Microservices Interview Questions

Tulshiram Sule

Chapter 1. Core Concepts in Microservices

Before we delve deep into microservices architecture, we must get familiar with few basic concepts. We will use terms like Cohesion, Coupling, Immutability, DRY, Open/Close Principle, Single Responsibility Principle in upcoming sections. If you are already aware of these basic terms, then probably you can skip this chapter.

1.1. Cohesion

Cohesion refers to the degree of focus that a particular software component has. A multi-purpose mobile phone with camera, radio, torch, etc. for example has low cohesion compared to a dedicated DLSR that does one thing better.

> **Cohesion - wikipedia.org**
>
> Cohesion in software engineering is the degree to which the elements of a certain module belong together. Thus, it is a measure of how strongly related each piece of functionality expressed by the source code of a software module is.

Example of cohesion:

Lets suppose we have a monolithic application for a fictitious e-shop, that does order management, inventory management, user management, marketing, delivery management etc. This monolithic software has very low cohesion compared to a microservices based architecture where each microservice is responsible for a particular business functionality, for example -

1. User management microservice
2. Inventory management microservice
3. Order management microservice
4. Demand generation/marketing microservice
5. Delivery/shipping tracking microservice

High cohesion often correlates with loose coupling, and vice versa.

Benefits of High Cohesion:

1. Reduced modular complexity, because each module does one thing at a time.
2. Increased system maintainability, because logical changes in the domain affect fewer modules, and because changes in one module require fewer changes in other modules.
3. Increased module reusability, because application developers will find the component they need more easily among the cohesive set of operations provided by the module.
4. Easy understandability and testability.

Microservices in general should have a high cohesion i.e. each microservice should do one thing

and do it well.

1.2. Coupling

Coupling refers to the degree of dependency that exists between two software components.

A very good day to day example of coupling is Mobile handset that has battery sealed into the handset. Design in this case is tightly coupled because battery or motherboard can not be replaced from each other without affecting each other.

In Object Oriented Design we always look for *low coupling* among various components so as to achieve flexibility and good maintainability. Changes in one components shall not affect other components of the system.

Important

High cohesion is almost always related to low coupling.

1.3. Immutability in Microservices

Immutable services can, by definition, be deployed without any heavy weight installers or configuration management. This makes it easy to scale, load balance and making services highly available. Docker, a light weight container (compared to a virtual machine) can be used as immutable infrastructure enabler.

1.4. Open/Close Principle

Our classses should be open for extension but closed for modifications. To put this more concretely, you should write a class that does what it needs to flawlessly and not assuming that people should come in and change it later. It's closed for modification, but it can be extended by, for instance, inheriting from it and overriding or extending certain behaviors.

1.5. DRY (Don't Repeat Yourself)

DRY stands for Don't Repeat Yourself. It promotes concept of code reusability.

In DRY people develop libraries and share these libraries. But we shall always keep in mind the Bounded Context Principle along with DRY principle, we shall never share a code that violates the Bounded Context Principle. And we shall never create shared unified models across Bounded Contexts, for example its really a bad idea to create a single large unified model for Customer class and share it across microservices.

The basic design principle behind DRY is that we should not try to reinvent the wheel. At the same time, we should not share the same wheel for multiple purposes.

In Microservices architecture we shall avoid creating unified models that are shared across microservices boundary, as these models defeats the principle of Bounded Context. But if there is some reusable logic that can be shared across services, we shall create a shared library and use it as

a verisoned jar dependency everywhere.

1.6. SOLID

SOLID is a mnemonic acronym for five design principles intended to make software designs more understandable, flexible and maintainable. The five design principles are,

1. Single responsibility principle - a class should have only a single responsibility (i.e. changes to only one part of the software's specification should be able to affect the specification of the class).
2. Open/Close Principle - "software entities … should be open for extension, but closed for modification."
3. Liskov substitution principle - "objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program."
4. Interface segregation principle - "many client-specific interfaces are better than one general-purpose interface."
5. Dependency inversion principle - one should "depend upon abstractions, [not] concretions."

The principles are a subset of many principles promoted by Robert C. Martin.

1.7. Single Responsibility Principle

Single Responsibility Principle is based on principle of cohesion, where every module or class should have responsibility over a single part of functionality provided by the software, and that responsibility should be entirely encapsulated by the class.

A real world example

Consider a reporting module that creates content for a report and send it through email. If we embed both these functionalities into single class/module, then we are not following Single Responsibility Principle. Such software should be split into two module, one for creating the report content, another for sending the email, each one having single responsibility.

> A class should have only one reason to change
>
> — Robert C. Martin

1.8. 8 Fallacies of Distributed Computing

In 1994, Peter Deutsch, a sun fellow at the time, drafted 7 assumptions architects and designers of distributed systems are likely to make, which prove wrong in the long run - resulting in all sorts of troubles and pains for the solution and architects who made the assumptions. In 1997 James Gosling added another such fallacy [JDJ2004]. The assumptions are now collectively known as the "The 8 fallacies of distributed computing" [Gosling]:

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous

These fallacies written 20 years back are increasingly becoming irrelevant with the advancement of science & technology.

However, we shall make sure that our distributed system design is resilient to the above mentioned 8 fallacies. In the upcoming chapters, we will see how Spring Cloud provide us with the appropriate libraries (circuit-breaker, OAuth 2.0 JWT, API Gateway, etc.) to take care of these issues.

References

- https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing
- https://www.infoworld.com/article/3114195/system-management/the-8-fallacies-of-distributed-computing-are-becoming-irrelevant.html
- http://www.rgoarchitects.com/Files/fallacies.pdf

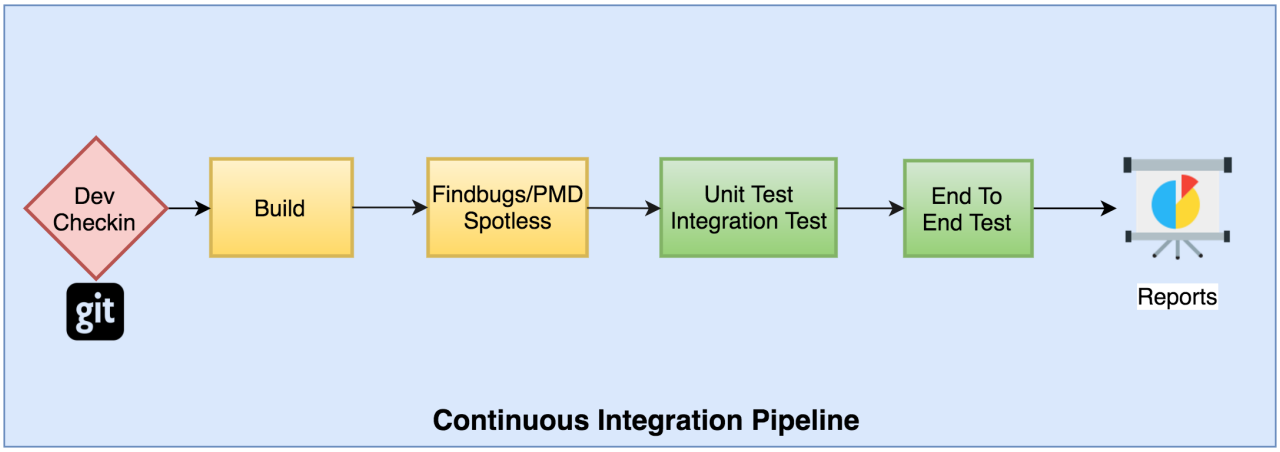
1.9. Continuous Integration (CI)

Continuous Integration is a software development approach where developers merge their code into a common repository, several times in a day.

CI provides the benefit of early feedback for any code that is developed and merged into common repository.

Every checkin should run a build pipeline, that includes

1. running set of unit tests
2. running integration Tests
3. build pre-checks - findbugs/PMD rules/code formatting/etc.
4. code coverage tools if any. (JaCoCo)
5. end-to-end tests. (Selenium/HtmlUnit/etc.)



Continuous Integration Pipeline (Jenkins or Other CI Tool)

This ensures that the common repository is always ready for the production deployment.

Tools like Git (Github/BitBucket, etc.), Jenkins and Maven/Gradle are normally used together to setup the build pipelines.

Jenkins Build Pipeline

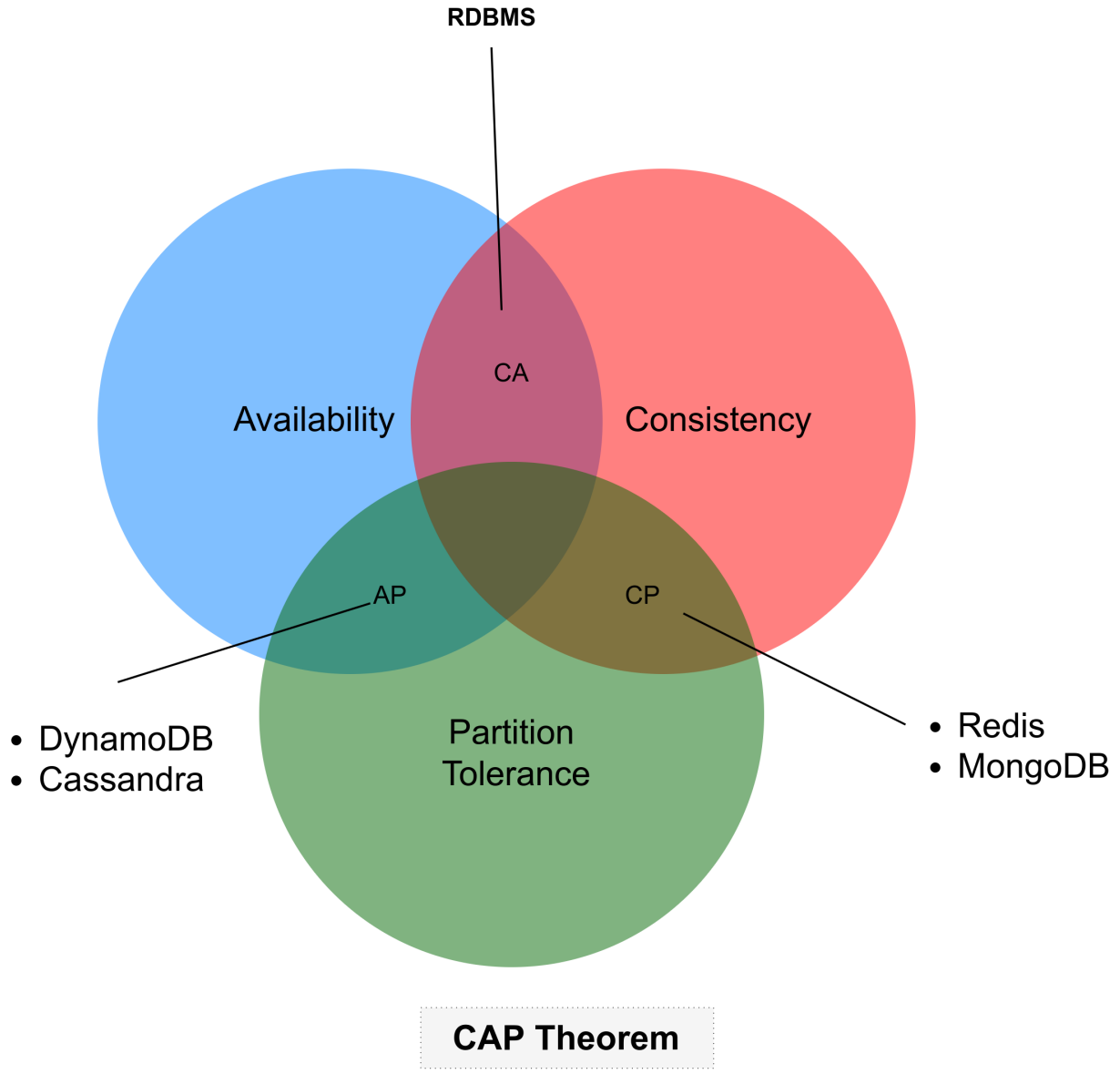
You can setup a Jenkins job that continuously monitors for commit on a pre-specified branch of git repository. Once developer commits his changes to this branch, Jenkins will trigger the build & test job and provide the feedback as early as possible. Dependency among two or more jobs can also be specified so that any affected dependency is also built & tested.

1.10. CAP Theorem

CAP Theorem

In theoretical computer science, the CAP theorem, also named Brewer's theorem after computer scientist Eric Brewer, states that it is impossible for a distributed data store to simultaneously provide more than two out of the following three guarantees

1. Consistency - Every read receives the most recent write or an error.
2. Availability - Every request receives a (non-error) response – without guarantee that it contains the most recent write.
3. Partition tolerance - the system continues to operate despite arbitrary partitioning due to network failures.



CAP (Consistency, Availability and Partition Tolerance) Theorem

CA databases

Most RDBMS (MySQL, PostreSQL, etc.) offer Consistency and High Availability. These databases are good for transactional needs (order management, payment service, etc.)

AP & CP databases

Partition tolerance is a required attribute of distributed scalable databases. So only CP and AP are the valid options for distributed databases. Out of these two, AP systems are easier to scale at the cost of consistency. These AP systems generally rely on eventual consistency.

Examples of AP systems are Cassandra and Amazon DynamoDB.

CP systems have to compromise on Availability, but these systems provides Strong Consistency and Partition Tolerance.

Examples of such systems are MongoDB and Redis.

1.11. 12 Factor App

The Twelve-Factor App is a recent methodology (and/or a manifesto) for writing web applications which run as a service.

Codebase

One codebase, multiple deploys. This means that we should only have one codebase for different versions of a microservices. Branches are ok, but different repositories are not.

Dependencies

Explicitly declare and isolate dependencies. The manifesto advices against relying on pre-installed softwares or libraries on the host machine. Every dependency should be put into `pom.xml` or `build.gradle` file.

Config

Store config in the environment. Do never commit your environment-specific configuration (most importantly: password) in the source code repo. Spring Cloud Config provides server and client-side support for externalized configuration in a distributed system. Using Spring Cloud Config Server you have a central place to manage external properties for applications across all environments.

Backing services

Treat backing services as attached resources. A microservice should treat external services equally, regardless of whether you manage them or some other team. For example, never hard code the absolute url for dependent service in your application code, even if the dependent microservice is developed by your own team. For example, instead of hard coding url for another service in your RestTemplate, use Ribbon (with or without Eureka) to define the url:

Correct approach

Creating named alias for a service, in `bootstrap.yml`

Listing 1. /src/main/resources/bootstrap.yml

```
product-service:
  ribbon:
    eureka:
      enabled: false
    listOfServers: localhost:8090,localhost:9092,localhost:9999
    ServerListRefreshInterval: 15000
```

And then using named alias to call the service inside code.

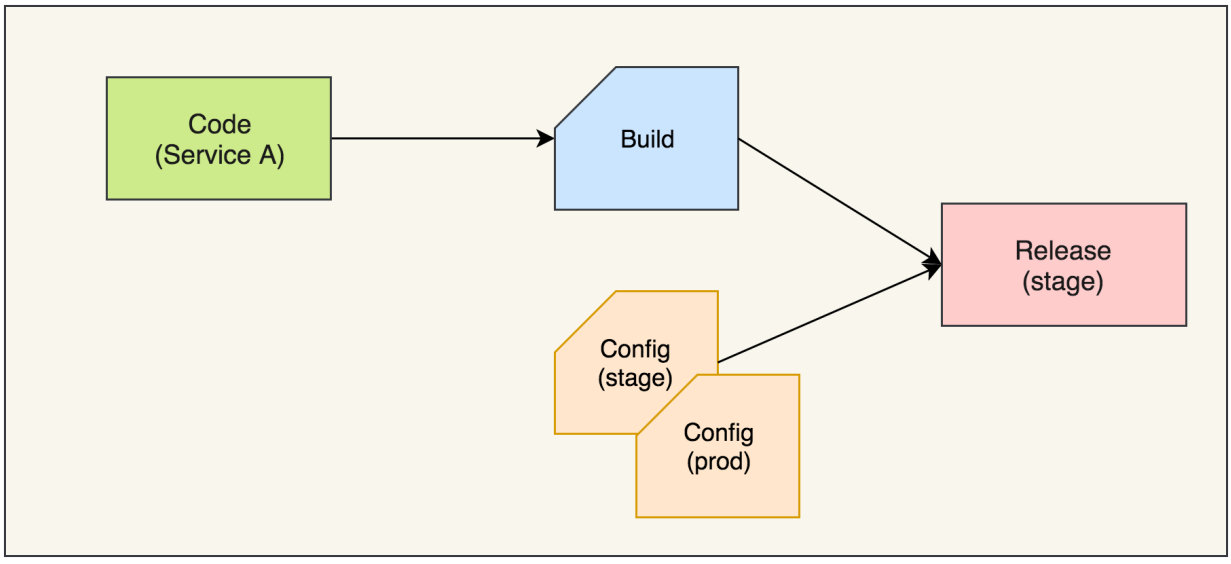
```
String greeting = this.restTemplate.getForObject("http://product-service/info", String.class);
```

Listing 2. Bad Approach (hardcoded host/port of dependent service)

```
String greeting = this.restTemplate.getForObject("http://localhost:8090/info", String.class);
```

Build Release & Run

Strictly separate build and run stages. In other words, you should be able to build or compile the code, then combine that with specific configuration information to create a specific release, then deliberately run that release. It should be impossible to make code changes at runtime, for e.g. changing the class files in tomcat directly. There should always be a unique id for each version of release, mostly a timestamp. Release information should be immutable, any changes should lead to a new release.



Build-Release-Run Principle

Processes

Execute the app as one or more stateless processes. This means that our microservices should be stateless in nature, and should not rely on any state being present in memory or in filesystem. Indeed the state does not belong in the code. So no sticky sessions, no in memory cache, no local filesystem storage, etc. Distributed cache like memcache, ehcache or Redis should be used

instead.

Port Binding

Export services via port binding. This is about having your application as standalone, instead of relying on a running instance of an application server, where you deploy. Spring boot provides mechanism to create an self-executable uber jar that contains all dependencies and embedded servlet container (jetty or tomcat).

Concurrency

Scale out via the process model. In the twelve-factor app, processes are a first class citizen. This does not exclude individual processes from handling their own internal multiplexing, via threads inside the runtime VM, or the async/evented model found in tools such as EventMachine, Twisted, or Node.js. But an individual VM can only grow so large (vertical scale), so the application must also be able to span multiple processes running on multiple physical machines. Twelve-factor app processes should never write PID files, rather it should rely on operating system process manager such as systemd - a distributed process manager on a cloud platform.

Disposability

Maximize robustness with fast startup and graceful shutdown. The twelve-factor app's processes are disposable, meaning they can be started or stopped at a moment's notice. This facilitates fast elastic scaling, rapid deployment of code or config changes, and robustness of production deploys. Processes should strive to minimize startup time. Ideally, a process takes a few seconds from the time the launch command is executed until the process is up and ready to receive requests or jobs. Short startup time provides more agility for the release process and scaling up; and it aids robustness, because the process manager can more easily move processes to new physical machines when warranted.

Dev/Prod parity

Keep development, staging, and production as similar as possible. Your development environment should almost identical to a production one (for example, to avoid some "works on my machine" issues). That doesn't mean your OS has to be the OS running in production, though. Docker can be used for creating logical separation for your microservices.

Logs

Treat logs as event streams, sending all logs only to stdout. Most Java Developers would not agree to this advise, though.

Admin processes

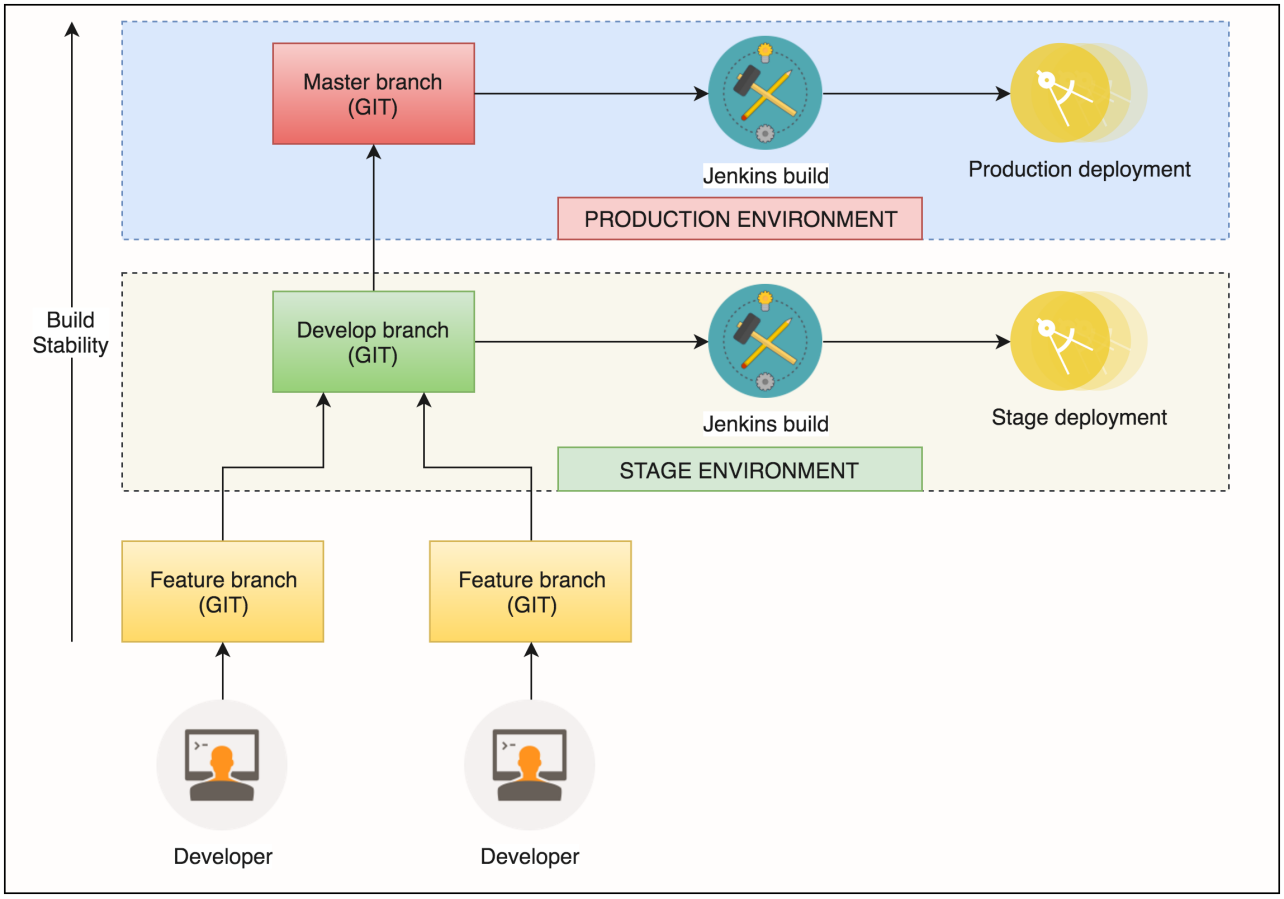
Run admin/management tasks as one-off processes. For example, a database migration should be run using a separate process altogether.

References

- https://12factor.net/
- https://cloud.spring.io/spring-cloud-config/
- https://spring.io/guides/gs/client-side-load-balancing/
- https://12factor.net/build-release-run

1.12. Typical Git workflow for a real project

You can setup git workflow as per project's need. Most common git workflow in any enterprise grade project would be a variant of the the following:



Git workflow and deployment pipeline

The overall process works like this:

1. Sprint is created with set of user stories. These user stories will map to set of features in your application.
2. Each developer will pick up a feature and create a corresponding feature branch in git repository. He/she will continue to commit his work to the respective feature branch throughout the sprint.
3. Once the feature branch is stable enough, changes will be merged to develop branch by the team lead.
4. A Jenkins job is configured for each environment (dev, stage, production) that will listen to git commits in respective branches and it will trigger the build → test → release jobs. So once feature branch is merged into develop branch, Jenkins will automatically execute the build, test it and deploy the changes to stage environment for manual/UAT testing.
5. Once changes in develop branch are tested up to satisfaction level, team lead will merge the changes to master branch. Another Jenkins job will listen to the commits and execute the build/test/deploy cycle for production environment.

Git vs SVN branch performance

If you have recently migrated from SVN, then you might think that creating separate branch for each feature might be an overhead. But that's not the case with Git. Git branches are very light weight and you can create branches on the fly for each bug you resolve, each feature/task you complete. In fact you should.

About GIT

Git was created by Linus Torvalds in 2005 for development of the Linux kernel, with other kernel developers contributing to its initial development. Its current maintainer since 2005 is Junio Hamano.

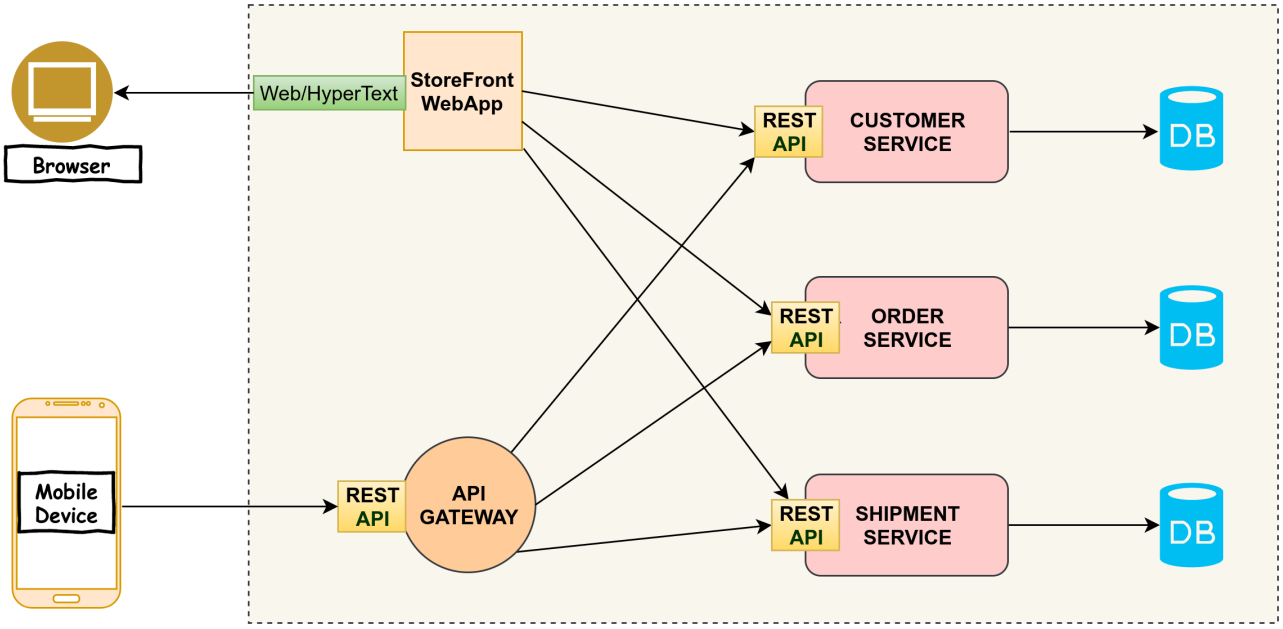
There is a beautiful book for learning git functionality, its freely hosted at:

https://git-scm.com/book/en/v2

Chapter 2. Introduction to Microservices

The term *microservices* became popular in late 2000 after big giants started moving their existing monolithic/SOA application into smaller autonomous services. As of this writing (2018), any new enterprise grade application in Java is potentially follows a microservices based architecture. Its a trend that will not stop any sooner, until we find a better way to craft software applications.

A typical microservices architecture is shown in the following diagram.



A Typical Architecture Microservices using Spring Cloud

Browser, mobile applications (iOS, Android), IOT devices talking to a distributed application based on microservices architecture using API gateway pattern. Each microservice has its own private data store, if there is a need for persistence. In the coming sections we will explore more on this.

2.1. Characteristics of a microservices architecture

1. High Cohesion - Small and focussed on doing one thing well. Small does not mean less number of lines of code because few programming languages are more verbose than others, but it means the smallest functional area that a single microservices caters to.
2. Loose Coupling - Autonomous - the ability to deploy different services independently, and reliability, due to the ability for a service to run even if another service is down.
3. Bounded Context - A Microservice serves a bounded context in a domain. It communicates with the rest of the domain by using an interface for that Bounded context.
4. Organisation around business capabilities instead of around technology.
5. Continuous Delivery and Infrastructure automation.
6. Versioning for backward compatibility. Even multiple versions of same microservices can exist in a production environment.
7. Fault Tolerance - if one service fails, it will not affect rest of the system. For example, if a

microservices serving the comments and reviews for a e-commerce fails, rest of the website should run fine.

8. Decentralized data management with each service owning its database rather than a single shared database. Every microservice has freedom to choose the right type of database appropriate for its business use-case (for example, RDBMS for Order Management, NoSql for catalogue management for an e-commerce website)
9. Eventual Consistency - event driven asynchronous updates.
10. Security - Every microservices should have capability to protect its own resource from unauthorised access. This is achieved using stateless security mechanisms like JSON Web Token (JWT pronounced as jot) with OAuth2.

2.2. Benefits of using Microservices Architecture

1. Each microservice is focussed on one business capability making them easier to maintain and develop.
2. Each microservice can be developed and deployed independently by different teams, thus they are autonomous.
3. Microservices can be created using heterogeneous technology stack. Appropriate technology stacked can be chosen for a given microservices as per business needs. A high performance engine can be written in Go while rest of the system can be developed using Spring Boot.
4. Scalability is better compared to monolithic application because most used microservice can be scaled more compared to least used microservice.
5. Resilience - Is one of the service goes down, it will not affect the entire application. Outage in service serving the static images content will not bring down the entire e-commerce web site.

2.3. Challenges in Microservices

1. DevOps is must because of explosion of number of processes in a production system. How to start and stop fleet of services?
2. Complexity of distributed computing such as “network latency, fault tolerance, message serialization, unreliable networks, handling asynchronous o/p, varying loads within our application tiers, distributed transactions, etc.”
3. How to make configuration changes across the large fleet of services with minimal effort.
4. How to deploy multiple versions of single microservice and route calls appropriately.
5. How to disconnect a microservice from ecosystem when it starts to crash unexpectedly.
6. How to isolate a failed microservice and avoid cascading failures in the entire ecosystem.
7. How to discover services in an elastic manner considering that services may be going UP or DOWN at any point in time.
8. How to aggregate logs/metrics across the services. How to identify different steps of a single client request spread across span of microservices.

2.4. Difference between Microservices and SOA

Microservices are continuation to SOA.

SOA started gaining ground due to its distributed architecture approach and it emerged to combat the problems of large monolithic applications, around 2006.

Both (SOA and Microservices) of these architectures share one common thing that they both are distributed architecture and both allow high scalability. In both, service components are accessed remotely through remote access protocol (RMI, REST, SOAP, AQMP, JMS, etc.). both are modular and loosely coupled by design and offer high scalability. Microservices started gaining buzz in late 2000 after emergence of light weight containers, Docker, Orchestration Frameworks (Kubernetes, mesos). Microservices differ from SOA in a significant manner conceptually -

1. SOA uses Enterprise Service Bus for communication, while microservices uses REST or some other less elaborate messaging system (AQMP, etc). Also, microservice follow "Smart endpoints and dumb points", which means that when a microservice needs another one as a dependency, it should use it directly without any routing logic / components handling the pipe.
2. In microservices, the service deployment and management should be fully automated, whereas SOA services are often implemented in deployment monoliths.
3. Generally microservices are significantly smaller than what SOA tends to be. Here we are not talking about the codebase here because few languages are more verbose than the other ones. We are talking about the scope (problem domain) of the service itself. Microservices generally do one thing in better way.
4. Microservices should own their own data while SOA may share common database. So one Microservices should not allow another Microservices to change/read its data directly.
5. Classic SOA is more platform driven, while we have lot of technology independence in case of microservices. Each microservice can have its own technology stack based on its own functional requirements. So microservices offers more choices in all dimensions.
6. Microservices makes as little assumption as possible on the external environment. A Microservice should manage its own functional domain and data model.

> The value of term Microservices is that it allows to put a label on a useful subset of the SOA terminology.
>
> — Martin Fowler

Microservices are essentially built over single responsibility principle. Robert C. Martin mentioned:

"Gather together those things that change for the same reason, and separate those things that change for different reasons"

2.5. References

1. http://www.oracle.com/technetwork/issue-archive/2015/15-mar/o25architect-2458702.html

Chapter 3. Microservices Interview Questions

This chapter is mostly Q&A oriented and our focus area would be:

- questions on microservices design patterns, anti-patterns and common pitfalls.
- questions on development and deployment of microservices.
- questions on monitoring of microservices.
- questions on Spring Boot, Spring Cloud and Netflix OSS.

Lets move forward…

3.1. How will you define Microservices Architecture?

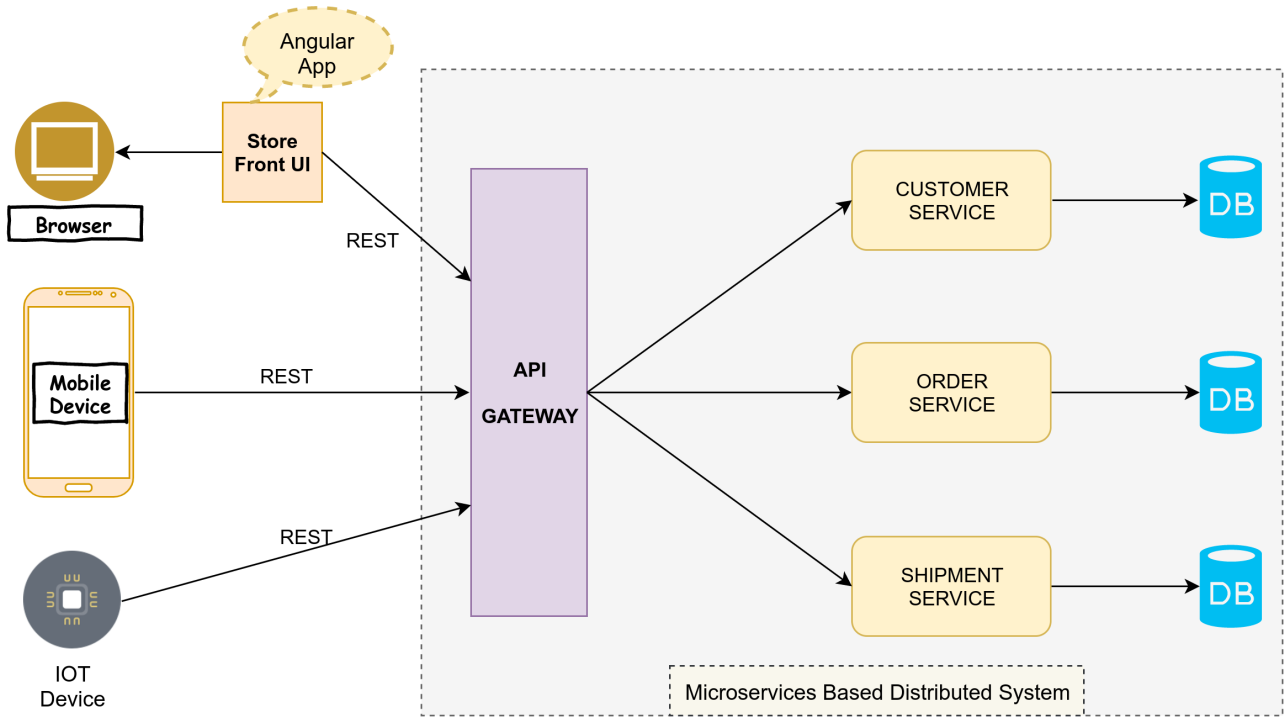
Microservices Architecture is a style of developing a scalable, distributed & highly automated system made up of many small autonomous services. It is not a technology but a new trend evolved out of SOA.

There is no single definition that fully describes the term "microservices". Famous authors have tried to define it in following way:

> Microservices are small, autonomous services that work together.
>
> — Sam Newman

> Loosely coupled service-oriented architecture with bounded contexts.
>
> — Adrian Cockcroft

> A microservice architecture is the natural consequence of applying the single responsibility principle at the architectural level.
>
> — Toby Clemson



Typical Microservices Architecture Spring Boot

3.2. What is Domain Driven Design?

Domain Driven Design (DDD) is a modelling technique for organized decomposition of complex problem domains. Eric Evans's book on Domain Driven Design has greatly influenced modern architectural thinking. DDD technique can be used while partitioning a monolith application into

microservices architecture.

Key principles of Domain Driven Design are:

1. placing the project's primary focus on the core domain and domain logic
2. basing complex designs on a model of the domain
3. initiating a creative collaboration between technical and domain experts to iteratively refine a conceptual model that addresses particular domain problems.

The term "Domain Driven Design" was coined by **Eric Evans** in his book of the same title.

Book link

Domain-Driven Design: Tackling Complexity in the Heart of Software

Reference

https://en.wikipedia.org/wiki/Domain-driven_design

3.3. What is Bounded Context?

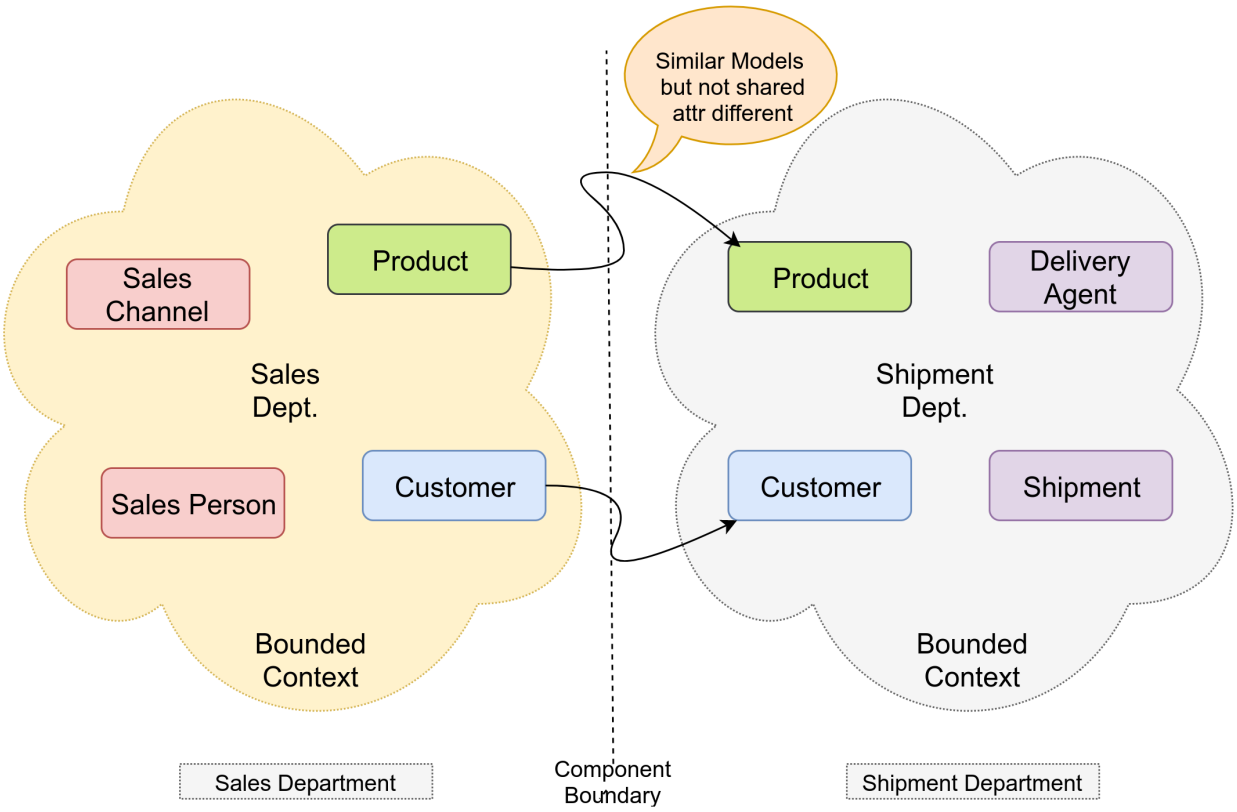
Bounded Context is a central pattern in Domain Driven Design. In Bounded Context, everything related to the domain is visible within context internally but opaque to other bounded contexts. DDD deals with large models by dividing them into different Bounded Contexts and being explicit about their interrelationships.

Monolithic Conceptual Model Problem

A single conceptual model for the entire organization is very tricky to deal with. The only benefit of such unified model is that integration is easy across the whole enterprise, but drawbacks are many, for example:

1. At first, its very hard to build a single model that works for entire organization.
2. Its hard for others (teams) to understand it.
3. Its very difficult to change such shared model to accommodate the new business requirements. The impact of such change will be widespread across team boundaries.
4. Any large enterprise needs a model that is either very large or abstract.
5. Meaning of a single word may be different in different department of a organization, so it may be really difficult to come up with a single unified model. Such model, even if created, will lead to lot of confusion across the teams.

DDD solves this problem by decomposing a large system into multiple Bounded Contexts, each of which can have a unified model, opaque to other bounded contexts.



Bounded Context for sales and shipment departments of an enterprise

As shown in image above, the two different bounded contexts, namely sales department and shipment department have duplicate models for customer and product that are opaque to other bounded context. In non DDD (domain driven design) world, we could have created a single unified model for customer and product and shared it using libraries across team boundaries.

Key points about Domain Driven Design

- DDD is about designing software based on models of the underlying domain.
- Bounded Context lays stress on the important observation that each entity works best within a localized context. So instead of creating a unified Product and Customer class across the fictitious eshop system, each problem domain (Sales, Support, Inventory, Shipment & Delivery etc.) can create its own, and reconcile the difference at the integration points.

3.4. What is polyglot persistence? Can this idea be used in monolithic applications as well?

Polyglot persistence is all about using different databases for different business needs within a single distributed system. We already have different database products in the market each for a specific business need, for example:

RDBMS

Relational databases are used for transactional needs (storing financial data, reporting requirements, etc.)

MongoDB

Documented oriented databases are used for documents oriented needs (for e.g. Product Catalog). Documents are schema free so changes in the schema can be accommodated into application without much headache.

Cassandra/Amazon DynamoDB

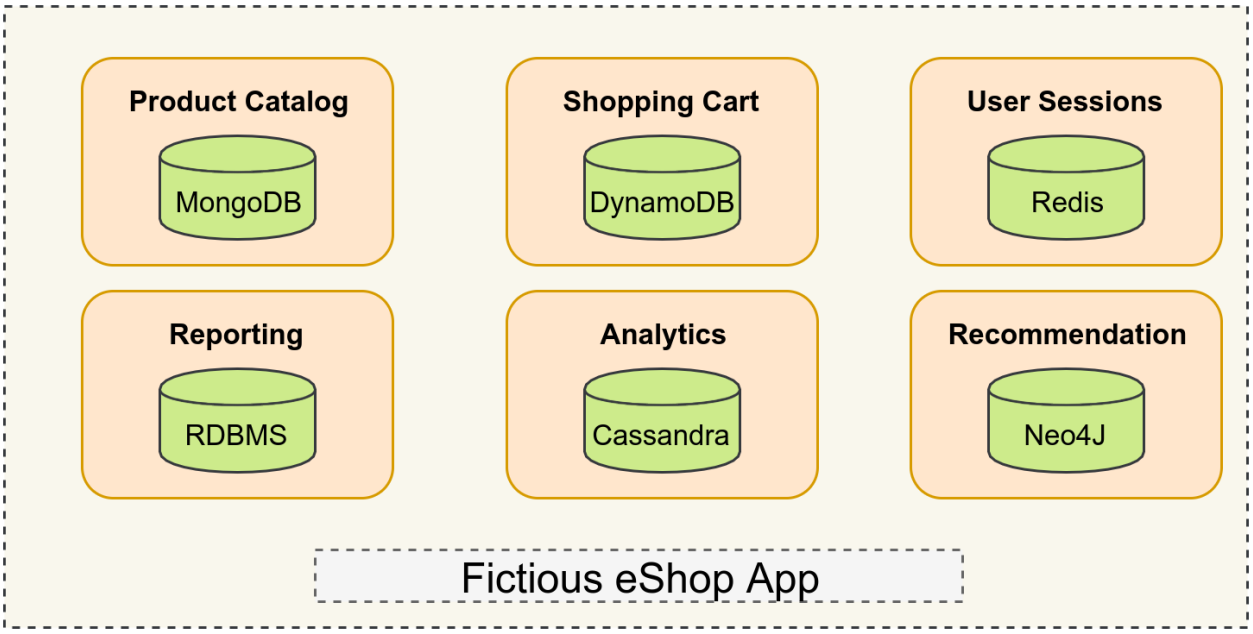
Key-value pair based database (User activity tracking, Analytics, etc.). DynamoDB can store documents as well as key-value pairs.

Redis

In memory distributed database (user session tracking), its mostly used as distributed cache among multiple microservices.

Neo4j

Graph DB (social connections, recommendations, etc)



Polyglot persistence in fictitious eShop Application

Benefits of *Polyglot Persistence* are manifold and can be harvested in both monolithic as well as microservices architecture. Any decent sized product will have variety of needs which may not be fulfilled by single kind of database alone. For example, if there are no transactional needs for a particular microservice, then its way better to use key-value pair or document oriented NoSql rather than using a transactional RDBMS database.

The Guardian Case Study

Case Study of "The Guardian" - Moving from Mysql to MongoDB - How it improved their performance.

https://www.mongodb.com/customers/guardian?c=2b0a518bc6

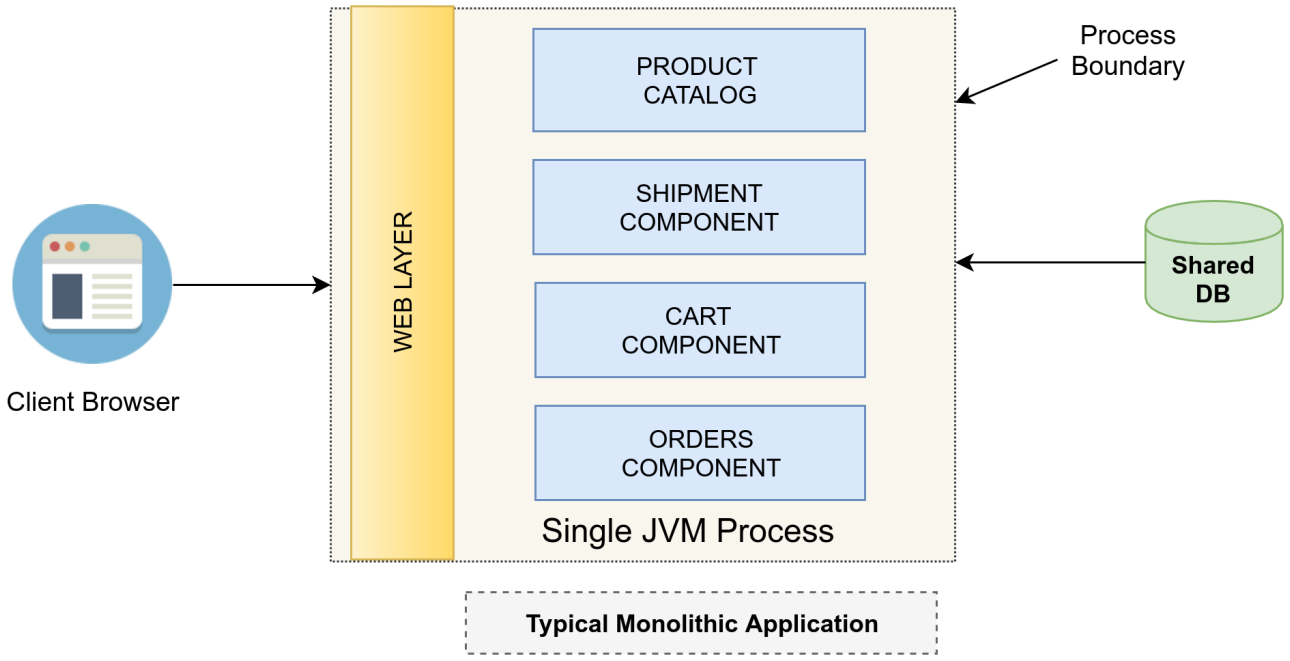
References

https://martinfowler.com/bliki/PolyglotPersistence.html

3.5. Why Microservices are better than Monoliths?

Microservices architecture is meant for developing large distributed systems that scale with safely. There are many benefits of microservices architecture over monoliths, for example:

1. Monolith application is built as a single unit, it is usually composed of 3 components -
    - a database (usually a RDBMS),
    - a server side executable (war file deployed in tomcat, websphere, etc)
    - and a client interface (JSP, etc.)
2. Whenever we want to add/update functionality, developers need to change at-least one of these three components and deploy the new version to production. The entire system is tightly coupled, have limitations in choosing technology stack, have low cohesion.
3. When we need to scale a monolith, we deploy the same version of monolith on multiple machines, by copying the big war/ear file again and again. Everything is contained into single executable file.
4. Microservices Architecture on the other hand is composed of small autonomous services, divided over business capabilities that communicate with each other over network mostly in async fashion.



Typical monolithic e-shop

As illustrated in above example, a typical monolith eShop application is usually a big war file deployed in a single JVM process (tomcat/jboss/websphere, etc). Different components of a monolith communicate with each other using inprocess communication like direct method invocation. One or more databases are shared among different components of a monolith application.

3.6. Isn’t in process communication in monolithic application faster than tons of remote network calls in microservices architecture?

Network communication brings some sort of latency to api calls, but that’s not a hurdle for developing a scalable distributed system anymore. VPC (Virtual Private Cloud) can now have in-house Optical Fibre network with speed of petabyte per second supporting millions of API calls per second between two microservices.

The advantages we get after adopting microservices architecture are manifold and they ought-weigh the overhead caused due to latency of network calls. In nutshell, a monolith application may be faster in the beginning, but microservices architecture is highly scalable and its performance does not drop with scalability.

3.7. How microservices are different than SOA?

Service Oriented Architecture gained its popularity in early 2000 due to its distributed system design approach that allowed building a highly scalable system. in SOA, different services deployed as separate processes collaborate with each other over the network to provide a set of business capabilities.

However, there was lack of consensus on how to do SOA well. Problems with SOA revolve around communication protocol used for inter-service communication (which is SOAP mostly), vendor middleware lockin (Enterprise Service Bus) and there was no clear guidance on strategy to partition system into fleet of services. SOA does not help you understand how to partition your system correctly into multiple microservices, it does not give emphasis to bounded context (and low coupling high cohesiveness principle).

Microservice architecture has emerged from real world use of SOA, taking our understanding of system and architecture of doing SOA well.

In nutshell, *microservices architecture* is SOA without vendor middleware, using simple communication protocol (mostly REST over HTTP) and with clear guidance on how to divide a big system into microservices based on business capabilities (bounded context).

You can consider microservices architecture as a specific approach for SOA rather than competitor of it, the same way as scrum is specific approach for Agile Software Development.

> SOA and Microservices are movements rather than Technology. These are styles of architecting a scalable application.
>
> — Martin Fowler

3.8. What is difference between small-services and microservices?

Microservices are usually small but not all small services are microservices. If any service is not following Bounded Context Principle, Single Responsibility Principle, etc. then it is not a microservice irrespective of its size. So the size is not the only eligibility criteria for a service to become microservice.

In fact size of a microservice is largely dependent on the language (Java, Scala, PHP) you choose, as few languages are more verbose than others.

3.9. What are benefits of using microservices architecture?

Embracing microservices architecture brings many benefits compared to using monolith architecture in your application, including:

Autonomous Deployments

The decentralized teams working on individual microservices are mostly independent from each other, so changing a service does not require coordination with other teams. This can leads to significantly faster release cycles. It is very hard to achieve the same thing in a realistic monolithic application where a small change may require regression of the entire system.

Culture Shift

Microservices style of system architecture emphasizes on culture of freedom, single responsibility, autonomy of teams, faster release iterations and technology diversification.

Technology Diversification

Unlike in monolithic applications, microservices are not bound to one technology stack (Java, .Net, Go, Erlang, Python, etc). Each team is free to choose a technology stack that is best suited for its requirements. For example, we are free to choose Java for a microservice, c++ for other and Go for another one.

DevOps Culture

The term comes from an abbreviated compound of "development" and "operations". It is a culture that emphasizes effective communication and collaboration between product management, software development and operations team. DevOps culture, if implemented correctly can lead to shorter development cycles and thus faster time to market.

3.10. How to partition a large application into microservices architecture, correctly?

Microservices should be autonomous and divided based on business capabilities. Each software component should have single well defined responsibility (a.k.a Single Responsibility Principle) and principle of Bounded Context (as defined by Domain Driven Design) should be used to create highly cohesive software components.

Partitioning microservice by business capabilities

For example, an e-shop can be partitioned into following microservices based on its business capabilities:

Product catalogue

responsible for product information, searching products, filtering products & products facets

Inventory

responsible for managing inventory of products (stock/quantity and facet)

Product review and feedback

collecting feedback from users about the products

Orders

responsible for creating and managing orders

Payments

process payments both online and offline (Cash On Delivery)

Shipments

Manage and track shipments against orders

Demand generation

market products to relevant users

User Accounts

manage users and their preferences

Recommendations

suggest new products based on user's preference or past purchases

Notifications

Email and SMS notification about orders, payments and shipments

The client application (browser, mobile app) will interact with these services via API gateway and render the relevant information to user.

How Amazon website works?

When you open Amazon website, the api-gateway may be interacting with 100s of microservices to just render a page that is most relevant for you. other interesting facts about Amazon as of 2015:

- Fifty million deployments per year
- Mostly black box between services
- Run by two-pizza teams (team size limited to two pizzas for a team meal)
- A team owns one or more microservices; no microservice should be bigger than what a two-pizza team can run
- Teams control their own destiny (product planning, Dev, Ops, and QA)
- Teams exist as part of larger orgs, such as Amazon.com (retail) or Prime, for example
- Built tooling as a set of microservices
- Mostly no coordination between teams when deploying services

Reference

https://apigee.com/about/blog/developer/microservices-amazon

3.11. How big a single microservice should be?

> A good, albeit non-specific, rule of thumb is as small as possible but as big as necessary to represent the domain concept they own.
>
> — Martin Fowler

Size should not be a determining factor in microservices, **instead bounded context principle and single responsibility principle should be used to isolate a business capability into single microservice boundary.** Few languages are more verbose than others, so lines of code required to implement a given functionality may vary from language to language.

3.12. 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 AQMP, JMS, Kafka, Protobuf or Thrift.

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 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. AQMP (like RabbitMQ) or Kafka can be used for asynchronous communication across microservices to achieve eventual consistency.

3.13. What shall be preferred communication style in microservices: synchronous or asynchronous?

Synchronous communication involves blocking call where the calling thread is blocked from doing anything else till the response is returned. Asynchronous communication, on the other hand is essentially event based, where the calling thread does not need to wait for the immediate response. Thus higher throughput can be achieved using asynchronous communication on the same hardware.

Synchronous communication between two services make them tightly coupled, and thus should be generally avoided. It also indicates a smell in your overall microservice partitioning. If the bounded context is clearly defined, such need should be among the rarest. However, this advise does not apply to API Gateway Aggregation Pattern where we aggregate the response from multiple microservices so as to avoid the network latency associated with round trips from mobile network.

There are other problems with synchronous communication - that the networks are not reliable. So client has posted a response that needs to be stored in multiple microservices, then data might become inconsistent if synchronous communication failure occurs.

Final advise

1. You must use asynchronous communication while handling HTTP POST/PUT (anything that modifies the data) requests, using some reliable queue mechanism (RabbitMQ, AQMP etc.)
2. Its 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 Queue. Events contains data about the domain, it should not tell what to do (action) on this data.
3. 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.

3.14. What is difference between Orchestration and Choreography in microservices context?

In Orchestration, we rely on a central system to control and call other Microservices in certain fashion to complete a given task. The central system maintains the state of each step and sequence of 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 microservices architecture. Whereas, Choreography's loose coupling approach should be adopted where-ever possible.

Practical example

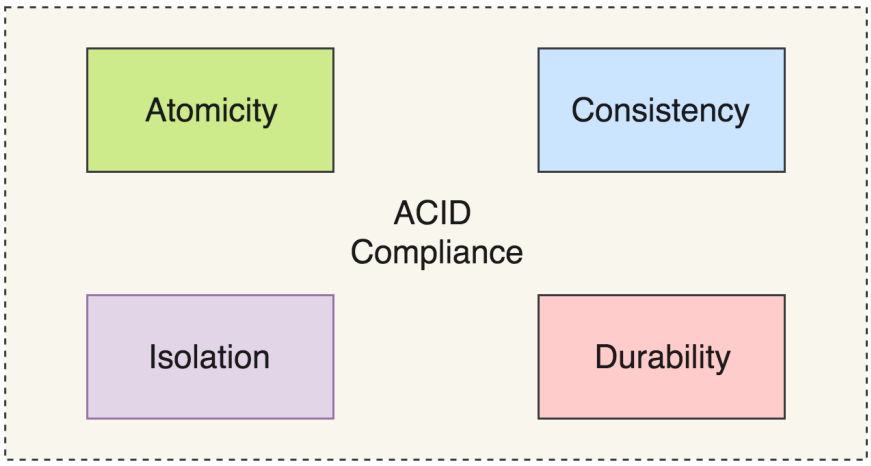
Let's say we want to develop a microservice that will send product recommendation email in a fictitious e-shop. In order to send Recommendations, we need to have access to user's order history which lies in a different microservices.

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 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 loosely coupled approach and highly scalable. The event in this case does not tell about the action, but just the data.

3.15. How to maintain ACID in microservice architecture?

ACID is an acronym for four primary attributes namely atomicity, consistency, isolation, and durability ensured by database transaction manager.



ACID compliance

Atomicity

In a transaction involving two or more entities, either all of the records are committed or none are.

Consistency

A database transaction must change affected data only in allowed ways following specific rules including constraints/triggers etc.

Isolation

Any transaction in progress (not yet committed) must remain isolated from any other transaction.

Durability

Committed records are saved by database such that even in case of a failure or database restart, the data is available in its correct state.

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

1. 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.
2. 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.

3.16. How frequent a microservice be released into production?

There is no definitive answer to this question, there could be a release every ten minutes, every hour or once a week. It all depends on the extent of automation you have at different level of software development lifecycle - build automation, test automation, deployment automation and monitoring. And of course on the business requirements - how small low-risk changes you care making in a single release.

In an ideal world where boundaries of each microservices are clearly defined (bounded context), and a given service is not affecting other microservices, you can easily achieve multiple deployments a day without major complexity.

Examples of deployment/release frequency

1. Amazon is on record as making changes to production every 11.6 seconds on average in May of 2011.

   https://www.thoughtworks.com/insights/blog/case-continuous-delivery

2. Github is well known for its aggressive engineering practices, deploying code into production on an average 60 times a day.

   https://githubengineering.com/move-fast/

3. Facebook releases to production twice a day.
4. Many Google services see releases multiple times a week, and almost everything in Google is developed on mainline.
5. Etsy Deploys More Than 50 Times a Day.

   https://www.infoq.com/news/2014/03/etsy-deploy-50-times-a-day

Right practices can make you deliver faster

1. Each microservices must be autonomous with least possible dependency on other services. Principles like Bounded-Context, Single Responsibility Principle, async communication using messages can help here.
2. Good level of automation at all levels of software development, so that a single commit in git repository can trigger the build, run automated testcases, deploy builds automatically.
3. Each change must contain small set of low-risk business requirements.

3.17. How to achieve zero-downtime during the deployments?

As the name suggests, zero-downtime deployments do not bring outage in production environment. It is a clever way of deploying your changes to production where at any given point in time, atleast one service will remain available to customers.

Blue-green deployment

One way of achieving this is **blue/green deployment**. In this approach, two versions of a single microservice are deployed at a time. But only one version is taking real requests. Once the newer version is tested to the required satisfaction level, you can switch from older version to newer version.

You can run a smoke-test suite to verify that the functionality is running correctly in the newly deployed version. Based on the results of smoke-test, newer version can be released to become the live version.

Changes required in client code to handle zero-downtime

Lets say you have two instances of a service running at same time, and both are registered in Eureka registry. Further, both the instances are deployed using two distinct hostnames:

Listing 3. /src/main/resources/application.yml

```
spring.application.name: books-service

---
spring.profiles: blue
eureka.instance.hostname: books-service-blue.example.com

---
spring.profiles: green
eureka.instance.hostname: books-service-green.example.com
```

Now the client app that needs to make api calls to `books-service` may look like below:

Listing 4. /sample/ClientApp.java

```
@RestController
@SpringBootApplication
@EnableDiscoveryClient
public class ClientApp {

    @Bean
    @LoadBalanced
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }

    @RequestMapping("/hit-some-api")
    public Object hitSomeApi() {
        return restTemplate().getForObject("https://books-service/some-uri", Object.class);  ①
    }
```

① We are calling the `books-service` using LoadBalanced RestTemplate.

Now, when `books-service-green.example.com` goes down for upgrade, it gracefully shuts down and delete its entry from Eureka registry. But these changes will not be reflected in the ClientApp until it fetches the registry again (which happens every 30 seconds). So for upto 30 seconds, ClientApp's `@LoadBalanced` RestTemplate may send the requests to `books-service-green.example.com` even if its down.

To fix this, we can use Spring Retry support in Ribbon client-side load balancer. To enable Spring Retry, we need to follow the below steps:

Listing 5. Add spring-retry to build.gradle dependencies

```
compile("org.springframework.boot:spring-boot-starter-aop")
compile("org.springframework.retry:spring-retry")
```

Now enable `spring-retry` mechanism in ClientApp using `@EnableRetry` annotation, as shown below:

```
@EnableRetry
@RestController
@SpringBootApplication
@EnableDiscoveryClient
public class ClientApp {

...
}
```

Once this is done, Ribbon will automatically configure itself to use retry logic and any failed request to `books-service-green.example.com` will be retried to next available instance (in round-robins fashion) by Ribbon. You can customize this behavior using the below properties:

Listing 6. /src/main/resources/application.yml

```
ribbon:
 MaxAutoRetries: 5
  MaxAutoRetriesNextServer: 5
  OkToRetryOnAllOperations: true
  OkToRetryOnAllErrors: true
```

Two Service Instances on a Single Host

If you have a single host machine and want to enable blue-green deployment strategy, then you need to start microservice at a dynamically available random port (in a predefined range). So two versions can work parallelly without causing port conflicts.

Smiliar discussions

1. https://github.com/spring-cloud/spring-cloud-netflix/issues/1290
2. https://www.quora.com/What-is-the-role-played-by-Netflix-Eureka-in-blue-green-deployment-done-on-Pivotal-Cloud-Foundry

3.18. How to slowly move users from older version of application to newer version?

If you have two different versions of a single microservice, you can achieve slow migration by maintaining both version of the application in production and then using API gateway to forward only selected endpoints to newly developed microservice.

This technique is also known as Canary Releasing. In Canary Releasing two versions coexist for a much longer time period compared to blue/green deployment.

3.19. How will you monitor fleet of microservices in production?

Monitoring is an essential aspect when it comes to any production deployment. In Microservices Architecture, we have fleet of services deployed on different or same host machine, so it becomes absolutely necessary to keep track of health of each individual service and take proper action in case of a failure. There are various tools that can help us here -

1. Graphite is an open source database that captures metrics from services in a time series database. By using Graphite we can capture essential metrics of multiple Microservices simultaneously. And we can use a dashboard like- Grafana to monitor these metrics in almost real time fashion.

   Graphite is built to handle large amount of data. It can record and show trends from few hours to few months within a few seconds. Many organizations like- Facebook etc use Graphite for monitoring. More information can be found here-

https://graphiteapp.org/

2. Spring Boot Admin is an open source project provided by Spring Boot Team. Spring Boot provides a basic UI for monitoring Health of service, logs/hystrix stats, JVM & memory Metrics, Datasource Metrics, Thread dump etc. More information can be found here

   https://github.com/codecentric/spring-boot-admin

3. Some of the other tools to monitor Microservices are: AppDynamics, DynaTrace, etc. These are paid tools with some premium features.

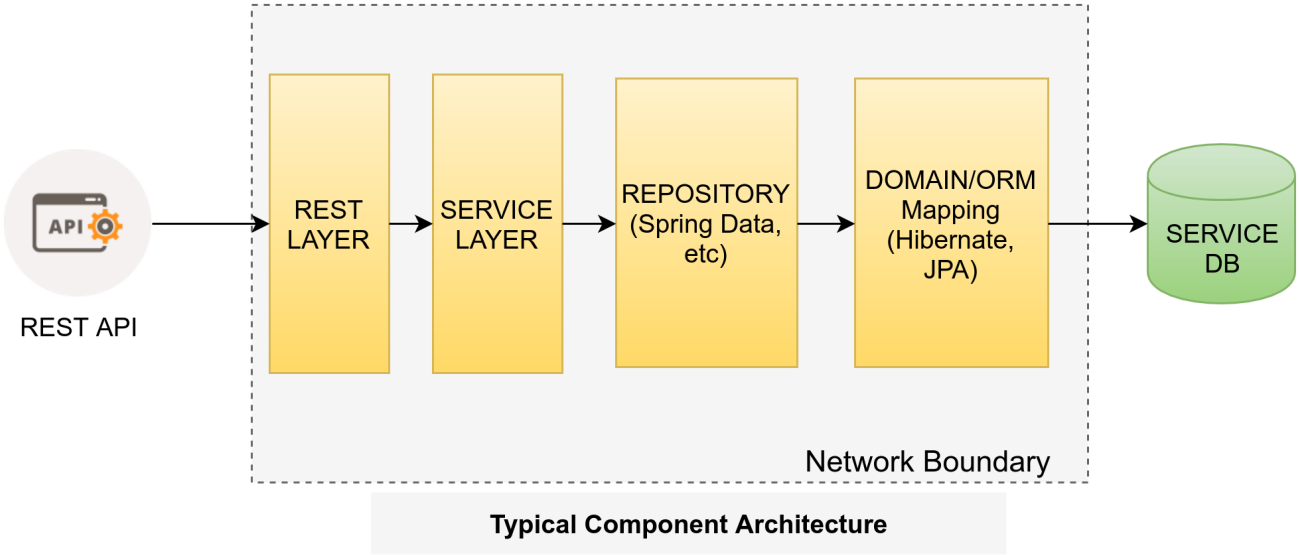
3.20. How will you troubleshoot a failed API request that is spread across multiple services?

In distributed systems its really hard to find the place where service actually failed first. The cascading effects of failure will lead to error in upstream services. We can use Correlation IDs in microservices that are unique for each request and gets injected at the API gateway level. So if failure occurs, we can aggregate the logs for a single request and figure out the exact place where it failed first.

These are unique IDs like GUID that can be passed from one service to another service during an API call. By using a GUID and failure message associated with it we can find the source of failure as well as the impact of failure.

3.21. What are different layers of a single microservice?

Like any typical Java application, a microservice has layered architecture. Most common layers in any microservice are:



Different Layers in a Single Microservice

From layering perspective, any microservice is no different than any other 3-tier architecture application.

1. Resource Layer (Rest Endpoint) - this layer is exposed to external world over HTTP.
2. Service Layer - this layer contains the business logic
3. Domain and Repository (Spring Data)
4. ORM/Data Mapper (Hibernate, JPA) - contains mostly POJO that map a Java Object with database table(RDBMS) or collection(NoSql).

3.22. How will you develop microservices using Java?

Spring Boot along with Spring Cloud is a very good option to start building microservices using Java language. There are lot of modules available in Spring Cloud that can provide boiler plate code for different design patterns of microservices, so Spring Cloud can really speedup the development process. Also Spring boot provides out of the box support to embed a servlet container (tomcat/jetty/undertow) inside a self contained executable jar (uber jar), so that these jars can be run directly from the command line, eliminating the need of deploying war files into a servlet container.

You can also use Docker container to ship and deploy the entire executable package onto a cloud environment. Docker can also help eliminate "works on my machine" problem by providing logical separation for the runtime environment during the development phase. That way you can gain portability across on premises and cloud environment.

3.23. Is it a good practice to deploy multiple microservices in a single tomcat container (servlet container)?

As stated in 12 factor app, we should deploy each microservice in a separate process to allow independent scaling. Therefore, it is best to run each microservice in its own servlet container (Tomcat, Jetty, Undertow, etc.).

Define port as 0 in config file will make your application take a random port. The actual server port can be captured inside a variable using `${local.server.port}`

In real cloud environment, Eureka client discovery can be used to locate the host and port number dynamically, thereby eliminating the need for knowing the host and port in advance.

Spring Boot provides us with options to use Tomcat, Jetty or Undertow servlet container with just few lines of code in `build.gradle` file.

Listing 7. build.gradle

```
dependencies {
    compile('org.springframework.boot:spring-boot-starter-web') {
        exclude module: "spring-boot-starter-tomcat"    ①
    }
    compile("org.springframework.boot:spring-boot-starter-undertow")
    //...other dependencies
}
```

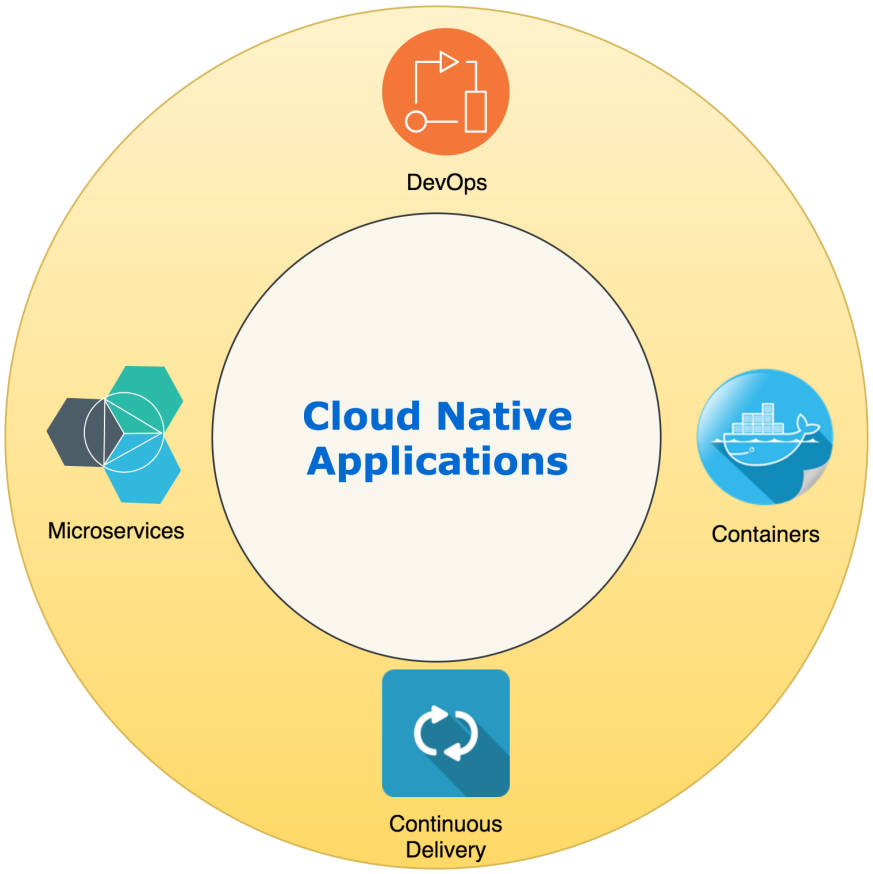
① In above example we have excluded the default tomcat and opted for undertow as the embedded servlet container for Spring Boot Application.

3.24. What are Cloud Native applications?

Cloud Native Applications (NCA) is a style of application development that encourages easy adoption of best practices in the area of continuous delivery and distributed software development. These applications are designed specifically for a cloud computing architecture (AWS, Azure, CloudFoundary, etc).

DevOps, continuous delivery, microservices and containers are the key concepts in developing cloud native applications.

Spring Boot, Spring Cloud, Docker, Jenkins, Git are few tools that can help you write Cloud Native Application without much effort.



Cloud Native Applications

Microservices

It is an architectural approach for developing a distributed system as a collection of small services. Each service is responsible for a specific business capability, runs in its own process and communicates via HTTP REST API or messaging (AQMP).

DevOps

It is collaboration between software developers and IT operations with a goal of constantly delivering high quality software as per customer needs.

Continuous Delivery

Its all about automated delivery of low-risk small changes to production, constantly. This makes

it possible to collect feedback faster.

Containers

Containers (e.g. Docker) offer logical isolation to each microservices thereby eliminating the problem of "run on my machine" forever. Its much faster and efficient compared to Virtual Machines.

Cloud-native is about how applications are created and deployed, not where.

References

https://pivotal.io/cloud-native

3.25. What is Spring Boot?

***Spring Boot makes it easy to create stand-alone, production grade Spring** based applications that you can "just run" with an opinionated view of the Spring platform and third-party libraries so you can get started with minimum fuss.*

Main features of Spring Boot

1. Create stand-alone Spring applications (12 factor app style)
2. Embed Tomcat, Jetty or Undertow directly (no need to deploy WAR files)
3. Provide opinionated `starter` POMs to simplify your Maven or Gradle configuration
4. Automatically configure Spring whenever possible
5. Provide production-ready features such as metrics, health checks and externalized configuration
6. Absolutely no code generation and no requirement for XML configuration

Current version of Spring Boot as of writing this document is 1.5.10

You can create a Spring Boot starter project by selecting the required dependencies for you project using online tool hosted at https://start.spring.io/

Bare minimum dependency for any spring boot application is:

```
 dependencies {
    compile("org.springframework.boot:spring-boot-starter-web:1.5.10.RELEASE")
}
```

The main java class for Spring Boot application will look something like the following:

Listing 8. SampleApplication

```
import org.springframework.boot.*;
import org.springframework.boot.autoconfigure.*;
import org.springframework.stereotype.*;
import org.springframework.web.bind.annotation.*;

@Controller
@EnableAutoConfiguration
public class SampleController {

    @RequestMapping("/")
    @ResponseBody
    String home() {
        return "Hello World!";
    }

    public static void main(String[] args) throws Exception {
        SpringApplication.run(SampleController.class, args);
    }
}
```

You can directly run this class, without deploying it to a servlet container.

Useful References

- Spring Boot Project
- Spring Boot Starter
- Building an Application with Spring Boot
- Spring Boot Documentation

3.26. What is Spring Cloud?

Spring Cloud empowers Java developers with libraries & tools to quickly build some of the common design patterns of distributed system, including the following:

1. Distributed/versioned configuration (Config Server)
2. Service Registration & Discovery (using Consul or Eureka Server)
3. Circuit Breaker (using Hystrix Library)
4. Distributed Sessions (OAuth2 and Redis)
5. Intelligent Routing
6. API Gateway Pattern
7. Service to Service Calls (RestTemplate and Feign)
8. Client side load balancing (Ribbon)
9. Distributed messaging (AQMP, Kafka)

Spring Cloud makes it easy to develop, deploy and operate JVM applications for the Cloud.

Different release trains in Spring Cloud at the time of writing this handbook are (newest to oldest) - Finchley, Edgware, Dalston and Camden. Spring Cloud is always used in conjunction with Spring Boot.

A bare minimum `build.gradle` for any Spring Cloud project will look like:

Listing 9. build.gradle

```
buildscript {
    ext {
        springBootVersion = '1.5.10.RELEASE'
    }
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-plugin:${springBootVersion}")
    }
}

apply plugin: 'java'
apply plugin: 'spring-boot'

dependencyManagement {
  imports {
    mavenBom ':spring-cloud-dependencies:Edgware.SR2'   ①
  }
}

dependencies {
    compile ':spring-cloud-starter-config'  ②
    compile ':spring-cloud-starter-eureka'
}
```

① `Edgware.SR2` is the `spring-cloud` train version.

② Spring cloud dependencies (eureka client and config client)

And a minimal version of `spring-cloud` Application:

```
@SpringBootApplication  ①
@EnableDiscoveryClient  ②
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

① Enables `spring-boot` in your application.

② Enables `discovery-client`: a `spring-cloud` feature in your microservice.

Reference

http://projects.spring.io/spring-cloud/

3.27. What is difference between application.yml and bootstrap.yml?

application.yml

`application.yml`/`application.properties` file is specific to Spring Boot applications. Unless you change the location of external properties of an application, spring boot will always load `application.yml` from following location:

```
/src/main/resources/application.yml
```

You can store all the external properties for you application in this file. Common properties that are available in any Spring Boot project can be found at: https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html You can customize these properties as per your application needs. Sample file is shown below:

Listing 10. /src/main/resources/application.yml

```
spring:
    application:
        name: foobar
    datasource:
        driverClassName: com.mysql.jdbc.Driver
        url: jdbc:mysql://localhost/test
server:
    port: 9000
```

Reference

https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-external-config.html

bootstrap.yml

`bootstrap.yml` on the other hand is specific to `spring-cloud-config` and is loaded before the `application.yml`

`bootstrap.yml` is only needed if you are using Spring Cloud and your microservice configuration is stored on a remote Spring Cloud Config Server.

Important points about bootstrap.yml

1. When used with Spring Cloud Config server, you shall specify the application-name and config git location using below properties

Listing 11. /src/main/resources/bootstrap.yml

```
spring.application.name: <application-name>
spring.cloud.config.server.git.uri: <git-uri-config>
```

2. When used with microservices (other than cloud config server), we need to specify the application name and location of config server using below properties

Listing 12. /src/main/resources/bootstrap.yml

```
spring.application.name: <application-name>
spring.cloud.config.uri: <http://localhost:8888>
```

3. This properties file can contain other configuration relevant to Spring Cloud environment for e.g. eureka server location, encryption/decryption related properties.

Internal working

Upon startup, Spring Cloud makes an HTTP(S) call to the Spring Cloud Config Server with the name of the application and retrieves back that application's configuration.

`application.yml` contains the default configuration for the microservice and any configuration retrieved (from cloud config server) during the bootstrap process will override configuration defined in `application.yml`

As per official Spring Cloud Documentation

> A Spring Cloud application operates by creating a "bootstrap" context, which is a parent context for the main application. Out of the box it is responsible for loading configuration properties from the external sources, and also decrypting properties in the local external configuration files.
>
> http://cloud.spring.io/spring-cloud-config/single/spring-cloud-config.html

Using YAML instead of Properties

YAML is a superset of JSON format and it is a very convenient format for specifying hierarchical configuration data. To support yaml based configuration in your Spring Boot project, you just need to add the following dependency:

Listing 13. build.gradle

```
compile('org.springframework.boot:spring-boot-starter')
```

It is recommended to use YAML format over conventional properties format because yaml is compact & easy to read.

3.28. How will you implement service discovery in microservices architecture?

Servers come and go in a cloud environment, and new instances of same services can be deployed to cater increasing load of requests. So it becomes absolutely essential to have service registry & discovery that can queried for finding address (host, port & protocol) of a given server. We may also need to locate servers for the purpose of client side load balancing (Ribbon) and handling failover gracefully (Hystrix).

Spring Cloud solves this problem by providing few ready made solutions for this challenge. There are mainly two options available for the service discovery - Netflix Eureka Server and Consul. Lets discuss both of these briefly:

Netflix Eureka Server

Eureka is a REST (Representational State Transfer) based service that is primarily used in the AWS cloud for locating services for the purpose of load balancing and failover of middle-tier servers. The main features of of Netflix Eureka are:

1. It provides `service-registry`.
2. zone aware service lookup is possible.
3. `eureka-client` (used by microservices) can cache the registry locally for faster lookup. The client also has a built-in load balancer that does basic round-robin load balancing.

Spring Cloud provides two dependencies - `eureka-server` and `eureka-client`.

Eureka server dependency is only required in eureka server's `build.gradle`

Listing 14. build.gradle - Eureka Server

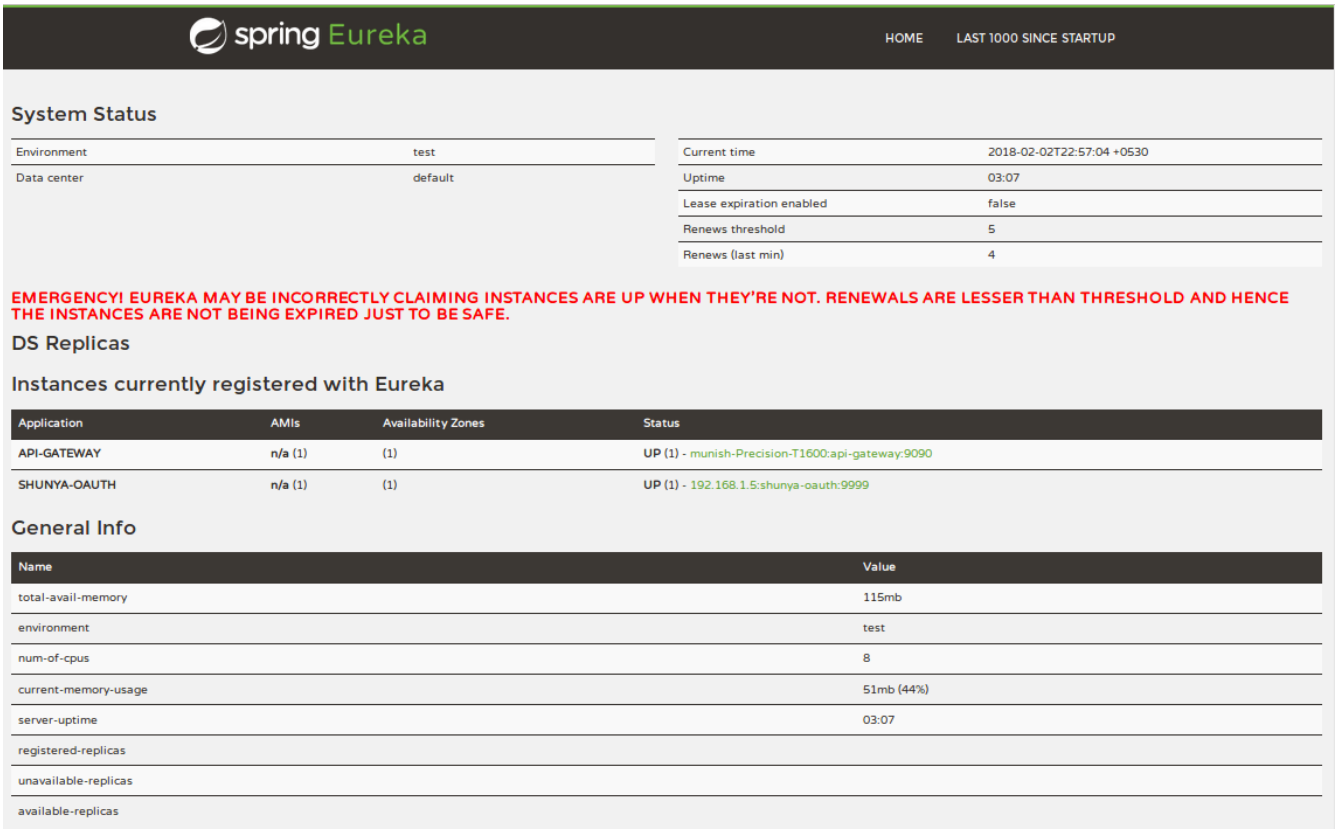
```
compile('org.springframework.cloud:spring-cloud-starter-netflix-eureka-server')
```

On the other hand, each microservice need to include the `eureka-client` dependencies to enables eureka discovery.

Listing 15. build.gradle - Eureka Client (to be included in all microservices)

```
compile('org.springframework.cloud:spring-cloud-starter-netflix-eureka-client')
```

Eureka server provides a basic dashboard for monitoring various instances and their health in service registry. The ui is written in `freemarker` and provided out of the box without any extra configuration. Screenshot for Eureka Server looks like the following.



Eureka Registry Screenshot

It contains list all services that are registered with Eureka Server. Each server has information like zone, host, port and protocol.

Consul Server

It is a REST based tool for dynamic service registry. It can be used for registering a new service, locating a service and health checkup of a service.

You have a option to choose any one of the above in your spring cloud based distributed application. In this book, we will focus more on the **Netflix Eureka Server** option.

3.29. How does Eureka Server work?

There are two main components in Eureka project: `eureka-server` and `eureka-client`.

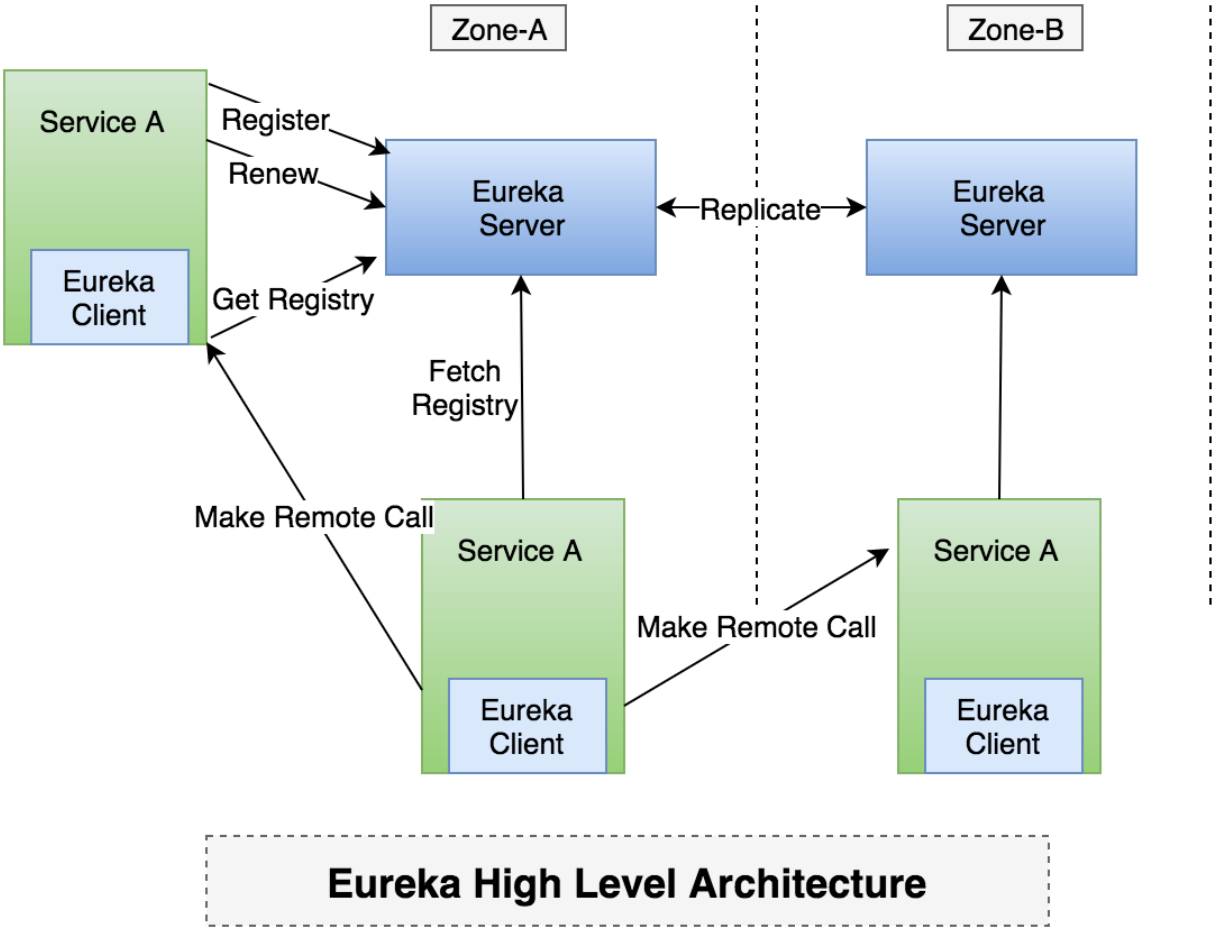
Eureka Server

The central server (one per zone) that acts as a service registry. All microservices register with this eureka server during app bootstrap.

Eureka Client

Eureka also comes with a Java-based client component, the `eureka-client`, which makes interactions with the service much easier. The client also has a built-in load balancer that does basic round-robin load balancing. Each microservice in the distributed ecosystem much include this client to communicate and register with `eureka-server`.

Typical usecase for Eureka



High Level Eureka Architecture

There is usually one eureka server cluster per region (us, asia, europe, australia) which knows only about instances in its region. Services register with Eureka and then send heartbeats to renew their leases every 30 seconds. If the service can not renew their lease for few times, it is taken out of server registry in about 90 seconds. The registration information and the renewals are replicated to all the eureka nodes in the cluster. The clients from any zone can look up the registry information (happens every 30 seconds) to locate their services (which could be in any zone) and make remote calls.

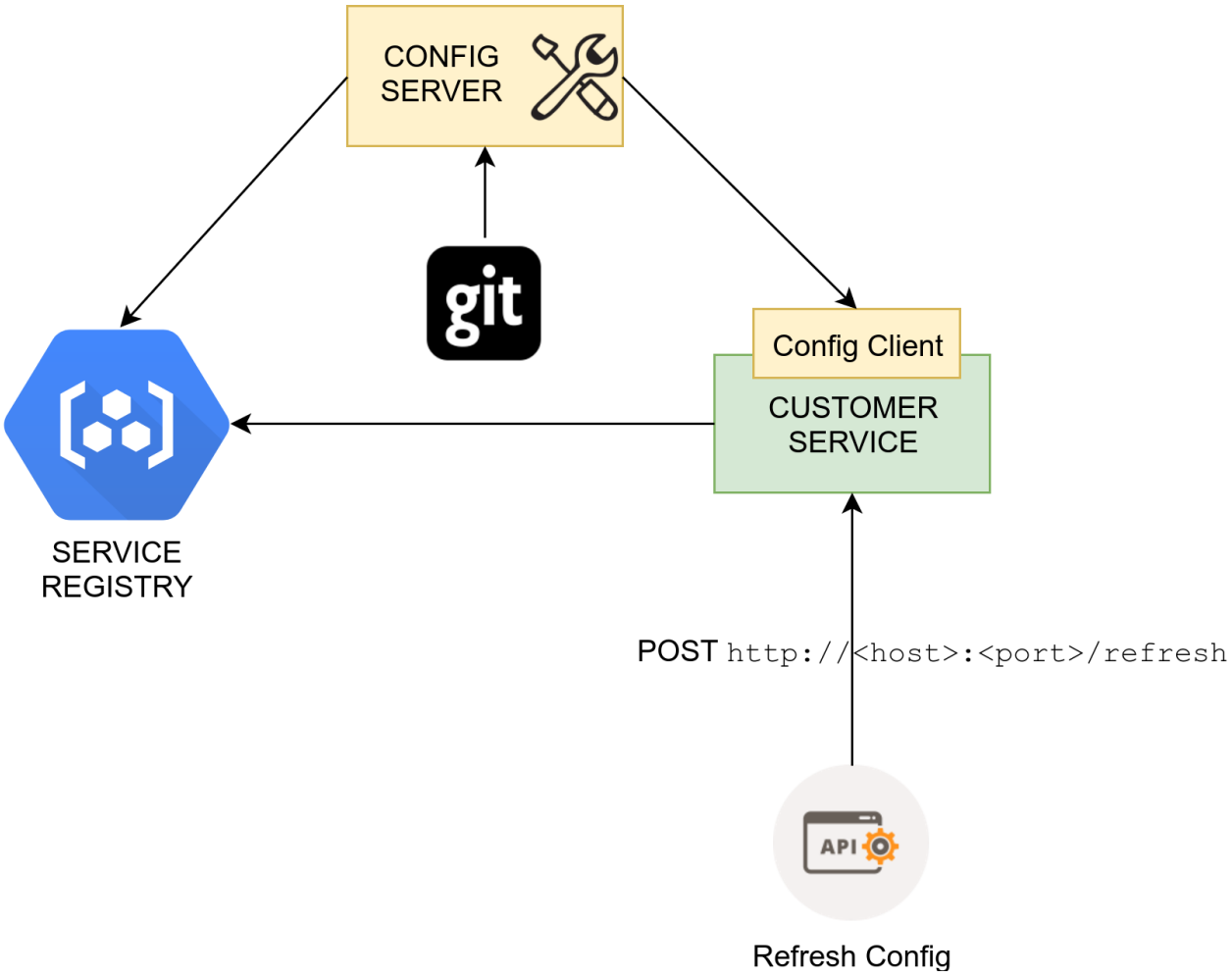
Eureka clients are built to handle the failure of one or more Eureka servers. Since Eureka clients have the registry cache information in them, they can operate reasonably well, even when all of the eureka servers go down.

More Information about eureka

https://github.com/Netflix/eureka/wiki/Eureka-at-a-glance

3.30. How to externalize configuration in a distributed system?

Spring Cloud Config provides server and client side support for externalized configuration in a distributed system. With a config server we have a central place to manage external properties for applications across all environments. The default implementation of the `config-server` storage uses git so it easily supports labelled versions of configuration environments.



Config Server Architecture

Key features of Spring Cloud config-server

1. HTTP, resource-based API for external configuration (YAML or properties syntax)
2. Encrypt and Decrypt property values - both symmetric and asymmetric encryption is supported
3. Very easy to create a standalone config server with use of `@EnableConfigServer`

Key features in Spring Cloud config-client

1. Encrypt and Decrypt property values
2. Bind to Config Server and Initialize Spring `Environment` with remote property sources.

3. `@RefreshScope` for Spring `@Beans` that want to be re-initialized when configuration changes.

3.31. How will you use config-server for your development, stage and production environment?

If you have 3 different environment (develop/stage/production) in your project setup, then you need to create three different config storage projects. So in total you will have four project:

config-server

It is the config-server that can de deployed in each environment. It is the Java Code without configuration storage.

config-dev

It is the git storage for your development configuration. All configuration related to each microservices in development environment will fetch its config from this storage. This project has no Java code, and t is meant to be used with config-server.

config-qa

same as config-dev but its meant to be used only in qa environment.

config-prod

same as config-dev, but meant for production environment.

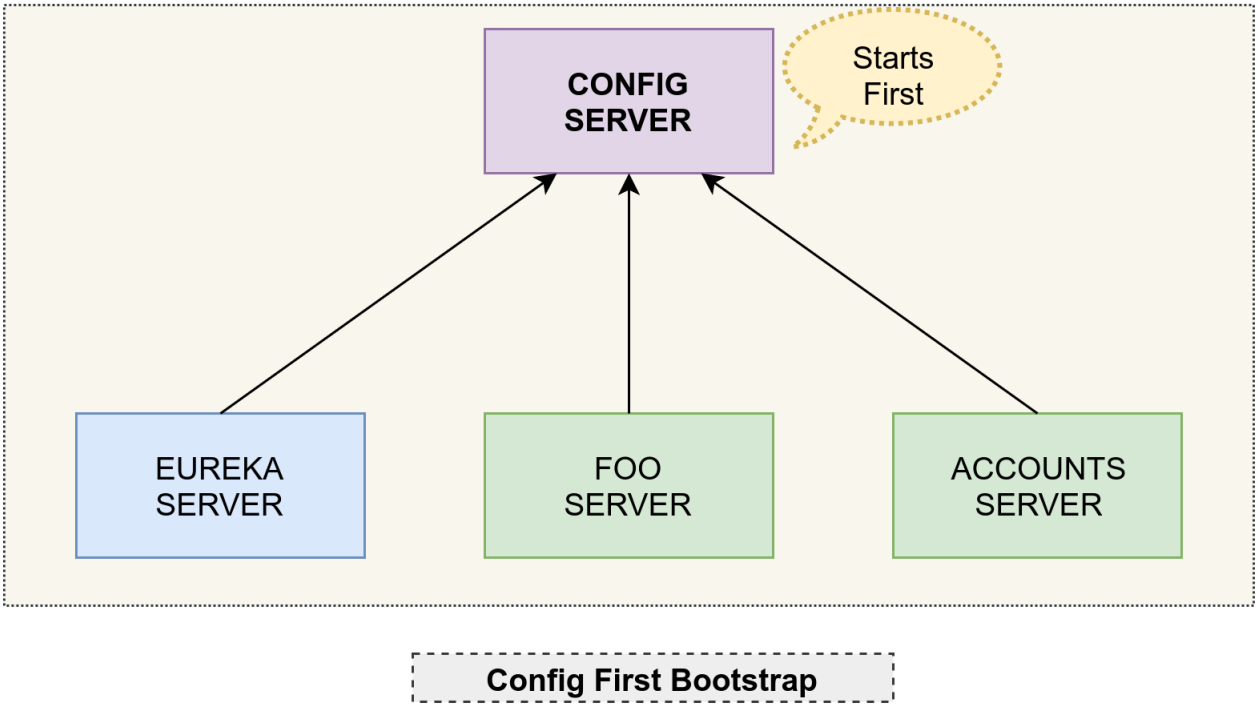
So depending upon the environment, we will use config-server with either `config-dev`, `config-qa` or `config-prod`.

3.32. What is difference between config first bootstrap and discovery first bootstrap in context of Spring Cloud Config client?

Config first bootstrap and *discovery first bootstrap* are two different approaches for using Spring Cloud Config client in Spring Cloud powered microservices. Let's discuss both of them:

Config First Bootstrap

This is the default behavior for any spring boot application where Spring Cloud Config client is on the classpath. When a config client starts up it binds to the Config Server using the bootstrap configuration property and initializes Spring `Environment` with remote property sources.



Config-first approach

The only configuration that each microservice (except `config-server`) needs to provide is the following:

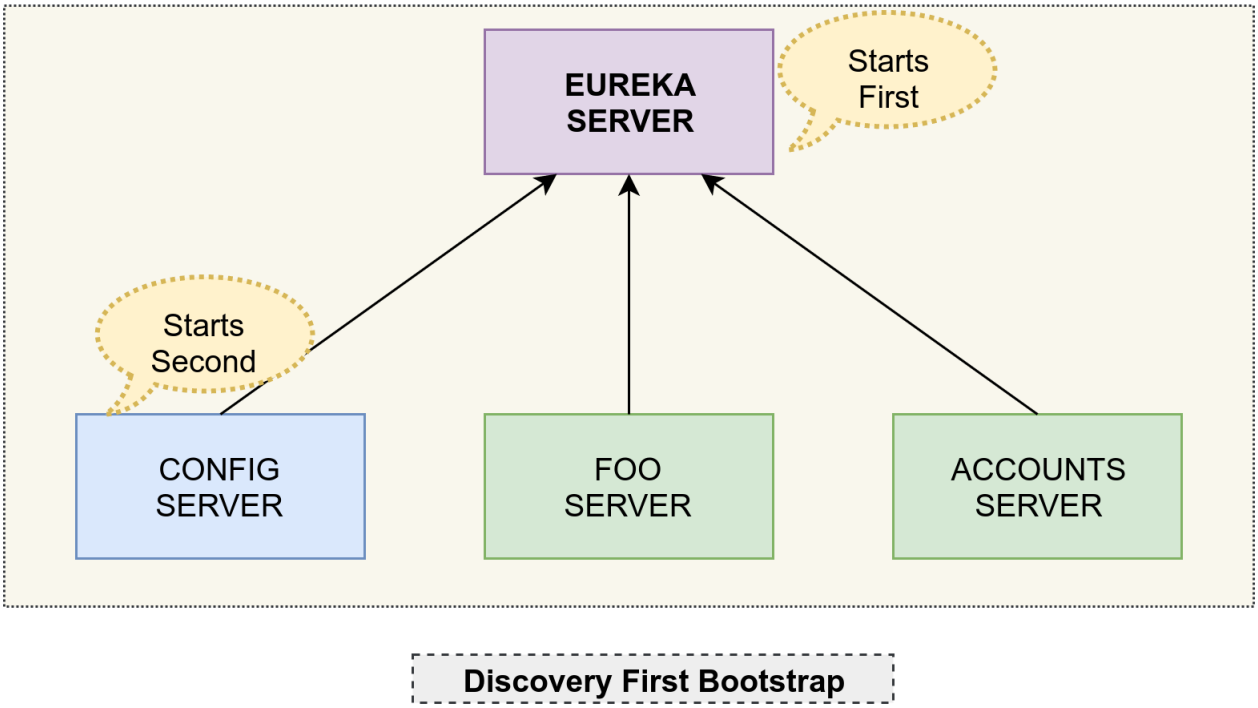
Listing 16. /src/main/resources/bootstrap.yml

```
spring.cloud.config.uri: http://localhost:8888
```

In config-first approach, even the `eureka-server` can fetch its own configuration from `config-server`. Point worth noting down here is that `config-server` must be the first service to boot up in the entire ecosystem, because each service will fetch its configuration from `config-server`.

Discovery First Bootstrap

If you are using Spring Cloud Netflix and Eureka Service Discovery then you can have Config Server register with the Discovery Service and let all clients get access to config server via discovery service.



Discovery-first approach

This is not the default configuration in Spring Cloud applications, so we need to manually enable it using the below property in bootstrap.yml

Listing 17. /src/main/resources/bootstrap.yml

```
spring:
    cloud:
        config:
            discovery:
                enabled: true
```

This property should be provided by all microservices so that they can take advantage of discovery first approach.

The **benefit** of this approach is that now `config-server` can change its host/port without other microservices knowing about it, since each microservice can get the configuration via eureka service now. The **downside** of this approach is that an extra network round trip is required to locate the service registration at app startup.

3.33. How to halt a Spring Boot based microservice at startup if it can not connect to Config Server during bootstrap?

If you want to halt the service when it is not able to locate the `config-server` during bootstrap, then you need to configure the following property in microservice's `bootstrap.yml`:

Listing 18. /src/main/resources/bootstrap.yml

```
spring:
    cloud:
        config:
            fail-fast: true
```

Using this configuration will make microservice startup fail with an exception when `config-server` is not reachable during bootstrap.

We can enable retry mechanism where microservice will retry 6 times before throwing an exception. We just need to add `spring-retry` and `spring-boot-starter-aop` to the classpath to enable this feature.

Listing 19. build.gradle

```
...

dependencies {
 compile('org.springframework.boot:spring-boot-starter-aop')
 compile('org.springframework.retry:spring-retry')
 ...
}
```

3.34. How to refresh configuration changes on the fly in Spring Cloud environment?

Using `config-server`, its possible to refresh the configuration on the fly. The configuration changes will only be picked by `Beans` that are declared with `@RefreshScope` annotation.

The following code illustrates the same. The property `message` is defined in the `config-server` and changes to this property can be made at runtime without restarting the microservices.

```
package hello;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.context.config.annotation.RefreshScope;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication
public class ConfigClientApplication {

    public static void main(String[] args) {
        SpringApplication.run(ConfigClientApplication.class, args);
    }
}

@RefreshScope   ①
@RestController
class MessageRestController {

    @Value("${message:Hello World}")
    private String message;

    @RequestMapping("/message")
    String getMessage() {
        return this.message;
    }
}
```

① `@RefreshScope` makes it possible to dynamically reload the configuration for this bean.

3.35. How to achieve client side load balancing in Spring Microservices using Spring Cloud?

Naive approach would be to get list of instances of a microservice using discovery client and then randomly pickup one instance and make the call. But that's not easy because that service instance may be down at the moment and we will have to retry the call again on next available instance.

Ribbon does that all internally. Ribbon will do at-least the following-

1. A central concept in Ribbon is that of the named client. more about named clients.
2. Load balance based on some algorithm - Round Robbin rule by default or WeightedResponseTimeRule, AvailabilityFilteringRule, etc can be chosen.
3. Make calls to next server if first one fails
4. Its easy to use hystrix with Ribbon and use circuit breaker for managing fault tolerance.
5. Establishing affinity between clients and servers by dividing them into zones (like racks in a data center) and favor servers in the same zone to reduce latency.
6. Keeping statistics of servers and avoid servers with high latency or frequent failures.
7. Keeping statistics of zones and avoid zones that might be in outage.

Under the hood, Ribbon Load Balancer uses the following 3 components-

- Rule - a logic component to determine which server to return from a list
- Ping - a component running in background to ensure liveness of servers
- ServerList - this can be static or dynamic. If it is dynamic (as used by DynamicServerListLoadBalancer), a background thread will refresh and filter the list at certain interval.

We shall always use Ribbon, a client side load balancing library to distribute the load among various instance of a microservice deployment.

1. Using Ribbon with RestTemplate.
2. Using Ribbon with Feign Client.

3.36. How to use client side load-balancer Ribbon in your microservices architecture?

Ribbon - a client side load-balancer can be used in Spring Cloud powered microservices to distribute the API call load among different instances of a single service. Following steps need to be taken:

Add required dependency in build.gradle

Listing 20. build.gradle

```
compile('org.springframework.cloud:spring-cloud-starter-ribbon')
```

Now we need to configure the application.yml for proper Ribbon settings.

Listing 21. /src/main/resources/application.yml

```
ribbon:
  ConnectTimeout: 60000
  ReadTimeout: 60000
  MaxAutoRetries: 1
  MaxAutoRetriesNextServer: 1
  OkToRetryOnAllOperations: true
  http:
    client:
      enabled: true
```

Create LoadBalanced RestTemplate Bean

Finally, you need to annotate `RestTemplate` bean with `LoadBalanced` annotation to enable Ribbon client side load-balancer.

```
    @LoadBalanced
    @Bean
    RestTemplate restTemplate(RestTemplateBuilder restTemplateBuilder) {
        return restTemplateBuilder.build();
    }
```

Now whatever calls you make using this `RestTemplate`, will be load balanced using default round-robbin fashion.

3.37. How to use both LoadBalanced as well as normal RestTemplate object in the single microservice?

We can create two different beans with different qualifiers, one with load balanced configuration another without ribbon load balancer.

Listing 22. Defining Normal and LoadBalanced RestTemplate with two different Bean names.

```
    @LoadBalanced
    @Bean(name = "loadBalancedRestTemplate")    ①
    RestTemplate restTemplate(RestTemplateBuilder restTemplateBuilder) {
        return restTemplateBuilder.build();
    }

    @Bean(name = "simpleRestTemplate")  ②
    public RestTemplate simpleRestTemplate(RestTemplateBuilder restTemplateBuilder) {
        RestTemplate restTemplate = restTemplateBuilder.build();
        return restTemplate;
    }
```

① Load Balanced RestTemplate bean

② Normal RestTemplate bean

Listing 23. Wiring Normal RestTemplate using @Qualifier

```
    @Autowired
    @Qualifier("simpleRestTemplate")
    private RestTemplate restTemplate;
```

3.38. How will you make use of Eureka for service discovery in Ribbon Load Balancer?

If both Ribbon and Eureka libraries are present on the classpath, Ribbon is automatically configured with Eureka i.e. Server list is populated from Eureka, Ping functionality is delegated to Eureka to check if a server is up. This approach is very convenient in any enterprise grade application.

3.39. Can we use Ribbon without eureka?

Yes, Ribbon can be used without eureka discovery. But, if classpath contains both eureka and ribbon, then Ribbon will be automatically configured with Eureka. Eureka in that case picks up server list from eureka server registry, and uses eureka to fetch the status of server instead of using ping.

Listing 24. application.yml

```
stores:
  ribbon:
    listOfServers: example.com,google.com,microsoft.com
```

We can also explicitly disable the use of Eureka in Ribbon using the following configuration.

Listing 25. application.yml

```
ribbon:
  eureka:
   enabled: false
```

Once this configuration is done, you can start using a `@LoadBalanced` resttemplate instance, in the following way:

Listing 26. Using Ribbon without eureka service registry

```
ResponseEntity<Book> responseEntity = restTemplate.exchange("http://stores/api/book?sku={param1}", HttpMethod.GET, null,
typeReference, uriParams);    ①
```

① `stores` is mapped to `example.com,google.com,microsoft.com` in `application.yml`

In the above code, api calls to stores will be load-balanced among three servers - `example.com,google.com,microsoft.com`

3.40. How will you use ribbon load balancer programmatically?

You can use Ribbon load-balancer client to fetch appropriate instance of a given microservice. Following example code demonstrate the same:

Listing 27. Ribbon Load Balancer (Manual Invocation)

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.ApplicationArguments;
import org.springframework.boot.ApplicationRunner;
import org.springframework.cloud.client.ServiceInstance;
import org.springframework.cloud.client.loadbalancer.LoadBalancerClient;
import org.springframework.stereotype.Component;

import java.net.URI;

@Component
public class RibbonLoadBalancer implements ApplicationRunner {

    @Autowired
    private LoadBalancerClient loadBalancer;

    public void doStuff() {
        ServiceInstance instance = loadBalancer.choose("product-service");
        URI storesUri = URI.create(String.format("http://%s:%s", instance.getHost(), instance.getPort()));
        // ... do something with the URI
        System.out.println("storesUri = " + storesUri);
        System.out.println("serviceId = " + instance.getServiceId());
    }

    @Override
    public void run(ApplicationArguments args) throws Exception {
        doStuff();
    }
}
```

`choose(<server-key>)` method of `LoadBalancerClient` will pick up next available server based on some criteria which could be as simple as round-robbin fashion or something complex like `ZoneAwareLoadBalancer` that takes care of zone stats of all available zones.

3.41. What is difference between @EnableEurekaClient and @EnableDiscoveryClient?

Both annotations are used to enable discovery configuration in Spring Cloud based microservice.

@EnableDiscoveryClient

`@EnableDiscoveryClient` is a generic annotation to enable a DiscoveryClient implementation. This will pick up appropriate implementation based on classpath dependencies. If eureka server is available on the class-path then it will enable Eureka discovery configuration, if consul library is present then it will enable configuration for consul discovery. Basically, EnableDiscoveryClient is

generic version that can be used with both Eureka and Consul.

@EnableEurekaClient

`@EnableEurekaClient` is a convenience annotation for clients to enable Eureka discovery configuration (specifically). This annotation is very much specific to Eureka based clients.

When using Eureka Server, you can use any of these convenient annotation to enable discovery configuration.

3.42. How to make microservices zone aware so as to prefer same zone services for inter-service communication using Spring Cloud?

If you have deployed Eureka clients to multiple zones than you may prefer that those clients leverage services within the same zone before trying services in another zone. To do this you need to configure your Eureka clients correctly.

First, you need to make sure you have Eureka servers deployed to each zone and that they are peers of each other. See the section on zones and regions for more information.

Next you need to tell Eureka which zone your service is in. You can do this using the metadataMap property. For example if service 1 is deployed to both zone 1 and zone 2 you would need to set the following Eureka properties in service 1

Listing 28. Service 1 in Zone 1

```
eureka.instance.metadataMap.zone: <zone1>
eureka.client.preferSameZoneEureka: true
```

Listing 29. Service 1 in Zone 2

```
eureka.instance.metadataMap.zone: <zone2>
eureka.client.preferSameZoneEureka: true
```

3.43. How to list all instances of a single microservice in Spring Cloud environment?

You can use `org.springframework.cloud.client.discovery.DiscoveryClient` which provides a simple API for discovering microservices registered in `service-registry`.

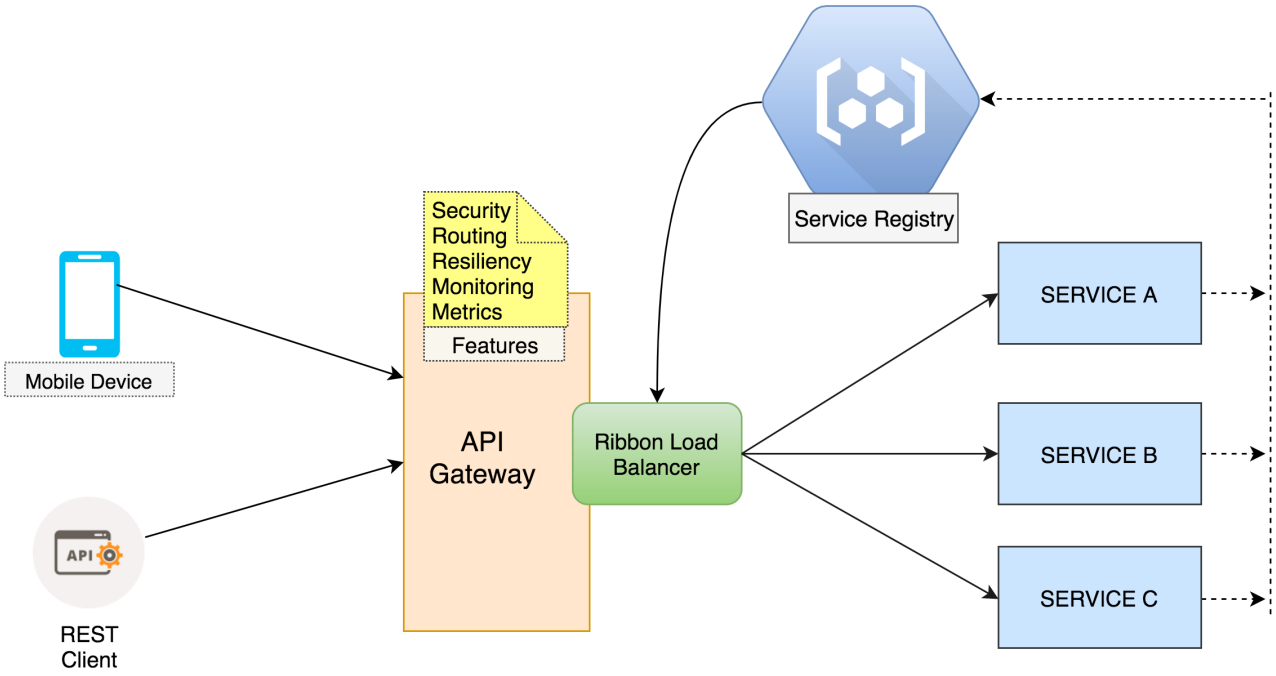
Listing 30. Discovering all instances of a given service

```
@Autowired
private DiscoveryClient discoveryClient;

public String serviceUrl() {
    List<ServiceInstance> list = discoveryClient.getInstances("STORES");
    if (list != null && list.size() > 0 ) {
        return list.get(0).getUri();
    }
    return null;
}
```

In the above code, `discoveryClient` will scan the service-registry for given server-key i.e. `STORES` and fetch all the instances registered with this key. `DiscoveryClient` may cache the results for performance reasons.

3.44. What is API Gateway?

API Gateway is a special class of microservices that meets the need of a single client application (such as android app, web app, angular JS app, iPhone app, etc) and provide it with single entry point to the backend resources (microservices), providing cross cutting concerns to them such as security, monitoring/metrics & resiliency.



API Gateway

Client Application can access tens or hundreds of microservices concurrently with each request , aggregating the response and transforming them to meet the client application's needs. Api Gateway can use a client side load balancer library (Ribbon) to distribute load across instances based on round robin fashion. It can also do protocol translation i.e. HTTP to AMQP if necessary. It can handle security for the protected resources as well.

Features of API Gateway

1. Spring Cloud DiscoveryClient integration
2. Declare intelligent routing for services, like shown below:

```
info.component: API Gateway

zuul.routes:
  customer-service:
    path: /customer/**
    serviceId: customer-service     ①

  product-service:
    path: /product/**
    serviceId: product-service
```

① customer-service is the application name of customer microservice, and fetched from eureka service registry using DiscoveryClient.

3. Request Rate Limiting (available in Spring Boot 2.x)
4. Path Rewriting
5. Hystrix Circuit Breaker integration for resiliency

3.45. How to protect internal endpoints leaking from API Gateway?

We can ignore certain endpoints at API Gateway and thus protect them from the external world.

Zuul provides a property named `zuul.ignored-patterns` that can be set to let zuul know what all we want to ignore.

Listing 31. application.properties

```
zuul.ignored-patterns: /*/health, /*/sensitive-endpoint
```

3.46. How to protect Sensitive Security Tokens from leaking into downstream system?

We can define what headers are sensitive at the API Gateway configuration.

Listing 32. /src/main/resources/application.yml

```
zuul.ignoreSecurityHeaders: false
zuul.sensitiveHeaders: Q
```

Here we are instructing zuul to relay token to the downstream systems. We can also configure the sensitive headers that should not be relayed.

3.47. How to retry failed requests at some other available instance using Client Side Load Balancer?

Spring Cloud Netflix provides two main mechanisms for making a HTTP requests using load balanced client (Ribbon) - RestTemplate and Feign. There is always a chance that a network call may fail due to any reason, and we may want to retry that request automatically on the next available server.

To enable this feature in Spring Cloud Netflix, we just need to include Spring Retry library on the classpath. When Spring Retry is present, the load balanced RestTemplate, Feign and Zuul will automatically retry any failed requests.

Listing 33. build.gradle

```
compile('org.springframework.retry:spring-retry')
```

We can customize the retry behavior by configuring certain Ribbon properties,

Listing 34. /src/main/resources/application.yml

```
client.ribbon.MaxAutoRetries: 3
client.ribbon.MaxAutoRetriesNextServer: 1
client.ribbon.OkToRetryOnAllOperations: true
```

Incase of Zuul Reverse Proxy, we can turn off retry functionality by setting

Listing 35. /src/main/resources/application.yml

```
zuul.retryable: false
```

Retry functionality can be disabled per route basis as well,

Listing 36. /src/main/resources/application.yml

```
zuul:
   routes:
       <routename>.retryable: false
```

For more information-

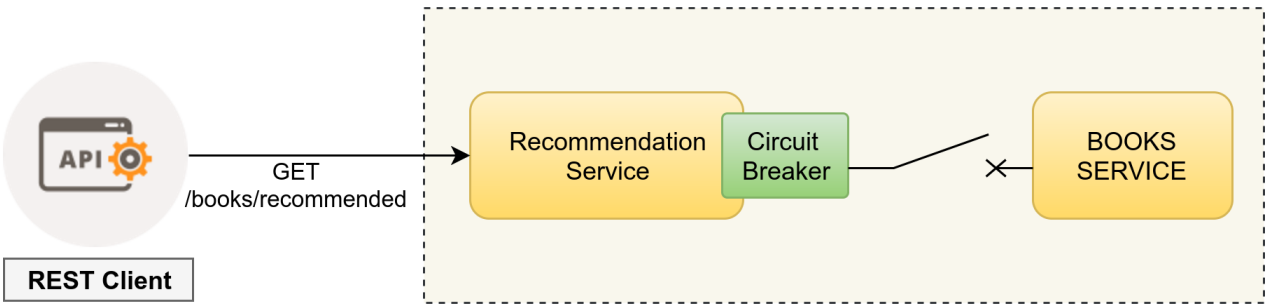
http://cloud.spring.io/spring-cloud-static/Camden.SR7/#retrying-failed-requests

3.48. What is Circuit Breaker Pattern?

Microservices often need to make remote network calls to another microservices running in a different process. Network calls can fail due to many reasons, including-

1. Brittle nature of network itself
2. Remote process is hung or
3. Too much traffic on the target microservices than it can handle

This can lead to cascading failures in the calling service due to threads being blocked in the hung remote calls. Circuit breaker is a piece of software that is used to solve this problem. The basic idea is very simple - wrap a potentially failing remote call in a circuit breaker object that will monitor for failures/timeouts. Once the failures reach a certain threshold, the circuit breaker trips, and all further calls to the circuit breaker return with an error, without the protected call being made at all. This mechanism can protect the cascading effects of a single component failure in the system and provide a option to gracefully downgrade the functionality. A typical use of circuit breaker in microservices architecture looks like the following diagram-



Typical Circuit Breaker Implementation

Here a REST client calls the Recommendation Service which further communicates with Books Service using a circuit breaker call wrapper. As soon as the books-service API calls starts to fail, circuit breaker will trip (open) the circuit and will not make any further call to book-service until the circuit is closed again.

Martin Fowler has beautifully explained this phenomenon in detail on his blog.

Martin Fowler on Circuit Breaker Pattern

https://martinfowler.com/bliki/CircuitBreaker.html

3.49. What are Open, Closed and Half-Open states of Circuit Breaker?

Circuit Breaker wraps the original remote calls inside it and if any of these calls fails, the failure is counted. When the service dependency is healthy and no issues are detected, the circuit breaker is in Closed State. All invocations are passed through to the remote service.

If the failure count exceeds a specified threshold within a specified time period, the circuit trips

into the Open State. In the Open State, calls always fail immediately without even invoking the actual remote call. The following factors are considered for tripping the circuit to Open State -

- An Exception thrown (HTTP 500 error, can not connect)
- Call takes longer than the configured timeout (default 1 second)
- The internal thread pool (or semaphore depending on configuration) used by hystrix for the command execution rejects the execution due to exhausted resource pool.

After a predetermined period of time (by default 5 seconds), the circuit transitions into a half-open state. In this state, calls are again attempted to the remote dependency. Thereafter the successful calls transitions the circuit breaker back into the closed state, while the failed calls return the circuit breaker into the open state.

3.50. What are use-cases for Circuit Breaker Pattern?

1. Synchronous communication over the network that is likely to fail is potential candidate for circuit breaker.
2. Circuit breaker is a valuable place for monitoring, any change in the breaker state should be logged so as to enable deep monitoring of microservices. It can easily troubleshoot the root cause of failure.
3. All places where a degraded functionality can be acceptable to caller if actual server is struggling/down.

3.51. What are benefits of using Circuit Breaker Pattern?

1. Circuit breaker can prevent a single service from failing the entire system by tripping off the circuit to the faulty microservice.
2. Circuit breaker can help offloading requests from a struggling server by tripping the circuit, thereby giving it a time to recover.
3. In providing a fallback mechanism where a stale data can be provided if real service is down.

3.52. Can circuit breaker be used in asynchronous communication?

Circuit breaker is mostly used in synchronous communication, but it can also be used in asynchronous communication.

In Asynchronous Communication, we put all the requests in a queue which the consumer consumes at its speed - a useful technique to avoid overloading of servers. A circuit breaker can be used here when the queue fills up.

3.53. What is Hystrix?

Hystrix is Netflix implementation for **circuit breaker pattern**, that also employs **bulkhead design pattern** by operating each circuit breaker within its own thread pool. It also collects many useful metrics about the circuit breaker's internal state, including -

1. Traffic volume.
2. Request volume.
3. Error percentage.
4. Hosts reporting
5. Latency percentiles.
6. Successes, failures and rejections.

All these metrics can be aggregated using another Netflix OSS project called Turbine. Hystrix dashboard can be used to visualize these aggregated metrics, providing excellent visibility into the overall health of the distributed system.

Hystrix is all about fail-fast, fail-silent and fallback.

More information about Hystrix Library

https://medium.com/netflix-techblog/fault-tolerance-in-a-high-volume-distributed-system-91ab4faae74a

3.54. What are main features of Hystrix library?

Hystrix library makes our distributed system **resilient** (adaptable & quick to recover) to failures. It provides three main features:

Latency and fault-tolerance

It helps stop cascading failures, provide decent fallbacks and graceful degradation of service functionality to confine failures. It works on the idea of fail-fast and rapid recovery. Two different options namely Thread isolation and Semaphore isolation are available for use to confine failures.

Real-time operations

Using real-time metrics, you can remain alerted, make decisions, affect changes and see results.

Concurrency

Parallel execution, concurrent aware request caching and finally automated batching through request collapsing improves the concurrency performance of your application.

More information on Netflix hystrix library:

- https://github.com/Netflix/Hystrix/
- https://github.com/Netflix/Hystrix/wiki#principles
- https://github.com/Netflix/Hystrix/wiki/How-it-Works

3.55. How to use Hystrix for fallback execution?

Hystrix can be used to specify the fallback method for execution in case actual method call fails. This can be useful for graceful degradation of functionality incase of failure in remote invocation.

Listing 37. Add hystrix library to `build.gradle`

```
dependencies {
    compile('org.springframework.cloud:spring-cloud-starter-hystrix')
```

Listing 38. Enable Circuit Breaker in main application

```
@EnableCircuitBreaker   ①
@RestController
@SpringBootApplication
public class ReadingApplication {

...
}
```

① This is important step

Listing 39. Using HystrixCommand fallback method execution

```
  @HystrixCommand(fallbackMethod = "reliable")  ①
  public String readingList() {
    URI uri = URI.create("http://localhost:8090/recommended");

    return this.restTemplate.getForObject(uri, String.class);
  }

  public String reliable() {    ②
    return "Cached recommended response";
  }
```

① Using `@HystrixCommand` annotation, we specify the fallback method to execute in case of exception.

② fallback method should have same signature (return type) as that of original method. This method provides a graceful fallback behavior while circuit is in open or half-open state.

3.56. When not to use Hystrix fallback on a particular microservice?

You do not need to wrap each microservice call within hystrix, for example

1. The api's that will never be invoked from another microservice shall not be wrapped in hystrix command.

2. If there is a service to service communication on behalf of a Batch Job (not end user), then probably you can skip hystrix integration depending upon your business needs.

3.57. How will you ignore certain exceptions in Hystrix fallback execution?

`@HystrixCommand` annotation provides attribute `ignoreExceptions` that can be used to provide list of ignored exceptions.

Listing 40. Hystrix ignore exceptions

```
import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.client.ServiceInstance;
import org.springframework.cloud.client.loadbalancer.LoadBalancerClient;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;

import java.net.URI;

@Service
public class HystrixService {

    @Autowired
    private LoadBalancerClient loadBalancer;

    @Autowired
    private RestTemplate restTemplate;

    @HystrixCommand(fallbackMethod = "reliable", ignoreExceptions = IllegalStateException.class,
MissingServletRequestParameterException.class, TypeMismatchException.class)
    public String readingList() {
        ServiceInstance instance = loadBalancer.choose("product-service");
        URI uri = URI.create("http://product-service/product/recommended");
        return this.restTemplate.getForObject(uri, String.class);
    }

    public String reliable(Throwable e) {
        return "Cloud Native Java (O'Reilly)";
    }

}
```

In the above example, if the actual method call throws IllegalStateException, MissingServletRequestParameterException or TypeMismatchException then **hystrix will not trigger the fallback logic (`reliable` method), instead the actual exception will be wrapped inside HystrixBadRequestException and re-thrown to the caller.** It is taken care by `javanica` library under the hood.

Reference

https://github.com/Netflix/Hystrix/tree/master/hystrix-contrib/hystrix-javanica#error-propagation

3.58. What is Strangulation Pattern in microservices architecture?

Strangulation is used to slowly decommission an older system and migrate the functionality to a newer version of microservices.

Normally one endpoint is Strangled at a time, slowly replacing all of them with the newer implementation. Zuul Proxy (API Gateway) is a useful tool for this because we can use it too handle all traffic from clients of the old endpoints, but redirect only selected requests to the new ones.

Let's take an example use-case:

Listing 41. /src/main/resources/application.yml

```
zuul:
  routes:
    first:
      path: /first/**
      url: http://first.example.com     ①
    legacy:
      path: /**
      url: http://legacy.example.com    ②
```

① Paths in `/first/**` have been extracted into a new service with an external URL http://first.example.com

② legacy app is mapped to handle all request that do not match any other patterns (`/first/**`).

This configuration is for API Gateway (zuul reverse proxy), and we are strangling selected endpoints `/first/` from the *legacy* app hosted at http://legacy.example.com slowly to newly created microservice with external URL http://first.example.com

More information can be found here

https://projects.spring.io/spring-cloud/spring-cloud.html#_strangulation_patterns_and_local_forwards

3.59. What is Circuit Breaker?

Circuit breaker is used to gracefully degrade the functionality when one of the service/method call fails, without cascading the effect that can bring down the entire system.

The logic of Circuit Breaker is quite simple -

Wrap a protected (mostly remote) call inside a circuit breaker object, which monitors for failures. Once the failures reach a certain threshold, trip the circuit and immediately return the error (or fallback response) without even making a actual remote call, thus giving failing service time to recover.

How it helps?

If many of the caller threads are being timeout on a failed resource, we may quickly exhaust all our

threads and the caller application will also get impacted due to this. Such failure can cascade and bring down the entire system. Circuit breaker does not allow this to happen by tripping off the circuits that are potentially failing. We achieve this fault-tolerance at the expense of degraded functionality.

3.60. What is difference between using a Circuit Breaker and a naive approach where we try/catch a remote method call and protect for failures?

Lets say we want to handle service to service failure gracefully without using Circuit Breaker pattern. The naive approach would be to wrap the inter service REST call in try catch clause. But Circuit Breaker does lot more that try catch can not accomplish -

1. Circuit Breaker does not even try calls once failure threshold is reached, doing so reduces the number of network calls. Also, number of threads consumed in making faulty calls are freed up.
2. Circuit breaker provides fallback method execution for gracefully degrading the behavior. Try catch approach will not do this out of the box without additional boiler plate code.
3. Circuit Breaker can be configured to use a limited number of threads for a particular host/API, doing so brings all the benefits of bulkhead design pattern.

So instead of wrapping service to service calls with try/catch clause, we must use circuit breaker pattern to make our system resilient to failures.

3.61. What is Request Collapsing feature in Hystrix?

We can front a HystrixCommand with a request collapser (HystrixCollapser is the abstract parent) with which we can collapse multiple requests into a single back-end dependency call. Using request collapser reduces the number of threads and network connections needed to perform concurrent HystrixCommand executions, that too in an automated manner without forcing developers to coordinate the manual batching of requests.

More Information

https://github.com/Netflix/Hystrix/wiki/How-it-Works#RequestCollapsing

3.62. What is difference between Circuit Breaker and Hystrix?

Circuit Breaker is a fault tolerance design pattern, while Netflix's Hystrix library provides an implementation for the circuit breaker pattern. We can easily apply circuit breakers to potentially-failing method calls (in JVM or over the network) using the Netflix Hystrix fault tolerance library.

3.63. Where exactly should i use Circuit Breaker Pattern?

At all those places where we are making a service to service call, for example 1. API Gateway 2. Aggregator Services 3. Web Front that calls multiple microservices to render a single page.

All those places where remote call can potentially fail are good candidate for using Circuit Breaker Pattern.

Where it should not be used?

Lets say, we are calling a REST endpoint directly from a mobile client (probably Android), and there is no inter-service calls involved in this case, except at API Gateway. So there is no need for the circuit breaker except at API gateway level. Android client should be designed to gracefully handle service failures in this case.

3.64. What is bulkhead design pattern?

Origin of Term Bulkhead

A ship's hull is divided into different watertight compartments (called as bulkheads) so that if the hull is compromised, the failure is limited to that bulkhead as opposed to taking the entire ship down.

By partitioning our system for fault tolerance, we can confine failures (timeouts, OOME, etc.) to one area and prevent the entire system from failing due to cascading effects.

Consider combining bulkheads with retry, circuit breaker, and throttling patterns to provide more sophisticated fault handling.

3.65. How does Hystrix implements Bulkhead Design Pattern?

The bulkhead implementation in Hystrix limits the number of concurrent calls to a component/service. This way, the number of resources (typically threads) that is waiting for a reply from the component/service is limited.

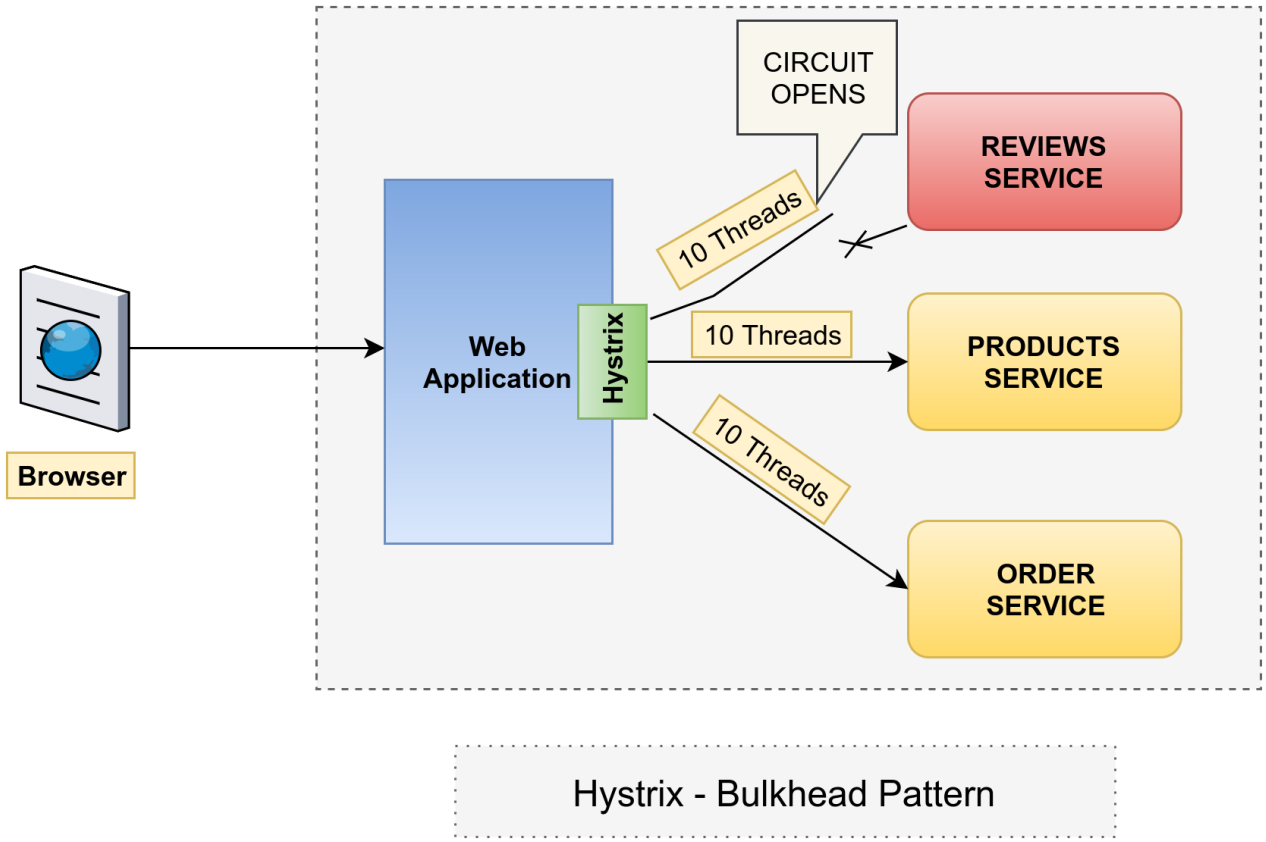
Lets assume we have a fictitious web application as shown in figure below. The WebFront communicates with 3 different components using remote network calls (REST over HTTP).

- Product catalogue Service
- Product Reviews Service
- Order Service

Now lets say due to some problem in Product Review Service, all requests to this service start to hang (or timeout), eventually causing all request handling threads in WebFront Application to hang on waiting for an answer from Reviews Service. This would make the entire WebFront Application non-responsive. The resulting behavior of the WebFront Application would be same if request volume is high and Reviews Service is taking time to respond to each request.

The Hystrix Solution

Hystrix's implementation for bulkhead pattern would limit the number of concurrent calls to components and would have saved the application in this case by gracefully degrading the functionality. Assume we have 30 total request handling threads and there is a limit of 10 concurrent calls to Reviews Service. Then at most 10 request handling threads can hang when calling Reviews Service, the other 20 threads can still handle requests and use components Products and Orders Service. This will approach will keep our WebFront responsive even if there is a failure in Reviews Service.



Bulkhead Design Pattern using hystrix

3.66. What is Hystrix approach to Bulkhead Pattern?

The bulkhead implementation in Hystrix limits the number of concurrent calls to a component. This way, the number of resources (typically threads) that is waiting for a reply from the component is limited.

Hystrix' has two different approaches to the bulkhead, thread isolation and semaphore isolation.

The advantage of the thread pool approach is that requests that are passed to C can be timed out, something that is not possible when using semaphores.

Thread Isolation

The standard approach is to hand over all requests to component C to a separate thread pool with a fixed number of threads and no (or a small) request queue.

Semaphore Isolation

The other approach is to have all callers acquire a permit (with 0 timeout) before requests to C. If a permit can't be acquired from the semaphore, calls to C are not passed through.

3.67. In microservices architecture, what are smart endpoints and dumb pipes?

Martin Fowler introduced concept of "smart endpoints & dumb pipes" while describing microservices architecture.

To give context, one of the main characteristic of a unix based system is to build small small utilities and connect them using pipes. For example, very popular way of finding all java processes in linux system is

Command pipeline in unix shell

```
ps elf | grep java
```

Here two commands are separated by pipe, the pipe's job is to forward the output of first command as an input to second command, nothing more. Its like a dumb pipe which has no business logic except the routing of data from one utility to another.

In his article Martin Fowler compares Enterprise Service Bus (ESB) to ZeroMQ/RabbitMQ, ESB is a pipe but has lot of logic inside it while ZeroMQ has no logic except the persistence/routing of messages. ESB is a fat layer that does lot of things like - security checks, routing, business flow & validations, data transformations, etc. So ESB is a kind of smart pipe that does lot of things before passing data to next endpoint (service). Smart endpoints & dumb pipes advocate an exactly opposite idea where the communication channel should be stripped of any business specific logic and should only distribute messages between components. The components (endpoints/services) should do all the data validations, business processing, security checks, etc on those incoming messages.

Microservices team should follow the principles and protocols that world wide web & unix is built on.

Reference

https://martinfowler.com/articles/microservices.html

3.68. What is difference between Semaphore and ThreadPool based configuration in Hystrix?

The default behavior of hystrix in spring cloud environment is to use Thread Isolation strategy for wrapping network calls. Using semaphore isolation is very easy when we want to propagate security context (AccessToken) to the downstream server.

Thread Isolation

The standard approach is to hand over all requests to component C to a separate thread pool with a fixed number of threads and no (or a small) request queue.

Semaphore Isolation

The other approach is to have all callers acquire a permit (with 0 timeout) before requests to C. If a permit can't be acquired from the semaphore, calls to C are not passed through.

3.69. How to handle versioning of microservices?

There are different ways to handle the versioning of your REST api to allow older consumers still consume the older endpoints. The ideal practice is that any non backward compatible change in a given REST endpoint shall lead to a new versioned endpoint.

Different mechanisms of versioning are:

- Add version in the URL itself
- Add version in API request header

Most common approach in versioning is the URL versioning itself. A versioned URL looks like the following:

Versioned URL

```
https://<host>:<port>/api/v1/...
https://<host>:<port>/api/v2/...
```

As a API developer you must ensure that only backward compatible changes are accommodated in a single version of URL. Consumer-Driven-Tests can help identify potential issues with API upgrades at an early stage.

3.70. What is difference between partitioning microservices based on technical capabilities vs business capabilities? Which one is better?

An enterprise level application should be partitioned into multiple microservices based on its business capabilities as laid down in Domain Driven Design (Bounded context principle).

Partitioning around the technical capabilities should not be followed in general, except for the scenarios that come under infrastructure services i.e. email service, sms service, storage service (Amazon S3), queue service (Amazon SQS), database service (Amazon RDS, DynamoDB, etc.)

So, my final advice is:

> Always partition around the business capabilities (orders management, user management, payment service, demand service, etc.).
>
> Partitioning around technical capabilities should only be permitted for creating infrastructural services.

3.71. Running Spring boot app at different port on server startup.

Why do we need that?

If we want to run multiple instances of single app on same server, then we need to assign a different port at runtime. To solve this problem we need to choose random available port at app startup. Since the app will register itself with eureka service registry, other apps can still discover this service through service registry.

Spring boot provides a convenient class to determine if a port is available or not.

Assigning a random port in custom port range

Listing 42. Utility class that search available port in custom range

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.util.SocketUtils;
import org.springframework.util.StringUtils;

public class SpringBootUtil {
    final static Logger log = LoggerFactory.getLogger(SpringBootUtil.class);

    public static void setRandomPort(int minPort, int maxPort) {
        try {
            String userDefinedPort = System.getProperty("server.port", System.getenv("SERVER_PORT"));
            if(StringUtils.isEmpty(userDefinedPort)) {
                int port = SocketUtils.findAvailableTcpPort(minPort, maxPort);
                System.setProperty("server.port", String.valueOf(port));
                log.info("Random Server Port is set to {}.", port);
            }
        } catch( IllegalStateException e) {
            log.warn("No port available in range 5000-5100. Default embedded server configuration will be used.");
        }
    }
}
```

Listing 43. Spring Boot Main Application

```
@SpringBootApplication
public class Application {
    private static final Logger logger = LoggerFactory.getLogger(Application.class);

    public static void main(String[] args) {
        SpringBootUtil.setRandomPort(5000, 5500);   ①
        ApplicationContext ctx = SpringApplication.run(Application.class, args);
        logger.info("Application " + ctx.getApplicationName() + " started");
    }
...
}
```

① Calling the custom method to set the random available port within range [5000-5500] and update the server.port property as well.

Always use Eureka Registry to fetch the service details e.g. host, port and protocol. Never use hardcoded host, port while communicating with one microservice from another. So you never need to know in advance what port and host a particular service has been deployed.

3.72. How will you run certain business logic at the app startup?

Often there is a need to execute certain business logic at servlet container startup for e.g. fetching feeds from a remote service, sending a notification about app startup by email, etc.

Spring boot provides two main mechanisms for wiring app startup code. This code will be executed once spring context has been loaded and servlet container is ready to serve requests.

1. Using `CommandLineRunner`
2. Using `ApplicationRunner`

Running more than one ApplicationRunner/CommandLineRunner in pre-defined order

We can use `@Order(<sequence>)` annotation to each ApplicationRunner Bean to run them in specified order.

For example, in the below code we have created two startup runners that execute in a sequence.

```
@Order(1)   ①
@Component
class StartupExecutorOne implements ApplicationRunner {
    @Override
    public void run(ApplicationArguments args) throws Exception {
        System.out.println("StartupExecutorOne invoked");
    }
}

@Order(2)
@Component
class StartupExecutorTwo implements ApplicationRunner {
    @Override
    public void run(ApplicationArguments args) throws Exception {
        System.out.println("StartupExecutorTwo invoked");
    }
}
```

① Bean with `Order` 1 will execute before bean with `Order` 2.

Program output

```
StartupExecutorOne invoked
StartupExecutorTwo invoked
```

`CommandLineRunner` can also be used in a similar fashion.

PostConstruct vs ApplicationRunner

Spring provides `@PostConstruct` annotation that can be used for post initialization work on a give component. But `@PostConstruct` method is invoked after the component has been created by spring container and before entire Spring context have been initialized.

On the other hand `ApplicationRunner` code will only run once the entire spring context have been initialized.

3.73. How to correctly implement a reporting microservice in a distributed system?

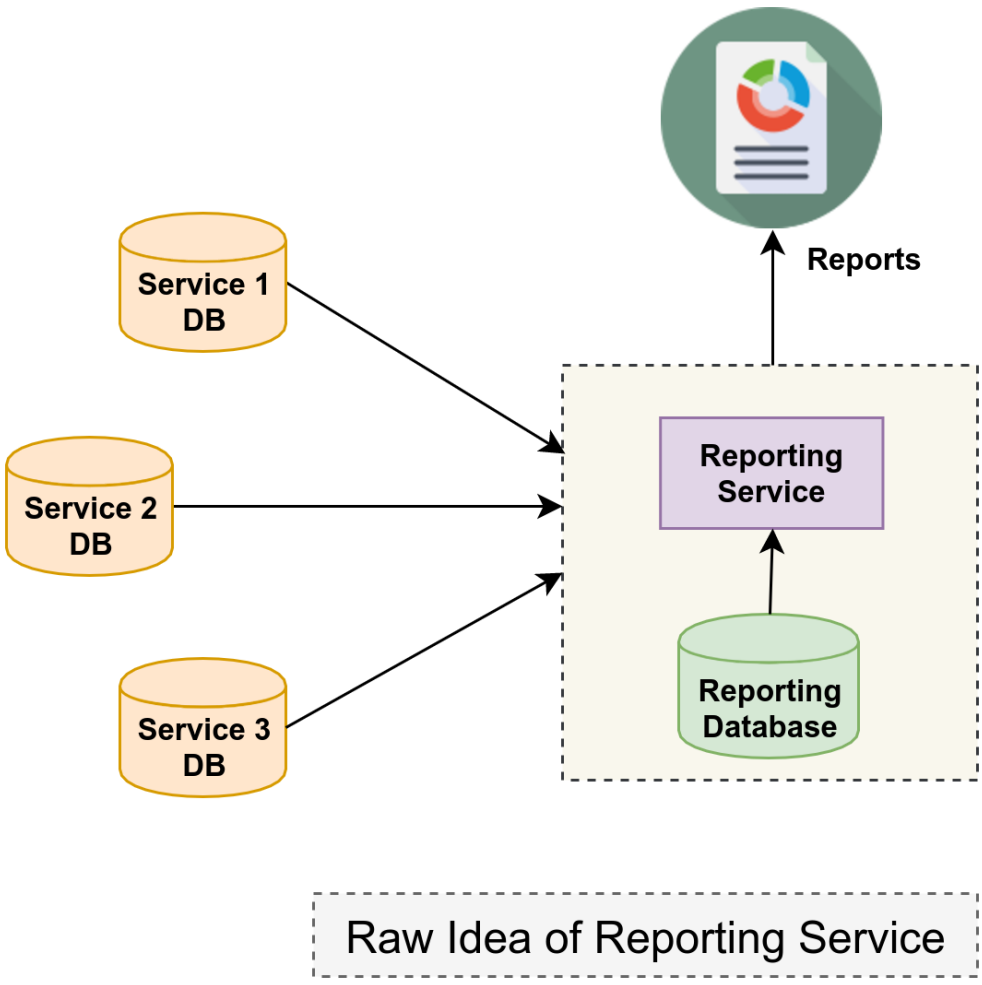
Implementing a reporting service requires collecting data from multiple microservices, hence its a unique problem in distributed systems. Preserving Bounded Context is very important aspect in microservices architecture, and specially in reporting kind of service.

There are multiple options to implement a reporting module in distributed system.

Why we need to encapsulate reporting features into a separate microservice?

Operational and the reporting needs are often different - with different requirements from schema, and different access patterns. In a complex system, it's always a good idea to separate reporting needs into a separate microservice having its own database. This microservice should take copy of the essential operational data and persist it in a different schema that is most appropriate for reporting needs.

In a situation like reporting service, we need to gather data from multiple microservices and then do analysis and reporting.



Reporting microservice - raw idea

Reporting database has number of advantages:

- The structure of reporting database could be designed just for reports.
- Reporting database is often read only, so there is no need for the normalization. Data duplication may make reporting easier and efficient.
- Each microservice can refactor its own database without affecting the reporting database.
- Queries run against the reporting database does not add load to operational database.
- You may have multiple reporting databases for different reporting needs.
- You can store derived data in the database, making it easier to write reports that use the derived data without having to introduce a separate set of derivation logic.

Let's first see what are the wrong ways to implement a reporting solution within microservices architecture.

HTTP pull model

A naive approach will be to make synchronous calls to other microservices and fetch the required data and then do the reporting. Though this model does not violate the principle of Bounded Context, but this approach is not efficient due to -

1. It will be tough to get huge data over synchronous calls.
2. It will put load on the operational database.

Database pull model

Pulling data directly from the database seems to be fastest option, as there is no other layer involved in this. But it leads to significant interdependence between individual microservice and the reporting service, making it hard to accommodate any refactoring in microservice. So this approach is efficient but tightly coupled.

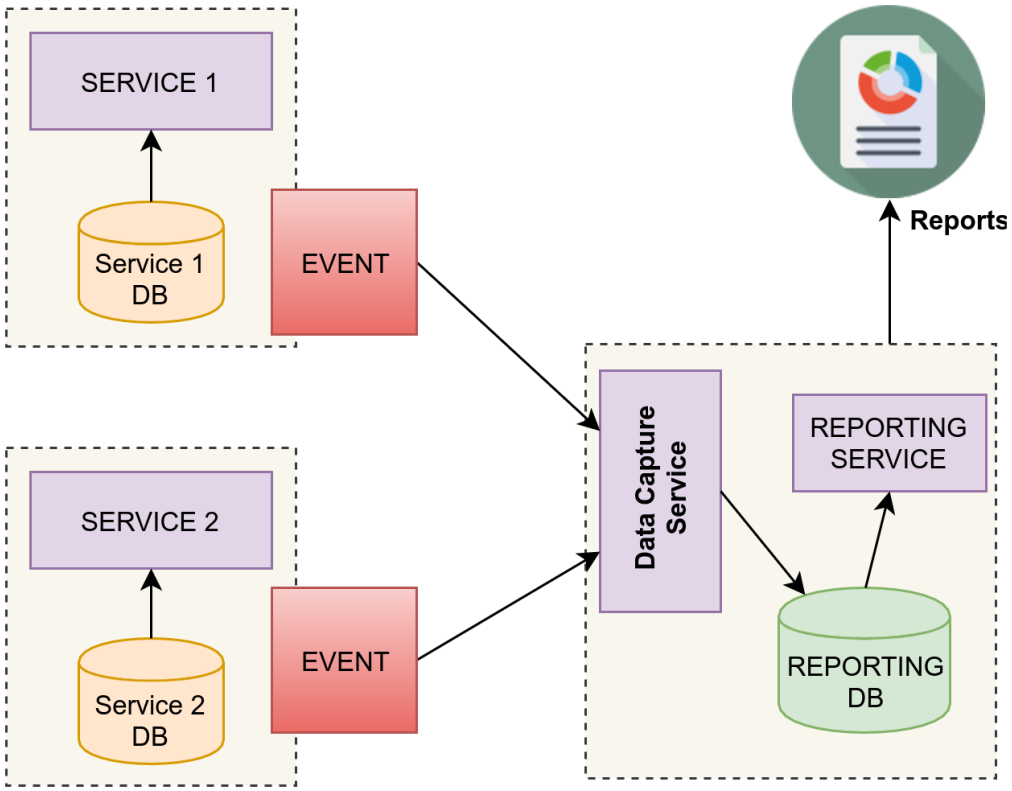
Lets now head towards the right approach for implementing Reporting Service Model.

Asynchronous event driven push model

The first two models were pull models where reporting microservice pulls data from individual microservice and that's a anti-pattern. Ideal way should be a push model where each microservice push data to reporting microservice.

Microservices should choose choreography over orchestration otherwise services will become tightly coupled. The right approach to implement such a reporting requirement is to use event based push model where each microservice transmits events to reporting microservice to keep data as real time as possible. This approach preserves the Bounded Context of each microservice.

So a better design would be to use Event Driven Approach, where each service will emit an event (OrderCreated, UserRegistered, etc.) and reporting service will listen to those events and create reporting. The following diagram illustrates a better design for correctly implementing a reporting service in any distributed system.



Right approach to implement Reporting Service Model

Data Pump

Sam Newman, in his book "Building Microservices", refers to this technique as a data pump.

References

- https://martinfowler.com/bliki/ReportingDatabase.html

3.74. What is Event Sourcing and CQRS? When should it be used? Should be use it for the entire system?

Event Sourcing should only be used for the specific scenarios, not everywhere in the system, otherwise it will complicate the implementation of the system.

Ideal Candidates for Event Sourcing

Shipping Tracker microservice could be an ideal candidate for the event sourcing technique.

3.75. How to send business errors from a RESTful microservice to client application?

In RESTful applications, often is not enough to just send the HTTP error codes for representing the business errors. HTTP error codes can tell about the kind of failure but will not be able to point to business error codes. The solution to this is to wrap our service response inside a custom object that can reveal more information about the failure reason, error codes for machine and developers.

The below class is an example showing how we can wrap additional information about the business exception such as errorCode, userMessage, developerMessage, etc. inside a service response.

Listing 44. Typical Implementation of ServiceResponse Wrapper Class

```
public class ServiceResponse<T> {

    T data;

    boolean success;
    int errorCode;
    String moreInfo;
    String userMessage;
    String developerMessage;

    List<String> errors = new ArrayList<>();

    //Getter and setters removed for brevity
}
```

Now in our Rest Controller, we can create an instance of this ServiceResponse and send it back to the RestClient.

Listing 45. Rest Controller that Fetches Order Information

```
    ServiceResponse<Order> response = new ServiceResponse<>();
    Order order = //Get Order from remote microservices
    if(order!=null) {
        response.setData(order);
        response.setSuccess(true);
    } else {
        response.setErrorCode(12254);   ①
        response.setSuccess(false);
    }
    return response;
```

① 12254 here is the business error code that api client would know how to deal with.

3.76. Is it a good idea to share common database across multiple microservices?

In microservices architecture, each microservice shall own its private data which can only be accessed by outside world through owning service. If we start sharing microservice's private datastore with other services, then we will violate principle of Bounded Context.

Practically we have three approaches -

1. Database server per microservice - Each microservice will have its own database server

instance. This approach has overhead of maintaining database instance and its replication/backup, hence its rarely used in practical environment.

2. Schema per microservice - Each microservice owns a private database schema which is not accessible to other services. Its most preferred approach for RDMS database (MySql, Postgres, etc.)
3. Private Table per microservice - Each microservice owns a set of tables that must only be accessed by that service. Its a logical separation of data. This approach is mostly used for hosted database as service solution (Amazon RDS).

Table-Per-Service approach - Amazon DynamoDB

If we are using database as a service (like AWS dynamoDB), then you shall prefer private table-per-service approach, where each microservice owns a set of tables that must only be accessible by that service. It is mostly a logical separation of data. In this way we can have a single DynamoDB instance for entire fleet of microservices.

3.77. How will you make sure that the email is only sent if the database transaction does not fail?

Spring provides two main mechanisms to handle this situation -

1. Using @TransactionalEventListener for listening transaction success event.

```
@Component
public class MyComponent {

  @TransactionalEventListener
  public void handleOrderCreatedEvent(CreationEvent<Order> creationEvent) {
    ...
  }

}
```

2. Using TransactionSynchronizationManager to attach a listner that gets invoked when transaction completes successfully.

Listing 46. Using TransactionSynchronizationManager

```
    @Transactional
    public void create(String firstName, String lastName, String email) {
        User user = new User();
        user.setEmail(email);
        user.setFirstName(firstName);
        user.setLastName(lastName);
        userRepository.save(user);

        TransactionSynchronizationManager.registerSynchronization(new TransactionSynchronizationAdapter() {
            public void afterCommit() {
                //do stuff right after commit is successful
                //TODO: Send welcome email to user.
            }
        });
    }
```

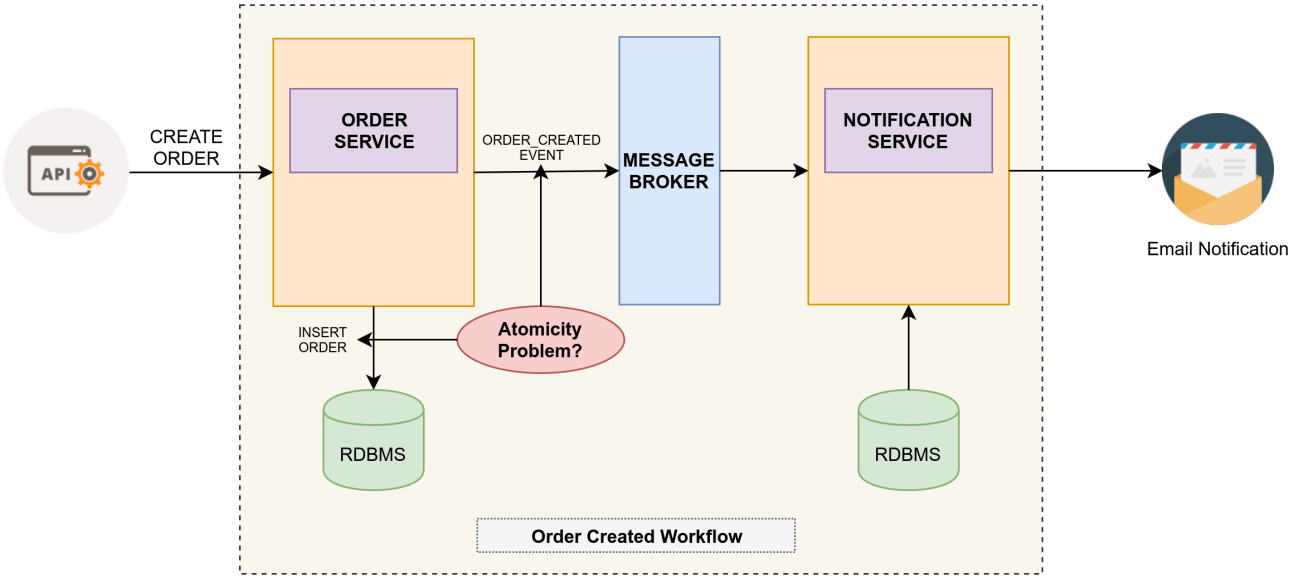
But the above mentioned code does not guarantee the atomicity of two operations i.e. if the transaction is successful but email send fails then spring will not rollback the transaction.

3.78. How will you atomically update the database and publish an event to message broker from single transaction?

Practical Problem

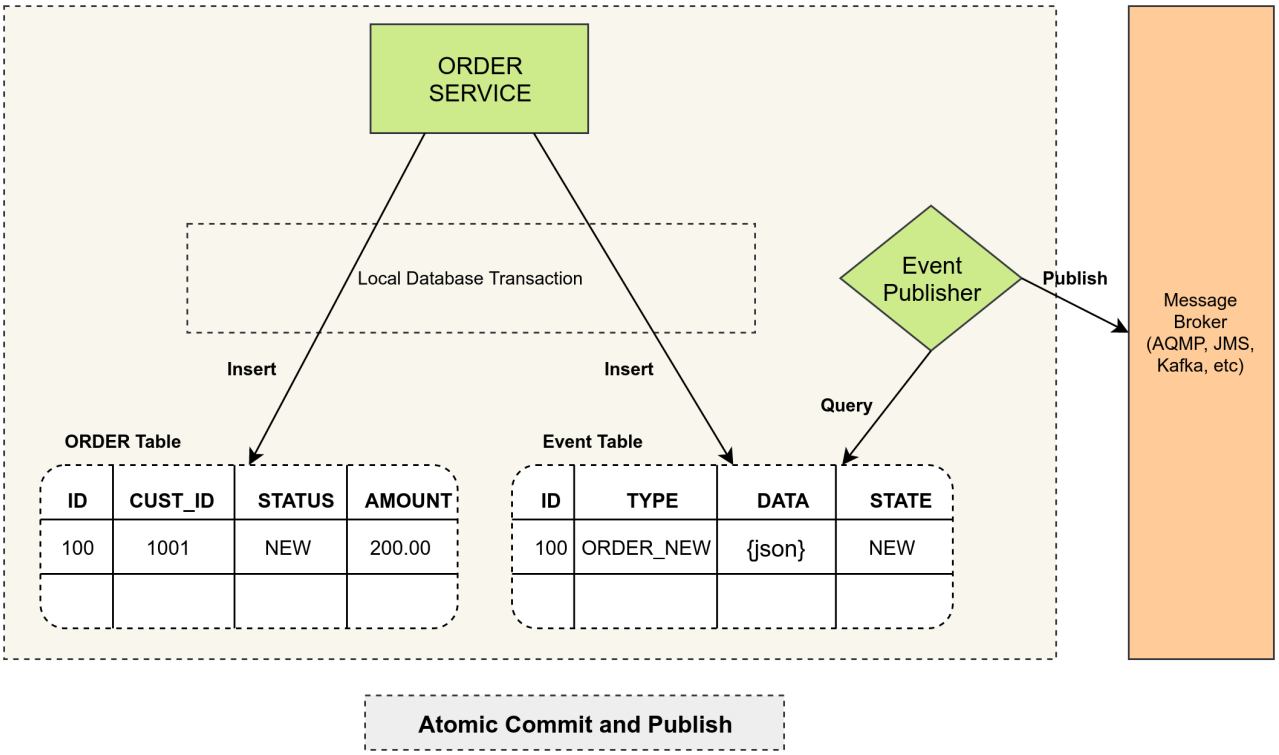
We have two microservices - ORDER Service and NOTIFICATION Service, whenever a new order arrives, we need to trigger an email in reliable fashion. How to achieve this?

In microservices styled applications there is a common problem of atomically updating the database and publishing an event to message broker. For example, whenever a new order is created Order Service must insert a new record into its local database while sending a Event to Email Service that will send the Email/SMS notification to the end user. It is very essential that these two operations are done atomically, otherwise if the service fails after updating the database but before publishing the event, the system will become inconsistent and user will never receive any email/SMS notification of the new order.



Atomicity Problem b/w Insert Order and send email

The standard way to ensure atomicity is to use a distributed transaction involving the database and the Message Broker. However, for the reasons described above, such as the CAP theorem, this is exactly what we do not want to do.



Using Local DB Transaction to Achieve Atomicity

Using Local Transactions

One way to achieve atomicity in this case is to publish events using a multi-step process involving local transactions.

3.79. How will you propagate security context of user when one microservice calls another microservice on behalf of user?

There are multiple ways the security context can be passed over service to service call. If Oauth2 is configure for security, then OAuth2 Token Relay can be used to propagate the user's accessToken to the next server. If we are using Basic Auth, then authorization header can be passed to the downstream system.

Configuring Token Relay, or Authorization Header Propagation at Zuul Proxy

You can control the authorization behaviour downstream of an @EnableZuulProxy through the proxy.auth.* settings. Example:

Listing 47. application.yml

```
proxy:
  auth:
    routes:
      customers: oauth2
      stores: passthru
      recommendations: none
```

In this example the "customers" service gets an OAuth2 token relay, the "stores" service gets a passthrough (the authorization header is just passed downstream), and the "recommendations" service has its authorization header removed. The default behaviour is to do a token relay if there is a token available, and passthru otherwise.

3.80. What is Token Relay in Spring Security?

A Token Relay happens when an OAuth2 consumer acts as a client and forwards the incoming OAuth2 Token to the outgoing resource request. The Consumer can be a pure client (like an SSO application) or a Resource Server.

3.81. How to Enable Token Relay?

If your app is a user facing OAuth2 client (i.e. has declared @EnableOAuth2Sso or @EnableOAuth2Client) then Spring Boot will automatically create a `OAuth2ClientContext` and `OAuth2ProtectedResourceDetails` in request scope. You can create your own `OAuth2RestOperations` using these two beans, as shown in below code. And then the context will always forward the access token downstream, also refreshing the access token automatically if it expires. (These are features of Spring Security and Spring Boot.)

Listing 48. Creating a OAuth2RestTemplate that relays Token

```
    @LoadBalanced
    @Bean
    @Autowired
    public OAuth2RestTemplate loadBalancedOauth2RestTemplate(OAuth2ClientContext oauth2ClientContext,
OAuth2ProtectedResourceDetails details) {
        return new OAuth2RestTemplate(details, oauth2ClientContext);
    }
```

Spring Boot (1.4.1) does not create an OAuth2ProtectedResourceDetails automatically if you are using client_credentials tokens. In that case you need to create your own ClientCredentialsResourceDetails and configure it with @ConfigurationProperties("security.oauth2.client").

3.82. How to revoke Access and Refresh Tokens on data breach to limit the damage?

In stateless security using JWT/OAuth2, its not possible to revoke the tokens because there is no centralized system where tokens will be checked for validity. It all happens through cryptography.

How JWT works?

JWT uses cryptography to validate the access token validity and its claims. JWT tokens are signed by authentication server that has private key, all resource servers can validate the signed JWT token using their public key. So JWT is essentially a decentralized security mechanism.

JWT & Currency

JWT is like Currency, where you need not to go to RBI whenever someone hands over 100 Rs Note to you, every citizen can check the validity of a currency note using security features provided by RBI. Thats the power of cryptography and distributed computing.

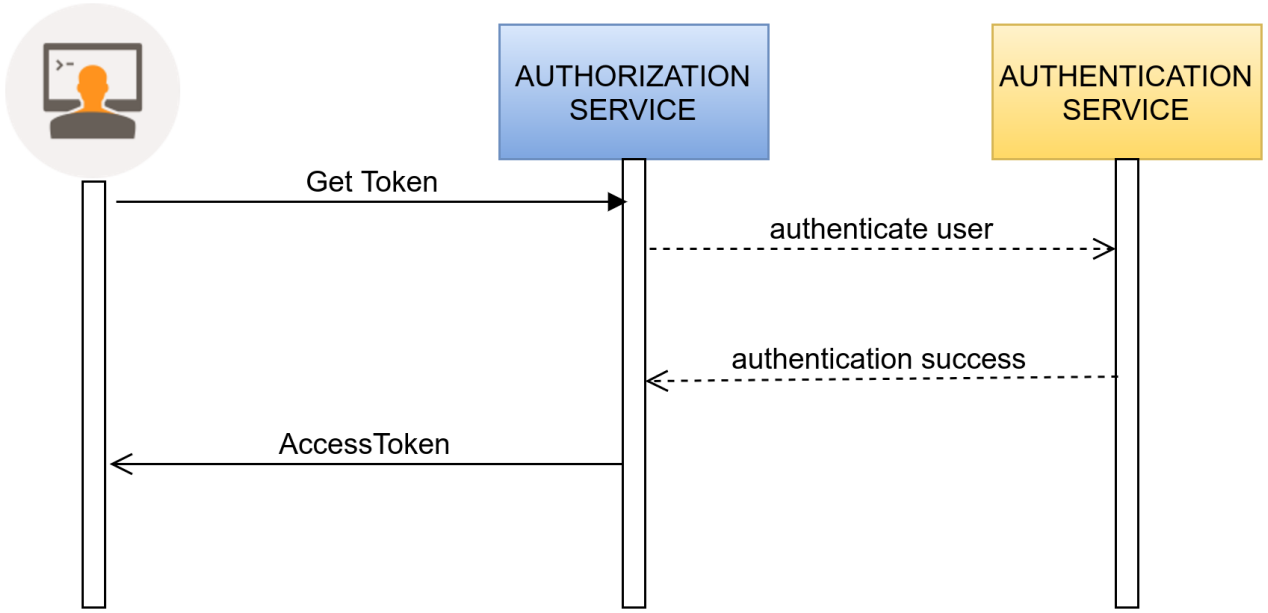
If the damage is system wide, then its better to change the private key used in authentication service to generate the Access and RefreshToken. Thus any token that had been issued before data breach will become invalid and resource server will deny access to resource. The only side effect of this approach will be that all the users be logged out of the system and they need to login again to access protected resources.

If damage is client wide (one of the client is compromised), then we should change the client credentials. Changing client credentials will invalidate refresh tokens, since all new access token calls using refresh token will start failing. But whatever AccessToken has been issued, they will continue to work until their expiry. Please be noted here that Refresh Token endpoint requires Basic Authentication Against Client Credentials.

If the damage is local, say a single person's access/refresh token has been stolen, then we shall store a device id or some other identifier for each user client device. Now instead of blocking the JWT, we can block the user's particular device that had been stolen.

3.83. Shall Authentication and Authorization be one service?

Not necessarily. Authentication and authorization are two different things. Authentication needs to know user identity (username and credentials), authorization service's responsibility is to issue tokens. So you can create two different services - one for authentication and another for authorization. But authorization service will need to talk to authentication service before issuing tokens so REST communication will be required between the two.



3.84. What is API Key security?

API Key is simple form of security for modern web APIs. Its simple because all we have to do is to set the API Key in Authorization header or the URI itself to get authenticated. There are downsides to using API Key based security.

API Key example

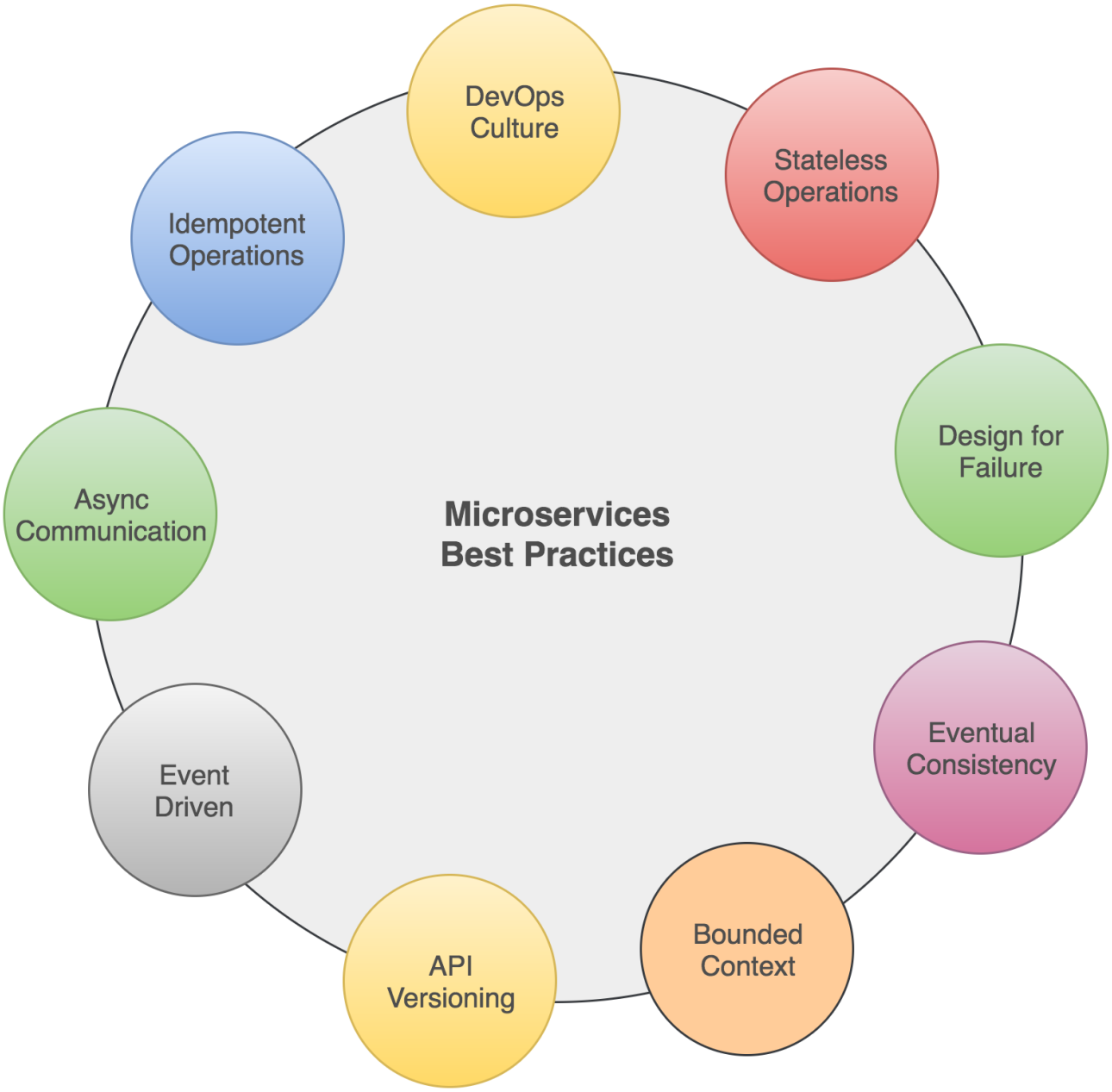
```
Authorization: Apikey 1234567890abcdef
```

Downsides of API Key security

1. Typically API key gives full access to operations that an API can perform.
2. It does not distinguish between Client and End User.
3. Its not easy to add custom data to API Key, like we can do in JWT AccessToken.
4. You need to keep them secret, because these keys does not have expiry. If you share it, you need to deactivate the existing one and generate a new one.

3.85. What are best practices for microservices architecture?

Microservices Architecture can become cumbersome & unmanageable if not done properly. There are best practices that help design a resilient & highly scalable system. The most important ones are -



Best Practices in Microservices Architecture

Partition correctly

Get to know the domain of your business, thats very very important. Only then you will be able to define the bounded context and partition your microservice correctly based on business capabilities.

DevOps culture

Typically, everything from continuous integration all the way to continuous delivery and deployment should be automated. Otherwise its a big pain to manage large fleet of microservices.

Design for stateless operations

We never know where a new instance of a particular microservice will be spun up for scaling out or for handling a failure, so maintaining a state inside service instance is a very bad idea.

Design for failures

Failures are inevitable in distributed systems, so we must design our system for handling failures gracefully. failures can be of different types and must be dealt accordingly, for example -

1. Failure could be transient due to inherent brittle nature of the network, and the next retry may succeed. Such failures must be protected using retry operations.
2. Failure may be due to a hung service which can have cascading effects on the calling service. Such failures must be protected using Circuit Breaker Patterns. A fallback mechanism can be used to provide degraded functionality in this case.
3. A single component may fail and affect the health of entire system, bulkhead pattern must be used to prevent entire system from failing.

Design for versioning

We should try to make our services backward compatible, explicit versioning must be used to cater different versions of the RESt endpoints.

Design for asynchronous communication b/w services

Asynchronous communication should be preferred over synchronous communication in inter microservice communication. One of the biggest advantages of using asynchronous messaging is that the service does not block while waiting for a response from another service.

Design for eventual consistency

Eventual consistency is a consistency model used in distributed computing to achieve high availability that informally guarantees that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value.

Design for idempotent operations

Since networks are brittle, we should always design our services to accept repeated calls without any side effects. We can add some unique identifier to each request so that service can ignore the duplicate request sent over the network due to network failure/retry logic.

Share as little as possible

In monolithic applications sharing is considered to be a best practice but thats not the case with Microservices. Sharing results in violation of Bounded Context Principle, so we shall refrain from creating any single unified shared model that works across microservices. For example, if different services need a common Customer model, then we should create one for each microservice with just the required fields for a given bounded context rather than creating a big model class that is shared in all services. The more dependencies we have between services, the harder it is to isolate the service changes, making it difficult to make change in a single service without effecting other service. Also, creating a unified model that works in all services brings complexity and ambiguity to the model itself, making it hard for anyone to understand the model.

In a way are want to violate the DRY principle in microservices architecture when it comes to domain models.

Reference

https://blogs.oracle.com/developers/getting-started-with-microservices-part-three

3.86. Shall we share common domain models or DTOs across microservices?

Ideally you should not. Creating and sharing unified DTO or Models violates the principle of Bounded Context. These shared libraries introduces dependencies between microservices taking away their autonomous behaviour. Moreover these unified models becomes too complex in any real enterprise application that hardly any single team can understand it.

For example, if we have a Customer model that will be used in Payment Service, Shipment Service and Notification Service then we shall create three different copies of Customer model with attributes appropriate for each individual service.

Listing 49. Customer Model in Payment Service

```
public class Customer {
    private String id;
    private String name;
    private Address billingAddress; ①
}
```

① Payment service needs a billing address

Listing 50. Customer Model in Shipment Service

```
public class Customer {
    private String id;
    private String name;
    private String mobile;
    private Address shippingAddress;    ①
}
```

① Shipment service is interested in Shipping Address of customer

Listing 51. Customer Model in Notification Service

```
public class Customer {
    private String id;
    private String name;
    private String mobile;  ①
    private String email;
}
```

① Notification service needs to know mobile and email for sending alerts about order to customer.

Here Payment Service is interested in billingAddress while Shipment Service is interested in ShippingAddress. So each Bounded Context (Payments, Shipment and Notification in this case) can have its own copy of Customer Model with appropriate fields.

3.87. How to share common code across multiple microservices?

There may be a need to share common utility classes across the services. For example, you might have created a utility class to start application on random port, or a customized RestTemplate with SelfSignedCertificate handling that can be used in multiple services. These classes can be put together into a separate shared library module and published to privately hosted maven repository using maven or gradle. Now all interested projects can use versioned jar dependencies to include common code. This way we can make sure that any new change does not break any other service using previous version.

Always share versioned jar

Never distribute Shared Library Modules as source code (without versioned jar) across microservices, instead use versioned jars published into private maven repository.

Do not create Unified Model

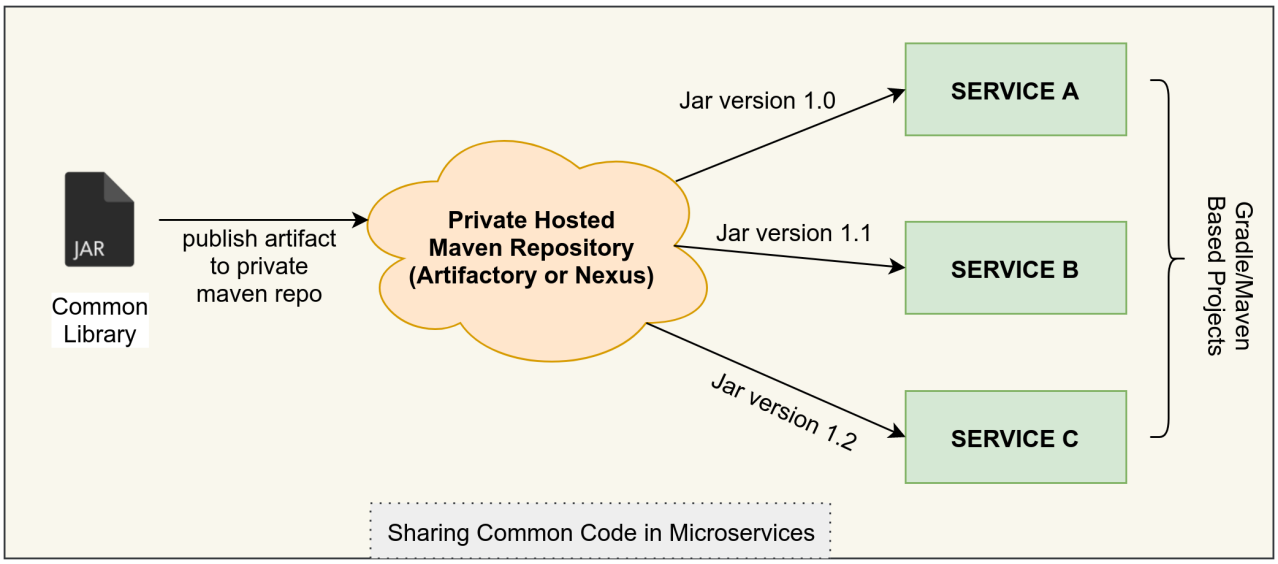
Do not create shared libraries for unified models across microservices, that's an anti-pattern. Each microservice should have its own version of a model that is shared across services. For example, Customer domain model should have its own definition in each microservice, with attribute only relevant to that business unit.

Reference

https://martinfowler.com/bliki/BoundedContext.html https://martinfowler.com/tags/domain%20driven%20design.html

Setting up private maven repository for library sharing

There are two open source options available for hosting your own private maven repository - Artifactory OSS and Nexus. You can use anyone of these in your own project.



Privately hosted maven repository for Common Library

Once repository is setup, you can use maven-publish to publish shared library artifacts using below code in gradle project.

Listing 52. build.gradle of shared module

```
apply plugin: 'maven-publish'

group = 'com.shunya.commons'
version = '1.0.0'

publishing {
    publications {
        mavenJava(MavenPublication) {
            from components.java
        }
    }
    repositories {
        maven {
            credentials {
              username 'admin'
              password 'password'
            }
            url "http://localhost:8081/artifactory/libs-release-local"
        }
    }
}
```

Now this uploaded artifact can be used by all other gradle projects using the below code:

Listing 53. build.gradle - top level

```
allprojects {
    repositories {
        maven { url "http://localhost:8081/artifactory/libs-release-local" }
    }
}
```

Listing 54. build.gradle - main project

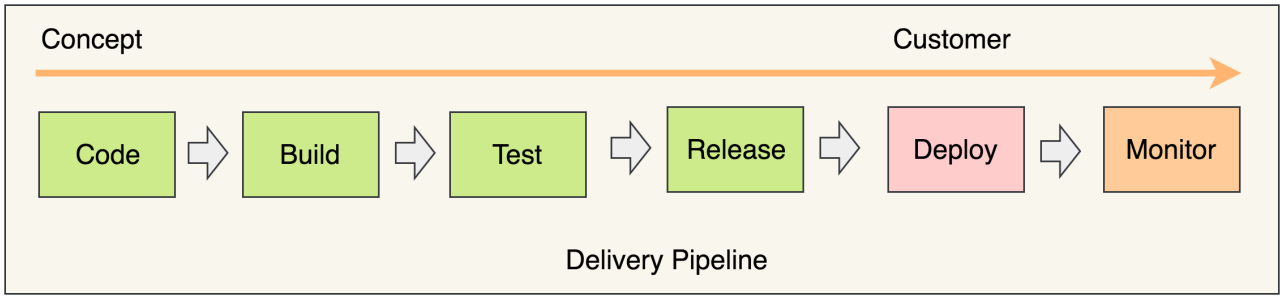
```
dependencies {
    compile 'com.shunya.commons:1.0.0'
}
```

That's it. This way each microservice can choose which version of common library to use. It gives flexibility to rollback to previous version if something goes wrong.

3.88. What is continuous delivery?

Continuous delivery is a software engineering practice in which agile teams produce software in continuous low-risk short cycles, ensuring that software can be reliably released at any time. This makes it possible to continuously adapt software inline with user feedback and changes in business requirements.

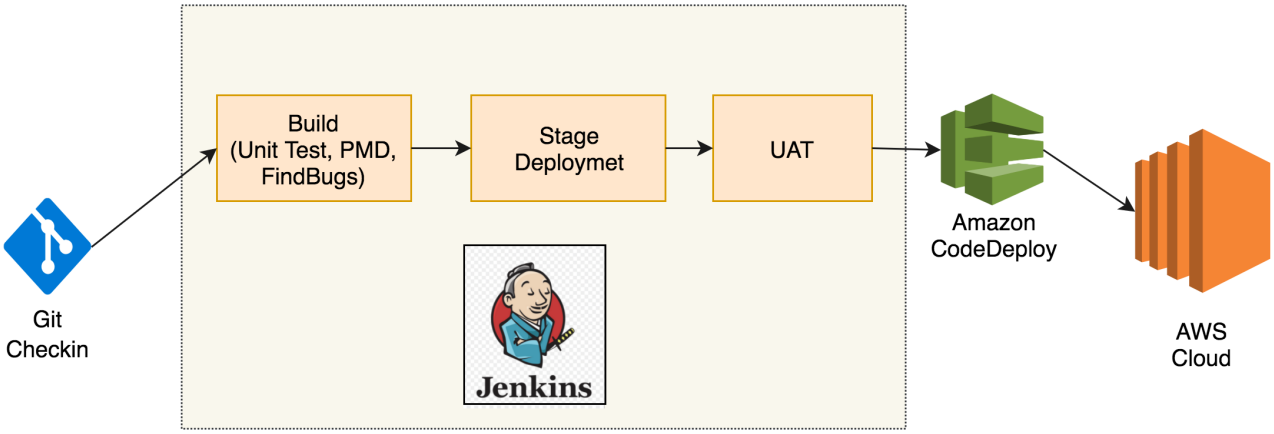
In this methodology Development, Test, Support and Operations work together as one delivery team to automate the streamline the build-test-release process as much as possible.



Continuous Delivery Pipeline

By following continuous delivery approach, a single commit in git repository can trigger the automated delivery pipeline taking your code to production, making it available to customers.

If you are using Jenkins, AWS, and CodeDeploy, then the pipeline might look like below:



AWS Code Deploy pipeline

References

- https://www.thoughtworks.com/continuous-delivery
- https://en.wikipedia.org/wiki/Continuous_delivery

3.89. How will you improve the performance of distributed system?

Introduce Caching at different layers

Techniques like Caching (client side caching, server side caching, proxy caching) can help improving the query performance of microservices. Asynchronous Integration can also help bringing performance and resiliency to the system. If hardware is under your control, then prefer to use optical fibre communication between microsevrices intranet.

Configuring threadpool size for servlet container

We need to correctly configure the number of threads for the underlying servlet container. Critical point to consider here is that hystrix when used as semaphore, will share threads with servlet container. Default value for hystix is 10 threads. We can configure number of worker threads in spring boot application using container specific properties defined in application.yml

Listing 55. application.yml

```
server:
  port: 9090

  jetty:
    acceptors: 2 (default = 1 + noCores/16)
    selectors: 8 (default = noCores)

  undertow:
    io-threads: 4 (default = noCores/2)
    worker-threads: 40 (default = 10)

  tomcat:
    max-threads: 40
```

selector and acceptors in Jetty

Every time a socket connects, the acceptor threads accepts the connection and assign the socket to a selector, chosen in a round robin fashion. The higher the connection open/close rate, higher the number of acceptor threads you need. For many many connections, very very active, you want a high number of selectors. For many many connections, but not very active, fewer selectors are typically needed.

For 5k clients for normal HTML pages you can easily go by with 1 selector.

You can refer to common properties in spring boot documentation

https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html

HTTP2 Protocol

HTTP2 protocol can offer major performance improvements by reducing the amount of traffic over the network due to compression of header and body. Moreover http2 protocol reduces the number of round trips required by client to load multiple resources from single host.

3.90. How will you implement caching for microservices?

Caching is a technique of performance improvement for getting query results from a service. It helps minimize the calls to network, database, etc. We can use caching at multiple levels in microservices architecture -

1. Server Side Caching - Distributed caching softwares like Redis/MemCache/etc are used to cache the results of business operations. The cache is distributed so all instances of a microservice can see the values from shared cache. This type of caching is opaque to clients.
2. Gateway Cache - central API gateway can cache the query results as per business needs and provide an improved performance. This way we can achieve caching for multiple services at one place. Distributed caching software like Redis or Memcache can be used in this case.

3. Client Side Caching - We can set cache-headers in http response and allow clients to cache the results for pre-defined time. This will drastically reduce load on servers since client will not make repeated calls to same resource. Servers can inform the clients when information is changed, thereby any changes in the query result can also be handled. E-Tags can be used for client side load balancing. If the end client is a microservice itself, then Spring Cache support can be used to cache the results locally.

3.91. Which protocol is generally used for client to service and inter-service communication?

client to service communication

Generally the preferred communication protocol is REST over HTTPS. If client is Android based, then libraries like OkHttp,HttpClient etc. are used for HTTP communication and Retrofit/RestTemplate for making REST calls.

inter-service communication

Preferred protocol for inter-service communication in microservices is generally AMQP(Advanced Message Queueing Protocol). AMQP using RabbitMQ is currently the most popular choice for messaging within a distributed system based on microservices architecture, mostly because of its platform independent nature. Amazon SQS, Kafka are few other options worth consideration. REST over HTTP can also be used for synchronous communication among microservices.

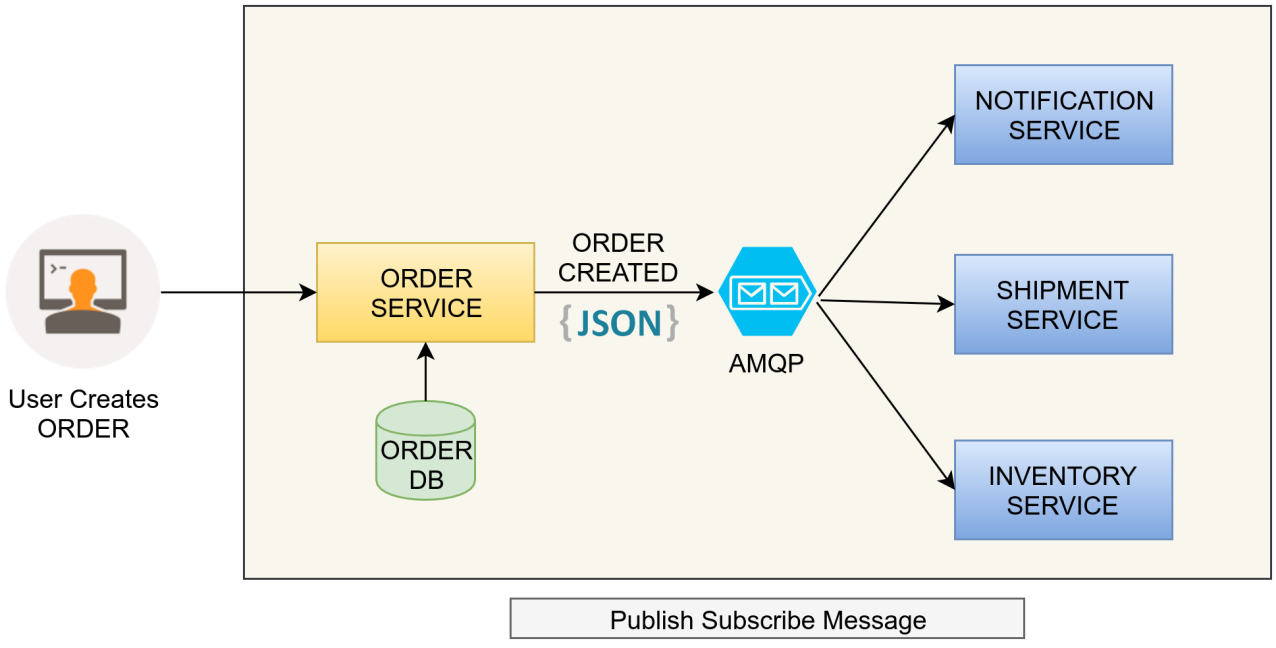
3.92. What are advantages of using asynchronous messaging within microservices architecture?

Asynchronous communication (using AMQP/JMS/Kafka, etc.) not only increases performance of the system but also increases reliability of the system.

- Performance is increased because the caller will just fire and forget the request, it does not need to wait for the response from other service thereby consuming lesser resources (threads and network traffic)
- Reliability of the system is increased due to guaranteed delivery of message to the consumer service. The caller service does not need to set the timeout values for the REST calls to another service and handle retry logic. Even circuit breaker pattern becomes an optional thing if you are using asynchronous communication.

Another powerful advantage of messaging that is not available if you are using REST communication is the capability to broadcast a message to multiple services in one go a.k.a publish and subscribe messaging. Publish and Subscribe usually involves *Fanout* or *Topic Exchange* and *Subscribers*. In event driven programming, producer just broadcast the message to a topic without knowing about the consumers and the way message will be consumed by consumers.

For example, in Financial system, a message Producer may broadcast a message that Google Alphabet Inc. Stock (GOOG) has been split. The message producer's responsibility is only to publish the message to a topic, any number of consumers can consume the message and take appropriate action without producer know about it at all.



CREATE_ORDER Event - Publish & Subscribe

In the example shown above, as soon as user creates an order, OREDR SERVICE will generate an CREATE_ORDER event and publish it onto AMQP topic. Now all the interested consumers (notification service, inventory service and shipment service) will start consuming this message in asynchronous fashion. Here the Order Service does not need to know the action that all consumer

services will take on CREATE_ORDER event. Notification service will send SMS/Email notification for the order, Inventory Service will adjust inventory for purchased items, and shipment service will start tracking for the order.

Important takeaway

CREATE_ORDER event carries information in JSON/Binary format about the newly created order e.g. order no, order amount, order items, customer information. It does not tell what action to perform on this event. Its responsibility of consumer application to perform an action on event, producer need not be aware of this at all.

3.93. What is good tool for documenting Microservices?

Swagger is a very good open source tool for documenting REST based APIs provided by microservices. It provides very easy to use interactive documentation.

By the use of swagger annotation on REST endpoint, api documentation can be auto-generated and exposed over the web interface. Internal and external team can use web interface, to see the list of APIs and their inputs & error codes. They can even invoke the endpoints directly from web interface to get the results.

Swagger UI is a very powerful tool for your microservices consumers to help them understand set of endpoints provided by a given microservice.

3.94. How will you integrate Swagger into your microservices?

Integrating swagger into Spring Boot based application should be straight forward. You need to add swagger dependencies into `build.gradle`, provide swagger configuration and finally make some tweaks into `WebMvcConfig` to allow swagger-ui into your project.

Listing 56. build.gradle - add swagger dependencies

```
dependencies {
    compile('org.springframework.cloud:spring-cloud-starter-config')

    // https://mvnrepository.com/artifact/io.springfox/springfox-swagger2
    compile group: 'io.springfox', name: 'springfox-swagger2', version: '2.8.0'
    compile group: 'io.springfox', name: 'springfox-swagger-ui', version: '2.8.0'
```

Second step is to define swagger configuration:

Listing 57. SwaggerConfig.java

```
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.context.annotation.*;
import springfox.documentation.builders.*;
import springfox.documentation.service.*;

@Configuration
@EnableSwagger2
@EnableAutoConfiguration
public class SwaggerConfig {

    @Bean
    public Docket productApi() {
        return new Docket(DocumentationType.SWAGGER_2)
                .groupName("Product Service")
                .apiInfo(apiInfo())
                .select()
                .apis(RequestHandlerSelectors.basePackage("hello"))
                .paths(PathSelectors.any())
                .build();
    }

    private ApiInfo apiInfo() {
        return new ApiInfoBuilder()
                .title("Product Service with Swagger")
                .description("Spring REST Sample with Swagger")
                .termsOfServiceUrl("http://www-03.ibm.com/software/sla/sladb.nsf/sla/bm?Open")
                .contact(new Contact("Munish Chandel", "","munish.chandel@outlook.com"))
                .license("Apache License Version 2.0")
                .licenseUrl("https://github.com/IBM-Bluemix/news-aggregator/blob/master/LICENSE")
                .version("1.0")
                .build();
    }
}
```

Lastly, add the below WebMvcConfig to enable swagger UI

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.ResourceHandlerRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;

@Configuration
public class WebMvcConfig extends WebMvcConfigurerAdapter {

    private static final Logger logger = LoggerFactory.getLogger(WebMvcConfig.class);

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        super.addResourceHandlers(registry);
        registry.addResourceHandler("swagger-ui.html")
                .addResourceLocations("classpath:/META-INF/resources/");
        /*registry.addResourceHandler("/webjars/**")
                .addResourceLocations("classpath:/META-INF/resources/webjars/");*/
    }

}
```

Now swagger is configured for use in your application.

3.95. What are common properties for a Spring Boot project?

You can configure spring boot application properties in your `application.properties` /`application.yml` file. The reference common properties are listed in the below link:

Common application properties

https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html

You can pick the one that are needed in your project and customize them according to your business/technical needs.

Chapter 4. Security in Microservices

4.1. Why Basic Authentication is not suitable in Microservices Context?

Basic Authentication is natively supported by almost all servers and clients, even Spring security has very good support for it and its configured out of the box. But it is not a good fit for Microservices due to many reasons, including -

1. We need credentials (username and password) every time we authenticate. This may be fine where all the participants can share the secrets securely, but Users may not be willing to share their credentials with all the applications.
2. There is no distinction between Users and Client Apps (application that is making request). In a realistic environment, we often need to know if a real user is making request or a client app is making request (for inter service communication).
3. It only covers authentication. what about scopes, Authorizations? Basic Auth does not support adding additional attributes in the authentication headers. There is no concept of Tokens in basic auth.
4. Performance reasons for BCrypt Matching. Passwords are often stored in database using one way hash i.e. Bcrypt, it takes lot of cpu cycles dependening upon the strength (a.k.a. log rounds in BCrypt) to compare the user's plain password with db saved bcrypt password, so it may not be a efficient to match password on every request. The larger the strength parameter the more work will have to be done (exponentially) to hash the passwords. If you set strength to 12, then in total 2^{12} iterations will be done in Bcrypt Logic. Usually 4-8 passwords can be matched per second on a T2.Micro instance on Amazon AWS instance. See BCryptPasswordEncoder for more info.

   https://docs.spring.io/spring-security/site/docs/4.0.4.RELEASE/apidocs/org/springframework/security/crypto/bcrypt/BCryptPasswordEncoder.html

5. If we use Basic Auth for a mobile application client, then we might have to store user's credentials on the device to allow remember me feature. This is quite risky as anyone getting access to device may steal the plain credentials.

OAuth2.0 - defacto in Spring Based Microservices

The choice of authentication in Spring based microservices is to use OAuth2.0 with JWT (Json Web Token, pronounced as jot)

4.2. Why OAuth2?

1. Simple for Clients (client is often a microservice itself)
2. AcessTokens can carry information beyond identity (like clientId, roles, issuer, userid, resourceId, expiry, scope, etc.).
3. Distributed and Stateless. You can validate the token's authenticity yourself.

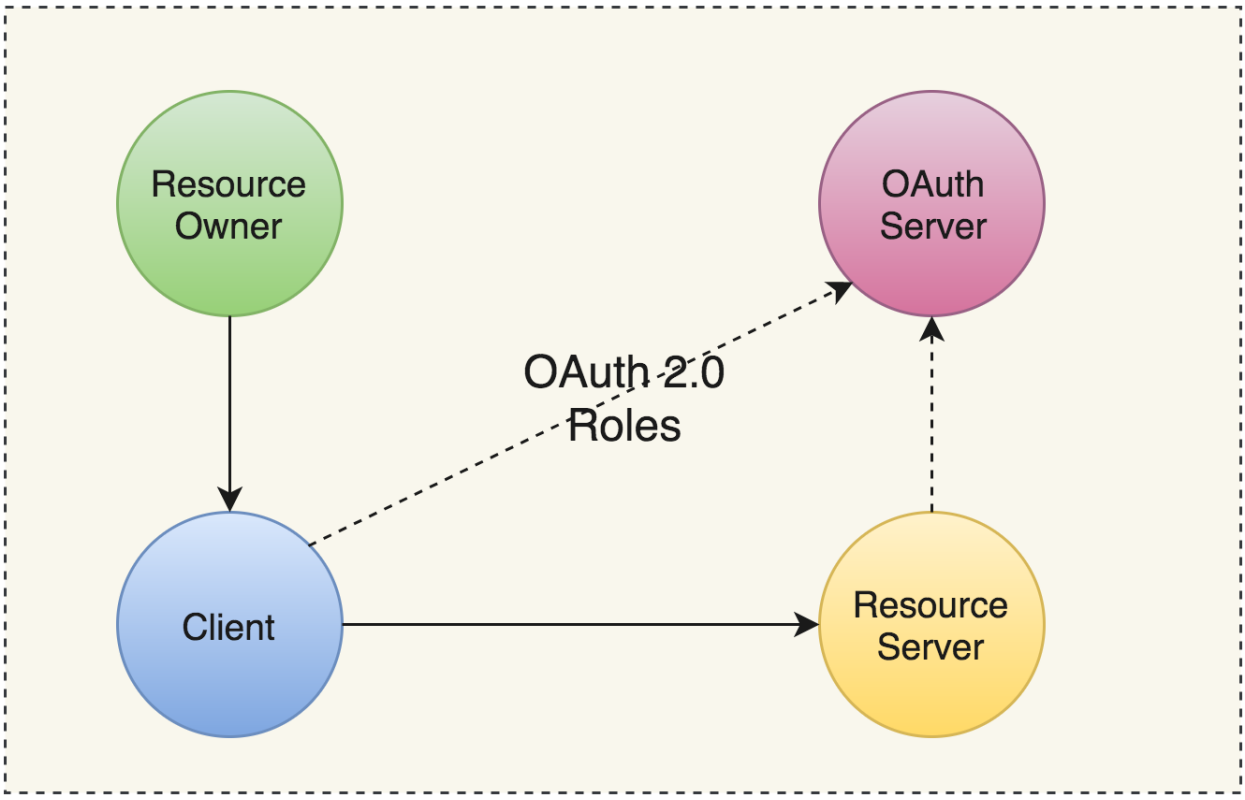
4. Refresh Token support
5. Clear distinction between user and machines.
6. Lightweight
7. A client application, often web application, acts on behalf of user, but with user's approval.
8. Interoperability with non-browser clients.

4.3. How OAuth2 Works?

Attach Power Point slides for OAuth2 Security.

OAuth2.0 is a delegation protocol where the Client (Mobile App or web app) does not need to know about the credentials of Resource Owner (end user).

4.4. What are different OAuth2 Roles?



Four different roles in OAuth 2.0 protocol

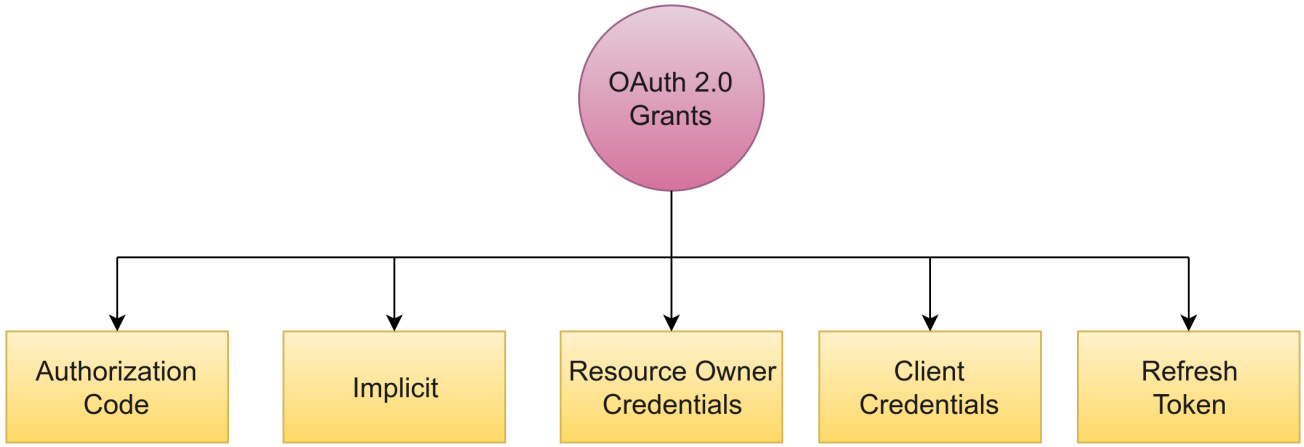
Oauth2 defines four roles.

1. `Resource Owner` - The person or the application that owns the data to be shared. When resource owner is a person, it is called as an end-user.
2. `Resource Server` - The application that holds the protected resources. It is usually a microservice.
3. `Authorization Server` - the application that verifies the identity of the resource owner (users/clients). This server issues access tokens after obtaining the authorization.

4. `Client` - the application that makes request to Resource Server on behalf of Resource Owner. It could be a mobile app or a web app (like stackoverflow).

4.5. What are different OAuth 2.0 grant types (OAuth flows)?

An authorization grant is a credential representing the resource owner's authorization (to access its protected resources) used by the client to obtain an access token.



Five Grants in OAuth 2.0 protocol

OAuth2 specification defines four grant types.

1. `Authorization Code` - its mostly used by web applications.
2. `Resource Owner Password Credentials` - It is used mostly by trusted clients like Android/Mobile Apps.
3. `Implicit` - Angular JS Applications and Mobile Apps where clientId and clientSecret can not be kept secret.
4. `Client Credentials` - mostly used for inter service communication by service clients.
5. `Refresh Token` - Used for generating a refresh token

4.6. When shall I use resource owner credentials?

When end user is a human, then resource resource owner credentials grant should be used. Important thing to note here is that resource owner's credentials will be exposed to the client application. Thus it should only be used where client app is trusted application. For example, if you have your own inhouse oauth 2.0 server, then you can use resource owner credentials in your company's android app.

But if you are integrating with Google or Facebook, then this type of grant is not feasible, because end user will never trust your android app to enter Google's credentials. Authorization Code grant is better alternative to such scenarios.

Listing 58. Using Curl to get Access Token based on `password` grant_type.

```
curl https://clientId:clientSecret@api.host.com/uaa/oauth/token -d grant_type=password -d username=<username> -d
password=<password> -d scope=openid
```

Listing 59. Sample Response

```
{
    "access_token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1aWQiOiI4NTkyM2RkZS0yNDIzLTRiYTMtOGNhOS1hODMzZjZiODBkODMiLCJhdWQiOlsiYWxsLWJyZWF
0aGUtcmVzb3VyY2VzIl0sInVzZXJfbmFtZSI6InBhdGllbnQxQG1haWwuY29tIiwic2NvcGUiOlsicmVhZCJdLCJleHAiOjE1MTk5MzgzNzQsImF1dGhvcml0
aWVzIjpbIlJPTEVfUEFUSUVOVCJdLCJqdGkiOiI5MTE0MjcwYy1lYTRlLTQ4MTQtYTViMi0zZjNlOTlkYzQyMzMiLCJjbGllbnRfaWQiOiJicmVhdGhlLW1vY
mlsZS1hcHAifQ.vL1nWBbHzSfoPaGglmHTG-ID8JmkdKznPKixsL2tCPQ",
    "expires_in": 2628000,
    "jti": "9114270c-ea4e-4814-a5b2-3f3e99dc4233",
    "refresh_token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1aWQiOiI4NTkyM2RkZS0yNDIzLTRiYTMtOGNhOS1hODMzZjZiODBkODMiLCJhdWQiOlsiYWxsLWJyZWF
0aGUtcmVzb3VyY2VzIl0sInVzZXJfbmFtZSI6InBhdGllbnQxQG1haWwuY29tIiwic2NvcGUiOlsicmVhZCJdLCJhdGkiOiI5MTE0MjcwYy1lYTRlLTQ4MTQt
YTViMi0zZjNlOTlkYzQyMzMiLCJleHAiOjE1NDg4Njc5NzQsImF1dGhvcml0aWVzIjpbIlJPTEVfUEFUSUVOVCJdLCJqdGkiOiJmNjdiNWY1OC1jMjIyLTQ0Y
TMtODhjYS1kZGI2OTI2OTZkNDgiLCJjbGllbnRfaWQiOiJicmVhdGhlLW1vYmlsZS1hcHAifQ.b1G-6VNF-zCJ-dc2fW4tiaqssQBowIMhGoUBBzBgw-0",
    "scope": "read",
    "token_type": "bearer",
    "uid": "85923dde-2423-4ba3-8ca9-a833f6b80d83"
}
```

4.7. When shall I use Authorization Code grant?

It should be used in applications where clientId and clientSecret can be kept securely i.e. webapps. In this flow the oauth 2.0 client is redirected back to authorization server's url for authentication purpose.

This grant type is used when you want to integrate with Google/Facebook on your server side webapp.

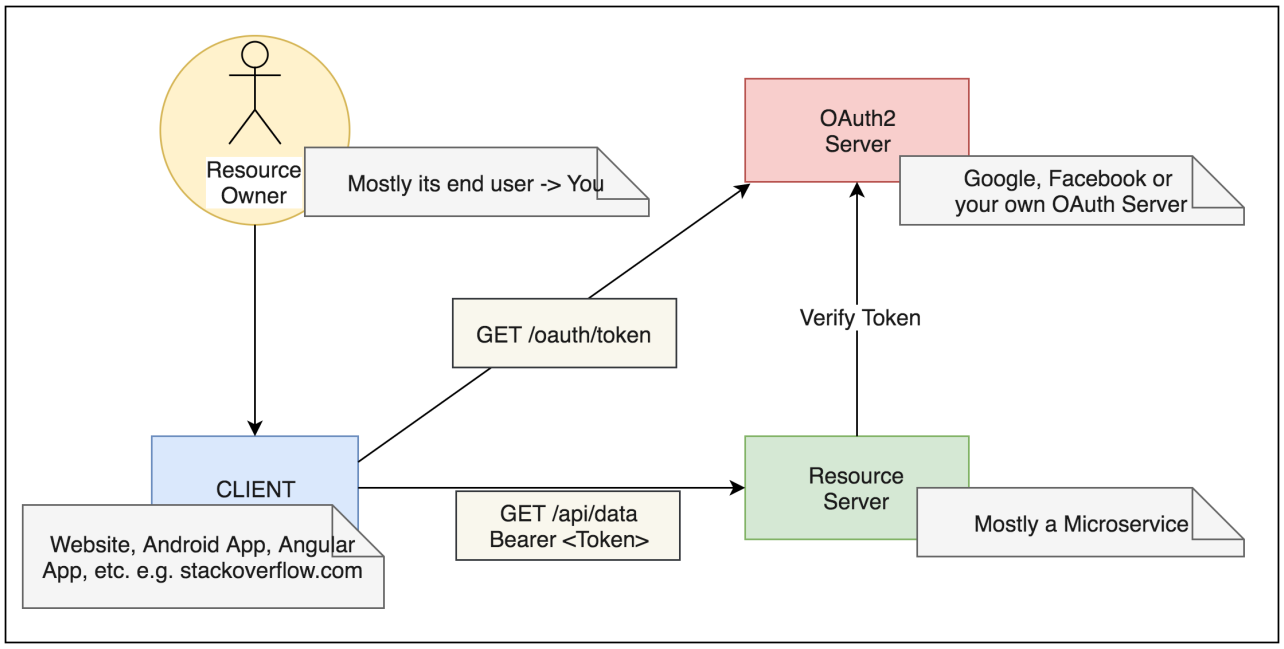
4.8. When shall I use client credentials?

Client credentials should be used for inter service communication that is not on behalf of users i.e. scheduled batch jobs, reporting jobs etc. Unlike Basic Auth, OAuth 2.0 protocol distinguishes between User (`Resource Owner`) and Machines (`Client`) based on Resource Owner Credentials and Client Credentials. Thus if the end consumer of your web services is not a human, then client credentials should be used.

Using Client Credentials to generate Access Token

```
curl service-account-1:service-account-1-secret@localhost:8080/auth/oauth/token -d grant_type=client_credentials
```

4.9. OAuth2 and Microservices



OAuth2 Use in Microservices Context

1. Resource Servers are often microservices.
2. Web App Clients uses Authorization Code Grant.
3. Browser Clients (single page apps - angular JS, etc) uses Authorization Code Grant or implicit Grant.
4. Mobile App and non-browser clients uses - password grant.
5. Service Clients (intra-system) uses - client credentials or relay user tokens depending upon the requirements.

4.10. What is JWT?

JWT is acronym for JSON Web Token. JWT are an open, industry standard RFC 7519 method for representing claims securely between two parties (even on unsecured networks).

JSON Web Token specification

JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA.

Sources

- https://tools.ietf.org/html/rfc7519
- https://jwt.io/introduction/

There are two important points here -

1. JWT are compact so they can be sent through a URL, preferrably as a POST paramater or inside HTTP header.
2. JWT contains all the required data about the user, so there is no need to query the database more than once (once during JWT generation).
3. Because JWTs can be signed—for example, using public/private key pairs—you can be sure the senders are who they say they are.

4.11. What are usecases for JWT?

There are many useful scenarios for leveraging power of JWT-

Authentication

Authentication is one of the most common scenario for using JWT, specifically in microservices architecture (but not limited to it). In microservices, oauth2 server generates a JWT at the time of login and all subsequent requests can include the JWT AccessToken as the means for authentication.

Implementing Single Sign On by sharing JWT b/w different applications hosted in different domains.

Information Exchange

JWT can be signed, using public/private key pairs, you can be sure that the senders are who they say they are. Hence JWT is a good way of sharing information between two parties. Example usecase could be -

1. Generating Single Click Action Emails e.g. Activate your account, delete this comment, add this item to favorites, Reset your password, etc. All required information for the action can

be put into JWT.

2. Timed sharing of a file download using a JWT link. Timestamp can be part of claim, so when the server time is past the time coded in JWT, link will automatically expire.

4.12. How does JWT looks like?

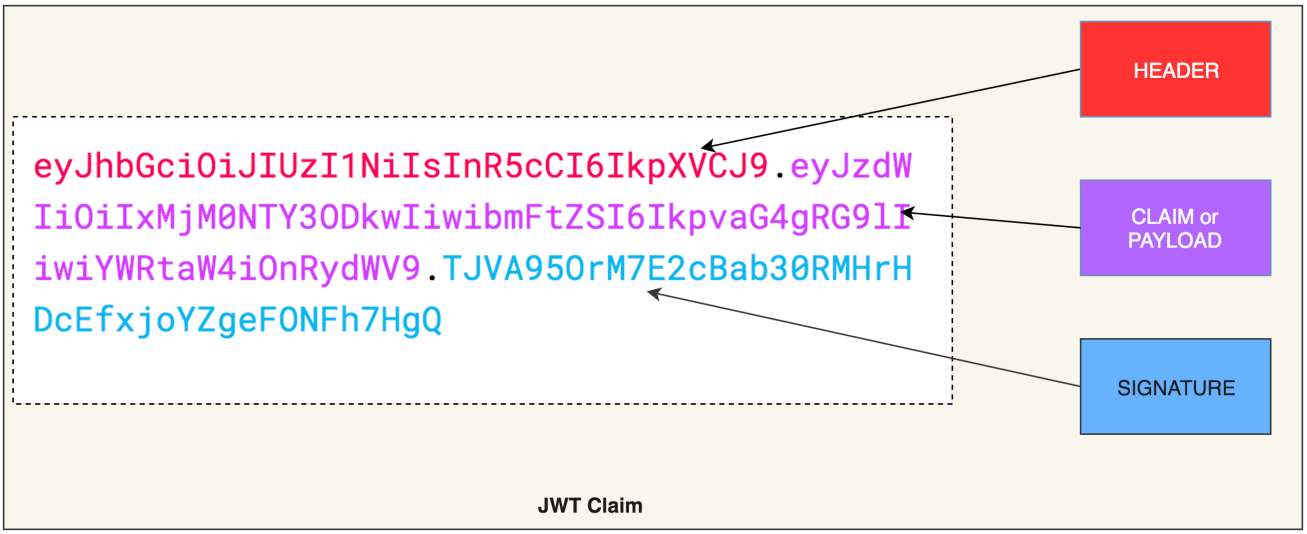
There are 3 parts in every JWT claim - Header, Claim and Signature. These 3 parts are separated by a dot. The entire JWT is encoded in `Base64` format.

```
JWT = {header}.{payload}.{signature}
```

A typical JWT is shown here for reference.

Encoded JSON Web Token

Entire JWT is encoded in Base64 format to make it compatibel with HTTP protocol. Encoded JWT looks like the following:



Encoded JWT Claim on https://jwt.io/

Decoded JSON Web Token

Header

Header contains algorithm information e.g. `HS256` and type e.g. `JWT`

Listing 60. Header Part

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Claim

claim part has expiry, issuer, user_id, scope, roles, client_id etc. It is encoded as a JSON object. You can add custom attributes to the claim. This is the information that you want to exchange with the third party.

Listing 61. Claim Part

```
{
  "uid": "2ce35360-ef8e-4f69-a8d7-b5d1aec78759",
  "user_name": "user@mail.com",
  "scope": ["read"],
  "exp": 1520017228,
  "authorities": ["ROLE_USER","ROLE_ADMIN"],
  "jti": "5b42ca29-8b61-4a3a-8502-53c21e85a117",
  "client_id": "acme-app"
}
```

Signature

Signature is typically a one way hash of (header + payload), is calculated using HMAC SHA256 algorithm. The secret used for signing the claim should be kept private. Pubic/private key can also be used to encrypt the claim instead of using symmetric cryptography.

Listing 62. Signature Part

```
HMACSHA256(base64(header) + "." + base64(payload), "secret")
```

Test & debug JWT claims

You can decode and test JWT using https://jwt.io

This is helpful for testing and debugging purpose.

4.13. What is AccessToken and RefreshToken?

In OAuth2, we receive two kind of tokens against authentication - AccessToken[mandatory] and RefreshToken[optional].

JWT Access token is used to authenticate against protected API resources.

JWT Refresh token is used to acquire new Access Token. Token refresh is handled by the following API endpoint: `/api/auth/token`. Refresh token should be used when existing accessToken has expired its validity.

Generate Access Token using Curl

```
curl trusted:secret@localhost:8081/oauth/token -d grant_type=password -d username=user -d password=password
```

Listing 63. Sample AccessToken Response


```
{
    "access_token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1aWQiOiIyY2UzNTM2MC1lZjhlLTRmNjktYThkNy1iNWQxYWVjNzg3NTkiLCJhdWQiOlsiYWxsLWJyZWF
0aGUtcmVzb3VyY2VzIl0sInVzZXJfbmFtZSI6ImFkbWluMUBtYWlsLmNvbSIsInNjb3BlIjpbInJlYWQiXSwiZXhwIjoxNTIwMDE3MjI4LCJhdXRob3JpdGll
cyI6WyJST0xFX0NST1NTIiwiUk9MRV9BRE1JTiJdLCJqdGkiOiI1YjQyY2EyOS04YjYxLTRhM2EtODUwMi01M2MyMWU4NWExMTciLCJjbGllbnRfaWQiOiJic
mVhdGhlLW1vYmlsZS1hcHAifQ.S0EMsoSFnMmDK7K6JZY5ldwKlTgXi2tNDOInE3wQlGk",
    "expires_in": 2627999,
    "jti": "5b42ca29-8b61-4a3a-8502-53c21e85a117",
    "refresh_token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1aWQiOiIyY2UzNTM2MC1lZjhlLTRmNjktYThkNy1iNWQxYWVjNzg3NTkiLCJhdWQiOlsiYWxsLWJyZWF
0aGUtcmVzb3VyY2VzIl0sInVzZXJfbmFtZSI6ImFkbWluMUBtYWlsLmNvbSIsInNjb3BlIjpbInJlYWQiXSwiYXRpIjoiNWI0MmNhMjktOGI2MS00YTNhLTg1
MDItNTNjMjFlODVhMTE3IiwiZXhwIjoxNTQ4OTQ2ODI4LCJhdXRob3JpdGllcyI6WyJST0xFX0NST1NTIiwiUk9MRV9BRE1JTiJdLCJqdGkiOiJlNmM5MTY5M
S01ZDViLTQ5NDctOTRlOS00ZjY2ZTQ0YzQ2MDAiLCJjbGllbnRfaWQiOiJicmVhdGhlLW1vYmlsZS1hcHAifQ.aEA7_yL21ykLia1RMS01JSk6gktBAtrP7ri
rY9u6VcY",
    "scope": "read",
    "token_type": "bearer",
    "uid": "2ce35360-ef8e-4f69-a8d7-b5d1aec78759"
}
```


4.14. How to use a RefreshToken to request a new AccessToken?

RefreshToken is used to create a new AccessToken once existing AccessToken is expired. The client requesting this operation must present its own clientId and clientSecret to oauth server to perform this operation.

Using curl to generate a new AccessToken using RefreshToken

Curl can be used at command line to generate the new AccessToken by presenting RefreshToken and client credentials to OAuth Server.

Request

```
curl <client-id>:<client-secret>@localhost:9999/uaa/oauth/token -d grant_type=refresh_token -d
refresh_token=$REFRESH_TOKEN
```

Listing 64. Response


```
{
    "access_token":"$ACCESS_TOKEN",
    "token_type":"bearer",
    "refresh_token":"$REFRESH_TOKEN",
    "expires_in":86399,
    "scope":"openid",
    "userId":1,
    "authorities":[ROLE_USER],
    "jti":"cd4ec2ad-ac88-4dd7-b937-18dd22e9410e"
}
```

Using RestTemplate to generate a new AccessToken using RefreshToken

Android Apps can use RestTemplate to renew their accessToken. The returned accessToken can be saved for subsequent calls to protected resources.

```
public OAuthTokenDto renewToken(final OAuthTokenDto existingToken) {
        String requestUrl = authBaseUrl + "oauth/token";
        MultiValueMap<String, String> map = new LinkedMultiValueMap<>();
        map.add("grant_type", "refresh_token");
        map.add("refresh_token", existingToken.getRefresh_token());
        map.add("scope", "read");
        HttpAuthentication authHeader = new HttpBasicAuthentication(CLIENT_ID, CLIENT_SECRET);
        HttpHeaders headers = new HttpHeaders();
        headers.setContentType(MediaType.APPLICATION_FORM_URLENCODED);
        headers.setAccept(Collections.singletonList(MediaType.APPLICATION_JSON));
        headers.setAuthorization(authHeader);
        HttpEntity<MultiValueMap<String, String>> entity = new HttpEntity<>(map, headers);
        RestTemplate restTemplate = new RestTemplate();
        final ResponseEntity<OAuthTokenDto> responseEntity =
                restTemplate.exchange(requestUrl, HttpMethod.POST, entity, OAuthTokenDto.class);
        return responseEntity.getBody();
    }
```

4.15. How to call the protected resource using AccessToken?

Protected resources from microservices can be accessed by passing Authorization Bearer token in request headers, using curl the request looks like below-

```
curl -H "Authorization: Bearer <AccessToken>" -v localhost:9090/user/info
```

Listing 65. JSON Response

```
{
    "key": "value"
    ...
}
```

The same call can be made using RestTemplate, Retrofit, FeignClient, RestAssured api or POSTMAN interface.

Calling Protected Resource using RestTemplate

Create HTTP headers that populates Bearer Token, like shown below:

```
public static HttpHeaders createHeaders() {
        final String access_token = $ACCESS_TOKEN; ①
        return new HttpHeaders() {
            {
                set("Authorization", "Bearer " + access_token);
            }
        };
    }
```

① `ACCESS_TOKEN` can be stored on Android Device SharedPreference and pciked up in this method.

And then use the above created Headers in RestTemplate, like shown below-

Listing 66. RestTemplate Call

```
HttpEntity<Object> httpEntity = new HttpEntity<>(ApplicationContext.createHeaders());

ResponseEntity<String> responseEntity =
        restTemplate.exchange(url, HttpMethod.GET, httpEntity, String.class, uriVariables);
```

Useful Links

- Downlaod POSTMAN

4.16. Can a refreshToken be never expiring? How to make refreshToken life long valid?

We can set the validity for refreshToken to a negative value, zero or `Integer.MAX_VALUE` to make it never expiring.

```
@Bean
public AuthorizationServerTokenServices tokenServices() throws Exception {
        DefaultTokenServices tokenServices = new DefaultTokenServices();
        tokenServices.setTokenStore(tokenStore);
        tokenServices.setSupportRefreshToken(true);
        tokenServices.setClientDetailsService(clientDetailsService);
        tokenServices.setRefreshTokenValiditySeconds(Integer.MAX_VALUE);
        return tokenServices;
}
```

It is not a good idea to make refreshToken never expiring due to security concerns. A better approach in this case could be to set `reuseRefreshTokens(false)` to create a new RefreshToken everytime a new AccessToken is created. This way, client credentials would still be required to generate the new refreshToken.

4.17. Generate AccessToken for Client Credentials.

Client Credentials can be used by clients (client applications or microservices, that are not human) for inter service communication.

Listing 67. Request using Curl

```
curl service-account-1:service-account-1-secret@localhost:8080/auth/oauth/token -d grant_type=client_credentials
```

Listing 68. Token Response

```
{
    "access_token":
"eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJzY29wZSI6WyJyZWFkIiwid3JpdGUiXSwiZXhwIjoxNTE4MjM1NzgzLCJhdXRob3JpdGllcyI6WyJST0x
FX1RSVVNURURfQ0xJRU5UIl0sImp0aSI6ImRhOTNjZTY3LWE5N2EtNGE2Ni04Nzc4LTQxNzUxOGM0YWEwOCIsImNsaWVudF9pZCI6InRydXN0ZWQifQ.dnuUZ
H4OlfP-MGm3b8gnXPrixFXFEpv4XWkCivlveHqOAUWGjK6_ZwhAsTfSIm4Ab5Bfy62KEyQ6cDpB8ENR85f5wcStGN8GWcPkMfet4TtU2PrgejHBWwZ9-
8vTRZY9JnL22m40YOatEMeuJ066Q2WJ7r7ywv_0r2d-kIu3X0M",
    "expires_in": 43199,
    "jti": "da93ce67-a97a-4a66-8778-417518c4aa08",
    "scope": "read write",
    "token_type": "bearer"
}
```

Now this token can be used for making protected endpoint call on resource server.

Using AccessToken to access protected resource

```
curl -H "Authorization: Bearer $TOKEN" -v http://localhost:8082/api
```

Where `$TOKEN` is the Access Token.

4.18. How to implement the Logout functionality using JWT?

In stateless security its impossible to invalidate the JWT accesstoken before they expire. So its not easy to implement logout funcitonality without side effects.

There are some alternatives that we can implement logout in browser and android/ios apps-

1. We can just throw away the access and refresh token from client's local storage upon logout.
2. We can use short lived AccessTokens paired with RefreshTokens.
3. We can implement stateful logout service that remembers either the JWT or a part of claim added in JWT.

First two approached used togather are easier to implement and are generally recommended.

4.19. Security in inter-service communication

There could be two usecases for inter service communication -

1. Calling one service from another on behalf of user request.
2. Using Service Client for internal service to service communication.

Token Relay

In first case, we shall propagate the security context of user from one service to another.

We can configure OAuth2RestTemplate to relay user token from service to service.

When to use Token Relay

This approach shall be used when a client (often a microservice) want to relay token to downstream service in order to access a protected resource on behalf of user.

Listing 69. A Load Balanced OAuth2RestTemplate that Relays Token

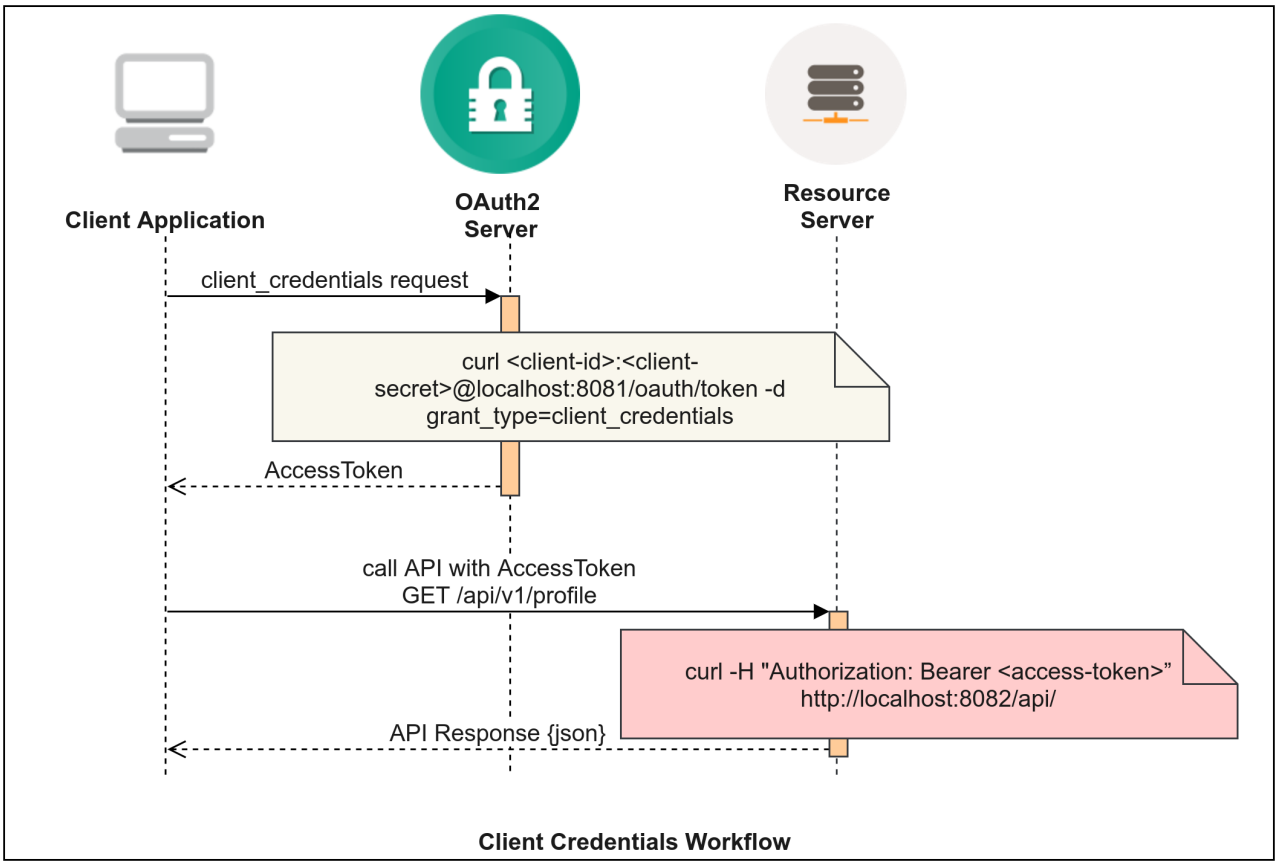
```
    @LoadBalanced
    @Bean
    @Autowired
    public OAuth2RestTemplate loadBalancedOauth2RestTemplate(OAuth2ClientContext oauth2ClientContext,
OAuth2ProtectedResourceDetails details) {
        return new OAuth2RestTemplate(details, oauth2ClientContext);
    }
```

Listing 70. Using OAuth2RestTemplate to make remote protected calls in another microservice.

```
@Service
public class RemoteMicroService {

    @Autowired
    private OAuth2RestTemplate oAuth2RestTemplate;

    //Use this oAuth2RestTemplate to make protected remote calls with user's token relay.
```

Client Credentials

In second case, we shall use Client Credentials issued by OAuth2 workflow for securing service to service communication. Here user's security context is not propogated to the downstream server, instead the client's credentials are used to secure the communication.



Client Crendtials Sequence Diagram

When to use Client Credentials

This approach shall mostly be used for scheduled jobs where we are not making remote service calls on behalf of end user.

Listing 71. Creating a RestTemplate based on Client Credentials

```
@Configuration
public class RestTemplateConfig {

    //Client Credentials based oAuth2 RestTemplate for Service Client Inter Microservice Communication.
    @Bean(name = "oauthRestTemplate")
    public RestTemplate oAuthRestTemplate() {
        ClientCredentialsResourceDetails resourceDetails = new ClientCredentialsResourceDetails();
        resourceDetails.setId("<Id>");
        resourceDetails.setClientId("<clientId>");
        resourceDetails.setClientSecret("<clientsecret>");
//      resourceDetails.setAccessTokenUri("<BaseUrl>/uaa/oauth/token");
        resourceDetails.setScope(Collections.singletonList("openid"));
        return new OAuth2RestTemplate(resourceDetails, new DefaultOAuth2ClientContext());
    }
```

This restTemplate can now be used for internal service communication, for example -

```
@Service
public class ProfileImageSyncService {
    private static final Logger logger = LoggerFactory.getLogger(ProfileImageSyncService.class);

    @Autowired
    @Qualifier("oauthRestTemplate")
    private RestTemplate restTemplate;

    public void doSomeRemoteWork() {
      ResponseEntity<String> responseEntity = restTemplate.exchange(targetUrl, HttpMethod.GET, null, String.class);
      ...Do something withr response.
      ...
    }
```

Do not use ResourceOwnerPasswordCredentials

We shall never use ResourceOwnerPasswordCredentials for internal service communication, thats a anti-pattern. Client Credentials should always be preferred when one service wants to talk to another service based on some scheduled job (not on behalf of user request), otherwise user's security token should be relayed to the next service if request is initiated by end user.

4.20. How to setup multiple authentications in Spring Security?

4.21. What is purpose of @EnableResourceServer?

`@EnableResourceServer` annotation marks your service (in terms of OAuth 2.0 roles) as a `Resource Server`. A Resource Server expects an access token in order to process the client request. Access token should be obtained from Authorization Server by OAuth 2.0 Client before calling the Resource Server.

Access token must be passed using Authorization Header in Http Request, like shown in below example -

Http Header

```
Authorization: Bearer <access token>
```

4.22. What is purpose of @EnableOAuth2Sso?

`@EnableOAuth2Sso` marks your service as an OAuth 2.0 client. This means that it will be responsible for redurecting the `Resource Owner` (end user) to the `Authorization Server` where the user has to enter the credentials. Once identity is proved, resource owner is redirected back to the client with Authorization Code. Then the Client takes the Authorization Code and exchanges it for an Access Token by calling Authorization Server. Only after that, the Client can make a call to a Resource Server with Access Token.

Normally a web app (like stackoverflow.com) is marked with @EnableOAuth2Sso.

4.23. What is purpose of @EnableOAuth2Client?

`@EnableOAuth2Client` annotation marks your application as OAuth 2.0 Client. It makes it possible to forward access token (after it has been exchanged for Authorization Code) to downstream services in case you are calling those services via OAuth2RestTemplate.

4.24. How can we add custom claims to JWT AccessToken?

We can add our own claims to JWT AccessToken progmatically using custom TokenEnhancer in OAuth2 Server.

Listing 72. Adding first and last name to Token

```
public class JwtTokenEnhancer implements TokenEnhancer {

    @Override
    public OAuth2AccessToken enhance(OAuth2AccessToken accessToken, OAuth2Authentication authentication) {
        if (authentication.getPrincipal() instanceof CustomUserDetails) {
            final CustomUserDetails user = (CustomUserDetails) authentication.getPrincipal();
            Map<String, Object> additionalInfo = new HashMap<>();
            additionalInfo.put("firstName", user.getFirstName());
            additionalInfo.put("lastName", user.getLastName());
            ((DefaultOAuth2AccessToken) accessToken).setAdditionalInformation(additionalInfo);
            return accessToken;
        } else {
            return accessToken;
        }
    }
}
```

Listing 73. Use this TokenEnhancer in OAuth2 server

```
@Configuration
@EnableOAuth2Client
@EnableAuthorizationServer
protected static class OAuth2AuthorizationConfig extends AuthorizationServerConfigurerAdapter {
    ...

    @Bean
    public TokenEnhancer tokenEnhancer() {
        return new JwtTokenEnhancer();
    }

    @Bean
    @Primary
    public DefaultTokenServices tokenServices() {
        DefaultTokenServices tokenServices = new DefaultTokenServices();
        tokenServices.setSupportRefreshToken(true);
        tokenServices.setReuseRefreshToken(true);
        tokenServices.setAccessTokenValiditySeconds(ONE_DAY * 3);
        tokenServices.setTokenStore(tokenStore());    ①
        tokenServices.setTokenEnhancer(tokenEnhancer());
        return tokenServices;
    }
    ...
}
```

① using JwtTokenEnhancer in tokenServices.

4.25. Security Best Practices

1. Add security at Service Layer

> In practice we recommend that you use method security at your service layer, to control access to your application, and do not rely entirely on the use of security constraints defined at the web-application level. URLs change and it is difficult to take account of all the possible URLs that an application might support and how requests might be manipulated. You should try and restrict yourself to using a few simple ant paths which are simple to understand. Always try to use a "deny-by-default" approach where you have a catch-all wildcard (/ or) defined last and denying access. Security defined at the service layer is much more robust and harder to bypass, so you should always take advantage of Spring Security's method security options.
>
> — Spring Security Docs

2. Default behaviour should be to authenticate each request and then selectively remove security from endpoints that needs to be public. Do not do otherwise.

3. Always use HTTPS. Basic auth over plain HTTP could prove to be disasterous.

4.26. How to enable spring security at service layer?

First of all we need to define a configuration that allows Spring Security at method level using annotations.

Listing 74. Enable Spring Security in Method Expressions

```
@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true, securedEnabled = true)
public class MethodSecurityConfig extends GlobalMethodSecurityConfiguration {

    @Override
    protected MethodSecurityExpressionHandler createExpressionHandler() {
        return new OAuth2MethodSecurityExpressionHandler();
    }
}
```

Now we can use the `@PreAuthorize` annotation to securely expose a method to certain Roles only. .Using Spring Security at Service Layer

```
import org.springframework.security.access.prepost.PostAuthorize;
import org.springframework.security.access.prepost.PreAuthorize;

import com.websystique.springsecurity.model.User;


public interface UserService {

    List<User> findAllUsers();

    @PostAuthorize ("returnObject.type == authentication.name")
    User findById(int id);

    @PreAuthorize("hasRole('ADMIN')")
    void updateUser(User user);

    @PreAuthorize("hasRole('ADMIN') AND hasRole('DBA')")    ①
    void deleteUser(int id);

}
```

① Only a user with `ROLE_ADMIN` or `ROLE_DBA` can access this service layer method.

Chapter 5. Testing Spring Boot based Microservices

Testing is as important as the Production Code itself.

The Outline

- Tools and Libraries available for testing.
- Why to do testing? 12 factor app example.
- Different aspects of Testing?
- Cohn's Test Pyramid
- Testing Strategies - Unit Testing, Integration Testing, Contract testing and End to End Testing.
- Testing in Production? Smoke Testing

5.1. Tools and Libraries available for testing

Important Tools and Libraries for testing Spring based Microservices are -

JUnit
: the standard test runners

TestNG
: the next generation test runner

Hemcrest
: declarative matchers and assertions

Rest-assured
: for writing REST Api driven end to end tests

Mockito
: for mocking dependencies

Wiremock
: for stubbing thirdparty services

Hoverfly
: Create API simulation for end-to-end tests.

Spring Test and Spring Boot Test
: for writing Spring Integration Tests - includes MockMVC, TestRestTemplate, Webclient like features.

JSONassert

An assertion library for JSON.

Pact

The Pact family of frameworks provide support for Consumer Driven Contracts testing.

Selenium

Selenium automates browsers. Its used for end-to-end automated ui testing.

Gradle

Gradle helps build, automate and deliver software, fastr.

IntelliJ IDEA

IDE for Java Development

Using spring-boot-starter-test

We can just add the below dependency in project's build.gradle

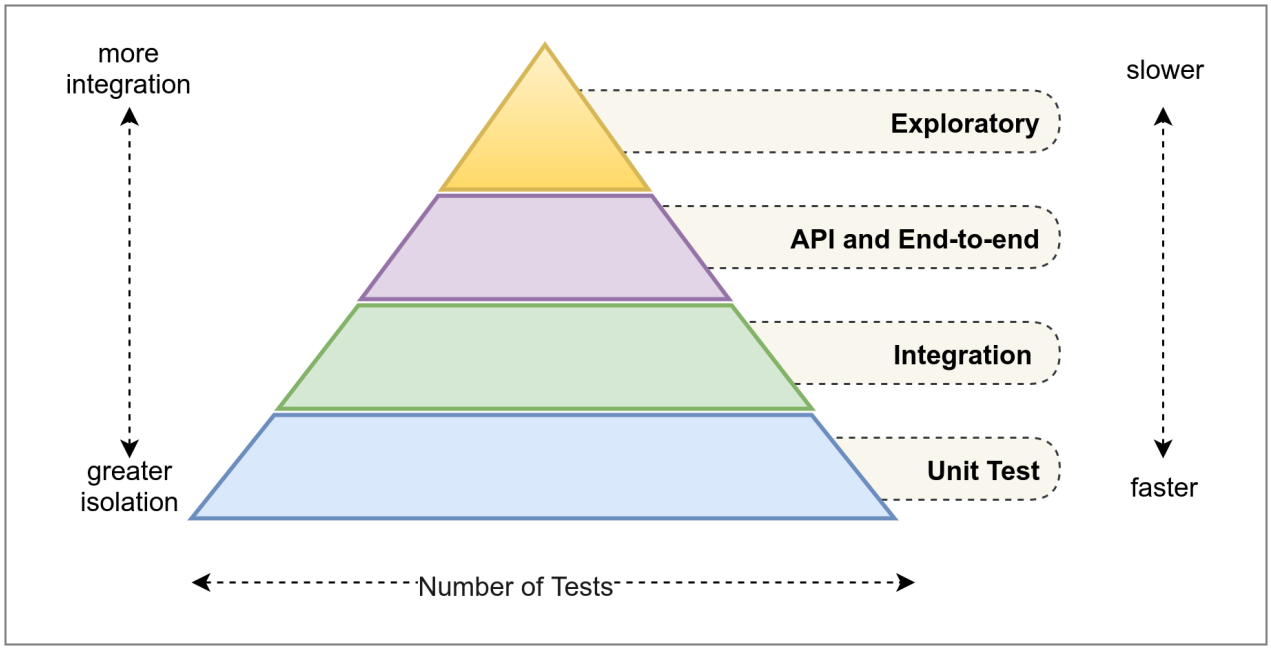
Listing 75. build.gradle

```
testCompile('org.springframework.boot:spring-boot-starter-test')
```

This starter will import two spring boot test modules `spring-boot-test` & `spring-boot-test-autoconfigure` as well as Junit, AssertJ, Hamcrest, Mockito, JSONassert, Spring Test, Spring Boot Test and a number of other useful libraries.

5.2. What is Mike Cohn's Test Pyramid?

Mike Cohn has provided a model called Test Pyramid that describes the kind of automated tests required for software development. There are four levels in Test Pyramid.



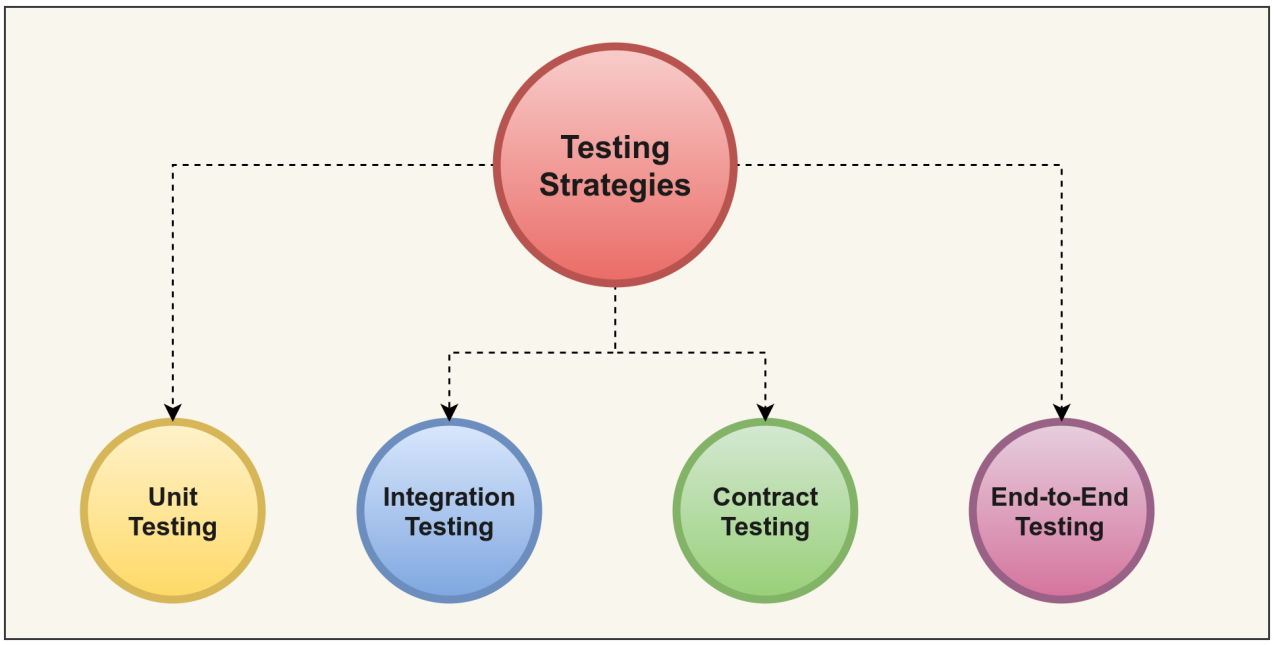
Mike Cohn Test Pyramid

1. First layer of Pyramid is Unit Tests. These are the result of Test Driven Development (TDD).
2. Second layer is Service tests. These tests are for testing the service directly under different inputs. These tests also verify the interface provided by a service against the client expectation.
3. Third layer is End-to-End tests that includes UI or API testing. These tests run against the entire system including UI, Front-end or third party clients. The purpose of these tests is to verify the actual production code.

Important Takeaways from Cohn's Test Pyramid:

1. We shall write tests with different granularity. Unit tests alone are not sufficient for any healthy system.
2. The more high level you get, fewer the tests you should have. The number of Unit tests should be highest in any application.

5.3. Testing Strategies



Testing Strategies in Microservices Architecture

1. Unit: a single component in isolation for its correctness. Junit, TestNG, Mockito.
2. Integration: An integration unit test focuses on testing the interaction between components in a "near real" environment. These tests will exercise the interactions with the container (embedded database, web container, and so on).
3. Contract Driven Tests: ensuring consistency across boundaries.
4. End-to-end: Verifies that a system meets external requirements and achieve its goals irrespective if the component architecture in use.

References, https://martinfowler.com/articles/microservice-testing

5.4. Mock vs Stub?

By writing mock, you discover the objects collaboration relationship by verifying that expectation are met, while stub only simulate the object's behavior. A Mock is just testing behaviour, making sure certain methods are called.

Mock

Mock is part of our unit tests and we explicitly setup the expectations on mocked object. These expectations may vary from test to test. A mock is not setup in a predetermined way so you have code that does it in your test. Mocks in a way are determined at runtime since the code that sets the expectations has to run before they do anything. Mock is also dummy implementation but its implementation done dynamic way by using mocking frameworks like Mockito.

For mocking every language provides some kind of framework. In Java we have Mockito framework available for creating mocks.

Stub

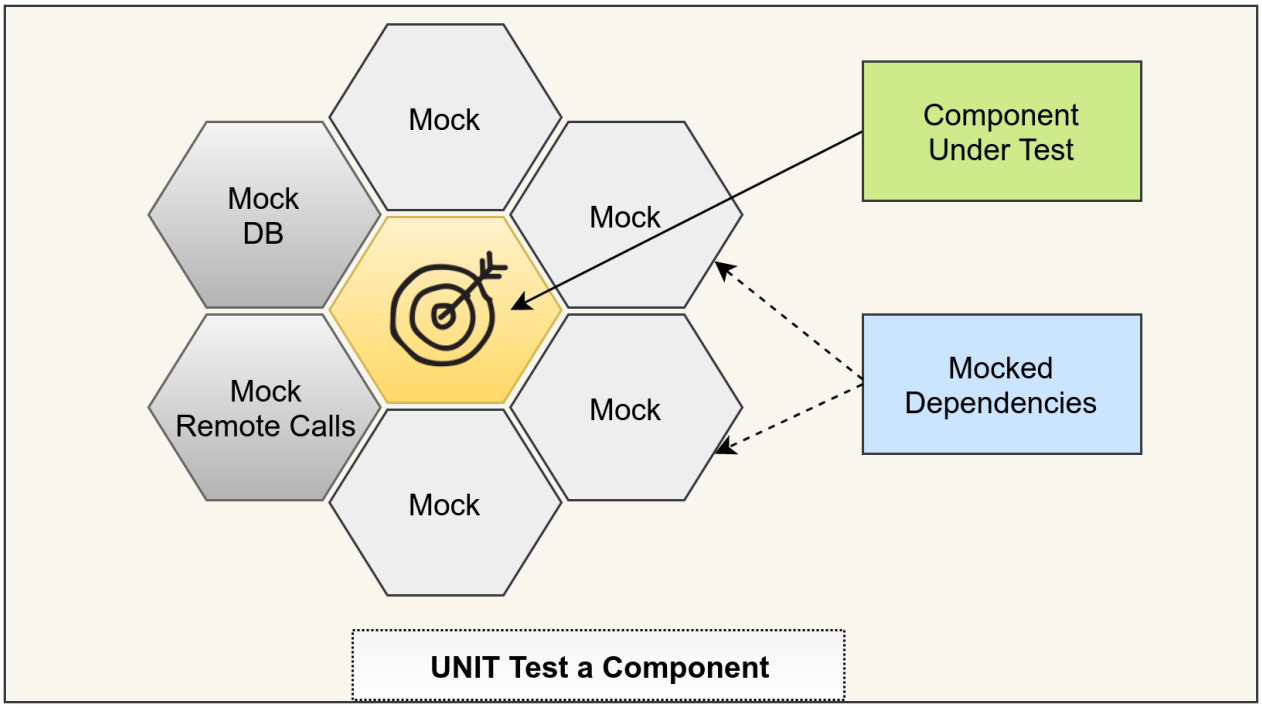
The biggest distinction is that a stub you have already written with predetermined behavior. So you would have a class that implements the dependency (abstract class or interface most likely) you are faking for testing purposes and the methods would just be stubbed out with set responses. They would not do anything fancy and you would have already written the stubbed code for it outside of your test. Stub is dummy implementation done by user in static way mean i.e in Stub writing the implementation code, normally this is done in Junit framework without mocking framework.

Reference

- https://martinfowler.com/articles/mocksArentStubs.html
- https://8thlight.com/blog/uncle-bob/2014/05/14/TheLittleMocker.html

5.5. Unit Testing

Unit tests are foundation of your test suite. The responsibility of unit test is to ensure that a certain unit (could be controller, service or utility class) works as intended. All the external dependencies are replaced with mocks in class under test. The following figure depcits the behavior of a unit test.



Unit Testing focus on component isolation

Few points that you should keep in mind while writing Unit Tests for Spring Boot based application:

- Unit Tests should not even load the Spring Context
- Service Layer and Utility classes are the ideal candidates for Unit Tests
- Unit Tests should focus on a single component under test, they shall mock every dependency that is injected into the component.
- Controller layer, DAO layer that does not add business logic is bad candidate for Unit Tests. They are more suited for Integration and End to End tests.

Isolation is important facet of Unit Tests

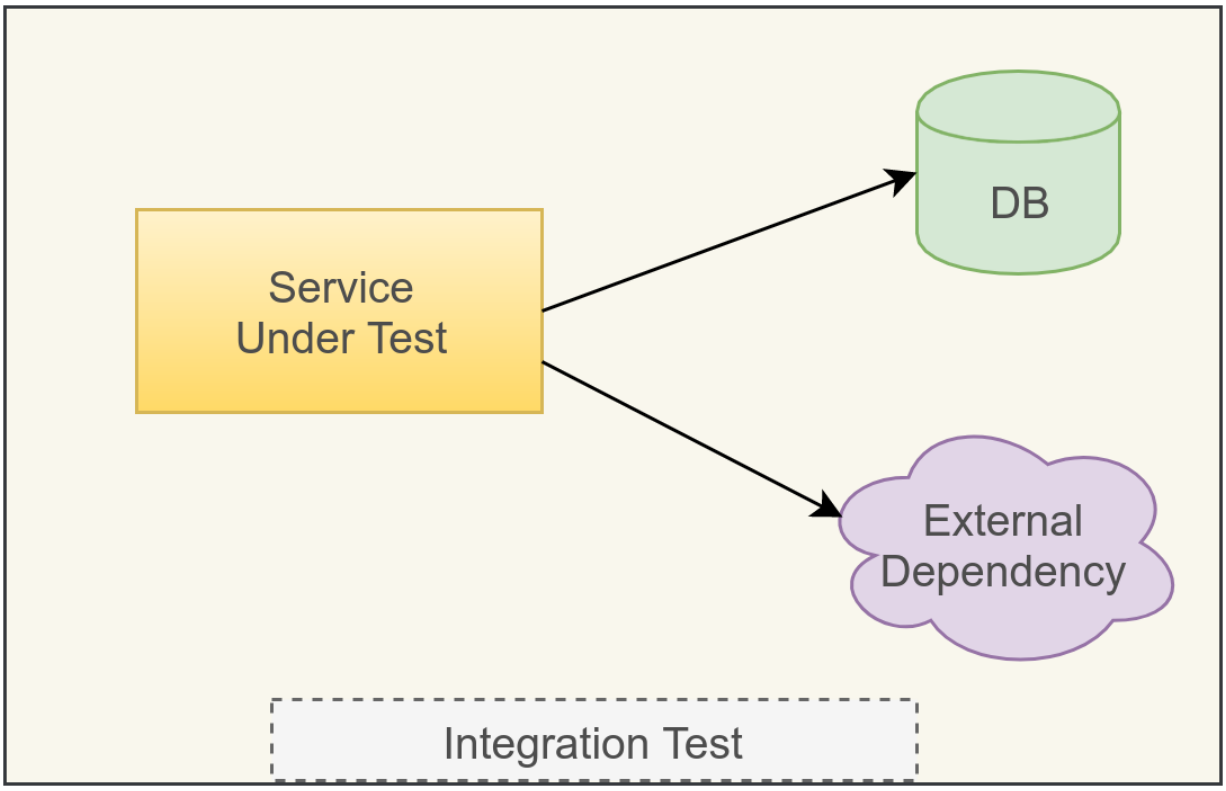
By testing the units in isolation by mocking/stubbing the dependencies, we make sure that only the class under test is being tested, not its dependencies!

Unit testing a controller

```
public class ExampleControllerTest {

    private ExampleController subject;

    @Mock
    private PersonRepository personRepo;

    @Before
    public void setUp() throws Exception {
        initMocks(this);
        subject = new ExampleController(personRepo);
    }

    @Test
    public void shouldReturnFullNameOfAPerson() throws Exception {
        Person peter = new Person("Peter", "Pan");
        given(personRepo.findByLastName("Pan"))
            .willReturn(Optional.of(peter));

        String greeting = subject.hello("Pan");

        assertThat(greeting, is("Hello Peter Pan!"));
    }

    @Test
    public void shouldTellIfPersonIsUnknown() throws Exception {
        given(personRepo.findByLastName(anyString()))
            .willReturn(Optional.empty());

        String greeting = subject.hello("Pan");

        assertThat(greeting, is("Who is this 'Pan' you're talking about?"));
    }
}
```

5.6. Integration Tests

Integration Tests are next higher level tests in the service. These tests ensure that our application can successfully integrate with database, network and filesystem, etc **with near realtime experience**. This also means that we have to bootstrap these dependent components along with our own service while running integration testcases. If we are using ORM, we can choose to run integration testcase using inmemory h2 database instead of a real RDBMS to reduce test execution time.



Integration Test checks that your application works well with a real database and other external dependencies

Integration Tests should be written for all places where our code interacts with database, filesystem, or network etc. Based on this we can categorize integration tests into different categories.

1. HTTP Integration Tests - tests that real REST call over HTTP hits your application successfully.
2. Database Integration Test - tests that you applications integrates with a real database successfully.
3. FileSystem integration Tests - checks that real filesystem integration works ok.
4. External Service Integration Tests - checks that your application can communicate to external service without issues.

5.7. Contract-Driven Tests

Consumer-Driven Contract Tests

Consumer driven contract tests are similar to Integration tests, except that they are shared among two teams. These tests ensure that both parties involved in an interface between provider and consumer service adhere to the shared contract. These tests will help both teams when they break the contract at very early stage.

Pact is an open source library to define producer and consumer side of CDC test using a language of your choice. Contract tests always include both sides of an interface — the consumer and the provider. Both parties need to write and run automated tests to ensure that their changes don't break the interface contract. <<< .Consumer-Driven Contract Tests image::cdc-test.png[]

Lets add pact dependencies to the Spring Boot Project.

Listing 76. build.gradle

```
    testCompile('au.com.dius:pact-jvm-consumer-junit_2.12:3.5.12')
    testCompile('au.com.dius:pact-jvm-provider-spring_2.12:3.5.12')
```

Now we can write the consumer side of CDC Test as follow:

Listing 77. Typical Consumer Driven Contract Test

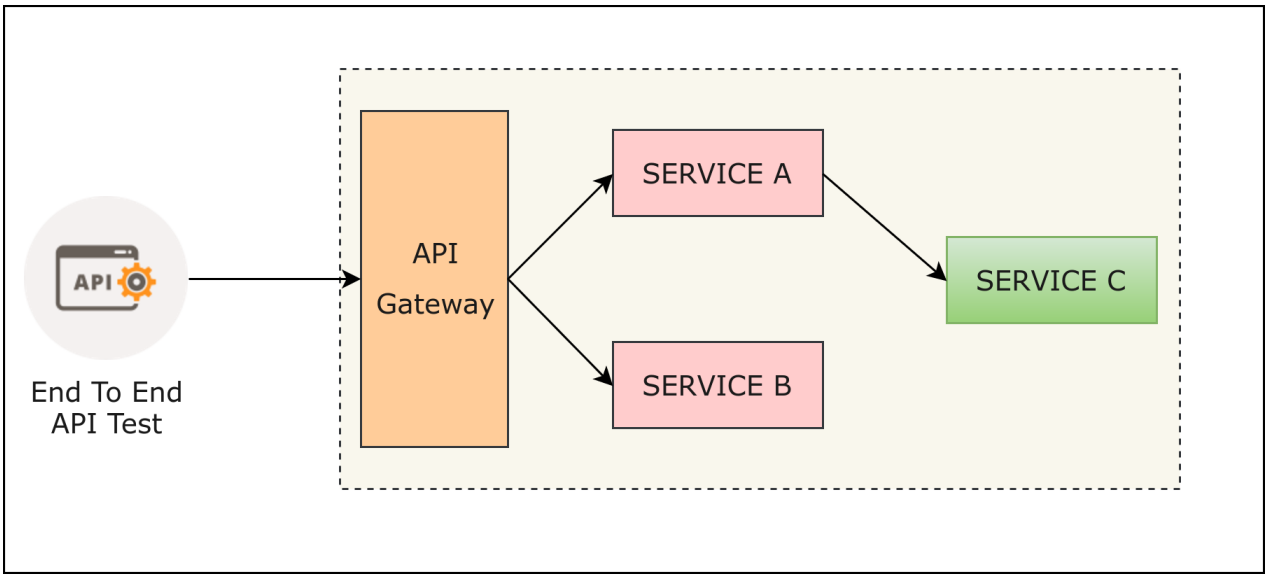
```
@RunWith(SpringRunner.class)
@SpringBootTest
public class GeoIpConsumerTest {

    @Autowired
    private GeoIpService subject;

    @Rule
    public PactProviderRuleMk2 weatherProvider = new PactProviderRuleMk2
            ("geoip_provider", "localhost", 9099, this);

    @Pact(consumer="sample_microservice")
    public RequestResponsePact createPact(PactDslWithProvider builder) throws IOException {
        return builder
                .given("geoip location data")
                .uponReceiving("a request for a geo ip location data")
                .path("/json/10.10.0.1")
                .method("GET")
                .willRespondWith()
                .status(200)
                .body(FileLoader.read("classpath:GeoIpApiResponse.json"), ContentType.APPLICATION_JSON)
                .toPact();
    }

    @Test
    @PactVerification("geoip_provider")
    public void shouldFetchGeoIpInformation() throws Exception {
        Optional<GeoIPDto> geoIpResponse = subject.getGeoIp("10.10.0.1");
        assertThat(geoIpResponse.isPresent(), is(true));
        assertThat(geoIpResponse.get().getCity(), is("Mohali"));
    }
}
```

consumer test generates a pact file (found in target/pacts/<pact-name>.json) each time it runs. This pact file describes our expectations for the contract in a special JSON format.

We can take this Pact JSON file and hand it over to the Team providing this interface.

5.8. End to End Tests

End to end testcase covers all the layers of system. These tests call our services through user interface. End-to-end testing treats the system as black box and testing is done on the public endpoints only (REST endpoints & GUI). System is generally deployed in a production like environment (have a database, communicates over REST/HTTP).



End to End Test covering entire system

We can use Selenium and Web Driver Protocol to run our end-to-end tests. First of all, we need to add selenium dependency in our build.gradle file

Listing 78. build.gradle

```
    //testCompile('org.seleniumhq.selenium:selenium-firefox-driver:3.9.3')
    testCompile('org.seleniumhq.selenium:selenium-chrome-driver:3.9.1')
    testCompile('org.seleniumhq.selenium:selenium-remote-driver:3.9.1')
    testCompile('org.seleniumhq.selenium:selenium-api:3.9.1')
```

Now we can create a test that opens chrome browser and calls our endpoint. Note that this test will only run on your system if you have Chrome/Firefox installed on the system you run this test on (your local machine, your CI server).

```
import io.github.bonigarcia.wdm.ChromeDriverManager;
import org.hamcrest.CoreMatchers;
import org.junit.After;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import org.springframework.boot.context.embedded.LocalServerPort;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

import static org.hamcrest.MatcherAssert.assertThat;

@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class HelloE2ESeleniumTest {
    private WebDriver driver;

    @LocalServerPort
    private int port;

    @BeforeClass
    public static void setUpClass() throws Exception {
        ChromeDriverManager.getInstance().setup();
    }

    @Before
    public void setUp() throws Exception {
        driver = new ChromeDriver();
    }

    @After
    public void tearDown(){
        driver.close();
    }

    @Test
    public void helloPageHasTextHelloWorld() {
        driver.navigate().to(String.format("http://localhost:%s/", port));

        WebElement body = driver.findElement(By.tagName("body"));

        assertThat(body.getText(), CoreMatchers.containsString("Hello World"));
    }
}
```

This test is quite straight forward, it spins the entire spring application on a random port using `@SpringBootTest` and navigate to the `/` root of our website. Then we check if application prints `hello world` or not.

5.9. Best Practices in Testing

- Automated Tests are as important as the production code. Do not write tests just for the sake of formality, create a proper design for testcases and do it properly with proper attention.
- You shall always avoid test duplication throughout your test pyramid.Duplicated tests will be more annoying than helpful in the long term.
- Have a minimum set of End to End Tests, do not completely eradicate end to end test altogether just to meet tight project timelines.
- Create different TestSuites for different needs - one for regression, another for smoke testing, may be another for Build verification and testing.
- Avoid Test Duplication - We have different strategies for different layers of testing, it becomes easy for you to fall in trap of test duplication. As with the Production code, you should strive for non overlapping tests. Whatever can be tested through unit tests, must not be tested again at high levels tests. Only if the higher level tests add additional value like database integration, security testing, etc. then its a good idea to test the behavior at higher level tests even if already tested the same in unit tests.
- Eradicate non-determinism from tests. Its dicussed in more detail in subsequent section.

5.10. Interview Questions

How can we eradicate non-determinism in tests?

Non-determinism in tests make them useless, teams will soon start ignoring the entire testsuite when few tests start fail randomly. It most often occurs in Integration Tests due to various reasons, including -

1. Dependency on remote services that may not be available at test execution or not stable enough. Using Test Doubles approach (using a stub instead of real remote service) can help getting rid of this issue. Contract Driven Tests is also a great alternative to solve this problem.
2. Asynchronous code is often hard to test. If we just assume x seconds for API call completion and use a sleep to get the results, non-determinism can often be the result. If a test needs to fire and wait for the API call, then best way to make such tests predictable is to use polling (short interval) in a while loop. Second option could be to provide a callback method that asynchronous API can invoke upon execution, if there is a support to do so.
3. Lack of Isolation e.g. a test can create some data in database table which causes test pollution with subsequent tests. Properly isolated tests can run in any sequence. Some ideas are:
    - We should aim to write tests that cleanup after its completion.
    - Tests should rebuild their starting state from scratch.
    - Run tests in database transaction that is rolled back after the test execution.

- Start full running server for tests on random ports to avoid any port conflict.

  https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-testing.html#boot-features-testing-spring-boot-applications-working-with-random-ports

More on this

https://martinfowler.com/articles/nonDeterminism.html

How will you write a Integration Test using Spring Boot?

Spring Boot provides various mechanisms to run Integration Tests at different layers. Database related code can be checked using @DataJpaTest, Rest controllers can be testing using either TestRestTemplate or MockMvc.

What is difference between using MockMvc and TestRestTemplate based approach for Integration testing in Spring Boot Applications?

Both MockMvc and TestRestTenplate helps in writing Interation tests for Rest Layer in Spring based microservices. MockMvc does not start the container to check the cotnroller behvior, while TestRestTemplate starts an embedded container on a random or predefined port. Both kind of tests load the spring context, so you are free to choose anyone of these mechanisms.

What kind of testing should be done for a DAO layer?

DAO layer should be checked using Inteartion tests for near real time behavior. Spring Boot automatically configures H2 in memory database for tests that are annotated with `@DataJpaTest`. Also this annotation makes sure that test focuses only on JPA components (it excludes other components).

```
import org.junit.After;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
import org.springframework.boot.test.autoconfigure.orm.jpa.TestEntityManager;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.test.context.ActiveProfiles;
import org.springframework.test.context.junit4.SpringRunner;

import static org.assertj.core.api.Assertions.assertThat;

@RunWith(SpringRunner.class)
@DataJpaTest(showSql = true)
@ActiveProfiles("test")
public class ProductRepositoryTests {

    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private ProductRepository productRepository;

    @After
    public void tearDown() throws Exception {
        productRepository.deleteAll();      ①
    }

    @Test
    public void testFindByLastName() {
        Product product = new Product("first");
        entityManager.persist(product);

        Iterable<Product> findBName = productRepository.findAll();

        assertThat(findBName).extracting(Product::getName).containsOnly(product.getName());
    }
}
```

① Cleaning up resources after the test execution bring isolation to tests, thus avoids the non-determinism in Integration tests.

How will you write an end-to-end test for microservices architecture?

Spring Boot provides a pre-configured `TestRestTemplate` that you can inject in your integration tests. All you need to do is annotate your test with `@SpringBootTest` and `TestRestTemplate` will be automatically available for Injection using `@Autowired`, like shown below:

Listing 79. Sample Integration test using Spring Boot

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.context.embedded.LocalServerPort;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment;
import org.springframework.boot.test.web.client.TestRestTemplate;
import org.springframework.test.context.junit4.SpringRunner;

import static org.assertj.core.api.Assertions.assertThat;

@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class HttpRequestTest {

    @LocalServerPort
    private int port;

    @Autowired
    private TestRestTemplate restTemplate;

    @Test
    public void greetingShouldReturnDefaultMessage() throws Exception {
        assertThat(this.restTemplate.getForObject("http://localhost:" + port + "/",
                String.class)).contains("Hello World");
    }
}
```

This test will start the entire Spring Application Context using embedded servlet container at a random port. TestRestTemplate will try to access rest endpoint, thereby verifying the code end-to-end.

What type of test will cover security aspects of microservices architecture?

Integration and end-to-end tests can cover the security aspects of restful microservices. For example, in below integration test we can check that only authorized user can access the protected resource.

```
import org.junit.Ignore;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.context.embedded.LocalServerPort;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment;
import org.springframework.boot.test.web.client.TestRestTemplate;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.test.context.junit4.SpringRunner;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;

@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment=WebEnvironment.RANDOM_PORT)
public class ApplicationTests {

    @LocalServerPort
    private int port;

    private TestRestTemplate template = new TestRestTemplate();

    @Test
    public void homePageProtected() {
        ResponseEntity<String> response = template.getForEntity("http://localhost:"
                + port + "/user", String.class);
        assertEquals(HttpStatus.UNAUTHORIZED, response.getStatusCode());
        String auth = response.getHeaders().getFirst("WWW-Authenticate");
        assertTrue("Wrong header: " + auth, auth.startsWith("Bearer realm=\""));
    }
}
```

How will you test a microservice that has dependency on legacy system?

If your microservice has dependency on a Legacy System, the communication b/w microservice and legacy can be of two types -

Asychnrous Communication

In case of asynchrnous communication, we can stub the interface with legacy system and test the microservice.

Synchronous Communication

Generally this is not a recommended approach for integration with microservices, hence should be avoided. When calling synchronously, generally an adaptor us used in b/w legacy system and microservice. This adaptor can be mocked to test the microservice.

While writing an Integartion Test for a controller, should you load the entire Spring Context with all controllers? How will you write such a test using Spring Boot?

No, only the controller under test and its dependencies should be loaded in integration test of that controller. Dependencies should either be mocked or stubbed unless you are planning for database integration testing.

```
@RunWith(SpringRunner.class)
@WebMvcTest(controllers = { MyController.class }, secure = false)
@ActiveProfiles({ "test" })
public class MyControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    MyService myService;

    @Test
    public void testBaseReq() throws Exception {
        Testing dummyData = new Testing();
        dummyData.setData("testing");
        when(myService.get(anyInt())).thenReturn(dummyData);

        this.mockMvc.perform(get("/")).andDo(print()).andExpect(status().isOk());
    }
}
```

Should we load Spring Context in Unit Tests?

No, unit tests should be very light weight. They should only focus on the single component and mock everything that this component is dependent on. Ideally services with mocked dependencies are best candidate for Unit tests on server side. Controller layer that is most dependent on Spring Context, should be tested using Integration tests.

How to do end-to-end testing of a microservices, considering there are 100s of microservices in system? Do we need to deploy all the services before test execution?

Running end-to-end tests can be quite expensive as it would require a staging kind of environment where all the services are deployed before running the tests. Better approach could be:

1. Use Test Doubles to mock the dependencies
2. Use Contract Driven Tests where dependent services are mocked

So there should be no need ti deploy all the microservices to perform integration testing.