FEBRUARY 2018

MICROSERVICES ARCHITECTURE

Interview Questions for Java Developers

100+ real problems and their solutions
Spring Boot, Spring Cloud, Cloud Native Applications

MUNISH CHANDEL

Microservices Interview Questions

For Java Developers (Spring Boot, Spring Cloud, Cloud Native Applications)

Munish Chandel

Version 1.0, 13.02.2018

Table of Contents

Who should read this book?

This book is intended for two type of candidates:

- First who are preparing for a job change and want to brush up their skills for Spring Boot & Spring Cloud based microservices architecture.
- Second, who want to use this book for its ready made recipes & solutions for various practical problems in any Spring Boot + Spring Cloud based microservices project. Various topics discussed in this book like design patterns, anti-patterns, common pitfalls, best practices, testing strategies, security handling can be useful for anyone working in microservices architecture.

Important Notice

If you do not know the basics of microservices, then probably this book is not for you. Throughout this book, we assume that you have prior understanding of basic concepts related to Spring Boot and Spring Cloud frameworks.

Who should not buy this book?

If you are looking forward for a sequential reading and tutorial kind of study material, then probably this book is not a right pick. This book does not contain tutorials from scratch, instead it discusses the important problems & challenges that one faces during design, development and deployment of microservices.

How to get PDF version of the ebook?

You can buy this ebook from the below website to obtain a PDF copy:

https://books.shunyafoundation.com/to/VWVTPZ3A

Already bought ebook in another format?

If you have already bought this ebook in another format (`epub`, `azw3`, `mobi` etc.), then you are eligible for PDF edition for free. Just login to https://books.shunyafoundation.com and create a ticket to get free PDF copy after producing a valid receipt of purchase.

Preface

Microservices Architecture has become a trending pattern for developing modern distributed systems across the IT industry. When it comes to Java, `Spring Boot` along with `Spring Cloud` is the obvious choice for developing distributed systems. Spring Boot takes care of most of the boiler plate code, so that teams can focus on the business functionality alone.

Adopting *microservices architecture* brings huge advantage to the teams and the enterprise in terms of productivity, maintainability and scalability.

Salient features of this book are:

- Practical problem oriented approach
- Complex scenario have been explained by use of diagrams
- To the point discussion without long boring texts
- Brief code snippets has been provided wherever required
- Important topics like best practices, design patterns, security, testing strategies have been discussed.

Various illustration diagrams and examples have been provided in this book to make concept self explanatory. Business use cases, patterns, anti-patterns and pitfalls are also an area of focus throughout the book.

The Outline

Book is mainly focussed on questions and answers with the following areas in focus:

Part 1 - Core Concepts

This section focuses on key concepts, terminology & design patterns used in Microservices Architecture.

Part 2 - Microservices Recipes

This section is question oriented and covers problems & recipes on various design patterns like API Gateway, Externalized configuration management, Circuit Breaker, Security OAuth2 and JWT etc. More than 100 questions have been covered with proper explanation.

Part 3 - Testing Aspects

This section covers the testing aspects of microservices.

How to contact us

The author would appreciate a feedback from the readers. You can send us your feedback at shunya.care@shunyafoundation.com

-Shunya Foundation

IDE and softwares required for microservices development

We need a set of tooling to start developing `spring-boot` based microservices. Please be noted that not all of these softwares are mandatory for microservices development.

Java 8 Development Kit

I don't think it requires any introduction. You can straight away download the latest Java 8 from the below location

http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

IntelliJ IDEA

It is one of the best Integrated Development Environment for developing Java based applications. IntelliJ IDEA is not a free software, so you must get a valid license for using it. You can buy it from

https://www.jetbrains.com/idea/

Eclipse with STS

Eclipse is a powerful free alternative to IntelliJ IDEA. You can use it along with Spring Tool Set (STS) for developing spring powered applications. You can download it from

http://www.eclipse.org/downloads/

Jenkins

Jenkins is an open source automation software for implementing Continuous Integration and Continuous Delivery in your project. You can get it from

https://jenkins.io/

Spring Boot and Spring Cloud

Spring Boot makes it easy to create stand-alone, production grade applications. Spring Cloud makes these application cloud ready.

https://projects.spring.io/spring-boot/

Swagger

Swagger is an open source framework that helps developers design, build, document and consume RESTful Web Services.

https://swagger.io

Postman

Postman makes API development faster, easier, and better. Its one of the best tools for testing restful apis. Standalone app for Mac/Ubuntu/Windows can be downloaded from:

https://www.getpostman.com/apps

Docker

Docker is an open platform for developers and sysadmins to build, ship, and run distributed applications, whether on laptops, data center VMs, or the cloud.

https://www.docker.com/

Git

Git is a distributed version control system for tracking changes in computer files and coordinating work on those files among multiple people.

https://git-scm.com/

Gradle

Gradle is an open-source build automation system that builds upon the concepts of Apache Ant and Apache Maven and introduces a Groovy-based domain-specific language (DSL) instead of the XML.

https://gradle.org/

RabbitMQ

RabbitMQ is the most widely deployed open source message broker. Get it from the official website:

- https://www.rabbitmq.com/
- https://www.rabbitmq.com/getstarted.html

Database

MySQL, PosgreSQL, MongoDB, Apache Cassandra, Amazon DynamoDB, Amazon SQS, Neo4J, In Memory Cache like Redis etc. as per business needs.

Let's Encrypt

Let's Encrypt is a certificate authority that provides free X.509 certificates for Transport Layer Security (TLS) encryption via an automated process designed to eliminate the hitherto complex process of manual creation, validation, signing, installation, and renewal of certificates for secure websites. You can use these certificates for free (90 days validity) in your production environment.

https://letsencrypt.org

Part I - Core Concepts

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 makign 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 resuable 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.

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

References

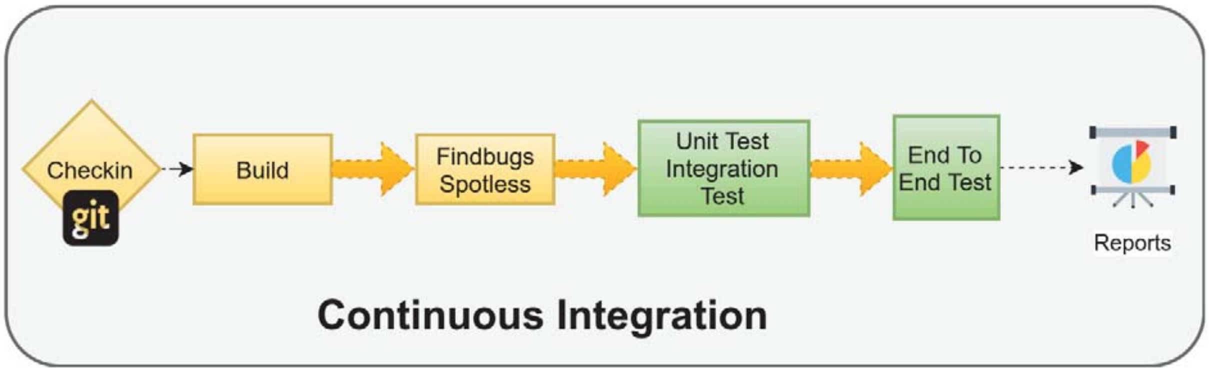
- https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing
- https://www.infoworld.com/article/3114195/system-management/the-8-fallacies-of-distributed-computing-are-becoming-irrelevant.html
- http://www.rgoarchitects.com/Files/fallacies.pdf

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.

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.

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

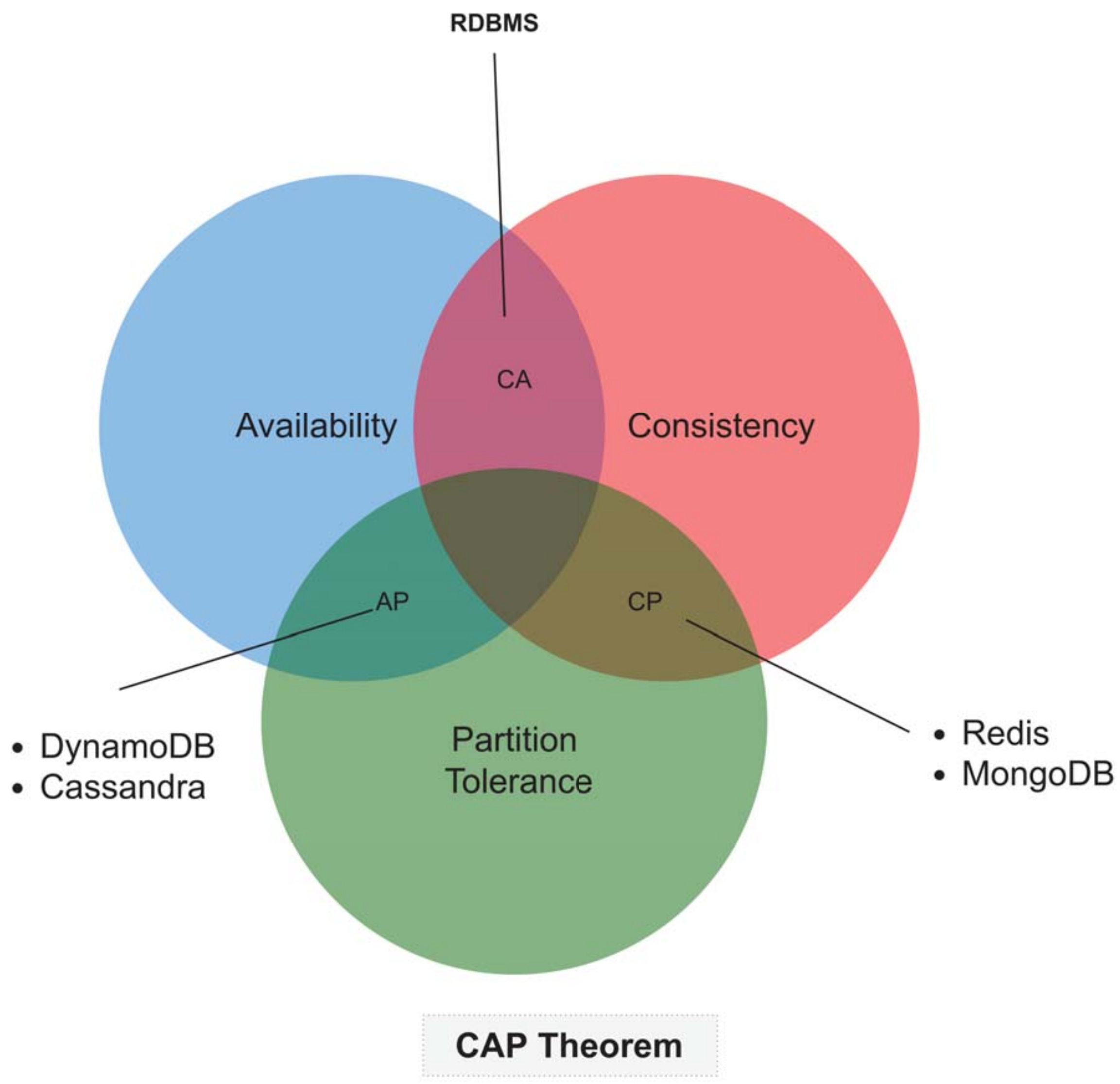
Tools like Git, Jenkins and Maven/Gradle are normally used together to setup the build pipelines.

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 is 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.

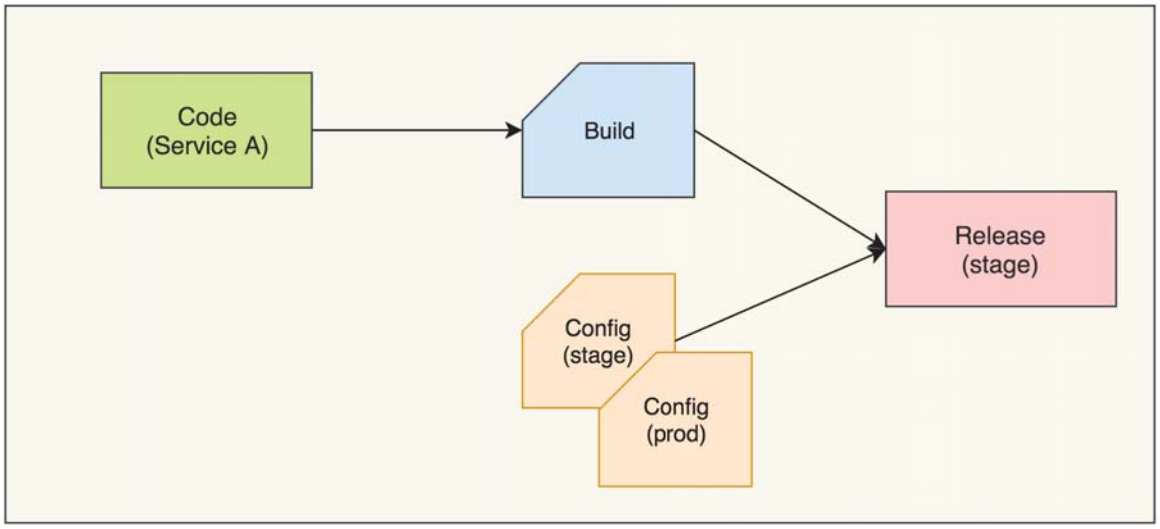
```
String greeting = this.restTemplate.getForObject("http://product-service/info", String.class);
```

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 standlone, 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 dependecies 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

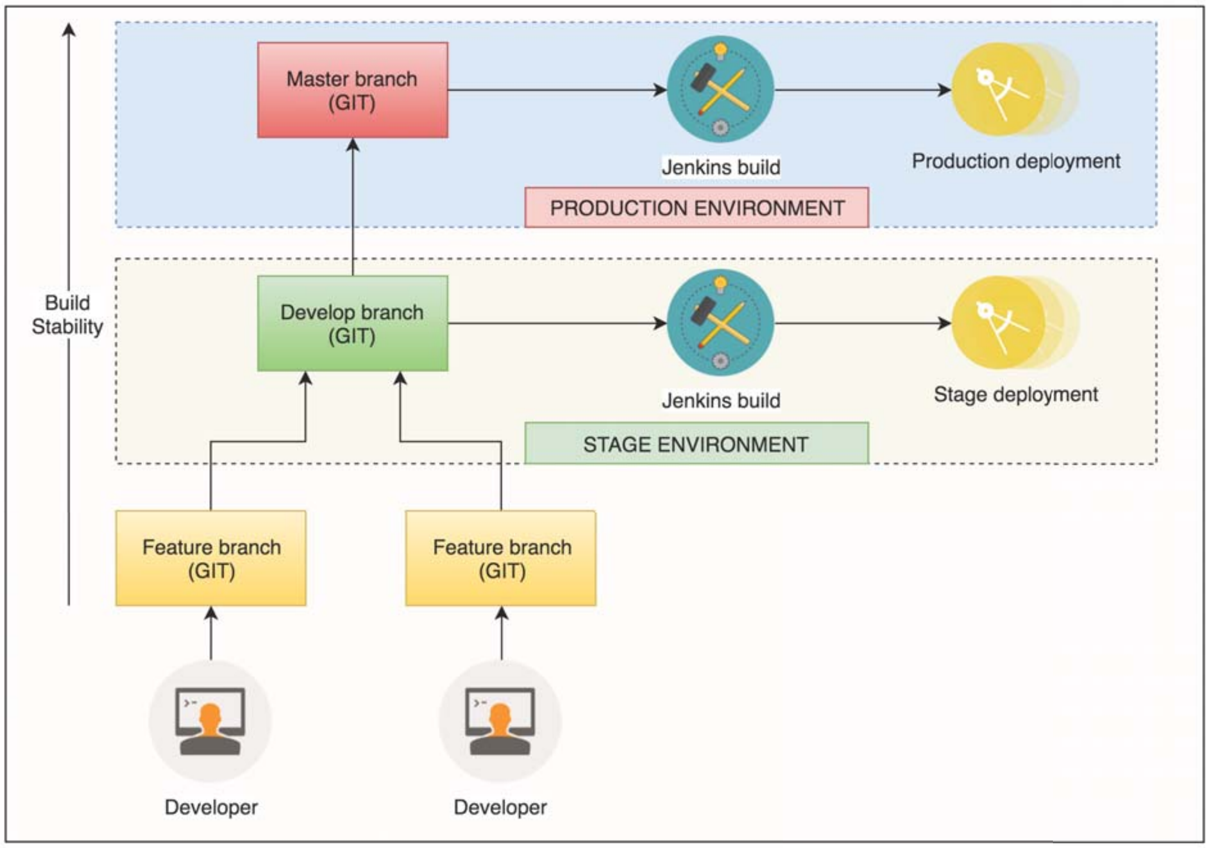
Run admin/management tasks as one-off processes. For example, a database migration should be run using a separate process altogether.

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.

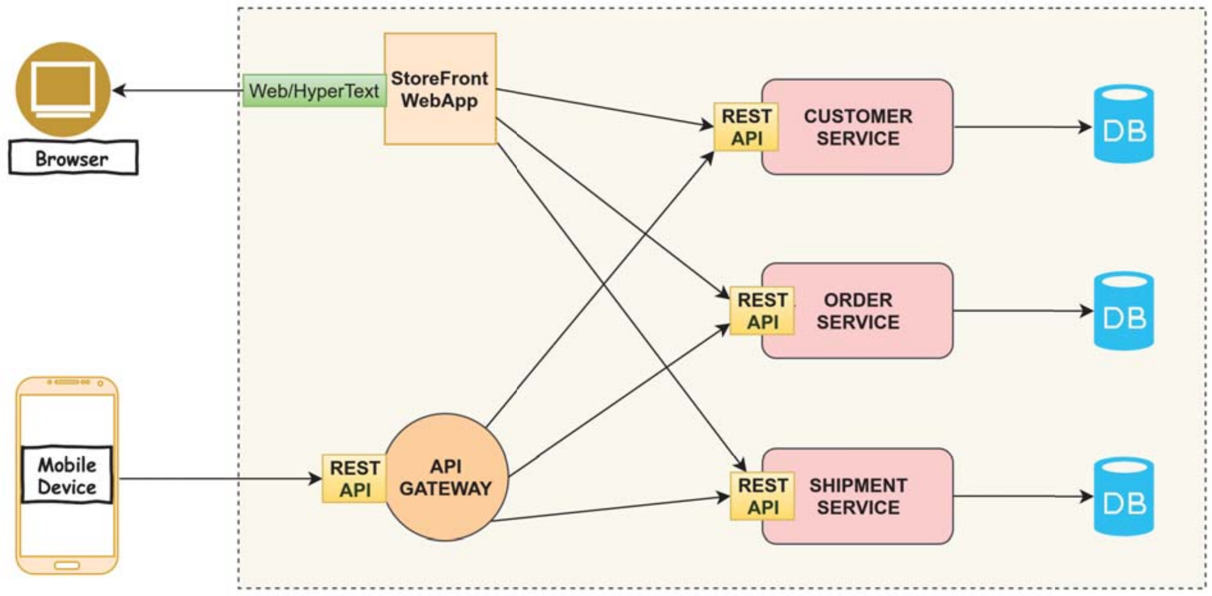
There is a beautiful book for learning git functionality, its freely hosted at:

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 talk to microservices architectured based distributed application 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

Part II - Microservices Recipes

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?

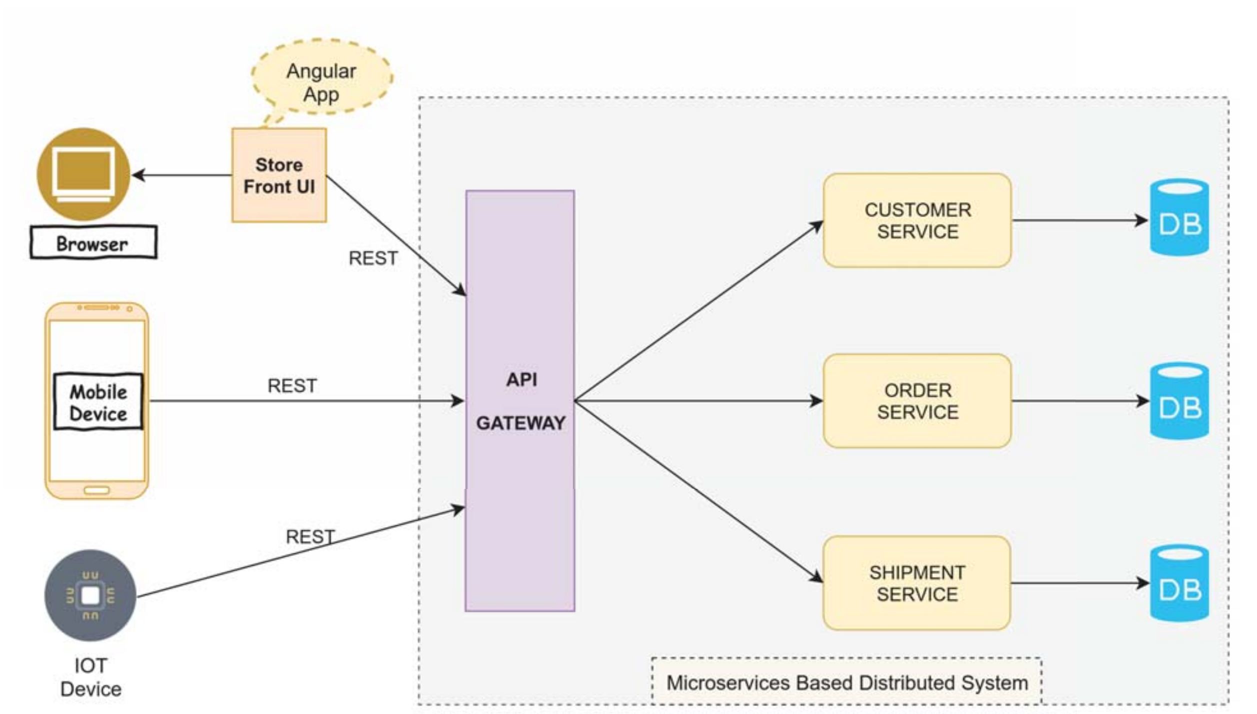
Microservices Architecture is a style of developing a scalable, distributed & highly automated system made up of many small autonomous services. It is not a technology but a new trend evolved out of SOA.

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?

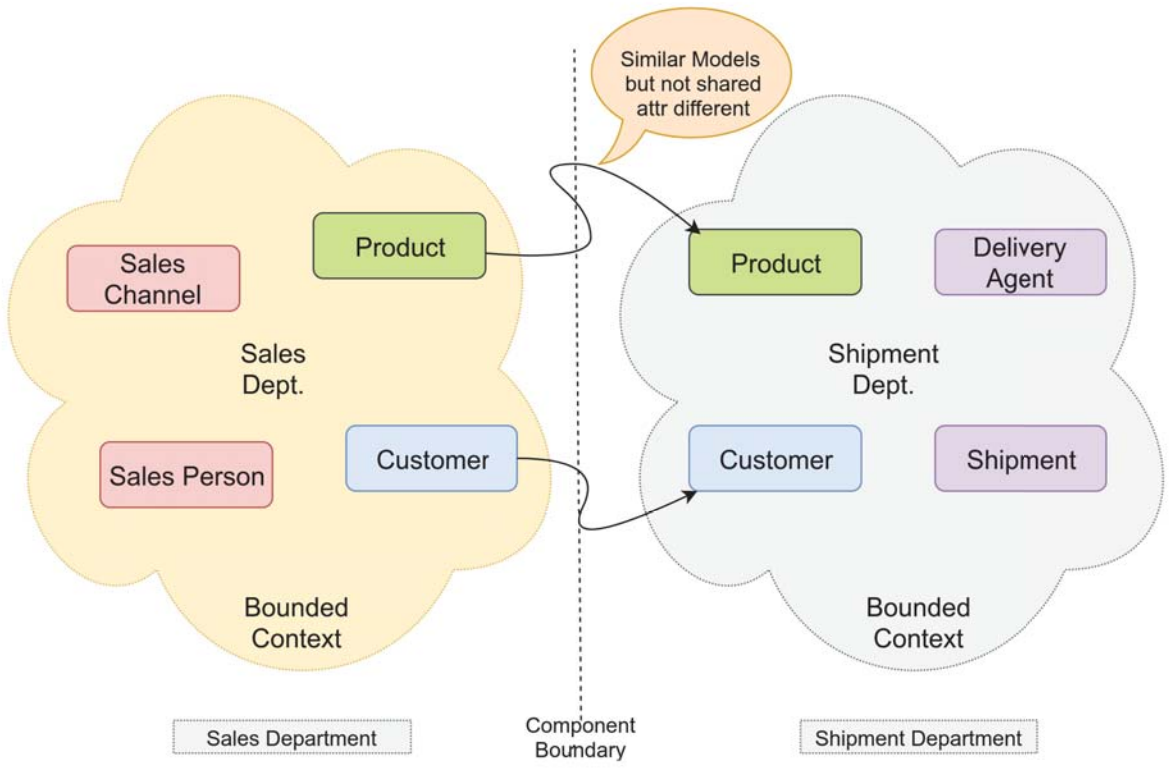
Bounded Context is a central pattern in Domain Driven Design. In Bounded Context, everything related to the domain is visible within context internally but opaque to other bounded contexts. DDD deals with large models by dividing them into different Bounded Contexts and being explicit about their interrelationships.

Monolithic Conceptual Model Problem

A single conceptual model for the entire orgnaization 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