1. Nhược điểm Microservices

- Khó khăn trong việc chia nhỏ các services

- Đảm bảo giao dịch phân tán

- Testing các services

- Triển khai microservices

- Config, quản lý deploy & hoạt động các services

- Track request, process của mỗi services

1. What is JHipster, and why use it

* Fully Open Source
* Built on Spring Boot + Angular/React
* Microservices support heavily uses the Netflix OSS libraries

1. Good reasons for choosing Microservices

* The app scope is large & not well defined and you are sure that the app will grow tremendously in terms of features
* The team size is large, there are enough members to effectively develop individual components independently
* The average skillset of the team is good and team members are confident about advanced Microservices patterns
* Time to market is not critical
* You are ready to spend more on infrastructure, monitoring and so so, in order to improve to product quality
* You user base is huge and you expect them to grow. For example, a social media app targeting users all over the world

1. Bad reasons for choosing Microservices

* You thought it was cool
* You wanted to impress someone
* Peer pressure
* You thought micro-services perform better than Monolithic automatically

1. Service Discovery

* Helps the API gateway to discover the right endpoints for a request
* It will have a load balancer to regulate the traffic to the services
* Based on location, where load balancing is done, can be classified into
* Client side discovery pattern: client is responsible for discovery and load balancing
* Server side discovery pattern: a dedicated server is responsible for discovery and load balancing
* Works hand in hand with a Service Registry
* JHipster uses Netflix Eureka for service discovery

1. Load balancing

* Load balancing in JHipster is done with Netflix Ribbon
* Support Fault tolerance
* Support multiple protocol (HTTP, TCP, UDP) support in an asynchronous and reactive model
* Supports caching and batching

1. Circuit breaking

* Circuit breaking in JHipster is done using Netfilix Hystrix
* Stop cascading failures
* Supports fallbacks and graceful degradation
* Enables fail fast and rapid recovery
* Supports real time monitoring and configuration changes
* Supports concurrency aware request caching
* Supports automated batching through request collapsing

1. The gateway

* “Edge service” or “gateway”, this is the entry to our microservices app
* Acts as a proxy
* Protects the micro-services
* Routes the requests
* Servers the front-end (Angular/React)
* There are often several gateways
* One for a client-facing front-office app
* One for the internal back-office
* One for a specific mobile app
* This is sometimes used with the “backend for frontend” pattern

1. API management

* They gateway can be an API management solution
* API management solution provide
  + Quality of service (Rate limiting)
  + Security (JWT, OAuth2, OpenID Connect & UAA)
  + Automatic documentation API (Swagger)
* As the number of micro-services grow, they become a very important part of an API strategy

1. Configuration management

* Spring Boot can be configured in many different ways
* Spring Cloud Configured offers centralized configuration
  + All micro-services can be automatically configured from one central location
  + Using Git, configurations can be tagged and roll-backed
  + The JHipster Registry adds an UI layer and a security layer on top of it

1. Eureka

* Netflix Eureka is a REST based service registry and discovery system
* It offers a client-server model
  + Eureka Server
    - Acts as registry for the services
    - Load balance among server instances
    - Useful in cloud-based environment where the availability is intermittent
* Eureka Client
  + Java based client for Eureka server
  + Does service discovery
  + Acts as middle tier client based load balancer
* Available as part of spring cloud Netflix

1. Feign and Ribbon

* Feign is a java to http client binder inspired by Retrofit, JAXRS-2.0, and WebSocket
* Feign is also a declarative web service client
* Spring Cloud Netflix Feign includes Ribbon to load balance the request made with Feign

1. Zuul

* Netflix Zuul is a gateway/edge service that provides dynamic routing, monitoring, resiliency, security and more
* It allows to code customized filters for use cases like
  + Authentication & Security
  + Insight & Monitoring
  + Dynamic routing
  + Stress testing & Load shedding
  + Static response handling
* Zuul 2 is on the pipeline with non-blocking IO
* It is used in the JHipster Gateway

1. Security

* An API management solution, like a JHipster gateway should secure the access to the back-end microservices
* JHipster support security mechanisms
  + JWT
  + OAuth2
  + JHipster UAA
  + OpenID Connect
* Requests are secured by default
  + The JHipster gateway adds the necessary security tokens to the HTTP requests
  + Micro-services either trust the gateway (JWT) or a third-party security system (JHipster UAA, OpenID Connect implementation) using either a shared secret or a public key

1. Rate limiting

* API management is also about Quality of Service
* JHipster provides a rate limiting filter, using Bucket4J
  + Uses a “token-bucket algorithm”
  + Can be distributed across a cluster using Hazelcast
* As a JHipster Gateway handles security and routing, it is very easy to add custom code
  + Example: allow more requests on a specific service for some users

1. Swagger aggregation

* A JHipster gateway can also aggregates Swagger configuration from all micro-services
  + It finds all microservices using the service discovery mechanism
  + It adds a Swagger UI on top of the Swagger definition
  + It handles security so requests can be tested

1. Consul

* Service discovery system from HashiCorp
* Open Source
* Written in Go
* Replaces Eureka
  + Works the same with Spring Cloud
  + JHipster provides a specific mechanism to load Spring Cloud Config data into the Consul K/V Store

1. Traefik

* HTTP reverse proxy and load balancer
* Open Source
* Written in Go
* 2 pattern are possible
  + Replace Zuul completely by Traefik
  + Use Zuul and Traefik together

1. HTTPS

* HTTPS support comes built-in with a JHipster app
  + See the application.yml configuration
  + It is also a requirement if you use HTTP/2
* Some people only secure the gateways
  + Internal networks are supposed to be secured
  + Do not add performance overhead
* Traefik supports HTTPS
* Let’s Encrypt provides free SSL certificates
  + Great solution, as long as your host is publicly available
  + An easy configuration is to use an Apache front-end, which has a official Let’s Encrypt support

1. JWT

* Our most popular and easy-to-use option
* Stateless, signed token that all micros-services can share and trust
* By default, the JHipster gateway generates a JWT
  + It sends it to the various micro-services
  + As they all trust the same key (which is shared from the JHipster Registry using Spring Cloud Config) they all accept the token
* Advanced options can make it more secure
  + Better encryption algorithms using Bouncy Castle
  + Public/private key pairs

1. JHipster UAA

* A mix of a JHipster app and CloudFoundry UAA (User Account and Authentication)
  + Security is handled by JHipster UAA, more secure
  + Easier to use when there are several gateways
  + Popular option for micro-services architectures
* Has to be generated for you micro-services architecture
  + Can easily be tuned and customized
* Provides OAuth2 tokens to all application

1. OpenID Connect

* Provides an identity layer on top of OAuth2
  + Standard with many implementations
  + Starts to be widely used across enterprises
* Great for micro-services architecture
  + User management, authentication and authorizations are handled by a third-party OpenID Connect implementation

1. JHipster Registry

* Spring Cloud Config server: with a UI and many tweaks
* Service discovery server: based on Netflix Eureka
* Management server: Monitoring and administration screens
* The JHipster Registry provides “live” monitoring screens
  + Metrics
  + Health
  + Live logs
  + Configuration
* It can also change log levels at runtime
* It is fully secured with JWT or OpenID connect

1. JHipster Console

* Based on the Elastic Stack
  + Logstack, Elasticsearch, Kibana
  + Specific, Logback tuning for better performance
* Provides many built-in dashboards
  + Performance, JVM, cache, available services…
* Aggregates all applications
* Stores data over time

1. Zipkin

* Zipkin is a distributed Tracing system
  + Zipkin helps to collect and search the timing data
  + All registered services will report the timing data to Zipkin and it creates a dependency diagram based on the received traced requests for each of the application or services
* Helps to troubleshoot latency problems in micro-services architectures
* Support in-memory, JDBC (MySQL), Cassandra and Elasticsearch as Storage options

1. Stateless vs Stateful

* Scaling stateless applications is easy
  + This is why JHipster uses a stateless design as much as possible
  + Basically you just need to run more instances
* Sometimes stateful is necessary
  + Security
  + Caches
* Sticky sessions is a usual solution to scaling stateful application, but it doesn’t work well in a micro-services architecture

1. Scaling cache

* No cache
  + The application scales easily
  + But sends all the load to the database, which doesn’t scale easily
* Ehcache
  + Add nodes on-the-fly by using network broadcasting: cannot work in most production environments
* Hazelcast
  + Adds nodes on-the-fly using the JHipster Registry
  + Can also do HTTP session clustering
  + Default option for JHipster micro-services
* Infinispan
  + Adds nodes on-the-fly using the JHipster Registry
  + Great alternative to Hazelcast

1. Deploying and scaling in Docker

* Use the JHipster docker-compose sub-generator
  + Generates a full Docker Compose configuration for the whole microservices architecture
  + Adds monitoring and log management
* Deploying is as simple as “docker-compose up -d”
* Scaling an application is done by Docker:
  + “docker-compose scale microservice-app=3”

1. Testing options - server side - Integration test

* Integration tests are created using Junit, Mockito and spring test context framework
* Spring Integration tests are generated for all the REST endpoints for the application and for entities.
* Mockito is excellent for creating mocks and spies.
* Spring provides any useful utilities and annotations for testing
* In memory database (H2, Mongo, Cassandra, Elasticsearch) is used for testing
* Run using *./mvnw* test or *./gradlew* test

1. Testing options - server side - Performance test

* Performance testing is done using Gatling
* Gatling is written in Scala
* Gatling tests can be generated for entities by choosing the option during generation
* Tests are written using Scala and the Gatling Scala DSL
* Provides great visualization in the test reports
* Ideal for performance and load testing
* Run using *./mvnw gatling:execute* or *./gradlew gatlingRun*

1. Testing options - server side - BDD test

* Behaviour driven tests are done using Cucumber
* Cucumber is the most widely used BDD testing framework
* The option can be enabled during generation
* Tests are written using Gherkin

1. Testing options - client side - unit tests

* Client side unit tests are done using Karma and Jasmine
* It is one of the most widely used combination for Angular unit testing
* Run using *yarn test*

1. Testing options - client side - e2e tests

* End-to-end tests are done using Protractor and Jasmine
* Protractor is one of the de facto option for Angular e2e testing
* Supports parallel testing and test suites
* Uses selenium webdriver to run the tests
* Can also be used with selenium grid easily
* Run using yarn e2e

1. The CI-CD sub-generator

* JHipster ci-cd sub generator can generate pipeline scripts for various CI/CD tools
* It currently supports
  + Jenkins pipeline
  + Travis CI
  + Gitlab CI
  + Circle CI
* The pipeline executes the following steps
  + Build the application
  + Test server side and client side tests including gatling tests if available
  + Package the application for production
  + Deploy to heroku if option is enabled.

1. Doing a production build

* In “prod” mode, JHipster creates a specific build
  + The Angular part uses a specific Webpack configuration to greatly optimize the front-end application
  + Spring Boot uses a specific configuration to remove hot reload, have higher cache values, etc.
* The final result is an “executable WAR file”
  + Uses an embedded Undertow server
  + Can be run directly as an executable file: “./microservice-0.0.1-SNAPSHOT.war”
* A Docker image can also be generated
  + “./mvnw package -Pprod dockerfile:build”
* The various JHipster “cloud” sub-generators either use the executable WAR file or the Docker image, with their own specific configuration