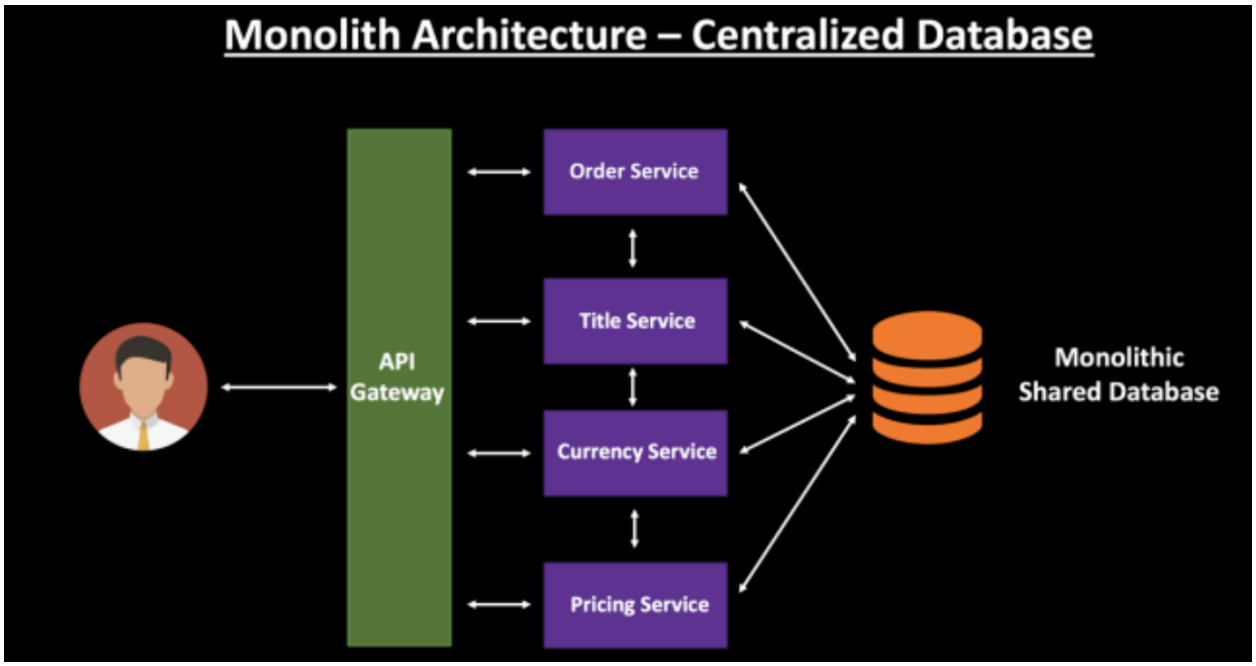
Microservice architecture **2**
Problems With Monolithic Database Design 2
Domain driven design 3
Event-Driven Architecture 3
Decomposing the Database 4
Web service API **7**
REST 7
Http Methods Reference 9
Idempotency 9
API versioning 10
SOAP 12
JAX-WS 12
Protobuf 14
Protobuf vs JSON 17
gRPC 19
gRPC vs REST 21

Microservice architecture

Problems With Monolithic Database Design

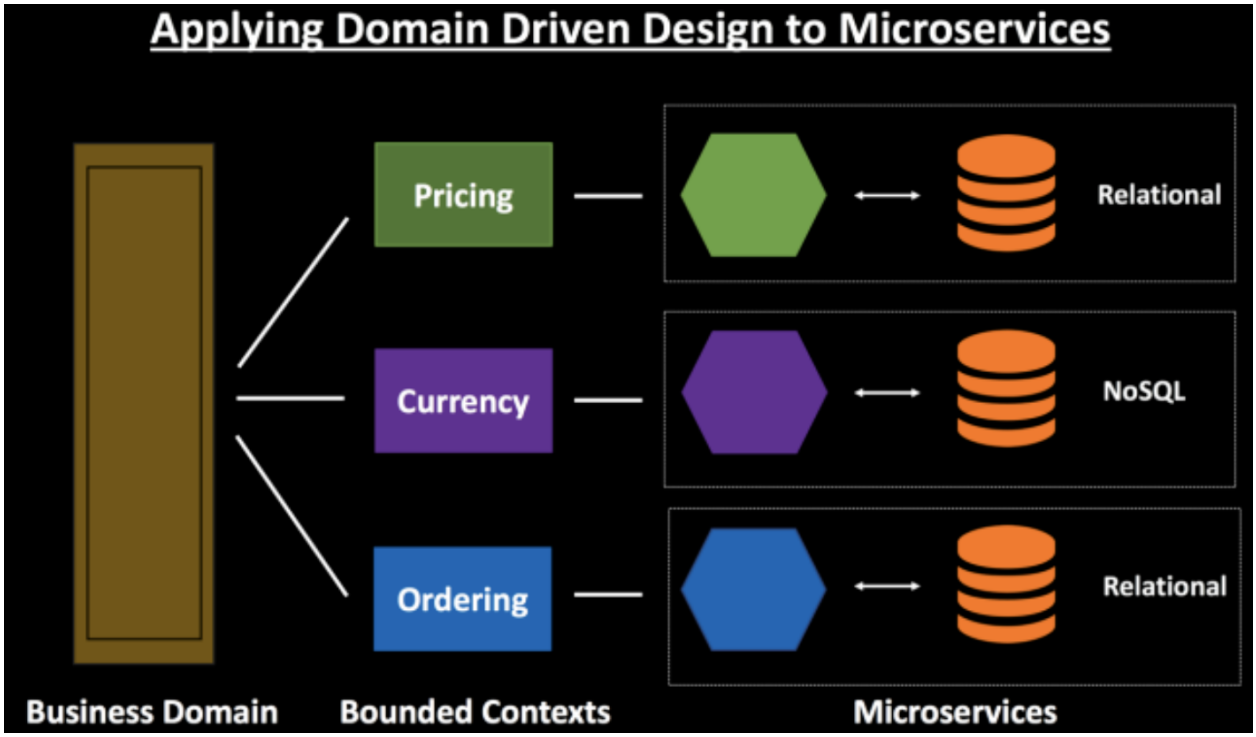
- The traditional design of having a Monolithic Database for multiple services creates a **tight coupling** and inability to deploy your service changes independently. If there are multiple services accessing the same database, any schema changes would need to be coordinated amongst all the services, which, in the real world, can cause additional work and delay in deploying changes.
- It is **difficult to scale** individual services with this design since you only have the option to scale out the entire monolithic database.
- Improving application performance becomes a challenge. With a single shared database, over a period of time, you end up having huge tables. This makes data retrieval difficult since you have to **join multiple big sized tables** to fetch the required data.
- Most of the times you have a relational store as your monolith database. **This constraints all your services to use a relational database**. However, there will be scenarios where a No-SQL datastore might be a better fit for your services and hence you don't want to be tightly coupled to a centralized datastore.



Domain driven design

Microservices should follow **Domain Driven Design** and have bounded contexts. You need to design your application **based on domains**, which aligns with the functionality of your application. It's like following the Code First approach over Data First approach - hence you **design your models first**. This is a fundamentally different approach than the traditional mentality of first designing your database tables when starting to work on a new requirement or greenfield project. You should always try to maintain the integrity of your Business model.

You should design your microservices architecture in such a way that each **individual microservice has its own separate database with its own domain data**. This will allow you to independently deploy and scale your microservices.



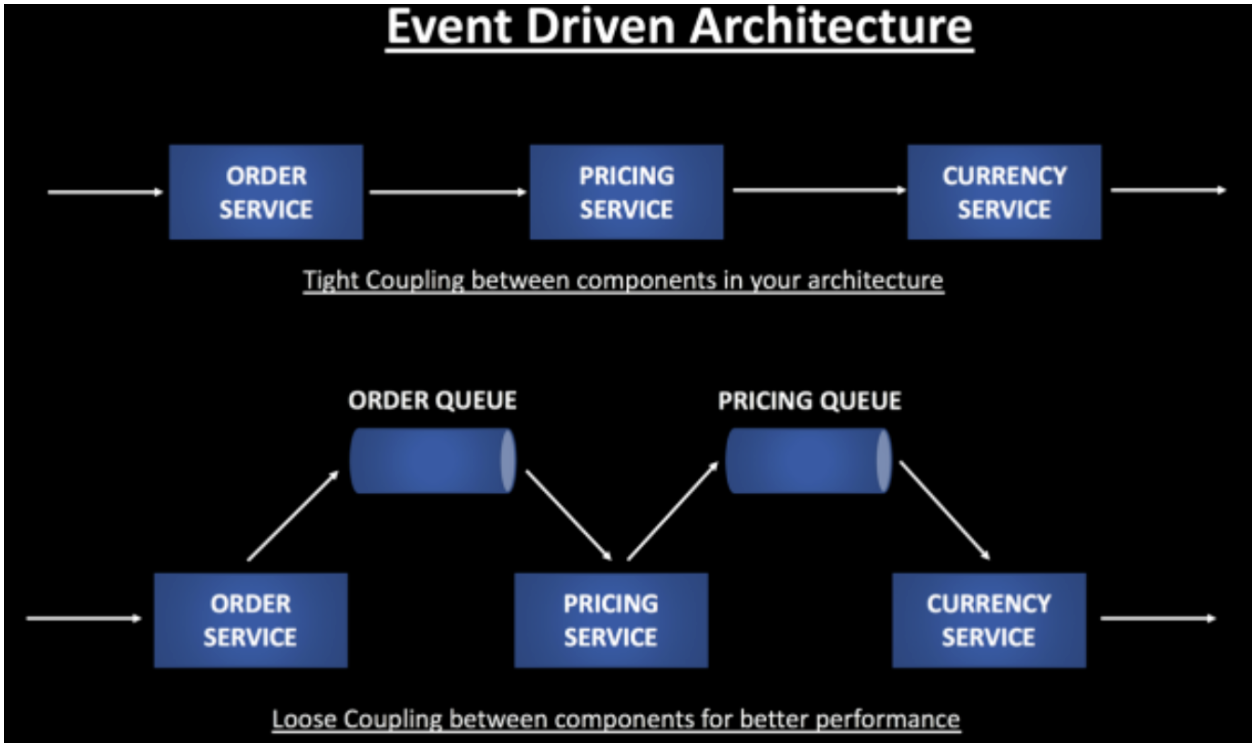
While designing your database, look at the application functionality and determine if it needs a relational schema or not. Keep your mind open towards a NoSQL DB as well if it fits your criteria.

Databases should be treated as private to each microservice. No other microservice can directly modify data stored inside the database in another microservice.

Event-Driven Architecture

Event-Driven Architecture is a common pattern to maintain data consistency across different services. Instead of waiting for an ACID transaction to complete processing and taking up system resources, you can make your application more available and performant by offloading the message to a queue. This provides loose coupling between services.

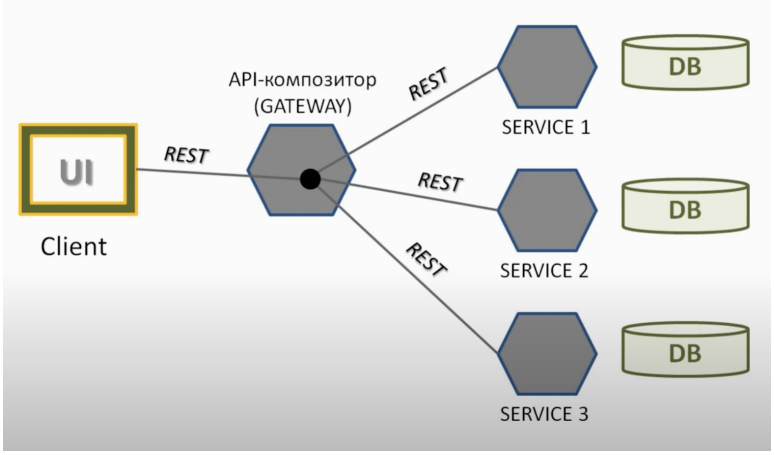
Messages to the queues can be treated as Events and can follow the Pub-Sub model. Publishers publishes a message and is not aware of the Consumer, who has subscribed to the event stream. **Loose coupling between components in your architecture enables to build highly scalable distributed systems**.



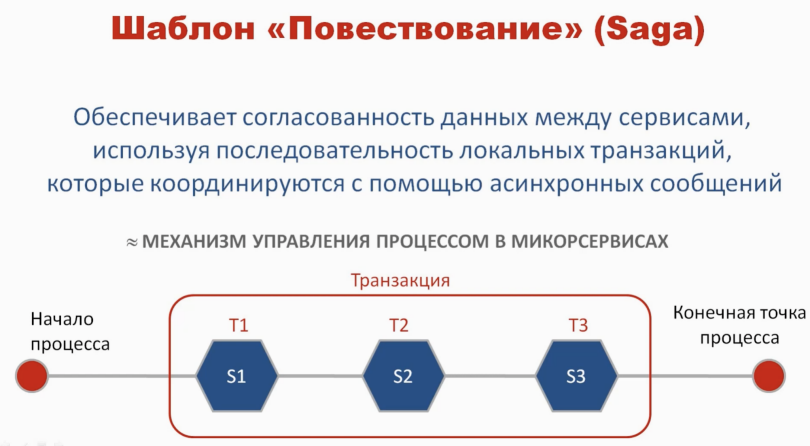
Decomposing the Database

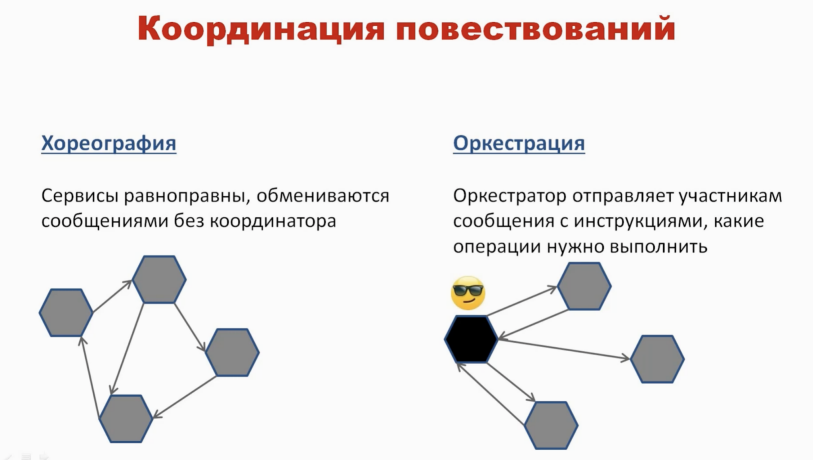
https://www.oreilly.com/library/view/monolith-to-microservices/9781492047834/ch04.html

Gateway pattern



Saga pattern





Хореография. Достоинства и недостатки

+ **Простота**
+ **Слабая связанность**
(Участники подписываются на события, не владея непосредственной информацией друг о друге)

- **Они сложнее для понимания**
- **Возникают циклические зависимости между сервисами**

Хореография может хорошо работать с простыми повествованиями, но в более сложных случаях лучше использовать оркестрацию

Оркестрация. Достоинства и недостатки

+ **Упрощенные зависимости**
+ **Улучшенное разделение ответственности и упрощенная бизнес-логика**

- **Риск избыточной централизации бизнес-логики**

Web service API

REST

Representational state transfer (REST) is a software architectural style which uses a subset of HTTP. It is commonly used to create interactive applications that use Web services. A Web service that follows these guidelines is called RESTful. REST is an alternative to, for example, SOAP as a way to access a Web service.

When a client request is made via a RESTful API, it transfers a representation of the state of the resource to the requester or endpoint. This information, or representation, is delivered in **one of several formats** via HTTP: JSON (Javascript Object Notation), HTML, XLT, Python, PHP, or plain text. **JSON** is the most generally popular programming language to use because, despite its name, it's **language-agnostic**, as well as **readable** by both humans and machines.

- **Client–server architecture**: The principle behind the client–server constraints is the separation of concerns. **Separating the user interface concerns from the data storage concerns** improves the portability of the user interfaces across multiple platforms. It also improves scalability by simplifying the server components. Perhaps most significant to the Web is that the separation allows the components to evolve independently, thus supporting the Internet-scale requirement of multiple organizational domains.

- **Statelessness**: In computing, a stateless protocol is a communications protocol in which no session information is retained by the receiver, usually a server. Relevant session data is sent to the receiver by the client in such a way that every packet of information transferred can be understood in isolation, without context information from previous packets in the session. This property of stateless protocols makes them ideal in high volume applications, **increasing performance by removing server load caused by retention of session information**.

- **Cacheability**: As on the World Wide Web, clients and intermediaries can cache responses. Responses must, implicitly or explicitly, define themselves as either cacheable or non-cacheable to prevent clients from providing stale or inappropriate data in response to further requests. Well-managed caching partially or completely eliminates some client–server interactions, further improving scalability and performance

- **Layered system**: A client cannot ordinarily tell whether it is connected directly to the end server or to an intermediary along the way. If a proxy or load balancer is placed between the client and server, it won't affect their communications, and there won't be a need to update the client or server code. Intermediary servers can improve system scalability by enabling **load balancing** and by providing **shared caches**. Also, security can be added as a layer on top of the web services, separating business logic from **security logic**. Adding security as a separate layer enforces security policies. Finally, intermediary servers can call multiple other servers to generate a response to the client (Facade pattern).

- **Uniform interface**: The uniform interface constraint is fundamental to the design of any RESTful system. It simplifies and decouples the architecture, which enables each part to evolve independently. The four constraints for this uniform interface are:
    - Resource identification in requests: Individual resources are identified in requests, for example using URIs in RESTful Web services. The resources themselves are conceptually separate from the representations that are returned to the client. For example, the server could send data from its database as HTML, XML or as JSON—none of which are the server's internal representation.
    - Resource manipulation through representations: When a client holds a representation of a resource, including any metadata attached, it has enough information to modify or delete the resource's state.
    - Self-descriptive messages: Each message includes enough information to describe how to process the message. For example, which parser to invoke can be specified by a media type.
    - Hypermedia as the engine of application state (HATEOAS): Having accessed an initial URI for the REST application—analogous to a human Web user accessing the home page of a website—a REST client should then be able to use server-provided links dynamically to discover all the available resources it needs. As access proceeds, the server responds with text that includes hyperlinks to other resources that are currently available. There is no need for the client to be hard-coded with information regarding the structure or dynamics of the application.

REST API examples:

curl -X POST localhost:8080/employees -H 'Content-type:application/json' -d '{"name": "Samwise Gamgee", "role": "gardener"}' {"id":3,"name":"Samwise Gamgee","role":"gardener"}
curl -X PUT localhost:8080/employees/3 -H 'Content-type:application/json' -d '{"name": "Samwise Gamgee", "role": "ring bearer"}' {"id":3,"name":"Samwise Gamgee","role":"ring bearer"}
curl -v localhost:8080/employees [{"id":1,"name":"Bilbo Baggins","role":"burglar"},{"id":2,"name":"Frodo Baggins","role":"thief"}]
curl -v localhost:8080/employees/99 {"id":99,"name":"Bilbo Baggins","role":"burglar"}
curl -X DELETE localhost:8080/employees/3

Http Methods Reference

GET	Retrieves information from the given server using a given URI. GET request can retrieve

	the data. It cannot apply other effects on the data
HEAD	The same as the GET method. It is used to transfer the status line and header section only
POST	sends the data to the server. For example, file upload, customer information, etc. using the HTML forms
PUT	Replace all the current representations of the target resource with the uploaded content
DELETE	Remove all the current representations of the target resource, which is given by URI
PATCH	Applies partial modifications to a resource

Idempotency

From a RESTful service standpoint, for an operation (or service call) to be **idempotent**, clients can make that same call repeatedly while producing the same result. In other words, making multiple identical requests has the same effect as making a single request. Note that while idempotent operations produce the same result on the server (no side effects), the response itself may not be the same (e.g. a resource's state may change between requests).

The PUT and DELETE methods are defined to be idempotent. However, there is a caveat on DELETE. The problem with DELETE, which if successful would normally return a 200 (OK) or 204 (No Content), will often return a 404 (Not Found) on subsequent calls, unless the service is configured to "mark" resources for deletion without actually deleting them. However, when the service actually deletes the resource, the next call will not find the resource to delete it and return a 404. However, the state on the server is the same after each DELETE call, but the response is different.

GET, HEAD, OPTIONS and TRACE methods are defined as safe, meaning they are only intended for retrieving data. This makes them idempotent as well since multiple, identical requests will behave the same.

API versioning

- **URI Versioning** - basic approach to versioning is to create a completely different URI for the new service:
  http://localhost:8080/v1/person
  http://localhost:8080/v2/person

```
@GetMapping("v1/student")
  public StudentV1 studentV1() {
    return new StudentV1("Bob Charlie");
 }

@GetMapping("v2/student")
  public StudentV2 studentV2() {
    return new StudentV2(new Name("Bob", "Charlie"));
 }
```

- **Request Parameter versioning** - use the request parameter to differentiate versions:
  http://localhost:8080/person/param?version=1
  http://localhost:8080/person/param?version=2

```
@GetMapping(value = "/student/param", params = "version=1")
  public StudentV1 paramV1() {
    return new StudentV1("Bob Charlie");
  }

  @GetMapping(value = "/student/param", params = "version=2")
  public StudentV2 paramV2() {
    return new StudentV2(new Name("Bob", "Charlie"));
  }
```

- **Headers versioning (custom)** - use a Request Header to differentiate the versions:
  http://localhost:8080/person/header
  +headers=[X-API-VERSION=1]

  http://localhost:8080/person/header
  +headers=[X-API-VERSION=2]

```
 @GetMapping(value = "/student/header", headers = "X-API-VERSION=1")
  public StudentV1 headerV1() {
    return new StudentV1("Bob Charlie");
  }

  @GetMapping(value = "/student/header", headers = "X-API-VERSION=2")
  public StudentV2 headerV2() {
    return new StudentV2(new Name("Bob", "Charlie"));
  }
```

- **Media type versioning (accept header)** - use the Accept Header in the request:
  http://localhost:8080/person/produces
  +headers[Accept=application/vnd.company.app-v1+json]

  http://localhost:8080/person/produces
  +headers[Accept=application/vnd.company.app-v2+json]

```
@GetMapping(value = "/student/produces", produces = "application/vnd.company.app-v1+json")
 public StudentV1 producesV1() {
   return new StudentV1("Bob Charlie");
 }

 @GetMapping(value = "/student/produces", produces = "application/vnd.company.app-v2+json")
 public StudentV2 producesV2() {
   return new StudentV2(new Name("Bob", "Charlie"));
 }
```

Pros and Cons:

- **URI Pollution** - URL versions and Request Param versioning pollute the URI space.
- Misuse of HTTP Headers - **Accept Header is not designed to be used for versioning**.
- Caching - If you use Header based versioning, **we cannot cache just based on the URL**. You would need take the specific header into consideration.
- Can we execute the request on the browser? - If you have non technical consumers, then the URL based version would be easier to use as they can be executed directly on the browser.
- API Documentation - How do you get your documentation generation to understand that two different urls are versions of the same service?

Some Major API providers that use different versioning approaches:

GitHub	Media type versioning (a.k.a “content negotiation” or “accept header”)
Microsoft	Headers versioning
Twitter	URI Versioning
Amazon	Request Parameter versioning

SOAP

Simple Object Access Protocol. It is an XML-based messaging protocol for exchanging information among computers. **SOAP is an application of the XML specification**. SOAP is designed to break traditional monolithic applications down into a multi-component, distributed form without losing security and control.

Although SOAP can be used in a variety of messaging systems and can be delivered via a variety of transport protocols, the **initial focus of SOAP is remote procedure calls transported via HTTP**.

Other frameworks including CORBA, DCOM, and Java RMI provide similar functionality to SOAP, but SOAP messages are written entirely in XML and are therefore uniquely **platform- and language-independent**.

SOAP is an integral part of the service-oriented architecture (SOA) and the Web services specifications associated with SOA.

SOAP's messages are defined at a high level in XML, but most SOAP applications use Web Services Definition Language (**WSDL**).

WSDL describes your webservice and its operations - what is the service called, which methods does it offer, what kind of parameters and return values do these methods have. It's a description of the behavior of the service - it's functionality.

XSD (Xml Schema Definition) describes the static structure of the complex data types being exchanged by those service methods. It describes the types, their fields, any restriction on those fields (like max length or a regex pattern) and so forth. It's a description of datatypes and thus static properties of the service - it's about data.

JAX-WS

Java API for XML Web Services (**JAX-WS**) is a standardized API for creating and consuming SOAP (Simple Object Access Protocol) web services.

There are two ways of building SOAP web services:

- Top-down (contract-first) approach - a WSDL document is created, and the necessary Java classes are generated from the WSDL
- Bottom-up (contract-last) approach - the Java classes are written, and the WSDL is generated from the Java classes.

In order to create a webservice from a WSDL, we need to use some JAX-WS compliant tool/framework which would generate JAX-WS portable artifacts we would be able to use in our Java code. Examples of such tools are listed below:

- Oracle wsimport (it was part of JDK till JDK 8) and wrappers like jaxws-maven-plugin
- Apache CXF - open source software project developing a Web services framework
- IBM Java2WSDL/WSDL2Java

After utilizing any wsdl-to-java tool we would get JAX-WS compliant service:

```
@WebService(
  name = "EmployeeServiceTopDown",
  targetNamespace = "http://topdown.server.jaxws.test.com/")
@SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE)
@XmlSeeAlso({
    ObjectFactory.class
})
public interface EmployeeServiceTopDown {
    @WebMethod(
      action = "http://topdown.server.jaxws.test.com/"
      + "EmployeeServiceTopDown/countEmployees")
    @WebResult(
      name = "countEmployeesResponse",
      targetNamespace = "http://topdown.server.jaxws.test.com/",
      partName = "parameters")
    public int countEmployees();
}
```

Examples of building a web service from WSDL and running it in Java code can be found here: https://github.com/ivanwolkow/wsdl-to-java .

Protobuf

Protocol buffers are **Google's** language-neutral, platform-neutral, extensible 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.

Protocol buffers currently support generated code in Java, Python, Objective-C, and C++. With our new proto3 language version, you can also work with Dart, Go, Ruby, and C#, with more languages to come.

Protobuf solves the problem of efficient data serialization/deserialization. How do you serialize and retrieve structured data structures? There are a few ways:

- Use Java Serialization. This is the default approach since it's built into the language, but it has a host of well-known problems (see Effective Java, by Josh Bloch pp. 213), and also doesn't work very well if you need to share data with applications written in C++ or Python. (language dependent)
- You can invent an ad-hoc way to encode the data items into a single string – such as encoding 4 ints as "12:3:-23:67". This is a simple and flexible approach, although it does require writing one-off encoding and parsing code, and the parsing imposes a small run-time cost. This works best for encoding very simple data.
- Serialize the data to XML. This approach can be very attractive since XML is (sort of) human readable and there are binding libraries for lots of languages. This can be a good choice if you want to share data with other applications/projects. However, XML is notoriously space intensive, and encoding/decoding it can impose a huge performance penalty on applications. Also, navigating an XML DOM tree is considerably more complicated than navigating simple fields in a class normally would be.

Protocol buffers are the flexible, efficient, automated solution to solve exactly this problem. With protocol buffers, you write a **.proto description** of the data structure you wish to store. From that, the protocol buffer compiler creates a class that implements automatic encoding and parsing of the protocol buffer data with an efficient **binary format**. The generated class provides getters and setters for the fields that make up a protocol buffer and takes care of the details of reading and writing the protocol buffer as a unit. Importantly, the protocol buffer format **supports the idea of extending the format over time** in such a way that the code can still read data encoded with the old format.

Protobuf data structure example:

```
syntax = "proto2";

package tutorial;

option java_multiple_files = true;
option java_package = "com.example.tutorial.protos";
option java_outer_classname = "AddressBookProtos";
```

```
message Person {
  optional string name = 1;
  optional int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    optional string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phones = 4;
}

message AddressBook {
  repeated Person people = 1;
}
```

Many standard simple data types are available as field types, including **bool, int32, float, double, and string**.

The " = 1", " = 2" markers on each element identify the **unique "tag"** that field uses in the binary encoding. Tag numbers 1-15 require one less byte to encode than higher numbers, so as an optimization you can decide to use those tags for the commonly used or repeated elements, leaving tags 16 and higher for less-commonly used optional elements. Each element in a repeated field requires re-encoding the tag number, so repeated fields are particularly good candidates for this optimization.

Each field must be annotated with one of the following modifiers:

- **optional**: the field may or may not be set. If an optional field value isn't set, a default value is used. For simple types, you can specify your own default value, as we've done for the phone number type in the example. Otherwise, a system default is used: zero for numeric types, the empty string for strings, false for bools. For embedded messages, the default value is always the "default instance" or "prototype" of the message, which has **none of its fields set**. Calling the accessor to get the value of an optional (or required) field which has not been explicitly set always returns that field's default value.
- **repeated**: the field may be repeated any number of times (including zero). The order of the repeated values will be preserved in the protocol buffer. Think of repeated fields as **dynamically sized arrays**.
- **required**: a value for the field must be provided, otherwise the message will be considered "uninitialized". Trying to build an uninitialized message will throw a RuntimeException. Parsing an uninitialized message will throw an IOException. Other than this, a required field behaves exactly like an optional field.

Required Is Forever! You should be very careful about marking fields as required. If at some point you wish to stop writing or sending a required field, it will be problematic to change the field to an optional field – old readers will consider messages without this field to be incomplete and may reject or drop them unintentionally. You should consider writing application-specific custom validation routines for your buffers instead. Within Google, **required fields are strongly disfavored**; most messages defined in proto2 syntax use optional and repeated only. (Proto3 does not support required fields at all.)

The message classes generated by the protocol buffer compiler are all **immutable**. Once a message object is constructed, it cannot be modified, just like a Java String. To construct a message, you must first construct a builder, set any fields you want to set to your chosen values, then call the builder's build() method.

```
Person john =
  Person.newBuilder()
    .setId(1234)
    .setName("John Doe")
    .setEmail("jdoe@example.com")
    .addPhones(
      Person.PhoneNumber.newBuilder()
        .setNumber("555-4321")
        .setType(Person.PhoneType.HOME))
    .build();
```

Each protocol buffer class has methods for writing and reading messages of your chosen type using the protocol buffer binary format:

- byte[] **toByteArray**();: serializes the message and returns a byte array containing its raw bytes.
- static Person **parseFrom**(byte[] data);: parses a message from the given byte array.

- void **writeTo**(OutputStream output);: serializes the message and writes it to an OutputStream.
- static Person **parseFrom**(InputStream input);: reads and parses a message from an InputStream.

Extending a Protocol Buffer rules:

- you must not change the tag numbers of any existing fields.
- you must not add or delete any required fields.
- you may delete optional or repeated fields.
- you may add new optional or repeated fields but you must use fresh tag numbers (i.e. tag numbers that were never used in this protocol buffer, not even by deleted fields).

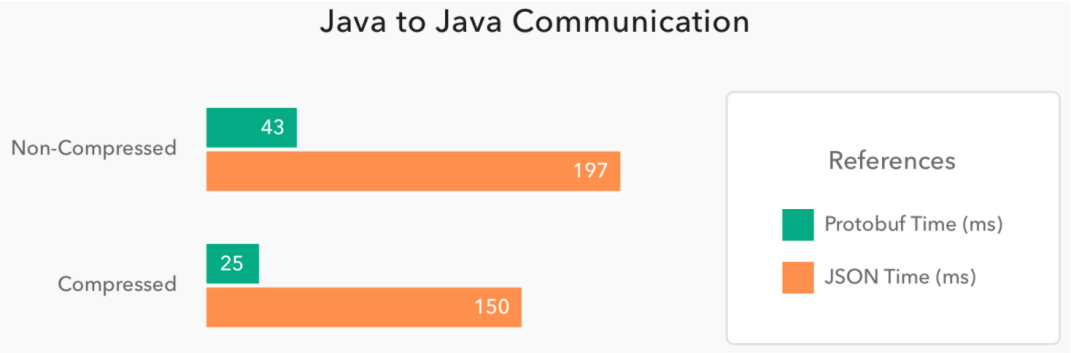
Protobuf vs JSON

- Protobuf is a **binary** data-interchange format developed by Google whereas JSON is **human-readable** data-interchange format. JSON is derived from JavaScript but as the name suggests, it is not limited to JavaScript only. It was designed in such a way that it can be used in multiple languages
- Protobuf supports **binary serialization** format whereas JSON is for simple **text serialization** format

- JSON is useful for common tasks and is **limited to certain types of data**(string, number, object (JSON object), array, boolean, null). It means JSON cannot serialize and de-serialize every python object. Protobuf covers a wide variety of data types when compared to JSON (scalar types, Nested Types, Unknown types - Any/Oneof, Maps) . Even **enumerations and methods** can be serialized with Protobuf.
- Both Protocol buffers and JSON are languages interoperable but **Protobuf are limited to subsets of programming language** whereas **JSON is widely accepted**.
- **JSON contains the only message** and not schema whereas Protobuf not only has messages but also **includes a set of rules and schemas** to define these messages.
- Protobuf is mostly useful for **internal services** whereas JSON is mostly useful for **web applications**
- Prior knowledge of schema is essential in decoding Protobuf messages whereas data can be easily decoded or parsed in JSON with knowing schemas in advance.

Using protobuf usually makes little difference for web applications (JavaScript to Java Communication), but can really boost efficiency in Java to Java Communication.

The chart below was generated with the average performance of 500 GET requests issued by one Spring Boot application to another Spring Boot application:



When using Protobuf in a non-compressed environment, the requests took **78%** less time than the JSON requests. This shows that the binary format performed almost 5 times faster than the text format. And, when issuing these requests in a compressed environment, the difference was even bigger. Protobuf performed 6 times faster, taking only 25ms to handle requests that took 150ms in a JSON format.

As you can see, when we have environments that JSON is not a native part of, the performance improvement is huge. So, whenever you face some latency issues with JSON, consider migrating to Protobuf.

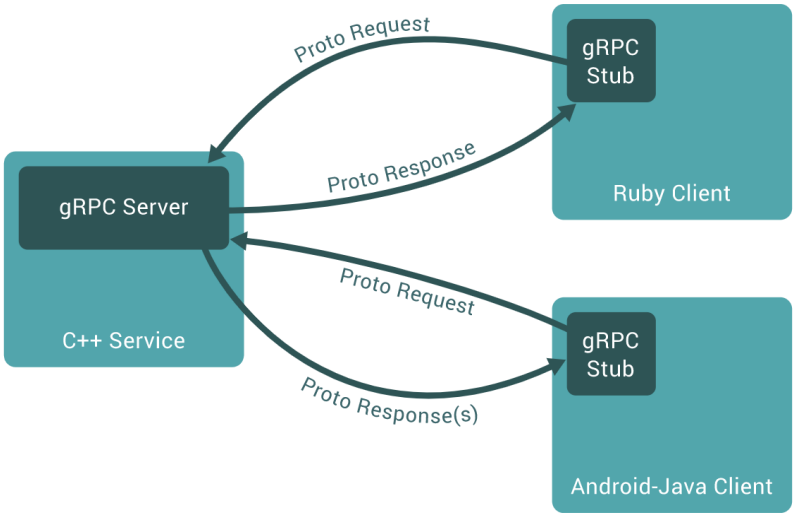
Disadvantages:

- **Lack of resources**. You won't find that many resources (do not expect a very detailed documentation, nor too many blog posts) about using and developing with Protobuf.
- **Smaller community**. Probably the root cause of the first disadvantage. On Stack Overflow, for example, you will find roughly 1.500 questions marked with Protobuf tags. While JSON have more than 180 thousand questions on this same platform.

- **Lack of support**. Google does not provide support for other programming languages like Swift, R, Scala and etc. But, sometimes, you can overcome this issue with third party libraries, like Swift Protobuf provided by Apple.
- **Non-human readability**. JSON, as exchanged on text format and with simple structure, is easy to be read and analyzed by humans. This is not the case with a binary format.

gRPC

gRPC is a modern open source high performance **Remote Procedure Call (RPC) framework** that can run in any environment. It can efficiently connect services in and across data centers with pluggable support for **load balancing, tracing, health checking and authentication**. It is also applicable in last mile of distributed computing to connect devices, mobile applications and browsers to backend services.



In gRPC, a client application can directly call a method on a server application on a different machine as if it were a local object, making it easier for you to create distributed applications and services. 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 (referred to as just a client in some languages) that provides the same methods as the server.

gRPC clients and servers can run and talk to each other in a variety of environments - from servers inside Google to your own desktop - and can be written in any of gRPC's supported languages. So, for example, you can easily create a gRPC server in Java with clients in Go, Python, or Ruby. In addition, the latest Google APIs will have gRPC versions of their interfaces, letting you easily build Google functionality into your applications.

By default, **gRPC uses Protocol Buffers**, Google's mature open source mechanism for serializing structured data (although it can be used with other data formats such as JSON).

```
message Person {
  string name = 1;
  int32 id = 2;
  bool has_ponycopter = 3;
}
```

You define gRPC services in ordinary proto files, with RPC method parameters and return types specified as protocol buffer messages:

```
// The greeter service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```

gRPC uses protoc with a special gRPC plugin to generate code from your proto file: you get generated gRPC client and server code, as well as the regular protocol buffer code for populating, serializing, and retrieving your message types.

gRPC lets you define four kinds of service method:

- **Unary RPCs** where the client sends a single request to the server and gets a single response back, just like a normal function call.

  rpc SayHello(HelloRequest) returns (HelloResponse);

- **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);

- **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);

- **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: for example, the server could wait to receive all the client messages before writing its responses, or it could alternately read a message then write a message, or some other combination of reads and writes. The order of messages in each stream is preserved.

  rpc BidiHello(stream HelloRequest) returns (stream HelloResponse);

Deadlines/Timeouts: gRPC allows **clients to specify how long they are willing to wait** for an RPC to complete before the RPC is terminated with a DEADLINE_EXCEEDED error. On the server side, the server can query to see if a particular RPC has timed out, or how much time is left to complete the RPC.

RPC termination: In gRPC, both the client and server make independent and local determinations of the success of the call, and their conclusions may not match. This means that, for example, you could have an RPC that finishes successfully on the server side ("I have sent all my responses!") but fails on the client side ("The responses arrived after my deadline!"). It's also possible for a server to decide to complete before a client has sent all its requests.

Cancelling an RPC: Either the client or the server can cancel an RPC at any time. A cancellation terminates the RPC immediately so that no further work is done.

Metadata is information about a particular RPC call (such as authentication details) in the form of a list of key-value pairs, where the keys are strings and the values are typically strings, but can be binary data. Metadata is opaque to gRPC itself - it lets the client provide information associated with the call to the server and vice versa. Access to metadata is language dependent.

Channel: A gRPC channel provides a connection to a gRPC server on a specified host and port. It is used when creating a client stub. Clients can specify channel arguments to modify gRPC's default behavior, such as switching message compression on or off. A channel has state, including connected and idle. How gRPC deals with closing a channel is language dependent. Some languages also permit querying channel state.

gRPC vs REST

gRPC largely follows HTTP semantics over HTTP/2 but we explicitly allow for full-duplex streaming. We diverge from typical REST conventions as we use static paths for performance reasons during call dispatch as parsing call parameters from paths, query parameters and payload body adds latency and complexity. We have also formalized a set of errors that we believe are more directly applicable to API use cases than the HTTP status codes.

gRPC provides plenty of advantages. Unlike REST, it can make the most out of HTTP 2, using **multiplexed streams** and following the **binary protocol**. Plus, it offers performance benefits due to the Protobuf message structure, and let's not forget the in-built code generation features which enable a multilingual environment.

Some key features of gRPC comparing to REST:

- Protobuf Instead of JSON
- Built on HTTP 2 (and making use of benefits of HTTP 2) Instead of HTTP 1.1
- In-Born Code Generation Instead of Using Third-Party Tools Like Swagger
- 7 to 10 times Faster Message Transmission (https://medium.com/@EmperorRXF/evaluating-performance-of-rest-vs-grpc-1b8bdf0b22da)
- Slower Implementation than REST

Some thoughts on when to use gRPC APIs:

- **Microservices connections**: gRPC's low-latency and high-speed throughput communication make it particularly useful for connecting architectures that consist of lightweight microservices where the efficiency of message transmission is paramount.
- **Multi-language systems**: With its native code generation support for a wide range of development languages, gRPC is excellent when managing connections within a polyglot environment.
- **Real-time streaming**: When real-time communication is a requirement, gRPC's ability to manage bidirectional streaming allows your system to send and receive messages in real-time without waiting for Unary client-response communication.
- **Low-power low-bandwidth networks**: gRPC's use of serialized Protobuf messages offers light-weight messaging, greater efficiency, and speed for bandwidth-constrained, low-power networks (especially when compared to JSON). IoT would be an example of this kind of network that could benefit from gRPC APIs.