[[https://www.javacodegeeks.com/2018/08/microservices-java-developers-microservices-communication.html]]

# Microservices for Java Developers: Microservices Communication

# 1. Introduction

Microservice architecture is essentially a journey into engineering of the distributed system. As more and more microservices are being developed and deployed, most likely than not they have to talk to each other somehow. And these means of the communication vary not only by transport and protocol, but also if they happen synchronously or asynchronously.

## Table Of Contents



In this section of the tutorial we are going to talk about most widely used styles of communication applied to microservice architecture. As we are going to see, each one has own pros and cons, and the right choice heavily depends on the application architecture, requirements and business constraints. Most importantly, you are not obligated to pick just one and stick to it. Embodying different communication patterns between different groups of microservices, depending on their role, specification and destiny, is absolutely possible. It worth reminding one of the core principles of the microservices we have talked about in the opening part of the tutorial: pick the right tool for the job.

# 2. Using HTTP

In present-day world, HTTP is very likely the most widely used communication protocol out there. It is one of the foundational pieces of the World Wide Web and despite being unchanged for quite a long time, it went through the major revamp recently to address the challenges of modern web applications.

The semantic of the HTTP protocol is genuinely simple but at the same time flexible and powerful enough. There are several major interaction paradigms (or styles) built on top of HTTP protocol (more precisely, HTTP/1.1) which are clearly in dominant position with respect to the microservice architecture implementations.

## 2.1 SOAP

SOAP (or **Simple Object Access Protocol**) is one of the first specifications for exchanging structured information in the implementation of the web services. It was designed way back in 1998 and is centered on XML messages transferred primarily over HTTP protocol.

One particularly innovative idea which came out of the evolution of SOAP protocol is Web Services Description Language(or just WSDL): an XML-based interface definition language that was used for describing the functionality offered by SOAPweb services. As we are going to see later on, the lessons learned from the WSDL taught us that, in some form or another, the notion of the explicit service contract (or schema, specification, description) is absolutely necessary to bridge providers and consumers together.

SOAP is over 20 years old, why even bother mentioning it? Surprisingly, there are quite a lot of systems which interface using SOAP web services and are still very heavily utilized.

## 2.2 REST

For many, the appearance of the REST architectural style signified the end of the SOAP era (which turned out not to be true strictly speaking).

***Representational State Transfer** (**REST**) is an architectural style that defines a set of constraints to be used for creating web services. Web services that conform to the REST architectural style, or RESTful web services, provide interoperability between computer systems on the Internet. REST-compliant web services allow the requesting systems to access and manipulate textual representations of web resources by using a uniform and predefined set of stateless operations. By using a stateless protocol and standard operations, REST systems aim for fast performance, reliability, and the ability to grow, by re-using components that can be managed and updated without affecting the system as a whole, even while it is running.*

*https://en.wikipedia.org/wiki/Representational_state_transfer*

The roots of term *representational state transfer* go back to 2000 when Roy Fielding introduced and defined it in his famous doctoral dissertation "Architectural Styles and the Design of Network-based Software Architectures".

Interestingly, REST architectural style is basically agnostic to the protocol being used but gained tremendous popularity and adoption because of HTTP. This is not a coincidence, since the web applications and APIs represent a significant chunk of the applications these days.

There are six constraints which the system or application should meet in order to qualify as RESTful. All of them actually play very well by the rules of the microservice architecture.

- **Uniform Interface:** it does not matter who is the client, the requests look the same.
- **Separation of the client and the server**: servers and clients act independently (separation of concerns).
- **Statelessness**: no client-specific context is being stored on the server between requests and each request from any client contains all the information necessary to be serviced.
- **Cacheable**: clients and intermediaries can cache responses, whereas responses implicitly or explicitly define themselves as cacheable or not to prevent clients from getting stale data.
- **Layered system**: a client cannot ordinarily tell whether it is connected directly to the end server or to an intermediary along the way.
- **Code on demand (optional):** servers can temporarily extend or customize the functionality of a client by transferring executable code (usually some kind of scripts)

REST, when used in the context of HTTP protocol, relies on resources, uniform resource locators (URLs), standard HTTP methods, headers and status codes to design the interactions between servers and clients. The table below outlines the typical mapping of the HTTP protocol semantics to the imaginable library management web APIs designed after REST architectural style.

| **URL: https://api.library.com/books/** | | | | | | |
|---|---|---|---|---|---|---|
| **GET** | **PUT** | **PATCH** | **POST** | **DELETE** | **OPTIONS** | **HEAD** |
| **Retrieve** all resources in a collection. | **Replace** the entire collection with another collection. | Not generally used. | **Create** a new entry in the collection. The new entry's URI is usually returned by the operation. | **Delete** the entire collection. | **List** available HTTP methods (and may be other options). | **Retrieve** all resources in a collection (should return headers only). |
| **URL: https://api.library.com/books/17** | | | | | | |
| **GET** | **PUT** | **PATCH** | **POST** | **DELETE** | **OPTIONS** | **HEAD** |
| **Retrieve** a representation of the single resource. | **Replace** the resource entirely (or **create** it if it does not exist yet). | **Update** the resource (usually, partially. | Not generally used. | **Delete** the resource. | **List** available HTTP methods (and may be other options). | **Retrieve** a single resource (should return headers only). |
| Idempotent: **yes** | Idempotent: **yes** | Idempotent: **no** | Idempotent: **no** | Idempotent: **yes** | Idempotent: **yes** | Idempotent: **yes** |
| Safe: **yes** | Safe: **no** | Safe: **no** | Safe: **no** | Safe: **no** | Safe: **yes** | Safe: **yes** |

There are other subtle but exceptionally important expectations (or implicit premises if you will) associated with each HTTP method in context of REST (and microservices in particular): idempotency and safety. An operation is considered idempotent when even if the same input is sent to it multiple times, the effect will be the same (as sending this input only once). Consequently, the operation is safe if does not modify the resource (or resources). The assumptions regarding idempotency and safety are critical to handling failures and making the decisions about mitigating them.

To sum up, it is very easy to get started building RESTful web APIs since mostly every programming language has HTTPserver and client baked into its base library. Consuming them is no brainer as well: either from command line (curl, httpie), using specialized desktop clients (Postman, Insomnia), or even from web browser (though not much you could do without installing the additional plugins).

This simplicity and flexibility of REST comes at a price: the lack of first-class support of discoverability and introspection. The agreement between server and client on the resources and the content of the input and output is out of band knowledge.

The API Stylebook with its Design Guidelines and Design Topics is a terrific resource to learn about the best practices and patterns for building magnificent RESTful web APIs. By the way, if you get an impression that REST architectural stylerestricts your APIs to follow the CRUD (Create/Read/Update/Delete) semantic, this is certainly a myth.

## 2.3 REST: Contracts on the Rescue

The lack of explicit, shareable, descriptive contract (besides the static documentation) for RESTful web APIs was always an area of active research and development in the community. Luckily, the efforts have been culminated recently into establishing the OpenAPI Initiative and releasing OpenAPI 3.0 specification (previously known as Swagger).

*The OpenAPI Specification (OAS) defines a standard, programming language-agnostic interface description for REST APIs, which allows both humans and computers to discover and understand the capabilities of a service without requiring access to source code, additional documentation, or inspection of network traffic. When properly defined via OpenAPI, a consumer can understand and interact with the remote service with a minimal amount of implementation logic. Similar to what interface descriptions have done for lower-level programming, the OpenAPISpecification removes guesswork in calling a service. – https://github.com/OAI/OpenAPI-Specification*

OpenAPI is not the de-facto standard everyone is obligated to use but a well-thought, comprehensive mean to manage the contracts of your RESTful web APIs. Yet another benefit it comes with, as we are going to see later on in the tutorial, is that the tooling around OpenAPI is just amazing.

Among alternative options it is worth to mention API Blueprint, RAML, Apiary and Apigee. Honestly, it does not really matter what you are going to use, the shift towards contract-driven development and collaboration does.

## 2.4 GraphQL

Everything is moving forward and the dominant positions of REST were being shaken by the new kid on the block, namely GraphQL.

*GraphQL is a query language for APIs and a runtime for fulfilling those queries with your existing data. GraphQLprovides a complete and understandable description of the data in your API, gives clients the power to ask for exactly what they need and nothing more, makes it easier to evolve APIs over time, and enables powerful developer tools. – https://graphql.org/*

GraphQL has an interesting story. It was originally created at Facebook in 2012 to address the challenges of handling their data models for client / server applications. The development of the GraphQL specification in the open started only in 2015 and since then this pretty much new technology is steadily gaining the popularity and widespread adoption.

*GraphQL is not a programming language capable of arbitrary computation, but is instead a language used to query application servers that have capabilities defined in this specification. GraphQL does not mandate a particular programming language or storage system for application servers that implement it. Instead, application servers take their capabilities and map them to a uniform language, type system, and philosophy that GraphQL encodes. This provides a unified interface friendly to product development and a powerful platform for tool-building.*
*– http://facebook.github.io/graphql/June2018/*

What makes GraphQL particularly appealing for microservices is a set of its core design principles:

- **It is hierarchical**: Most of the data these days is organized into hierarchical structures. To achieve congruence with such reality, a GraphQL query itself is structured hierarchically.
- **Strong-typing**: Every application declares own type system (also known as schema). Each GraphQL query is executed within the context of that type system whereas GraphQL server enforces the validity and correctness of such query before executing it.
- **Client-specified queries**: A GraphQL server publishes the capabilities that are available for its clients. It becomes the responsibility of the client to specifying exactly how it is going to consume those published capabilities so the given GraphQL query returns exactly what a client asks for.
- **Introspective**: The specific type system which is managed by a particular GraphQL server must be queryable by the GraphQL language itself.

GraphQL puts clients in control of what data they need. Although it has some drawbacks, the compelling benefits of strong typing and introspection often make GraphQL a favorable option.

Unsurprisingly, most of the GraphQL implementations are also HTTP-based and for good reasons: to serve as a foundation for building web APIs. In the nutshell, the GraphQL server should handle only HTTP `GET` and `POST` methods. Since the conceptual model in GraphQL is an entity graph, such entities are not identified by URLs. Instead, a GraphQL server operates on a single endpoint (usually `/graphql`) which handles all requests for a given service.

Surprisingly (or not?), many people treat GraphQL and REST as direct competitors: you have to pick one or another. But the truth is that both are excellent choices and can happily

coexist to solve the business problems in a most efficient ways. This is what microservices are all about, right?
The implementations of GraphQL exist in many programming languages (for example, graphql-java for Java, Sangria for Scala, just to name a few) but the JavaScript one is outstanding and set the pace for entire ecosystem.
Let us take a look on a how the RESTful web APIs from the previous section could be described in the terms of GraphQLschema and types.

```
schema {
  query: Query
  mutation: Mutation
}

type Book {
  isbn: ID!
  title: String!
  year: Int
}

type Query {
  books: [Book]
  book(isbn: ID!): Book
}

# this schema allows the following mutation:
type Mutation {
  addBook(isbn: ID!, title: String!, year: Int): Book
  updateBook(isbn: ID!, title: String, year: Int): Book
  removeBook(isbn: ID!): Boolean
}
```

The separation between mutations and queries provides natural explicit guarantees about the safety of the particular operation.
It is fair to say that GraphQL slowly but steadily is changing the web APIs landscape as more and more companies are adapting it or have adapted already. You may not expect it but RESTful and GraphQL are often deployed side by side. One of the new patterns emerged of such co-existence is backends for frontends (**BFF**) where the GraphQL web APIs are fronting the RESTful web services.

# 3. Not only HTTP

Although HTTP is the king, there are a couple of communication frameworks and libraries which go beyond that. Like, for example RPC-style conversations, the oldest form of inter-process communication.
Essentially, RPC is a request–response protocol where the client sends a request to a remote server to execute a specified procedure with supplied parameters. Unless the communication between client and server is asynchronous, the client usually blocks till the remote server sends a response back. Although quite efficient (most of the time the exchange format is a binary one), RPC used to have a huge issues with interoperability and portability across different languages and platforms. So why to rake over old ashes?

# 3.1 gRPC

The HTTP/2, a major revision of the HTTP protocol, unblocked the new ways to drive the communications on the web. gRPC, a popular, high performance, open-source universal RPC framework from Google, is the one who bridges the RPCsemantics with HTTP/2 protocol.

To add a note here, although gRPC is more or less agnostic to the underlying transport, there is no other transport supported besides HTTP/2 (and there are no plans to change that in the immediate future). Under the hood, gRPC is built on top of another widely adopted and matured piece of the technology from Google, called protocol buffers.

*Protocol buffers are a flexible, efficient, automated mechanism for serializing structured data – think XML, but smaller, faster, and simpler. You define how you want your data to be structured once, then you can use special generated source code to easily write and read your structured data to and from a variety of data streams and using a variety of languages. You can even update your data structure without breaking deployed programs that are compiled against the "old" format.*

*– https://developers.google.com/protocol-buffers/docs/overview*

By default, gRPC uses protocol buffers as both its Interface Definition Language (**IDL**) and as its underlying message interchange format. The IDL contains the definitions of all data structures and services and carry on the contract between gRPC server and its clients.

For example, here is very simplified attempt to redefine the web APIs from the previous sections using protocol buffersspecification.

```
syntax = "proto3";

import "google/protobuf/empty.proto";

option java_multiple_files = true;
option java_package = "com.javacodegeeks.library";

package library;

service Library {
  rpc addBook(AddBookRequest) returns (Book);
  rpc getBooks(Filter) returns (BookList);
  rpc removeBook(RemoveBookRequest) returns (google.protobuf.Empty);
  rpc updateBook(UpdateBookRequest) returns (Book);
}

message Book {
  string title = 1;
  string isbn = 2;
  int32 year = 3;
}

message RemoveBookRequest {
  string isbn = 1;
}

message AddBookRequest {
  string title = 1;
```

```
  string isbn = 2;
  int32 year = 3;
}

message UpdateBookRequest {
  string isbn = 1;
  Book book = 2;
}

message Filter {
  int32 year = 1;
  string title = 2;
  string isbn = 3;
}

message BookList {
  repeated Book books = 1;
}
```

gRPC provides bindings for many mainstream programming languages and relies on the protocol buffers tools and plugins for code generation (but if you are programming in Go, you are in a luck since the Go language ecosystem is the state of the art there). gRPC is an excellent way to establish efficient channels for internal service-to-service or service-to-consumer communication.
A lot of exciting developments are happing around gRPC these days. The most promising one is gRPC for Web Clients(currently in beta) which is going to provide a JavaScript client library that lets browser clients to access gRPC servers directly.

## 3.2 Apache Thrift

To be fair, gRPC is not the only RPC-style framework available. The Apache Thrift is another one dedicated to scalable cross-language services development. It combines a software stack with a code generation engine to build services that work efficiently and seamlessly between many languages.
Apache Thrift is specifically designed to support non-atomic version changes across client and server code. It is very similar to gRPC and protocol buffers and shares the same niche. While it is not as popular as gRPC, it supports bindings for 25 programming languages and relies on modular transport mechanism (HTTP included).
Apache Thrift has own dialect of the Interface Definition Language which resembles protocol buffers quite a lot. To compare with, here is another version of our web APIs definition, rewritten using Apache Thrift.

```
namespace java com.javacodegeeks.library

service Library {
  void addBook(1: Book book),
  list getBooks(1: Filter filter),
  bool removeBook(1: string isbn),
  Book updateBook(1: string isbn, 2: Book book)
}
```

```
struct Book {
  1: string title,
  2: string isbn,
  3: optional i32 year
}

struct Filter {
  1: optional i32 year;
  2: optional string title;
  3: optional string isbn;
}
```

## 3.3 Apache Avro

Last but not least, Apache Avro, a data serialization system, is often used for RPC-style communication and message exchanges. What distinguishes Apache Avro from others is the fact that the schema is represented in JSON format, for example, here is our web APIs translated to Apache Avro.

```
{
  "namespace": "com.javacodegeeks.avro",
  "protocol": "Library",

  "types": [
    {
      "name": "Book",
      "type": "record",
      "fields": [
          {"name": "title", "type": "string"},
          {"name": "isbn", "type": "string"},
          {"name": "year", "type": "int"}
      ]
    }
  ],

  "messages": {
     "addBook": {
       "request": [{"name": "book", "type": "Book"}],
       "response": "null"
     },
     "removeBook": {
       "request": [{"name": "isbn", "type": "string"}],
       "response": "boolean"
     },
     "updateBook": {
       "request": [
           {"name": "isbn", "type": "string"},
           {"name": "book", "type": "Book"}
       ],
       "response": "Book"
     }
```

```
    }
}
```

Another unique feature of Apache Avro is to make out different kind of specifications, based on the file name extensions, for example:

- ***.avpr**: defines a Avro Protocol specification
- ***.avsc**: defines an Avro Schema specification
- ***.avdl:** defines an Avro IDL

Similarly to Apache Thrift, Apache Avro supports different transports (which additionally could be either *stateless* or *stateful*), including HTTP.

# 4. REST, GraphQL, gRPC, Thrift … how to choose?

To understand where each of these communication styles fit the best, the Understanding RPC, REST and GraphQL article is a great starting point.

# 5. Message passing

The request–response is not the only method to structure the communication in distributed systems and microservices in particular. Message passing is another communication style, asynchronous by nature, which revolves around exchanging messages between all the participants.

Messaging is heart and soul of the even-driven applications and microservices. In practice, they are implemented primarily on the principles of Event Sourcing or Command Query Responsibility Segregation (**CQRS**) architectures however the definition of what it means to be event-driven goes broader than that.

To be fair, there is tremendous amount of different options to talk about and pick from. So to keep it sane, we are going to focus more on a core concept rather than concrete solutions.

## 5.1 WebSockets and Server-Sent Events

If your microservice architecture constitutes of RESTful web services, picking a native HTTP messaging solution is a logical way to go.

The WebSocket protocol enables bidirectional (full-duplex) communication channels between a client and a server over a single connection. Interestingly, the WebSocket is an independent TCP-based protocol but at the same time *"… it is designed to work over HTTP ports 80 and 443 as well as to support HTTP proxies and intermediaries …" (https://tools.ietf.org/html/rfc6455)*.

For non-bidirectional communication, server-sent events (or in short, **SSE**) is a great, simple way to enable servers to push the data to the clients over HTTP (or using dedicated server-push protocols).

With the raising popularity of HTTP/2, the role of WebSocket and server-sent events is slowly diminishing since most of their features are already backed into the protocol itself.

## 5.2 Message Queues and Brokers

Messaging is exceptionally interesting and crowded space in software development. Java Message Service (JMS), Advanced Message Queuing Protocol (AMQP), Simple (or

Streaming) Text Orientated Messaging Protocol (STOMP), Apache Kafka, NATS, NSQ, ZeroMQ, not to mention Redis Pub/Sub, upcoming Redis Streams and tons of cloud solutions. What to say, even PostgreSQL includes one!
Depending on your application needs, it is very likely you could find more than one message broker to choose from. However, there is an interesting challenge which you may need to solve:

- efficiently publish the message schemas (to share what is packed into message)
- evolve the message schemas over time (ideally, without breaking things)

Surprisingly, our old friends protocol buffers, Apache Thrift and Apache Avro could be an excellent fit for these purposes. For example, Apache Kafka is often used with Schema Registry to store a versioned history of all message schemas. The registry is built on top of Apache Avro.
Other interesting libraries we have not talked about (since they are purely oriented on message formats, not services or protocols) are FlatBuffers, Cap'n Proto and MessagePack.

## 5.3 Actor Model

The actor model, originated in 1973, introduces the concept of actors as the universal primitives of concurrent computation which communicate with each other by sending messages asynchronously. Any actor, in the response to a message it receives, can do concurrently one of the following things:

- send a finite number of messages to other actors
- instantiate a finite number of new actors
- change the designated behavior to process the next message it receives

The consequences of using message passing are that actors do not share any state with each other. They may modify their own private state, but can only affect each other through messages.
You may have heard about Erlang, a programming language to build massively scalable soft real-time systems with requirements on high availability. It is one of the best examples of successful actor model implementation.
On JVM, the unquestionable leader is Akka: a toolkit for building highly concurrent, distributed, and resilient message-driven applications for Java and Scala. It started as the actor model implementation but over the years has grown into full-fledged Swiss knife for distributed system developers.
Frankly speaking, the ideas and principles behind the actor model make it a serious candidate for implementing microservices.

## 5.4 Aeron

For a highly efficient and latency-critical communications the frameworks we have discussed so far may not be a best choice. You can certainly fallback to custom-made TCP/UDP transport but there is a good set of options out there.
Aeron is an efficient reliable UDP unicast, UDP multicast, and IPC message transport. It supports Java out of the box with performance being the key focus. Aeron is designed to be the highest throughput with the lowest and most predictable latency possible of any messaging system. Aeron integrates with Simple Binary Encoding (SBE) for the best possible performance in message encoding and decoding.

## 5.5 RSocket

RSocket is a binary protocol for use on byte stream transports such as TCP, WebSockets, and Aeron. It supports multiple symmetric interaction models via asynchronous message passing using just a single connection:

- request/response (stream of 1)
- request/stream (finite stream of many)
- fire-and-forget (no response)
- channel (bi-directional streams)

Among other things, it supports session resumption which allows to resume long-lived streams across different transport connections. This is particularly useful when network connections drop, switch, and reconnect frequently.

# 6. Cloud native

Cloud computing is certainly the place where the most of the applications are being deployed nowadays. The heated fights for the market share replenish the continuous streams of innovations. One of those is serverless computing where the cloud provider takes care of server management and capacity planning decisions dynamically. The presence of the term serverless is a bit confusing since the servers are still required, but the deployment and execution models change.

The exciting part is that serverless code can be used along with the application deployed as more traditional microservices. Even more, the whole application designed after microservice architecture could be built on top of purely serverless components, dramatically decreasing the operational burden.

In case the serverless computing sounds new to you, the good introduction to such architecture is given in this post on Martin Fowler's blog.

## 6.1 Function as a service

One of the best examples of serverless computing in action is function as a service (**FaaS**). As you may guess, the unit of deployment in such a model is a function (ideally, in any language, but Java, JavaScript and Go are most likely the ones you could realistically use right now). The functions are expected to start within a few milliseconds in order to handle the individual requests or to react on the incoming messages. When not used, the functions are not consuming any resources, incurring no charges at all.

Each cloud provider offers own flavor of function as a service platform but it is worth mentioning Apache OpenWhisk, OpenFaaS and riff projects, a couple of open-source well-established function as a service implementations.

## 6.2 Knative

This is literally a newborn member of the serverless movement, public announced by Google just a few weeks ago.

*Knative components extends Kubernetes to provide a set of middleware components that are essential to build modern, source-centric, and container-based applications that can run anywhere: on premises, in the cloud, or even in a third-party data center. … Knative components offer developers Kubernetes-native APIs for deploying serverless-style functions, applications, and containers to an auto-scaling runtime. – https://github.com/knative/docs*

Knative is in very early stages of development but the potential impact of it on the serverless computing could be revolutionary.

# 7. Conclusions

Over the course of this section we have talked about many different styles to structure the communication between microservices (and their clients) in the applications which follow microservice architecture. We have understood the criticality and importance of the schema or/and contract as the essential mean of establishing healthy collaboration between service providers and consumers (think teams within organization). Last but not least, the combination of multiple communication styles is certainly possible and makes sense, however such decisions should be driven by real needs rather than hype (sadly, it happens too often in the industry).

# 8. What's next

In the next section of the tutorial we are going to evaluate the Java landscape and most widely used frameworks for building production-grade microservices on JVM.
The complete set of specification files is available for download.