Cloud Applications Architecture

Course 13 - API Design

Definition

API = Application Programming Interface

Might refer to different layers:

- Operating system functionality (i.e. system call) (e.g. for reading files)
- Communication between tiers (e.g. ORMs for interacting with the DB)
- Communication between services/applications over the network
    - This is the subject of the course

Motivation

Adoption/Popularity of APIs in general

- Both consumption and production
  - Many SaaS products offer APIs (e.g. Stripe)
- Even for large corporations
  - Traditionally they implemented their own software

Microservices architecture

- Common approach for new projects
- Services need a way to communicate to each other

Consideration

While there are multiple alternatives for designing APIs, there is no approach or technology that fits all cases.

Our duty is to choose the approach that makes the most sense for our current use case/needs.

Some designs might facilitate performance, others discoverability while others might be built to support a wide range of use cases.

Contexts/Use-Cases

Mobile applications

Websites

Inter-service communication (e.g. microservices)

Legacy systems (e.g. older ERPs/CRMs)

High-performance compute

API Styles

Procedure/Action Oriented

- Data sits behind functions
- Similar to writing code in general
- Emphasizes the **behavior** of the system
- Usually results in **large** set of procedures (and is ever expanding)
- Each system has its own set of procedures

Entity/Resource Oriented

- Data is the main actor
- The behavior of the system is based on the data
- Similar to working with databases
  - System-specific data model
  - Common well-known features/capabilities
- Usually results in concise set of operations that can be used across multiple use cases

Data Formats

JSON

XML

YAML

Protocol Buffers (Protobuf)

Avro

Parquet

...and more - check out the comparison on wikipedia

Overview

API styles over time, Source: Rob Crowley

RPC

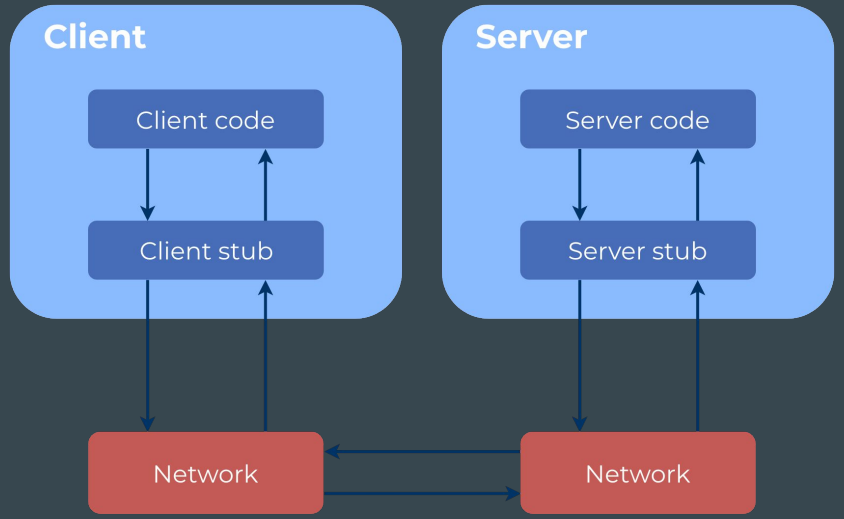
Remote **P**rocedure **C**all

Instead of calling a function/method locally, we call over the network in a different system.

Implemented by various protocols

- XML-RPC, JSON-RPC
- gRPC (used in many Google products, e.g. firebase; also heavily used in microservices architectures)

Tends to achieve high performance due to low overhead.




```
--> {"jsonrpc": "2.0", "method": "subtract", "params": [42, 23], "id": 1}
<-- {"jsonrpc": "2.0", "result": 19, "id": 1}

--> {"jsonrpc": "2.0", "method": "subtract", "params": [23, 42], "id": 2}
<-- {"jsonrpc": "2.0", "result": -19, "id": 2}
```

Example from the JSON-RPC Spec

SOAP

Stands for **S**imple Object **A**ccess **P**rotocol

One of the first protocols to standardize how information is exchanged over the network

Based on RPC

Only works with XML

WSDL (Web Service Description Language)

- Used to describe the capabilities of a web server
- Tools/frameworks can generate stubs/code based on it

SOAP

Pros

- Widely used and adopted
- Establishes an interface contract
  - Important for businesses
- Large number of extensions
  - WS-Addressing, WS-Security, etc.
  - See more here

Cons

- XML only
- Not suitable for high performance/throughput requirements
  - XML tends to be verbose and slow to parse
- Tight coupling (since it’s RPC)

REST

REpresentational **S**tate **T**ransfer

Introduced by Roy Fielding in his doctoral thesis in 2000 - link

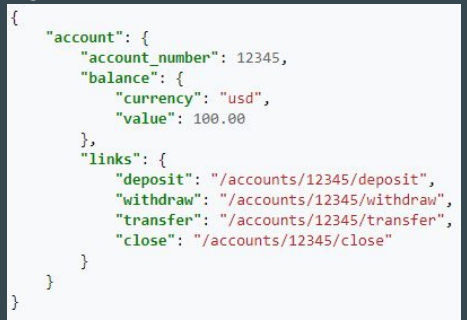
Most commonly performed over HTTP (Roy also contributed heavily to HTTP/1.1)

- Resources (e.g. cars, comments, movies) are represented by URLs (e.g. https://icdb/cars/1)
- Aspects such as cacheability and data format can be controlled through headers
- Each resource might support multiple operations, each corresponding to an HTTP verb
  - List: GET /movies
  - Read: GET /movies/1
  - Create: POST /movies
  - Update: PUT /movies/1 (expects the entire representation of the movie), PATCH /movies/1 (expects only the fields that must be updated)
  - Delete: DELETE /movies/1
- HTTP status codes indicate the result of the operations (e.g. 201 = resource created)

REST

It introduces a set of constraints such as **statelessness**, **cacheability**, and the most distinctive one, **uniform interface** which requires:

- **Resources are identified by URIs**
  - E.g. when reading the details about a car, besides the expected information such as model and engine, we also get a URL pointing to the details about the manufacturer and the URL of the car itself
- **Representations of the underlying resources provide enough information to enable us to change the resource's state.**
- **Self-descriptive messages (requests/responses)**
  - Each message contains all the information needed to be able to consume it (e.g. contains the format of the data)
- **HATEOAS**



REST

In practice, only a small percentage of APIS can truly be considered **RESTful** (mostly because of missing HATEOAS and URL identifiers).

For simplicity (and in lack of a better term), the term REST is commonly used to identify **resource-based** APIs that follow the **HTTP conventions** (verbs, status codes, headers).

While there are many good examples of REST APIs, SWAPI (Star Wars API) is among the first examples people give.

Learn more about REST here

REST Standardization

OData (Open Data Protocol)

- Rarely used by new projects, but still heavily used by Microsoft (e.g. the Azure REST API) and SAP among others.
- Standardizes aspects such as:
  - Filtering: e.g. *$filter=createdAt ge datetime'2017-06-01T00:00:00'* (entities created after 01.06.2017)
  - The service document (the root of the API): provides links to the resources
  - The metadata document (/$metadata): provides a representation of the data model and the supported operations.

Message improvement standards (format and/or description - hypermedia)

- JSON:API, HAL, JSON-LD, Collection+JSON, Siren
- See this article for more info and how to choose between them

OpenAPI/Swagger

A standardized way to describe RESTful services (somewhat similar to WSDL)

Useful for:

- Generating documentation (tools such as Swagger UI and ReDoc)
- Helping developers understand the API
- Generating clients/SDKs and server stubs (tools such as OpenAPI Generator)
- Exposing APIs through certain API Gateways (e.g. AWS API Gateway, Azure API Management, Google Apigee).
- Testing the API (e.g. Postman can import OpenAPI specs)

There are other alternatives such as RAML, API Blueprint, WADL

It’s recommended to maintain a specification when building a RESTful API.

REST

Pros

- **Widely used and adopted**
- (Somewhat) Loose coupling
- Well suited for public APIs
- Low overhead

Cons

- Multiple requests might be needed to retrieve all the data for a given page/screen
- Always returns all the fields of the resource (overfetching) (relevant especially for mobile devices and/or very large data sets)
- No shared definition/specification of how to build REST APIs

GraphQL

Open sourced by Facebook in 2015

- Internally used since 2012 for the mobile apps
- Designed specifically for optimizing APIs at their scale (and traffic)

Has 3 main operations:

- Queries - for reading data
- Mutations - for changing data
- Subscriptions - for real-time data reading

Examples: Star Wars GraphQL API, SpaceX API, Github

Do not confuse with Microsoft and Facebook Graph APIs (which are RESTful)

GraphQL

Pros

- Shared definition (everyone implements GraphQL APIs in the same way)
- Introspection (also provides strong data typing) (enables service discovery - this is why Graph*i*QL works) (think of OpenAPI for REST)
- **Specify what to fetch**
- Versioning (likely) not needed (Facebook didn't have to version their GraphQL API since 2014)

Cons

- Hard to reference entities from other systems
- Hard to cache (client libraries such as Apollo usually handle caching though)
- Adds complexity (on both the server and the client)
- Hard to scale (mostly due to real-time functionality)
- Doesn't support proxies

Related Cloud Services

API Gateways/Management Services

- AWS API Gateway
- Azure API Management
- Google Apigee
- Kong

Managed Services

- AWS AppSync
- Hasura

Callable Functions

- AWS and Google functions can be called as RPC (through their SDKs)

Resources

RPC vs REST vs GraphQL

Altexsoft comparison of API Styles

Designing Quality APIs (Cloud Next '18)

GraphQL: The Documentary

Building Modern APIs with GraphQL (great intro to GraphQL)

AWS Article: SOAP vs. REST