Web Services

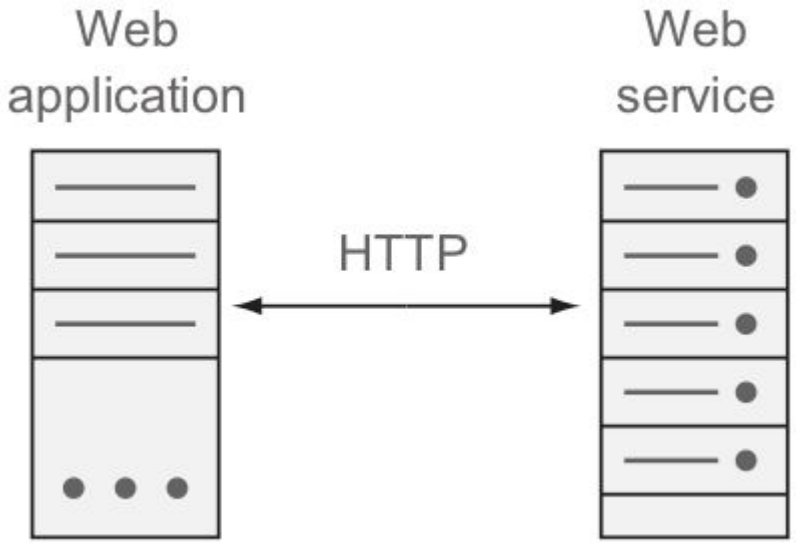
Learning Outcomes

After completing this session you should be able to explain

- Web services
- SOAP, REST, gRPC, GraphQL
- REST Concepts
- Best practices of designing RESTful API
- JSON

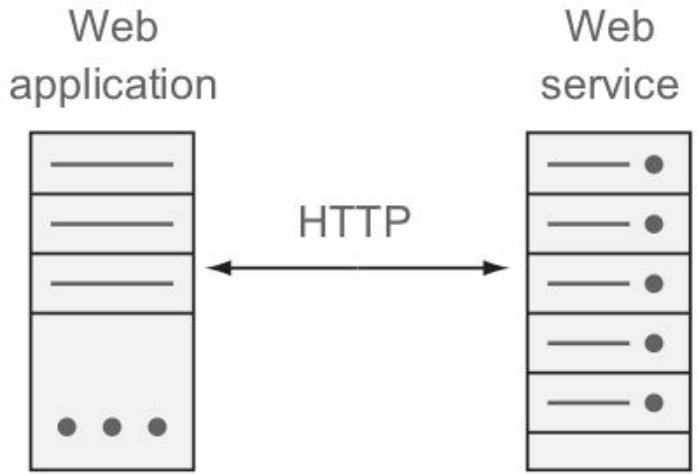
Web Services

Web services are **software programs that interact with other software programs**



Web Services

Web services **provide a standard means of interoperating between different software applications**, running on a **variety of platforms** and/or **frameworks**



Web Services

There are two popular approaches to web services: **SOAP (Simple Object Access Protocol)** and **REST (REpresentational State Transfer)**

Other approaches such as **gRPC** and **GraphQL** are also available which tries to alleviate the limitations of **SOAP** and **REST** services

Web Services: SOAP

SOAP is an **XML**-based protocol for making network API requests

Although it is most commonly used over **HTTP**, it aims to be independent from **HTTP** and avoids using most **HTTP** features

Instead, it comes with a sprawling and complex multitude of related standards (the web service framework, known as **WS-***) that add various features

Web Services: SOAP

The **API** of a **SOAP** web service is described using an **XML**–based language called the **Web Services Description Language**, or **WSDL**

WSDL enables code generation so that a client can access a remote service using local classes and method calls (which are encoded to **XML** messages and decoded again by the framework)

As **WSDL** is not designed to be human-readable, and as **SOAP** messages are often too complex to construct manually, users of **SOAP** rely heavily on tool support, code generation, and **IDEs**

Web Services: REST

REST (Representational State Transfer) is an architectural style (or design philosophy) used in designing programs that talk to each other by manipulating **resources** using a standard few **actions** (or verbs)

The REST architectural style describes six constraints

These six constraints defines the basis of **RESTful-style**

Client-Server

Stateless

Cache

Layered System

Uniform Interface

Code on Demand

Web Services: REST

Constraints	Induced Properties

• Client-Server
• Layered System
• Stateless
• Cache
• Code on Demand (Optional)
• Uniform Interface
  ○ Identification of Resources
  ○ Resource Representation
  ○ Self Descriptive Messages
  ○ HATEOAS
	• Performance
	• Scalability
	• Simplicity
	• Modifiability
	• Visibility
	• Portability
	• Reliability

Web Services: REST

When used over HTTP , a **URL** is used to **identify a resource** and **HTTP methods** are used as **verbs** to **manipulate** them

A definition format such as **`OpenAPI`**, also known as **Swagger**, can be used to **describe `RESTful` APIs** and **produce documentation**

The Uniform Interface

The central feature that distinguishes the REST architectural style from other network-based styles is its emphasis on a uniform interface between components

REST is defined by four **interface constraints**: **identification of resources**; manipulation of **resource**s through **representations**; **self-descriptive messages**; and, **hypermedia as the engine of application state**

What's a Resource?

The key abstraction of information in REST is a resource

Any information that can be named/identified can be a resource:

a document or image,

a temporal service (e.g. "today's weather in Addis Ababa"),

a non-virtual object (e.g. a person), and so on

Resource Names (URIs)

REST uses a resource identifier to identify the particular resource involved in an interaction between components

The **URI** is the **name** and **address** of a **resource**

The Relationship Between URIs and Resources

Can two **URIs** designate the same resource?

The Relationship Between URIs and Resources

Two different resources may point to the same data

Example:

> If the current software release is 1.0.3, then
> `http://www.example.com/software/releases/1.0.3.tar.gz` and
> `http://www.example.com/software/releases/ latest.tar.gz`
> will refer to the same file for a while

A resource may have one or many URIs

The Relationship Between URIs and Resources

Can a single URI designate two resources?

The Relationship Between URIs and Resources

Every URI designates exactly one resource

If a URI designated more than one resource, it wouldn't be a **Universal Resource Identifier**

Representations

REST components perform actions on a resource by using a **representation** to **capture the current or intended state of** that **resource** and transferring that representation between components

A representation is a sequence of bytes, plus representation metadata to describe those bytes

The server might **present** a list of products, for example, as an `XML/JSON` document, **a web page**, or as **comma-separated text**

Links

In the most RESTful services, **representations are hypermedia**: documents that contain not just data, but **links to other resources**

Also referred as **Hypermedia as the Engine of Application State (HATEOAS)**

HATEOAS allow clients to deal with discovering the resources and the identifiers

Links

Example:

HATOAS enabled reply for the following request

```
GET /hotels/xyz
```

The response, besides giving details about the hotel, also gives information to the client about which operations can be done against the resource and how they should be done


```
{
    "city": "Newyork",
    "display_name": "Hotel Xyz",
    "star_rating": 4,
    "links": [
        {
            "href": "xyz/book",
            "rel": "book",
            "type": "POST"
        },
        {
            "href": "xyz/rooms",
            "rel": "rooms",
            "type": "GET"
        }
    ]
}
```

Operations on Resource

There are only a few basic things you can do to a resource

HTTP provides four basic methods for the four most common operations

```
HTTP GET

HTTP PUT

HTTP POST

HTTP DELETE
```

Operations: GET

The **`HTTP GET`** method is used to retrieve (or read) a representation of a resource

In the **non-error case**, **`GET`** returns a representation in XML or JSON and an HTTP response code of **`200 (OK)`**

In an **error case**, it most often returns a **`404 (NOT FOUND)`** or **`400 (BAD REQUEST)`**

`GET` (along with **`HEAD`**) requests **should be safe and idempotent**

Operations: GET

Examples:

```
GET http://www.example.com/customers/12345

GET http://www.example.com/customers/12345/orders

GET http://www.example.com/buckets/sample
```

Operations: PUT

`PUT` is most-often utilized for **update capabilities**, `PUT`-ing to a known/existing resource `URI` with the request body containing the newly-updated representation of the original resource

However, `PUT` **can also be used to create a resource** in the case where the **resource ID is chosen by the client** instead of by the server

On successful update, return `200` (or `204` if not returning any content in the body) from a `PUT`

Operations: PUT

If using **`PUT` for create**, return **HTTP status `201`** on **successful creation**

A body in the response is optional — providing one consumes more bandwidth

It is **not necessary** to return a link via a Location header in the creation case since the client already set the resource ID

Operations: PUT

`PUT` is **not a safe operation**, in that it modifies (or creates) state on the server, but **it is idempotent**

if you create or update a resource using PUT and then make that same call again, the resource is still there and still has the same state as it did with the first call

Operations: PUT

Examples:

```
PUT http://www.example.com/customers/12345

PUT http://www.example.com/customers/12345/orders/98765

PUT http://www.example.com/buckets/secret_stuff
```

Operations: POST

The `POST` verb is most-often **utilized for creation of new resources**

In particular, **it's used to create subordinate resources**. That is, subordinate to some other (e.g. parent) resource

When creating a new resource with `POST` request to the parent resource, the service takes care of associating the new resource with the parent and assigning an ID (new resource URI) to it

Operations: POST

On **successful creation**, return **HTTP status** `201`, returning a `Location` **header with a link to the newly created resource**

`POST` is **neither safe or idempotent**

`POST` is recommended for **non-idempotent resource requests**

Making two identical `POST` requests will most-likely result in two resources containing the same information

Operations: POST

Examples:

```
POST http://www.example.com/customers

POST http://www.example.com/customers/12345/orders
```

Operations: DELETE

`DELETE` is used to remove a resource identified by a URI

On **successful deletion**, return **HTTP status `200 (OK)`** along with a response body, the representation of the deleted item (often demands too much bandwidth) or return **HTTP status `204 (NO CONTENT)`** with no response body

`DELETE` operations are **idempotent**

If you `DELETE` a resource, it's removed. Repeatedly calling `DELETE` on that resource ends up the same: the resource is gone

Operations: DELETE

Examples:

```
DELETE http://www.example.com/customers/12345

DELETE http://www.example.com/customers/12345/orders

DELETE http://www.example.com/buckets/sample
```

Operations: HEAD

`HTTP HEAD`: Retrieve a metadata-only representation

A client can use **`HEAD`** to check whether a resource exists, or find out other information about the resource, without fetching its entire representation

`HEAD` gives you exactly what a **`GET`** request would give you, but without the entity-body

Operations: OPTIONS

`HTTP OPTIONS`: Check which HTTP methods a particular resource supports

The **`OPTIONS`** method lets the client discover what it's allowed to do to a resource

The response to an **`OPTIONS`** request contains the **HTTP Allow header**, which lays out the subset of the operations this resource supports

Here's a sample Allow header:

```
Allow: GET, HEAD
```

Operations Summary

What should be effect of the following requests?

What status code (message) be returned to the client, when the operation succeeds or when it fails

HTTP Verb	/customers	/customers/{id}
GET		
PUT		
POST		
DELETE		

Operations Summary

HTTP Verb	`/customers`	`/customers/{id}`
`GET`	200 (OK), list of customers. Use pagination, sorting and filtering to navigate big lists	200 (OK), single customer. 404 (Not Found), if ID not found or invalid
`PUT`	404 (Not Found), unless you want to update/replace every resource in the entire collection	200 (OK) or 204 (No Content). 404 (Not Found), if ID not found or invalid
`POST`	201 (Created), 'Location' header with link to /customers/{id} containing new ID	404 (Not Found)
`DELETE`	404 (Not Found), unless you want to delete the whole collection (not often desirable)	200 (OK). 404 (Not Found), if ID not found or invalid

Going beyond the HTTP Operations

How do you activate a customer account?

REST doesn't allow you to have arbitrary operations on the resources, and you're more or less restricted to the list of available HTTP methods, so you can't have a request that looks like this:

```
ACTIVATE /user/456 HTTP/1.1
```

Two most common approaches to handle this issue

Convert action to a resource

Make the action a property of the resource

Convert action to a resource

Example: activating a user

```
POST /user/456/activation HTTP/1.1

{ "date": "2015-05-15T13:05:05Z" }
```

This code will create an activation resource that represents the activation state of the user

Doing this also gives the added advantage of giving the activation resource additional properties (date in this example)

Make the action a property of the resource

If activation is a simple state of the customer account, you can simply make the action a property of the resource, and then use the **`PATCH HTTP`** method to do a partial update to the resource

Example:

```
PATCH /user/456 HTTP/1.1

{ "active" : "true" }
```

REST Best Practices

Use HTTP Verbs to Mean Something

Any API consumer is capable of sending GET, POST, PUT, and DELETE verbs, and they greatly enhance the clarity of what a given request does

Also, GET requests must not change any underlying resource data.

Measurements and tracking may still occur, which updates data, but not resource data identified by the URI

REST Best Practices

Sensible Resource Names

Having sensible resource names or paths (e.g., `/posts/23` instead of `/api?type=posts&id=23`) improves the clarity of what a given request does

Appropriate resource names provide context for a service request, increasing understandability of the service API

Resources are viewed hierarchically via their URI names, offering consumers a friendly, easily-understood hierarchy of resources to leverage in their applications

REST Best Practices

Sensible Resource Names

Resource names should be nouns, avoid verbs as resource names.

Use the HTTP methods to specify the verb portion of the request

REST Best Practices

XML and JSON

Favor JSON support as the default, but unless the costs of offering both JSON and XML are staggering, offer them both

REST Best Practices

Create Fine-Grained Resources

It's much easier to create larger resources later from individual resources than it is to create fine-grained or individual resources from larger aggregates

Exercise

Let's say we're describing an order system with customers, orders, line items, products, etc.

Design the URIs involved in describing the resources in this service

To read a customer with Customer id 33245

For creating a new Product

For reading, updating, deleting Product with id 66432

For creating a new Order for a Customer with id 33245

Resource URI Examples

To insert (create) a new customer in the system

```
POST http://www.example.com/customers
```

To read a customer with Customer id 33245

```
GET http://www.example.com/customers/33245
```

The same URI would be used for PUT and DELETE, to update and delete, respectively

Resource URI Examples

For creating a new Product

```
POST http://www.example.com/products
```

For reading, updating, deleting Product with id 66432

```
GET|PUT|DELETE http://www.example.com/products/66432
```

For creating a new order for a customer with id 33245

```
POST http://www.example.com/customers/33245/orders
```

Resource URI Examples

For retrieving an order by its id without having to know the customer

```
GET http://www.example.com/orders/8769
```

To return only the first line item in same order

```
GET
http://www.example.com/customers/33245/orders/8769/lineitems
/1
```

Resource URI Examples

For getting list of orders that customer with id 33245 has created or owns

```
GET http://www.example.com/customers/33245/orders
```

To add a line item to order #8769 (which is for customer #33245)

```
POST
http://www.example.com/customers/33245/orders/8769/lineitems
```

Resource URI Examples

There aren't any hard and fast rules, only make sure the imposed structure makes sense to consumers of your services

More info at

https://www.restapitutorial.com/resources.html

Other Guidelines

https://github.com/microsoft/api-guidelines/blob/vNext/Guidelines.md
https://cloud.google.com/apis/design/resources#what_is_a_rest_api

Other Issues in RESTful API design

Querying, **Filtering,** and **Pagination**

Filtering and **Sorting** Results

Service **Versioning**

Date/Time Handling

Securing Services

Caching and **Scalability**

REST Reference

https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

https://www.restapitutorial.com/

JSON (ECMA-404, RFC-8259)

`JavaScript Object Notation (JSON)` is a lightweight, text-based, language independent data interchange format

It is a text format for the serialization of structured data

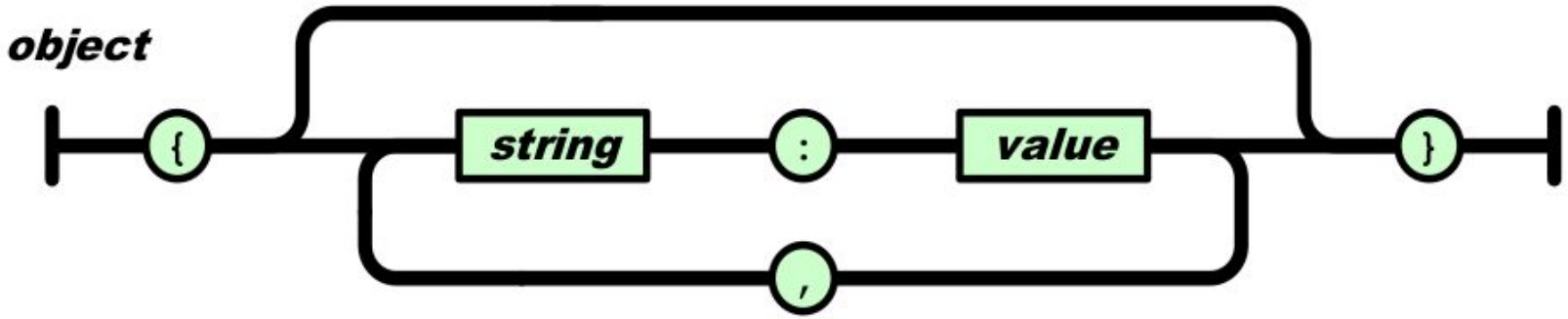
`JSON` can represent **four primitive types** (**`strings`**, **`numbers`**, **`booleans`**, and **`null`**) and **two structured types** (**objects** and **arrays**)

A **`string`** is a sequence of zero or more Unicode characters wrapped with quotation marks

JSON (ECMA-404, RFC-8259)

An **`object`** structure is represented as a **pair of curly brackets surrounding** unordered collection of **zero or more `name`/`value` pairs (or members)**

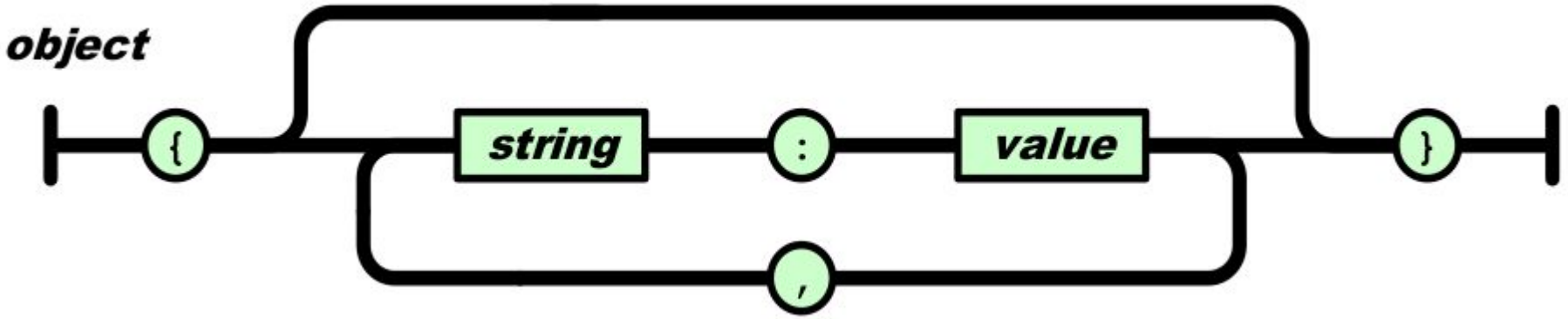
A **`name`** is a **`string`** and a **`value`** is a **`string`**, **`number`**, **`boolean`**, **`null`**, **`object`**, or **`array`**



JSON (ECMA-404, RFC-8259)

A **single colon** comes **after each name**, **separating the `name` from the `value`** and a **single comma** separates a `value` from a **following `name`**

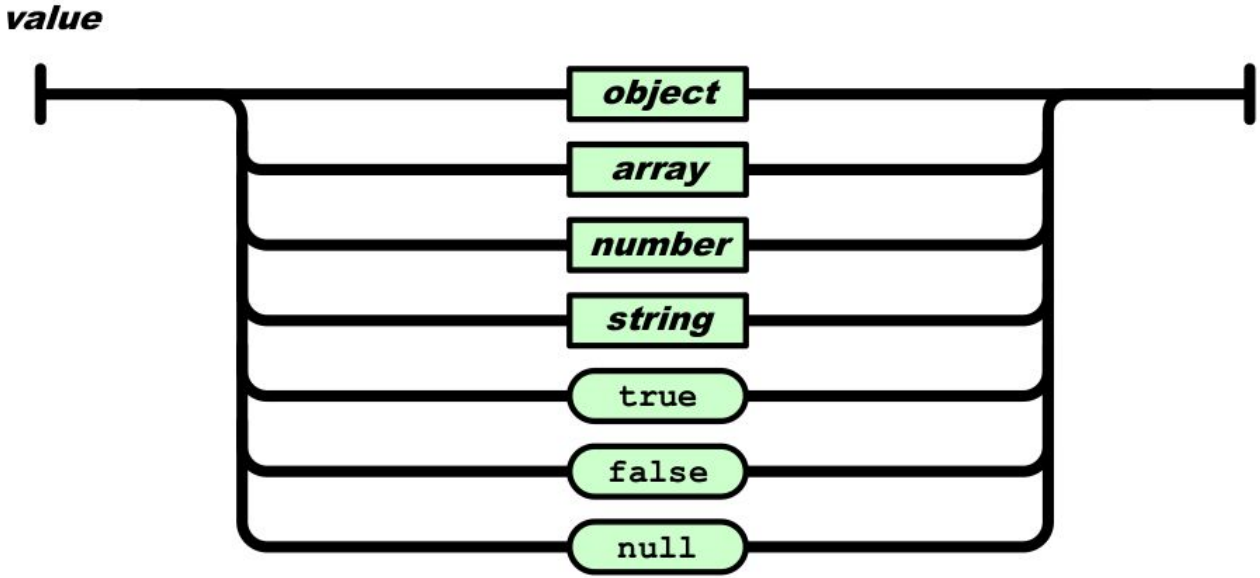
The **names** within an `object` **SHOULD be unique**



JSON (ECMA-404, RFC-8259)

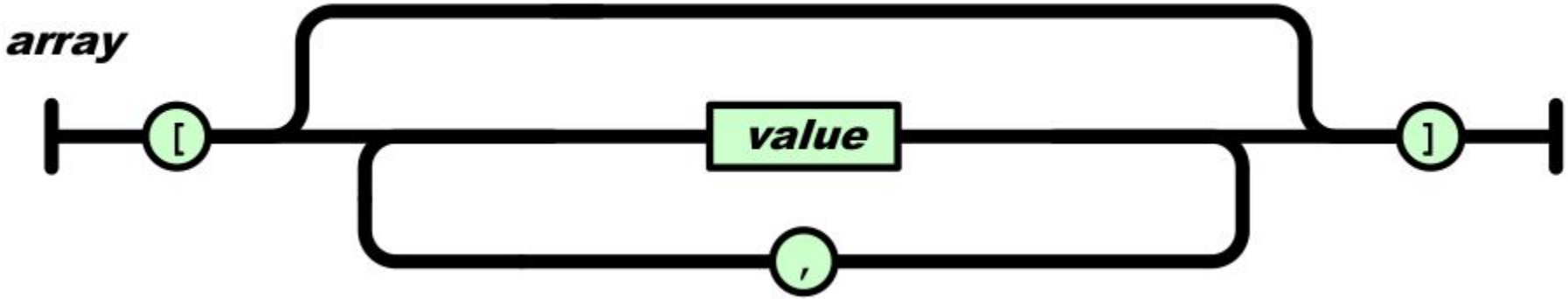
A **`value`** MUST be an **`object`**, **`array`**, **`number`**, or **`string`**, or one of the following **three literal names**: **`false`**, **`null`**, **`true`**

The literal names MUST be lowercase



JSON (ECMA-404, RFC-8259)

An **`array`** is an ordered sequence of zero or more **`values`**



JSON Object Example

```
{
    "Image": {
        "Width": 800,
        "Height": 600,
        "Title": "View from 15th Floor",
        "Thumbnail": {
            "Url": "http://www.expl.com/img/4",
            "Height": 125,
            "Width": 100
        },
        "Animated": false,
        "IDs": [116,943,234, 38793]
    }
}
```

Example JSON array containing two objects


```
[
    {
        "precision": "zip",
        "Latitude": 37.7668,
        "Longitude": -122.3959,
        "Country": "US"
    },
    {
        "precision": "zip",
        "Latitude": 37.371991,
        "Longitude": -122.026020,
        "Country": "US"
    }
]
```

Other Web Services

`Remote Procedure Call (RPC)` is one of the technology that allow one program to interact with another (SOAP is RPC like)

The **`RPC`** model tries to **make a request to a remote network service look the same as calling a function or method** in your programming language, **within the same process** (this abstraction is called **location transparency**)

Other Web Services

Some of the example of recent Web Service related technologies includes

- **gRPC** from Google
- **GraphQL** from Facebook

gRPC

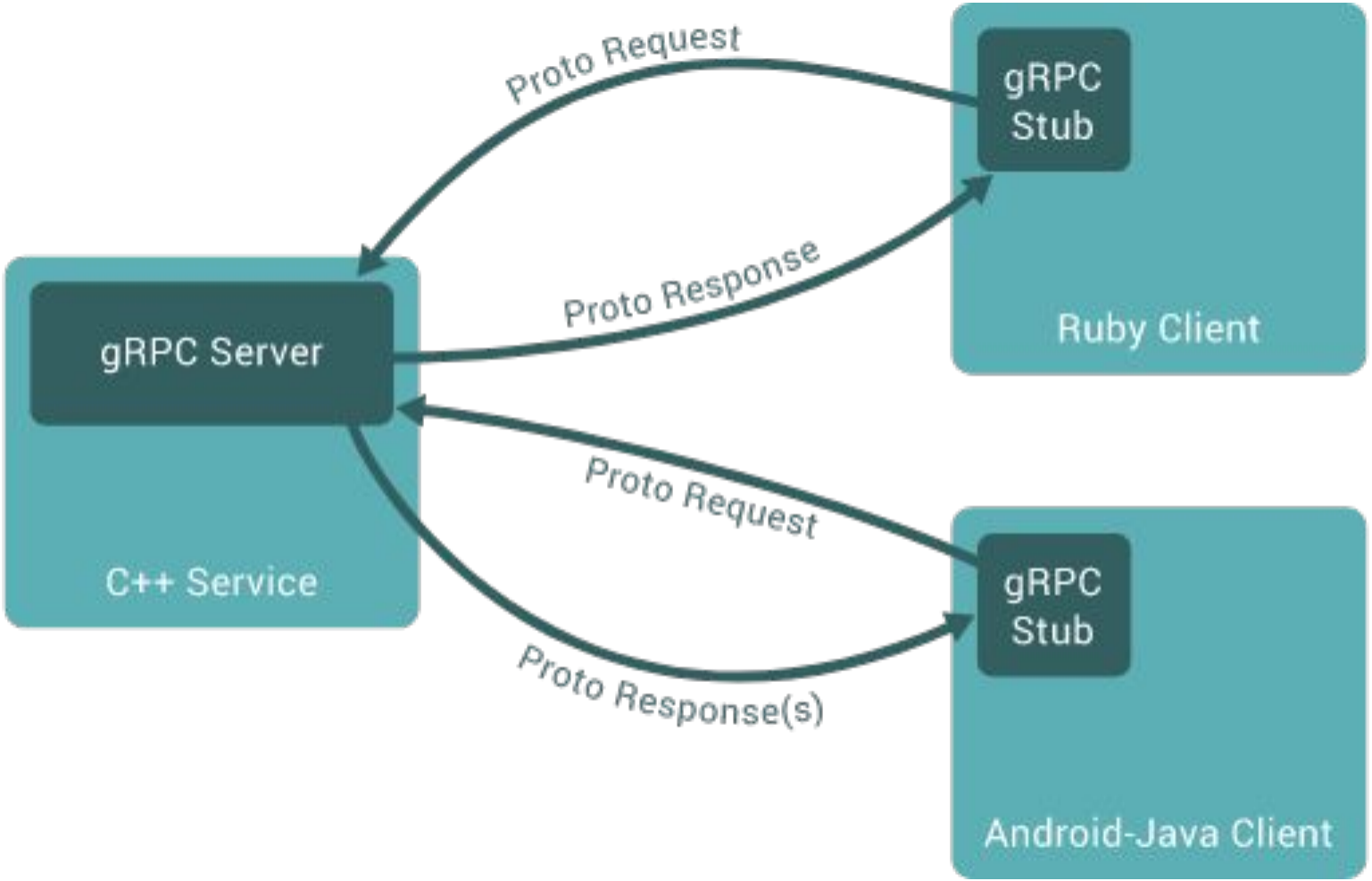
In `gRPC` a client application can directly call methods on a server application on a different machine as if it was a local object

As in many RPC systems, gRPC is based around the idea of **defining a service**, **specifying the methods that can be called remotely** with their parameters and return types

On the server side, the server implements this interface and runs a gRPC server to handle client calls

On the client side, the client has a stub that provides the same methods as the server

gRPC


Proto Request
gRPC Stub
Proto Response
gRPC Server
Ruby Client
Proto Request
gRPC Stub
Proto Response(s)
C++ Service
Android-Java Client

gRPC and Protocol Buffers

Protocol Buffer is Google's mature open source mechanism for **serializing** structured data

By default, **gRPC uses protocol buffers as the Interface Definition Language** (**IDL**) for describing both the service interface and the structure of the payload messages

gRPC can also be used with other data formats such as JSON

Service Definition

```
service HelloService {
  rpc SayHello (HelloRequest) returns (HelloResponse);
}

message HelloRequest {
  string greeting = 1;
}

message HelloResponse {
  string reply = 1;
}
```

Four Kinds of Services

Unary RPCs where the client sends a single request to the server and gets a single response back

```
rpc SayHello(HelloRequest) returns (HelloResponse){
}
```

Four Kinds of Services

Server streaming RPCs where the client sends a request to the server and gets a stream to read a sequence of messages back

The client reads from the returned stream until there are no more messages.

gRPC guarantees message ordering within an individual RPC call

```
rpc LotsOfReplies(HelloRequest) returns (stream HelloResponse){
}
```

Four Kinds of Services

Client streaming RPCs where the client writes a sequence of messages and sends them to the server, again using a provided stream

Once the client has finished writing the messages, it waits for the server to read them and return its response

Again gRPC guarantees message ordering within an individual RPC call

```
rpc LotsOfGreetings(stream HelloRequest) returns (HelloResponse) {
}
```

Four Kinds of Services

Bidirectional streaming RPCs where both sides send a sequence of messages using a read-write stream

The two streams operate independently, so clients and servers can read and write in whatever order they like

The order of messages in each stream is preserved

```
rpc BidiHello(stream HelloRequest) returns (stream HelloResponse){
}
```

Using the API

After defining the service, you can use protocol buffer compiler (called `protoc`) to generate client- and server-side interface

On the server side, the server implements the methods declared by the service and runs a gRPC server to handle client calls

The gRPC infrastructure decodes incoming requests, executes service methods, and encodes service responses

Using the API

On the client side, the client has a local object known as **stub** (for some languages, the preferred term is client) that implements the same methods as the service

The client can then just call those methods on the local object, wrapping the parameters for the call in the appropriate protocol buffer message type

gRPC looks after sending the request(s) to the server and returning the server's protocol buffer response(s)

gRPC and Protocol Buffer References

https://grpc.io/docs/guides/

https://developers.google.com/protocol-buffers/docs/gotutorial

Advantages of Using gRPC

With gRPC we can define our service once in a `.proto` file and **implement clients and servers in any of gRPC's supported languages**

The service can be run in environments ranging from servers inside Google to your own tablet - **all the complexity of communication between different languages and environments is handled for you by gRPC**

You also get all the **advantages of working with protocol buffers**, including **efficient serialization**, a **simple IDL**, and **easy interface updating**

GraphQL

`GraphQL` is a **query language for your API,** and a server-side runtime for executing queries by using a type system you define for your data

GraphQL isn't tied to any specific database or storage engine and is instead backed by your existing code and data

GraphQL

A **`GraphQL`** service is created by defining types, and fields on those types, then providing functions for each field on each type

Once a **`GraphQL`** service is running (typically at a URL on a web service), it can be sent **`GraphQL`** queries to validate and execute

A received query is first checked to ensure it only refers to the types and fields defined, then runs the provided functions to produce a result.

GraphQL

GraphQL uses the **`Schema Definition Language (SDL)`**

Example

```
type Hotel {
  id: String!
  displayName: String!
  city: String !
  noRooms: Int
  starRating: Int
}
```

GraphQL Queries

Lets, look at how retrieving data works in GraphQL

Below is shown example query for the id part of all hotels in the previous schema

This will be sent from the client to the server

```
{
  allHotels {
    id
  }
}
```

The **`allHotels`** field in this query is called the **root** field of the query

Everything under the root field is the **payload** of the query

GraphQL Queries

The server will respond with a **JSON** detailing all the hotels, ids in the database

```
{
  "allHotels": [
    { "id": "xyz" },
    { "id": "abc" },
    { "id": "pqr" }
  ]
}
```

GraphQL Queries

If the client needs the **`displayName`**, for example, it has to ask for it explicitly in the query

```
{
  allHotels {
    Id
    displayName
  }
}
```

Queries can also be explicitly specified by the root field name, and can take arguments

```
{
  allHotels (city: Delhi) {
    id
    displayName
  }
}
```