraphQL

Запросы могут быть составными (включать фрагменты) и принимать параметры, можно создавать псевдонимы атрибутов и объектов

Одна точка входа в приложение, без необходимости явно отслеживать версионирование (порядок полей определяют клиенты), ориентирован на продукт, а не на ресурсы

Пример GraphQL-запроса

```
query {
  user(id: 1) {
    name
    age
    friends {
      name
    }
  }
}
```

GraphQL

Каждый атрибут связан с функцией-генератором, которая может принимать параметры

Параметры имеют строгий тип (и можно указывать обязательность и значение по умолчанию), доступны метаданные запросов, мутаций и схемы

Не привязан ни к какому протоколу (можно использовать HTTP, WebSockets, MQTT)

Серверы GraphQL

Для функции преобразователя доступны переменные, глобальный контекст, информация о родительском контексте

Python: graphene (модель похожа на ORM, resolve функции определяются по имени)

PHP: graphql-php (описание схемы в словаре с анонимными функциями)

Java: graphql-java (специальный формат схемы graphqls)

NodeJS: apollo-server (включает graphiql, веб-инструмент для отладки), взаимодействует с express.js, connect, hapi, koa, restify, aws lambda.

Порядок описания схемы

Определяется список полей и их метаданных (типы, включая составные, необязательное описание, функцию-преобразователь resolve), можно создавать интерфейсы (и указывать реализации), новые типы (фрагменты).

Функция преобразователь получает доступ к переданным параметрам и глобальному контексту (например, подключению к БД)

Из типов данных создаётся схема, которая в дальнейшем передаётся в веб-сервер или непосредственно используется для запросов и изменений данных.

Запросы

Псевдонимы alias: field[(param: …, …)]

Директивы @include(if: ..., @skip(if: …)

Переменные $var:Type[!], ! - если обязательное поле, Type: Int, String, Boolean, любой сложный тип

Фрагменты fragment <fragment_name> on Type { структура }, ссылка ...fragment_name

Клиенты GraphQL

- **Python:** graphql-parser, django-graphql
- **PHP:** laraev-graphql, GraphQL Symfony Bundle, graphql-php
- **Java:** rbdms-to-graphql (создаёт схему по JDBC),
- **JavaScript:** graphql-js (Facebook), поддерживает Subscriptions (realtime updates), Apollo Client (есть связывание с React, Angular)
- **Базы данных:** GraphpostgresQL,
- **Kotlin:** ktq

Apollo

Универсальный клиент и сервер GraphQL, есть привязка к React, Angular, Android, iOS, meteor.

Полезно использовать: eslint-plugin-graphql (для проверки запросов по схеме), intellij-graphql (поддержка запросов внутри IDE)

Пример сервера на nodejs: https://github.com/apollographql/starwars-server

Пример ReactJS + Apollo

```
import { gql, graphql } from
'react-apollo';

function TodoApp({ data: {
todos, refetch } }) {
  return (
    <div>
      <button onClick={() =>
refetch()}>
        Refresh
      </button>
      <ul>
```

```
{todos.map(todo => (
          <li key={todo.id}>
            {todo.text}
          </li>
        ))}
      </ul>
    </div>
  );
}
```

```
export default graphql(gql`
  query TodoAppQuery {
    todos {
      id
      text
    }
  }
`)(TodoApp);
```

Что нового в Apollo 2.0?

- поддержка связывания с REST API, например, user(email: $email) @rest(route: '/users/email/:email', email: $email)
- улучшена поддержка progressive web apps и реализован поточный обмен данными (Apollo Link) для директив @defer, @live и @stream (React)
- модульная архитектура (теперь слой кэширования не зависит от Redux)
- управление состоянием приложения (для обновления локальных кэшей), пример: https://github.com/apollographql/apollo-link-state