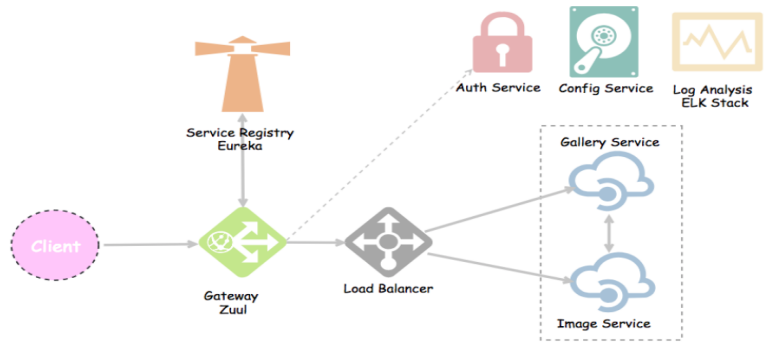


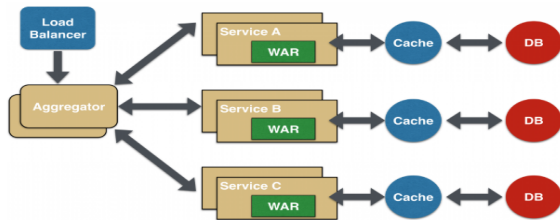
<p>Microservices: Is an architecture pattern that allows us to break our large system into number of independent collaborating components.</p> <p>A microservice is an engineering approach focused on decomposing applications into single-function modules with well-defined interfaces which are independently deployed and operated by small teams who own the entire lifecycle of the service.</p> <p>Microservices accelerate delivery by minimizing communication and coordination between people while reducing the scope and risk of change.</p> <p>While there is no single accepted definition for microservices, for me, there are a few important characteristics:</p> <ul style="list-style-type: none"> • REST - Built around RESTful Resources. Communication can be HTTP or event based. • Small Well Chosen Deployable Units - Bounded Contexts • Cloud Enabled - Dynamic Scaling 	<p>Spring cloud: which builds on top of Spring Boot, provides a set of features to quickly build microservices. It can quickly setup services, with minimal configurations.</p> <p>For e.g.:</p> <ul style="list-style-type: none"> • Service registration and discovery • Circuit breakers • Proxies • Logging and log tracing • Monitoring • Authentication, etc.
<p>Decomposing: Instead of having one large application, we decompose it into separate, different, mini-applications (services). Each service handles a specific business domain and provides the implementation for user interface, business logic, and connection to database.</p> <p>Single-function Each and every service has a specific function, or responsibility. And yes, a service can do many tasks, but all of them are nevertheless relevant to this single function.</p> <p>Well-defined interfaces: Services must provide an interface that defines how can we communicate with it. This basically defines a list of methods, and their inputs and outputs.</p> <p>Independent: Independent means services doesn't know about each other implementation. They can get tested, deployed, and maintained independently.</p> <p>Small Teams: We split the work up and team across the services. Each team focuses on a specific service, they don't need to know about internal workings of other teams.</p> <p>Entire Lifecycle: The team is responsible for the entire lifecycle of the service; from coding, testing, staging, deploying, debugging, maintaining.</p> <p>Minimizing Communication: Minimizing communication doesn't mean that the team members should ignore each other. It means the only essential cross-team communication should be through the interface that each service provides.</p> <p>The scope and risk of change:</p>	<p>Microservices using Spring cloud:</p>  <p>Eureka (Service Registry): It's the naming server, or called service registry. It's duty to give names to microservices. (@EnableEurekaServer)(@EnableEurekaClient)</p> <ul style="list-style-type: none"> • No need to hardcode the IP addresses of microservices. • What if service use dynamic ip addresses; when autoscaling. <p>Zuul (Gateway): A gateway is a single entry point into the system, used to handle requests by routing them to the corresponding service. It can also be used for authentication, monitoring, and more.</p> <p>Zuul is a proxy, gateway, an intermediate layer between the users and your services. (@EnableZuulProxy)</p> <ol style="list-style-type: none"> 1. It uses Eureka server to route the requested service. 2. It load balances (using Ribbon) between instances of a service running on different ports. 3. What else? We can filter requests, add authentication, etc. <p>Ribbon: Netflix Ribbon is an Inter Process Communication (IPC) cloud library. Ribbon primarily provides client-side load balancing algorithms.</p> <p>Apart from the client-side load balancing algorithms, Ribbon provides also other features:</p> <ul style="list-style-type: none"> • Service Discovery Integration – Ribbon load balancers provide service discovery in dynamic environments like a cloud.

<p>Services should be changed without breaking other services. And so long as we don't change the external interface there will be no problem for other services.</p>	<p>Integration with Eureka and Netflix service discovery component is included in the ribbon library</p> <ul style="list-style-type: none"> • Fault Tolerance – the Ribbon API can dynamically determine whether the servers are up and running in a live environment and can detect those servers that are down • Configurable load-balancing rules – Ribbon supports RoundRobinRule, AvailabilityFilteringRule, WeightedResponseTimeRule out of the box and also supports defining custom rules
<p>Monolithic vs Microservices/ Disadvantages of Monolith applications:</p> <ul style="list-style-type: none"> • Scalability challenges. • New technology adoption. • New processes - Agile? • Difficult to perform automation tests. • Difficult to adapt to modern development practices. • Adapting to device explosion. <p>Disadvantages of MS: Communication between methods in monolithic is much faster when compared to services asynchronous communications, which slower, harder to debug, and must be secured.</p> <p>And for sure, there will be extra effort for operations, deployment, scaling, configuration, monitoring and testing as each service is separate.</p> <p>For that being said, we need to have a skilled DevOps team to handle the complexity involved in deployment and monitoring automation.</p> <p>Advantages of microservices include</p> <ul style="list-style-type: none"> • New technology and process adoption becomes easier. You can try new technologies with the newer microservices that we create. • Faster release cycles. • Scaling with the cloud. 	<p>Ribbon API works based on the concept called “Named Client”. While configuring Ribbon in our application configuration file we provide a name for the list of servers included for the load balancing.</p> <p>Ribbon API enables us to configure the following components of the load balancer:</p> <ul style="list-style-type: none"> • Rule – Logic component which specifies the load balancing rule we are using in our application • Ping – A Component which specifies the mechanism we use to determine the server’s availability in real-time • ServerList – can be dynamic or static. In our case, we are using a static list of servers and hence we are defining them in the application configuration file directly <p>Authentication with JWT: The authentication flow is simple as:</p> <ol style="list-style-type: none"> 1. The user sends a request to get a token passing his credentials. 2. The server validates the credentials and sends back a token. 3. With every request, the user has to provide the token, and server will validate that token.
<p>Resource Overview:</p> <ul style="list-style-type: none"> • Retrieve all Students - @GetMapping("/students") • Get details of specific student - @GetMapping("/students/{id}") • Delete a student - @DeleteMapping("/students/{id}") • Create a new student - @PostMapping("/students") • Update student details - @PutMapping("/students/{id}") 	<p>JSON Based Token (JWT) is a JSON-based open standard for creating access tokens. It consists of three parts; header, payload, and signature. The header contains the hashing algorithm {type: “JWT”, hash: “HS256”} The payload contains attributes (username, email, etc) and their values. {username: "Omar", email: "omar@example.com", admin: true } The signature is hashing of: Header + “.” + Payload + Secret key</p>
<p>Challenges With Microservice Architectures:</p> <ul style="list-style-type: none"> • Quick setup needed • Automation • Visibility • Bounded Context • Configuration Management • Dynamic scale-up and scale-down • Pack of Cards • Debugging • Consistency <p>Solutions:</p> <ul style="list-style-type: none"> • Spring Boot <ul style="list-style-type: none"> ◦ Embedded servers (easy deployment with containers) ◦ Metrics (monitoring) ◦ Health checks (monitoring) ◦ Externalized configuration • Spring Cloud <ul style="list-style-type: none"> ◦ Dynamically scale up and down. using a combination of <ul style="list-style-type: none"> ■ Naming Server (Eureka) 	<p>Configuration Service: common configuration variables, enum classes, or logic, used by multiple services can be kept in separate service that can be included and used as a dependency in other services.</p> <p>Circuit Breakers & Log Tracing: Hystrix: If you have, let’s say 3 services, A calls → B, and B calls → C service. What if, a failure happened at B? The error will cascade down to service C, right?. A -> B (failure) -> C</p> <p>Another example, let's say a service A calls a remote service R, and for some reason the remote service is down. How can we handle such a situation? What we would like to do is stop failures from cascading down, and provide a way to self-heal, which improves the system’s overall resiliency.</p> <p>Hystrix is the implementation of Circuit Breaker pattern, which gives a control over latency and failure between distributed services.</p>

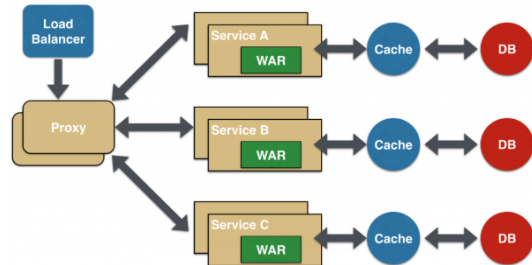
- Ribbon (Client Side Load Balancing)
- Feign (Easier REST Clients)
- Visibility and monitoring with
 - Zipkin Distributed Tracing
 - Netflix API Gateway
- Configuration Management with Spring Cloud Config Server.
- Fault Tolerance with Hystrix.

Microservice Design Pattern:

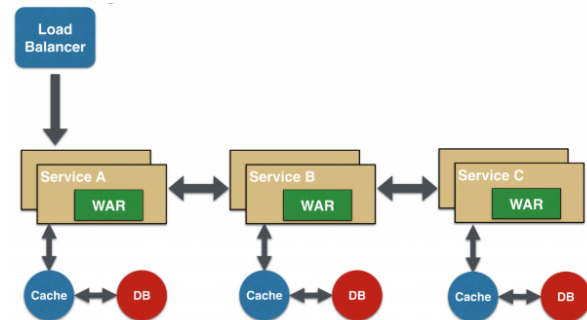
• Aggregator:



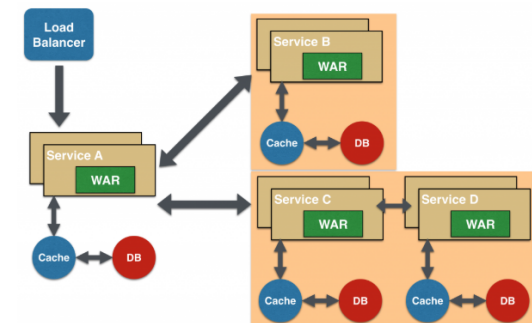
• Proxy:



• Chained:



• Branch:



• Shared:

The main idea is to stop cascading failures by failing fast and recover as soon as possible — Important aspects of fault-tolerant systems that self-heal.

Hystrix watches for failures in that method, and if failures reached a threshold (limit), Hystrix opens the circuit so that subsequent calls will automatically fail. Therefore, and while the circuit is open, Hystrix redirects calls to the fallback method.

Sleuth:

If you have, let's say 3 services, A, B and C. We made three different requests.

One request went from A → B, another from A → B → C, and last one went from B → C.

A → B

A → B → C

B → C

As the number of microservices grow, tracing requests that propagate from one microservice to another and figure out how a requests travels through the application can be quite daunting.

Sleuth makes it possible to trace the requests by adding unique ids to logs.

A trace id (1st) is used for tracking across the microservices; represents the whole journey of a request across all the microservices, while span id (2nd) is used for tracking within the individual microservice.

ELK Stack:

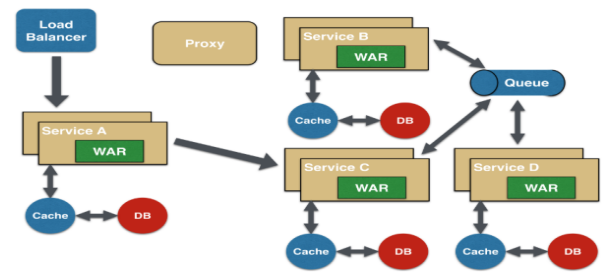
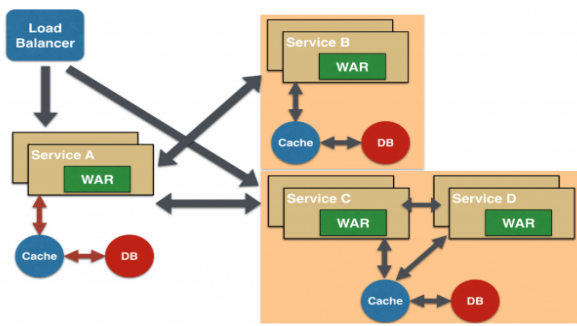


Logstash is a tool for managing logs. It supports virtually any type of log, including system logs, error logs, and custom application logs. It can receive logs from numerous sources, including syslog, messaging (for example, rabbitmq), and jmx, and it can output data in a variety of ways, including email, websockets, and to Elasticsearch.

Elasticsearch is a full-text, real-time search and analytics engine that stores the log data indexed by Logstash. It is built on the Apache Lucene search engine library and exposes data through REST and Java APIs. Elasticsearch is scalable and is built to be used by distributed systems.

Kibana is a web-based graphical interface for searching, analyzing, and visualizing log data stored in the Elasticsearch indices. It utilizes the REST interface of Elasticsearch to retrieve the data, and not only enables users to create customized dashboard views of their data, but also allows them to query and filter the data in an ad hoc manner.

• Asynchronous:



Note: Information gathered in this document has been collected from various sources on the Internet.

Sources:

<https://medium.com/omarelgabrys-blog/>

<http://blog.arungupta.me/microservice-design-patterns/>

<https://medium.com/oneclicklabs-io/> ,

<https://dzone.com/articles/microservices-with-spring-boot-part-1-getting-star>