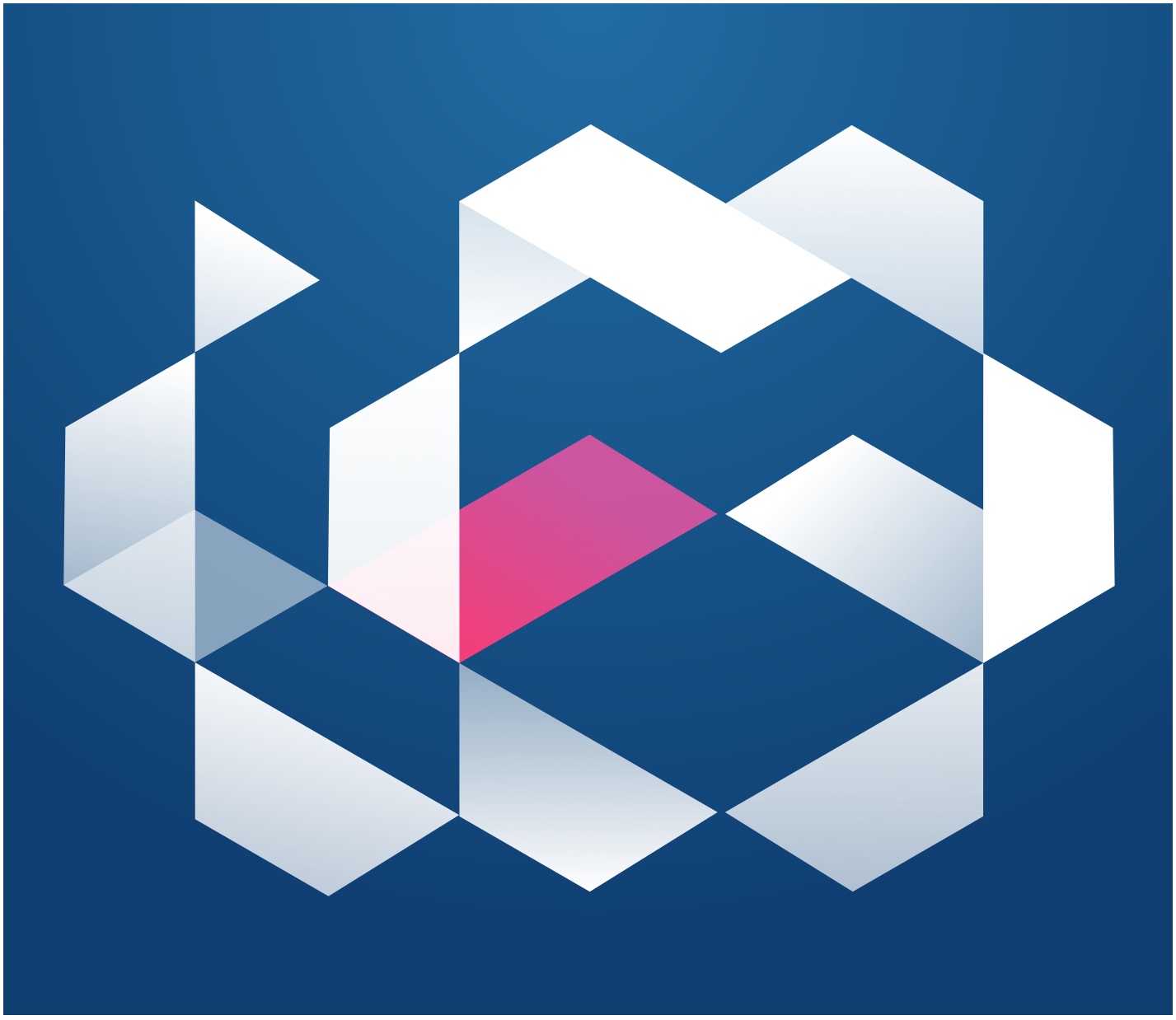
METADEV

API Technologies Review



Pedro J. Molina, PhD

Revision 1.0

https://metadev.pro

Published in Seville, 2023.03.29

Metadev Whitepaper: API Technologies Review.

Cover design & image: *Paco Soria, 2023.*
Produced with LaTeX / MiKTeX and pdfTeX.

Designed to be read in digital media.
Consider it twice before printing, to save trees.

METADEV

About

Metadev S.L. is a company devoted to creating high quality software on the cloud. Metadev helps its customers to improve processes, train technical staff, and recommend practices for excellence in development and operation optimizing ROI on Software investments.
https://metadev.pro | info@metadev.pro

Authors

Pedro J. Molina, PhD
Metadev's Founder

Pedro J. has more than 25 years of experience developing software and helping business customers to maximize ROI on technology. He also represent the ISA Group from the University of Seville in the OpenAPI Initiative (part of the *Linux Foundation*).

Linkedin: linkendin.com/in/pjmolina — Twitter: @pmolinam

Executive Summary

Review of the state of the art of API technologies. Recomendations & criteria for choosing a technological stack for building APIs.

Contents

1 Goals

- Review the state of the art of technologies for building APIs.
- Analize pros & cons of different foundations.

Expected audience: CTOs, Developers, & Software Architects.

2 APIs & Requirements

APIs *(Application Programmer Intefaces)* are pervasive nowadays. They evolved from old RPC *(Remote Procedure Calls)*. The success of many APIs is key to understand the development of distributed services on the Internet, the creation of ecosystems and platforms of services, & the raising of the API Economony on top of that.

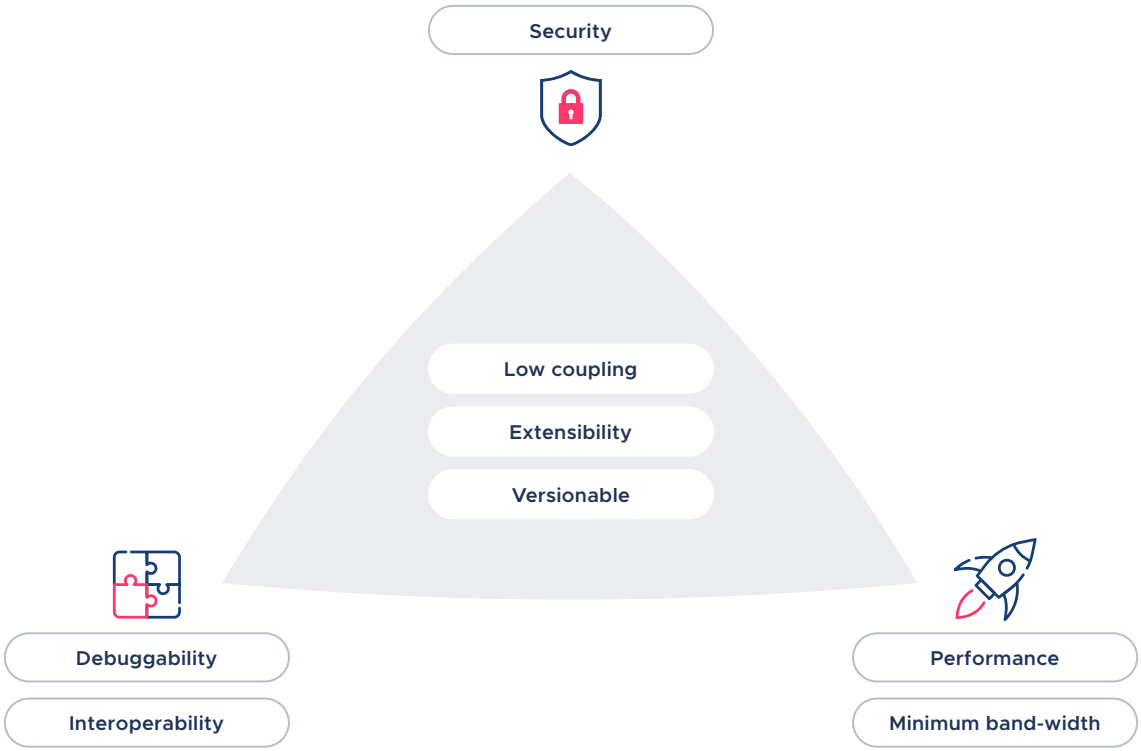
Clean APIs opens the doors to company integrations, collaboration, delegation of non-core tasks, processes outsourcing, etc. bringing over the table leverage points for growing business to optimize its costs and nurture an ecosystem around the services provided (the platforms).

2.1 Requirements

Traditional requirements for APIs comprise selecting a subset of the following ones. Tradeoffs between them do not allow to get some of them without sacrificing, at the same time, others.

- **Interoperability:** is the capability of been able to work well with different languages and technological stacks. This requires not-depending on specific language or stack features, using a neutral specification that needs to be mapped later to each language. The World Wide Web is based on this principle for data and APIs to perdure in the long term (years & decades). Open data like science, public data, and public domain sources should enforce this requirement to ensure it is still usable years later on after it was deployed.
- **Debuggability:** is the capability of been inspected to understand the messages involved when something is not working as expected. The more close to text form, the more easy is to review. On the contrary, closer to binary forms are less debuggable (at least directly) and requires support with specfic tooling to desearialize and inspect.
- **Performance:** the overal speed on serilizing, deserializing, and sending messages over the wire. The faster, the better.

- **Minimum band-width:** refers to the size of the message in-transit. Binary formats and compression helps to minimize the size of the packets, reducing comunication time, usage, & congestion in the network.
- **Security:** provides warraties to protect messages from the eyes of non-authorized recipients or interception by a thrid-party, to discover if the message has been tampered, and signed by the emitter to check its authenticity. The price involved is extra band-width and extra message exchanges.
- **Extensibility:** refers to the capability to extend the message in the future without needing to replace (redistribute and install) the client or the provider. This requirement is vital when the number of clients and servers is high enough to discard the costs of upgrading the full network or you loose control about when a client or server can be upgraded.
- **Versionable:** is the capability to send newer versions of the message and negotiate what versions to use between clients and providers.
- **Low coupling:** refers to the capability of disconnecting, as much as possible, server implementation from client implementation in a way both can evolve independiently without breaking each other (or needed to replace both at the same time).



Tension between requirements:

Interoperability & *Debuggability* are opposed to *Performance* & *Band-width*. In particular, any extra functional feature like *Extensibility*, *Versionable* or, *Low coupling* will have a cost both in terms of *Performance* and/or *Minimum band-width*. In the same way, in most of scenarios *Security* is no-negotiable and introduces similar costs.

Of course, adding more requirements, will add extra complexity to the mix. This explains why there are different use cases leading to picking different sets of requirements and, therefore, arriving to different solutions for a similar problem: *remote calls*.

Let's explore the main technologies available in the following section.

3 Technologies for APIs

In this section, we will review the technology stacks and standards available to setup an API, looking at encoding formats, protocols, and standars itselfs.

3.1 Encoding formats

An encoding format controls how data is send to & received from a given computation node. As discussed berofe, the main encoding formats prioritize tradeoffs depending of the use cases.

Let's review the more prominents:

3.1.1 XML

XML or *(eXtensible Markup Language)* is a pervasive format derived from SGML used in multiples enviroments. It was used as the fundation for SOAP.

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
  <person name="Jessica">
    <age>24</age>
    <online>true</online>
  </person>
</xml>
```

Pros:

- Human readable.
- Extensibility built-in via imported schemas and namespaces.
- Has a rich ecosystem of tooling for processing, validation, transform, etc.
- Documents can be validated using Schemas.

Cons:

- Very verbose (open and closing tags) implies a big size containing a lot of redundancy (it can be reduced adding compression on top of it).
- Slower to process than other formats.

3.1.2 JSON

JSON *(JavaScript Object Notation)* is a lightweigh encoding method using the rules for describing JavaScript objects. Therefore, the values can be direclty parsed or **eval**uated by any JS interpreter. JSON replaced XML in early smartphones due to its simplicity of parsing. XML parsers where out of scope for early smartphones with constrained computation capabilities (nowadays, this is not the case anymore).

Example:

```
{
  "name": "Jessica",
  "age": 24,
  "online": true,
  "keys": [5, 6, 7]
}
```

Pros:

- Human readable & simple.
- Easy to generate & consume from JavaScript.
- Extensible.
- Fast to emit (serialize) and to parse (deserialize).

Cons:

- Very limited data types forcing many types to be encoded as strings (f.e. dates and datimes).
- Verbose. Size with a lot of redundancy (can be reduce with compression).
- Untyped. Needs external optional tools to check its contents (like f.e. JSON Schema).

3.1.3 BSON

BSON *(Binary JSON)* was introduced by MongoDB to reduce the size of the message over the wire providing a more compact and smaller message (efficient in transmission). At the same time, it has the same semantics as JSON (same expressiveness power). Thus, able to transform from/to JSON as needed.

Example:

```
{"hello": "world"} ->
\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')
```

Pros:

- Compact, small size.
- Performant.
- Extensible.

Cons:

- No direct human-readable. Harder to debug. Requires conversion to be examined.
- Limited data types (example: dates as strings). Same limitations as JSON.
- Untyped messages.

3.1.4 Protocol Buffers

Protocol Buffers (also known as *Protobuff*)[5] is a binary serialization format created by Google. Nowadays, is used in gRPC and long-term data storage.

Example:

Protobuff declares typed structured messages in a language independent form. Different binding are provided to consume and produce these messages from different languages.

```
message Person {
    optional string name = 1;
    optional int32 age = 2;
    optional string email = 3;
    optional bool online = 4;
}
```

Pros:

- Compact.
- Best performance.
- Extensible.
- Strongly typed.

Cons:

- Non human readable. Harder to debug. Requires tooling to deserialize data to be examined.

3.1.5 Apache Avro

Apache Avro[2] is another popular binary serialization format created by the Apache Hadoop project. It is oriented to be used on Big Data contexts with data streaming with map/peduce operations.

Example:

In a similar maner to Protobuf, Avro defines schemas to type the message format. See example:

```
{
  "type" : "record",
  "namespace" : "Metadev",
  "name" : "User",
  "fields" : [
    { "name" : "Name" , "type" : "string" },
    { "name" : "Age" , "type" : "int" },
    { "name" : "Email" , "type" : "string" },
    { "name" : "Online" , "type" : "bool" }
  ]
}
```

Pros:

- Compact.
- Extensible.
- Strongly typed.
- Support for dynamic schemas.
- Oriented to big data.

Cons:

- Slower than Protobuff.
- Non human readable. Harder to debug. Requires tooling to deserialize data to be examined.

3.2 Transport Protocols

Peer to peer communication needs to be carried over a channel. Channels has precise rules for dealing with the traffic and interpret the messages. Let's review the most frequently used transport protocols for APIs.

3.2.1 HTTP

HTTP[3] *(HyperText Transfer Protocol)* is the foundational procotol for the WWW *(World Wide Web)*. Is ubiquitous in the latest 30 years and is expected it will to be continued be in use for many more.

It is frequently complemented with and additional layer called SSL/TLS *(Secure Socket Layer/Transport Layer Security)* to add security to the communication, then the combination is refered as HTTPS.

HTTP uses URIs *(Uniform Resource Identifiers)* to address the resources. HTTP verbs like GET, POST or PUT are used to describe the nature of the operation to perform on the resource. Request messages contains headers and an optional body. Headers are string based key-value pairs. HTTP defines many predefined HTTP Headers. At the same time, the user can extend the protocol with custom headers using the recommended preffix `x-` to avoid collisions with well-known headers.

The body can be encoded with any technique. To notify the server about the encoding been used there is a special HTTP header called `Content-Type` used for this porpuse.

The previus encodings described can be declared using MIME/Types, a standard way of declarying encoding types:

Encoding	MIME Type
XML	`application/xml`
JSON	`application/json`
BSON	`application/bson`
Protobuff	`application/x-protobuff`
Apache Avro	`application/avro`

HTTP is a unidirectional, synchronous message protocol implemented over TCP *(Transmission Control Protocol)* where a client initiates the communication with a message and a server responds another message. In its simplier form, the channel is clossed after the response is received (advanced uses cases like audio/video streaming, for example, is done maintaing the channel open and sending blocks of data asynchronously).

This lack of state in the procotol provides the foundation for scalability properties on HTTP.

3.2.2 WebSockets

WebSockets differs from HTTP in the way the communication channel is keep open after the initial handshake. Also, the server can take the lead and send message to the client without a prior client request.

Keeping open the channel introduces some problems for scalability on the server side to preserve same server and IP during the lifetime of the websocket. State must be preserved server-side.

3.2.3 Queues

Queues are the foundation for asynchronous communication. Messages are placed on a queue to be processed later on. The client does not have control or information about when and where. This provide a high level of decoupling between servers and clients. Popular queues products include RabbitMQ, ZeroMQ, Amazon SQS, Azure Queues, to cite a few. Most of them implements the stardard AMQP[8] *(Advanced Message Queuing Protocol)* to allow easy interoperability between different queue implementations.

3.2.4 MQTT

In the subdomain of IoT *(Internet of Things)*, MQTT *(*MQ Telemetry Transport) is a very popular queue protocol for small devices, low energy consumption gadgeds that needs to send telemetry regularly. The use this protocol optimized for its small packet size and simple processing (low energy requirements).

Sending MQTT traffic is simmilar to writting into a queue (a write and forget opperation, no need to await for a result). Subscribing to an MQTT queue is like opening a Websocket and subscribe to an specific topic: messages will arrive in realtime till the client closes the channel or send the unsubscription message.

3.2.5 SMTP

Much less popupar now, there were a time were command and messages were send over SMTP *Simple Mail Transport Protocol* using the email protocol. Any message can be encoded into an email and send to a remote mailbox. There a mail processor can extract the message and process it, maybe generating a response email if needed. The high quantity of *SPAM* in the mail makes this way of automation less frequent, but there is still uses cases for it like sending an email to an specific inbox to automatically subscribe, unsubscribe or to request a given resource.

Therefore, mail is not only for human-to-human communication. It can be used for human-to-machine scenarios for self-serving scenarios like mail lists.

3.2.6 Custom over TCP or UDP

For other usages, many custom protocols build on top of TPC and UDP to build real-time audio and video, RPC and more. Using the transport layer at raw reduces latencies that can be cricital for real time services. The cost is the extra complexity to implement and maintaing a bigger communication stack.

3.3 Standards

Many standars have emerged for APIs over the years, let's review the most relevant.

3.3.1 CORBA

CORBA *(Common Object Request Broker Architecture)* was an standard looking for a neutral RPC created by committe by the OMG *(Object Management Group)*. It was quite popular on the 80's and 90's in some enviroments. However, it was not trully achieve its goals due to proprietary protocol implementations, incompatible implementations and it is tricky to use with firewalls.

3.3.2 COM

Microsoft implemented its own RPC mecanimsh used in Windows for years known as COM *(Component Object Model)* and its sucessors COM+ and DCOM. It was hardly used or implemented outside of Windows OS, therefore is no a foundation for real interoperability among heterogeneous systems like CORBA.

3.3.3 SOAP

SOAP *(Simple Object Access Protocol)* was the first standard trully motivated to achieve real interoperability between any language (present or future). As fundation, SOAP uses XML as the encoding and can use many transports protocols like HTTP, queues or SMTP. The growing complexity of the stack (also designed by commitee) also impacted in its decadence after the 90's.

3.3.4 REST

REST *(REpresentational State Transfer)* was direct proposal from the PhD of Roy Fieldings[9]. Using resources and representation formats provides a standard widely used on the Internet promoting decoupling and long term evolution.

REST is typically transported over HTTP/S (although not limited to). It allows content negotiation. REST is extensible for encoding formats. Its most popular encodings used to be JSON or XML.

3.3.5 GraphQL

GraphQL is a direct RPC mechanism using a unique endpoint. GraphQL sacrifies the usage of URIs and resources and provides a fixed contract between server and consumers. It allows a quick discovery of operations for the client.

The price to pay is a high coupling between client and server, less interoperability and discoverability, and no support for versioning and HTTP caching mechanism.

3.3.6 OpenAPI

OpenAPI is an open standard to describe APIs over HTTP(S). Created by Tony Tam and named initially Swagger evolved to OpenAPI and nowadays is mantained by the OpenAPI Initiative[7] under the umbrella of the Linux Foundation[1].

The specification provides a formal way to describe the API, endpoints, parameters, types, error codes, etc. using a JSON or YAML document. This file can be mechanically processed by programs to document the API, create skeletons or stubs or SDKs using code generation techniques, or configure API middleware.

OpenAPI was a revolution on the API scene. Today it is the main standard to look at when building APIs. It provides a great degree of interoperability, extensibility, and discoverability.

3.3.7 gRPC

Google created gRPC *(Google Remote Procedure Call)* as an internal and propietary RPC mechanism inside Google. Later on, the company decided to publish it as open source. It allows to use both asynchronous and synchronous communications methods. It provides tooling for scalability and fault tolerence out of the box. Relay on Protocol Buffers for serialization.

3.3.8 AsyncAPI

Traditionally, OpenAPI deals with REST style APIs over HTTP where clients send a message and awaits for a response. This pattern is called synchronous communication in the way the client needs to await for the response.

AsyncAPIs[6] aware of other forms communications involving asynchronous patterns, build a similar specification to formalize and include such type of communications. Extending OpenAPI for gRPC, MQTT and more.

4 Conclusions

As discussed in the begining, different usage scenarios prioritize main requirements in different forms, leading to different solutions available for implementing APIs.

To sum up and syntetize our experience in a few lines:

- When *Interoperability* is key, REST using JSON or XML described with OpenAPI specification are well suited for the task. Industrial examples: standards for banking, open-data, and travel sectors.
- In cases where *Performance* is the leading requirement, the ones using binary serialization protocols like gRPC provides a good solution. Example: Kubernetes APIs are exposed mainly in gRPC format.
- IoT or low battery consumption scenarios can favour MQTT over competitor protocols.
- When *Time to market* is crucial and the client is in sync with the server (client is developer at the same time) GraphQL can be a good candidate for the task. The main price to pay is client-to-server hard coupling without versioning or cache mechanisms available.
- AsyncAPIs are a good fit where the mayority of the communications are asynchronous.

Of course, this was a rule of thumb approach from the sky, any specific scenario will need further analysis and assestment. As appointed by Fred Brooks in the 80's in his well-known paper, in software like many aspects in life, the is *No Silver-bullet*[4].

References

[1] The linux foundation. https://www.linuxfoundation.org. 11

[2] Avro Authors. Avro specification. https://avro.apache.org/docs/1.2.0/spec.html, 2012. 7

[3] Tim Berners-Lee. Hypertext transfer protocol – http/1.1. https://www.ietf.org/rfc/rfc2616.txt, 1990. 8

[4] Frederick P. Brooks. No silver bullet—essence and accident in software engineering. In *Proceedings of the IFIP Tenth World Computing Conference*, page 1069–1076, 1986. 12

[5] Google Corp. Protocol buffers. https://developers.google.com/protocol-buffers/docs/overview, 2008. 6

[6] Fran Mendez et al. Async api. https://www.asyncapi.com. 11

[7] OpenAPI Initiative. Openapi initiative. https://openapis.org. 11

[8] OASIS. Advanced message queueing protocol. https://www.oasis-open.org/news/pr/iso-and-iec-approve-oasis-amqp-advanced-message-queuing-protocol/, 2014. 9

[9] Fielding Roy Thomas. *Representational State Transfer (REST)*. PhD thesis, University of California, 2000. 10

If you liked this whitepaper:

Enroll into our newsletter for future whitepapers & techonology insight.

Feel free to **forward** this whitepaper to anyone interested in the topics covered.

Join the conversation & **write** us at info@metadev.pro.