1. What is Web Service?

Web Service is designed for application-to-application interaction. It should be interoperable – not platform dependent. It should also allow communication over network.

1. What is SOAP?

In SOAP, **XML request and response** is used for the communication between the applications. SOAP defines the specific XML request and response structure. We would need to define a **SOAP Envelope** which contains **SOAP Header** and **SOAP** **Body**. Header contains the meta information and body contains the actual request or response message. Header is optional but service need to use SOAP XML request and response for communication. SOAP can communicate over HTTP or MQ.

1. What is WSDL?

In SOAP, the service definition is provided using WSDL (web service definition language). WSDL contains the following:-

Endpoint

All Operations

Request Structure

Response Structure

1. What is Rest?

Rest stands for Representational State Transfer. It is a term coined by Roy Fielding. It’s an architecture style. In rest we need to think in terms of resources and make use HTTP methods. It communicates over HTTP.

**Rest Terminology**

**Verbs** - HTTP Methods: GET, PUT, POST, DELETE

**Messages** - the payload of the action (JSON/XML)

**URI** - Uniform Resource Identifier

A unique string identifying a resource

**URL** - Uniform Resource Locator

A URI with network information - http://www.example.com

1. Difference between SOAP and Rest?

|  |  |
| --- | --- |
| SOAP | Rest |
| Soap is a protocol which uses XML request and response. | Rest is an architectural style. |
| It uses XML request and response and it should adhere the SOAP structure using the SOAP Envelope, Header and Body. | Any exchange format can be used such as Json, XML, Text etc. |
| It uses WSDL for service definition. | It doesn’t have any standard service definition language. |
| It can use HTTP or MQ as transfer protocol. | Rest uses HTTP as transfer protocol |
| There are some complexities associated with parsing the SOAP xml. | Rest is easy to implementation. |

1. What are different HTTP methods or verbs?

**GET** – It is a **get** request for a resource.

**HEAD** – It is like GET, but it only asks for meta information without the body.

**POST** – It is a **create** request.

**PUT** – It is a **create or update** request.

**DELETE** – It is a **delete** request for a resource.

**TRACE** – It will **echo the received request**. Can be used to see if the request was altered by intermediate servers.

**OPTIONS** – Returns the **HTTP methods supported** by the server for the specified url.

**CONNECT** – Converts the request to a transparent TCP/IP tunnel, typically for HTTPS through an unencrypted HTTP proxy.

**PATCH** – **Applies the partial modifications** to the specified resource.

1. What are the safe HTTP methods?

GET, HEAD, OPTIONS and TRACE

1. What are Idempotent HTTP methods?

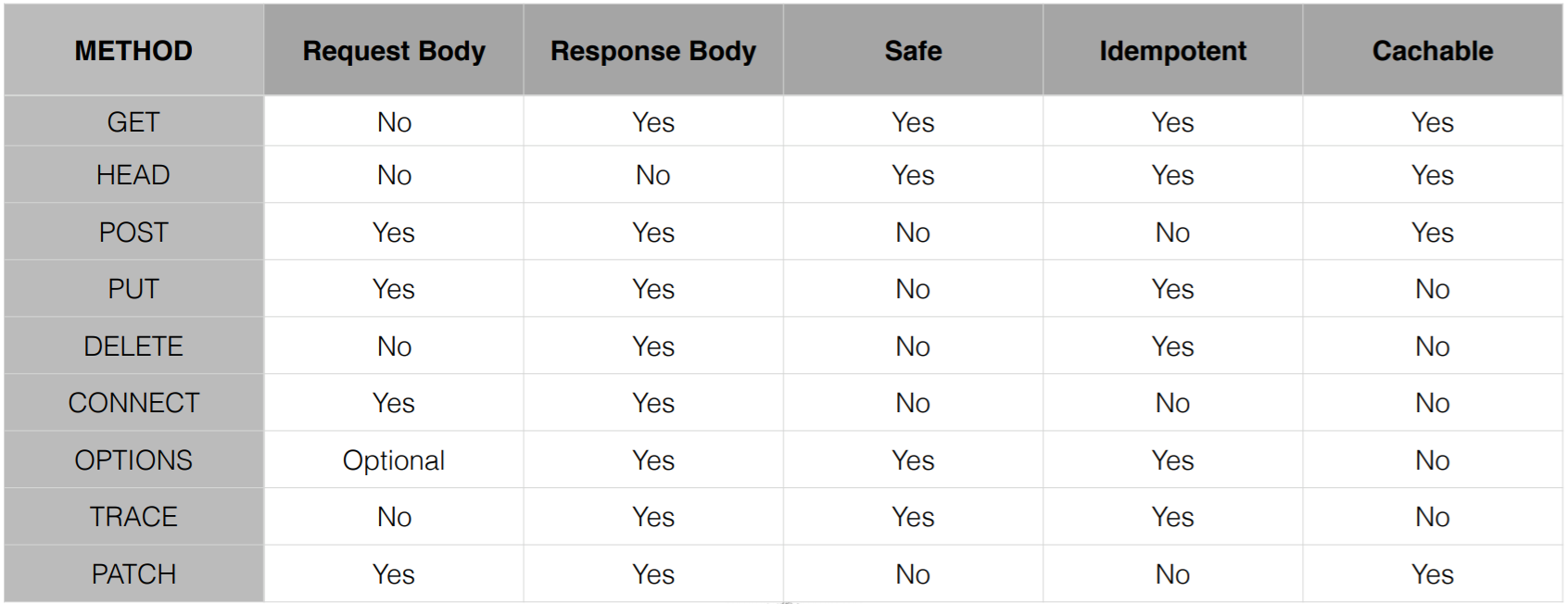
HTTP methods which can be repeatedly used without any further effect on the outcome.

All safe methods (**GET, HEAD, OPTIONS and TRACE**) are idempotent.

**PUT** and **DELETE** are idempotent.

1. What are non-idempotent HTTP methods?

**POST** is **non-Idempotent** method. Multiple posts are likely to create multiple resources.



1. What are the different HTTP status codes?

• **100** series are **informational** in nature

• **200** series indicate **successful** request

• **300** series are **redirections**

• **400** series are **client errors**

• **500** series are **server-side errors**

1. What are some common HTTP status codes?

• **200** Okay; **201** Created; **204** Accepted

• **301** Moved Permanently

• **400** Bad Request; **401** Not Authorized; **404** Not Found

• **500** Internal Server Error; **503** Service Unavailable

1. What is Richardson Maturity Model?

**Level 0: Swap of POX**

* + POX - Plain Old XML
  + Uses implementing protocol as a transport protocol
  + Typically **uses one URI** and **one kind of method**
  + Examples - RPC, SOAP, XML-RPC

**Level 1: Resources**

* Uses **multiple URIs** to identify **specific resources**
* Examples:

<http://www.example.com/product/1234>

http://www.example.com/product/5687

* Still uses a **single method** (ie GET)

**Level 2: HTTP Verbs**

* **HTTP Verbs are used with URIs for desired actions**
* Examples:

GET /products/1234 - to return data for product 1234

PUT /products/1234 (with XML body) to update data for product 1234

DELETE /products/1234 to delete product 1234

* Most common in practical use

**Level 3: Hypermedia**

* **Representation now contains URIs** which may be useful to consumers
* Helps client developers explore the resource
* No clear standard currently
* Spring provides an implementation of HATEOS (Hypermedia As The Engine Of application State)

Richardson Maturity Model

Level 0 - Expose SOAP webservice in Rest style

eg: http://server/getPosts

Level 1 - Expose Resources with proper URI

eg: http://server/accounts

Level 2 - Level 1 + HTTP methods

Level 3 - Level 2 + HATEOAS

1. What is SpringBoot Auto Configuration?

It configures the beans if class or jar is found on the classpath.

For e.g.: when we select **SpringBootWebStarter** which had the dependency on **Spring Web MVC** loads the **Dispatcher Servlet** class in the **classpath**. Hence, **SpringBootAutoConfiguration** configures **DispatcherServletAutoConfiguration**.

Similarly, other beans are configured/created like **ErrorMvcAutoConfiguration**, **HibernateJpaAutoConfiguration**, **HttpMessageConvertersAutoConfiguration**, **JacksonAutoConfiguration**, **ValidationAutoConfiguration**, **ViewResolver** etc.

**Microservices**

1. **What are microservices?**

* Microservices is an architectural style.
* Decomposition of single system into a suite of small services, each running as independent processes and intercommunicating via open protocols.
* Developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API.
* **Composing a single application using a suite of small services, each running as independent processes, intercommunicating via open protocols (like HTTP/REST, TCP, UDP or messaging), separately written, deployed, scaled and maintained. Microservices encapsulate business capability and they are independently replaceable and upgradeable.**

1. **Difference between SOA and Microservices?**

* SOA is about integrating various enterprise applications. Microservices are mainly about decomposing single application.
* SOA relies on Orchestration. Microservices rely on choreography.
* SOA relies on smart integration technology, dumb services. Microservices relies on smart services, dumb integration technology.

1. Microservices Example?

Consider a monolith Shopping Cart application having functions for Searching for products, Product Catalog, Inventory management, Shopping cart, Checkout, Fulfillment

Search

Reviews

Cart

Checkout

Contact

Catalog

Shipping

Customer

Payment

Order

Inventory

Product

1. What are advantages and disadvantages of **Monolithic application**?

**Advantages**

* Easy to comprehend but not to digest
* Easy to test as a single unit
* Easy to deploy as a single unit
* Easy to manage changes up to a certain limit
* Easy to scale (When care is taken)
* Complexity managed by language construct

**Disadvantages**

* **Language and Framework Lock** – Entire application written with single technology stack doesn’t let you take advantage of emerging technologies.
* **Digestion** – Single developer cannot digest the larger codebase. Single team can’t handle a single large application. Amazon’s “2 Pizza” rule.
* **Deployment** – Cannot independently deploy single change to a single component. Changes are “held-hostage” by other changes.

1. What are Micro Services advantages?

* Easy to digest each service
* Very easy to test, deploy, manage and version
* It’s easy scale single service.
* Change cycle is decoupled.
* No Language and Framework Lock.

1. What are disadvantages of Microservices?

* Complexity has moved out of the application, but into the operations layer. **Fallacies of Distributed Computing.**
* **Services may be unavailable**
* Remote calls are more expensive than in-process calls
* **Transactions: Must rely on eventual consistency over ACID**
* **Feature span multiple services**
* **Change management becomes difficult because it needs to consider the interaction of services.**
* **Refactoring module boundaries**.

1. **What are the fallacies of distributed computing?**

* The network is reliable
* Latency is zero
* Bandwidth is infinite
* The network is secure.
* Topology doesn’t change
* There is one administrator.
* Transport cost is zero.
* The network is homogenous.

1. How do you **break a monolith into Microservices**?

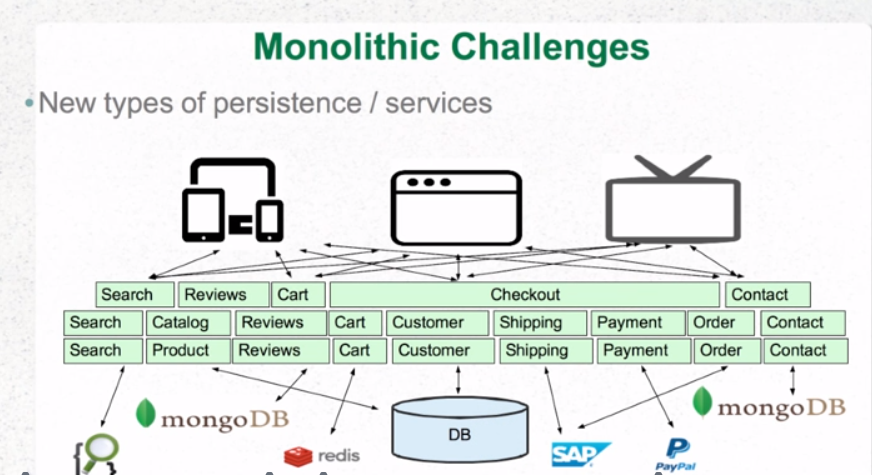
Primary Consideration is Business functionality.

* Noun-based
* Verb-based
* Single Responsibility Principle
* Bounded Context (Domain Driven Design)

1. What is Spring Boot?

Opinionated approach to configuration/ defaults

1. **What** **is Spring Cloud**?
   * Spring Cloud is a **sub-project within the Spring IO** umbrella.
   * It provides libraries to apply common patterns needed in distributed applications.
     + Distributed/ Versioned/ Centralized Configuration Management
     + Service Registration and Discovery
     + Load Balancing
     + Service-to-Service calls
     + Circuit Breakers
     + Routing
   * Spring Cloud provide enables easy use of Netflix libraries.
   * Spring Cloud is based on Spring Boot.



What is Spring Cloud Stream?

Spring Cloud Stream is a binding frameworks which binds the application code to the remote destination.

**What are the different patterns to migrate the monoliths to Microservices?**

**Strangler Application Pattern** – Modernizing an application by incrementally developing a new (strangler) application around the legacy application. The strangler application has a microservice architecture. Monoliths needs to be modular monoliths application. Strangler application grows over time.

**Domain Driven Design** – DDD brings clarity to complexity. It is a way of looking at software from top down.

**What is Domain Driven Design (DDD)?**

DDD brings clarity to complexity. It is a way of looking at software from top down. It enforces to thinking in terms of Context.

**Domain** - A sphere of knowledge, influence or activity. The subject area to which user applies a program is the domain of the software.

**Model** – A system of abstraction that describes the selected aspects of a domain and can be used to solve problems related to that domain.

**What is Ubiquitous language?**

A language structured around the domain model and used by all team members within a bounded context to connect all the activities of the team with software.

**What is Context?**

The setting in which a word or statement appears that determines its meaning. Statements about a model can only be understood in a context.

**What is Bounded Context?**

A description of a boundary (typically a subsystem, or the work of a particular team) within which a particular model is defined and applicable.

What is Entities?

Value Objects

Domain Events

Services

Aggregates

Repositories

Factories

**What is Cohesion?**

The degree to which the elements inside a module belong together is said to be cohesion.

**What is Coupling?**

The measure of the strength of the dependencies between components is said to be coupling. A good design is always said to have High Cohesion and Low Coupling.

**What is Distributed Transaction?**

Distributed Transaction is any situation where a single event results in the mutation of two or more separate sources of data which cannot be committed atomically. In the world of microservices, it becomes even more complex as each service is a unit of work and most of the time multiple services must work together to make a business successful.

**How to manage transactions in Microservices Architecture?**

One of the best ways to solve the problem of distributed transactions is to avoid them completely.

In a distributed system involving multiple databases, we have **two options** to achieve ACID compliance:

**Two-Phase Commit** - One way to achieve ACID compliance is to use a two-phase commit (a.k.a 2PC), which ensures that all involved services must commit to transaction completion or all the transactions are rolled back.

**Eventual Consistency** / **SAGA** - Use eventual consistency, where multiple databases owned by different microservices **become eventually consistent using asynchronous messaging** using messaging protocol. Eventual consistency is a specific form of weak consistency.

**2 Phase Commit** should ideally be **discouraged in microservices architecture** due to its fragile and complex nature. We can achieve some level of ACID compliance in distributed systems through eventual consistency and that should be the right approach to do it.

How would you implement SSO for Microservice Architecture?

**What is 12 Factor App?**

<https://12factor.net/>

**I. Codebase -** One codebase tracked in revision control, many deploys

**II. Dependencies** - Explicitly declare and isolate dependencies

**III. Config** - Store config in the environment

**IV. Backing services** - Treat backing services as attached resources

**V. Build, release, run** - Strictly separate build and run stages

**VI. Processes** - Execute the app as one or more stateless processes

**VII. Port binding** - Export services via port binding

**VIII. Concurrency** - Scale out via the process model

**IX. Disposability** - Maximize robustness with fast startup and graceful shutdown

**X. Dev/prod parity** - Keep development, staging, and production as similar as possible

**XI. Logs** - Treat logs as event streams

**XII. Admin processes** - Run admin/management tasks as one-off processes

**How do microservices communicate with each other?**

Microservices are often integrated using a simple protocol like REST over HTTP. Other communication protocols can also be used for integration like AMQP, JMS, Kafka, etc.

The communication protocol can be broadly divided into two categories- synchronous communication and asynchronous communication.

**Synchronous Communication**

RestTemplate, WebClient, FeignClient can be used for synchronous communication between two microservices. Ideally, we should minimize the number of synchronous calls between microservices because networks are brittle, and they introduce latency. Ribbon - a client-side load balancer can be used for better utilization of resource on the top of RestTemplate. Hystrix circuit breaker can be used to handle partial failures gracefully without a cascading effect on the entire ecosystem. Distributed commits should be avoided at any cost, instead, we shall opt for eventual consistency using asynchronous communication.

**Asynchronous Communication**

In this type of communication, the client does not wait for a response, instead, it just sends the message to the message broker. AMQP (like RabbitMQ) or Kafka can be used for asynchronous communication across microservices to achieve eventual consistency.

**What should be preferred communication style in microservices?**

* You must use asynchronous communication while handling HTTP POST/PUT (anything that modifies the data) requests, using some reliable queue mechanism (RabbitMQ, AMQP, etc.)
* It's fine to use synchronous communication for Aggregation pattern at API Gateway Level. But this aggregation should not include any business logic other than aggregation. Data values must not be transformed at Aggregator, otherwise, it defeats the purpose of Bounded Context. In Asynchronous communication, events should be published into a Queue. Events contain data about the domain, it should not tell what to do (action) on this data.
* If microservice to microservice communication still requires synchronous communication for GET operation, then seriously reconsider the partitioning of your microservices for bounded context, and create some tasks in backlog/technical debt.

**What is the difference between Orchestration and Choreography in microservices context?**

In Orchestration, we rely on a central system to control and call other Microservices in a certain fashion to complete a given task. The central system maintains the state of each step and sequence of the overall workflow. In Choreography, each Microservice works like a State Machine and reacts based on the input from other parts. Each service knows how to react to different events from other systems. There is no central command in this case.

Orchestration is a tightly coupled approach and is an anti-pattern in a microservices architecture. Whereas, Choreography’s loose coupling approach should be adopted where-ever possible.

Example

Let’s say we want to develop a microservice that will send product recommendation email in a fictitious e-shop. In order to send Recommendations, we need to have access to user’s order history which lies in a different microservices.

In Orchestration approach, this new microservice for recommendations will make synchronous calls to order service and fetch the relevant data, then based on his past purchases we will calculate the recommendations. Doing this for a million users will become cumbersome and will tightly couple the two microservices.

**In Choreography approach**, we will use event-based Asynchronous communication where whenever a user makes a purchase, an event will be published by order service. Recommendation service will listen to this event and start building user recommendation. This is a loosely coupled approach and highly scalable. The event, in this case, does not tell about the action, but just the data.

What is the use of PACT in Microservices architecture?

PACT is an open source tool to allow testing interactions between service providers and consumers in isolation against the contract made so that the reliability of Microservices integration increases.

Whether do you find GraphQL the right fit for designing microservice architecture?

GraphQL and microservices are a perfect fit, because GraphQL hides the fact that you have a microservice architecture from the clients. From a backend perspective, you want to split everything into microservices, but from a frontend perspective, you would like all your data to come from a single API. Using GraphQL is the best way I know of that lets you do both. It lets you split up your backend into microservices, while still providing a single API to all your application and allowing joins across data from different services.

**What is OAuth?**

OAuth stands for open authorization protocol. This allows accessing the resources of the resource owner by enabling the client applications on HTTP services such as third-party providers Facebook, GitHub, etc. So with this, you can share resources stored on one site with another site without using their credentials.

**What is DRY in Micro Services Architecture?**

DRY stands for **Don’t Repeat Yourself**. It basically promotes the concept of reusing the code. This results in developing and sharing the libraries which in turn result in tight coupling.

What is a Consumer-Driven Contract (CDC)?

This is basically a pattern for developing Microservices so that they can be used by external systems. When we work on microservices, there is a particular provider who builds it and there are one or more consumers who use Microservice.

Generally, providers specify the interfaces in an XML document. But in Consumer Driven Contract, each consumer of service conveys the interface expected from the Provider.

What are Reactive Extensions in Microservices?

Reactive Extensions is also called Rx. It is a design pattern which allows collecting results by calling multiple services and then compile a combined response. Rx is a popular tool in distributed systems which works exactly opposite to legacy flows.

**What is Canary Releasing?**

Canary Releasing is a technique to reduce the risk of introducing a new software version in production. This is done by slowly rolling out the change to a small subset of users before giving it out to the entire infrastructure, i.e. making it available to everybody.

**Explain Blue-Green Deployment?**

Blue-green deployment is a technique that reduces downtime and risk by running two identical production environments called Blue and Green. At any time, only one of the environments is live, with the live environment serving all production traffic. For this example, Blue is currently live and Green is idle.

As you prepare a new version of your software, deployment and the final stage of testing takes place in the environment that is not live: in this example, Green. Once you have deployed and fully tested the software in Green, you switch the router so all incoming requests now go to Green instead of Blue. Green is now live, and Blue is idle.

This technique can eliminate downtime due to application deployment. In addition, blue-green deployment reduces risk: if something unexpected happens with your new version on Green, you can immediately roll back to the last version by switching back to Blue.