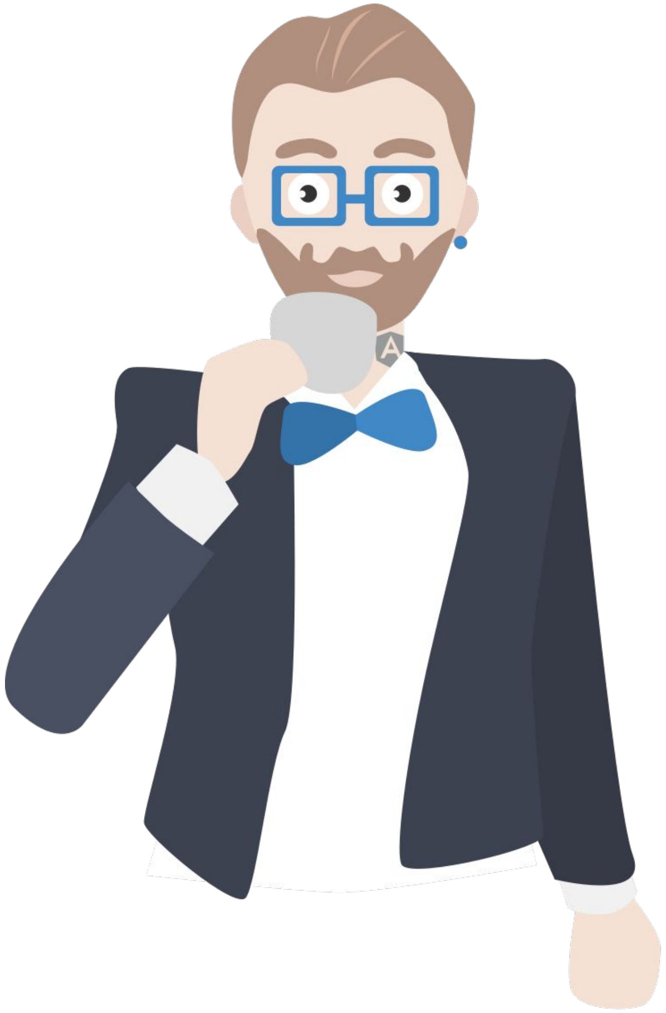


Easy microservices with JHipster

Julien Dubois
&
Deepu K Sasidharan

Julien Dubois

@juliendubois

JHipster creator & lead developer

Chief Innovation Officer, Ippon Technologies

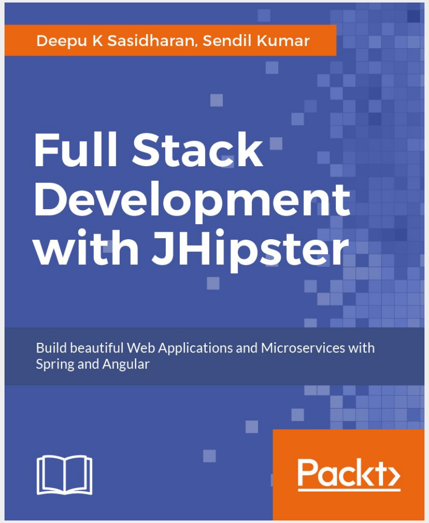
Deepu K Sasidharan

@deepu105

JHipster co-lead developer

Senior Product Developer, XebiaLabs

https://www.packtpub.com/application-development/full-stack-development-jhipster



Deepu K Sasidharan, Sendil Kumar

Full Stack Development with JHipster

Build beautiful Web Applications and Microservices with Spring and Angular

What you will learn in the next 3 hours

- How to create microservices quickly and efficiently
- Distributed architecture designs
- Scalability, failover, and best practices for managing microservices
- Microservices in production

What is JHipster, and why use it

- Most popular application generation tool in the Java world
  - 8,400+ GitHub stars, 375+ contributors
  - Nearly 1 million installations
  - 200+ companies officially using it on http://www.jhipster.tech/companies-using-jhipster/
  - Won this year a Duke's choice award for extreme innovation and a Jax Innovation Award
- Fully Open Source
- Built on Spring Boot + Angular (Soon React as well)
- Microservice support heavily uses the Netflix OSS libraries

JD

Talk is cheap, show me the code

-- Linus Torvalds

Let’s build our first microservice

- This is the simplest possible microservice
  - no database
- Go to https://start.jhipster.tech
  - Select the options
  - Generate the application
  - Open it up your IDE
  - Run it
  - See it live on http://localhost:8081/

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

JD

Adding a simple “Hello, world”

- Run `jhipster spring-controller Hello`
- Compile
- Check Swagger
- Remove the security to access the endpoint

JD

What did we learn?

- Creating a Spring Boot microservice with a few clicks
- Using the JHipster Registry
- Improving the microservice using a sub-generator

Microservice architectures

JD

Popular Microservice patterns

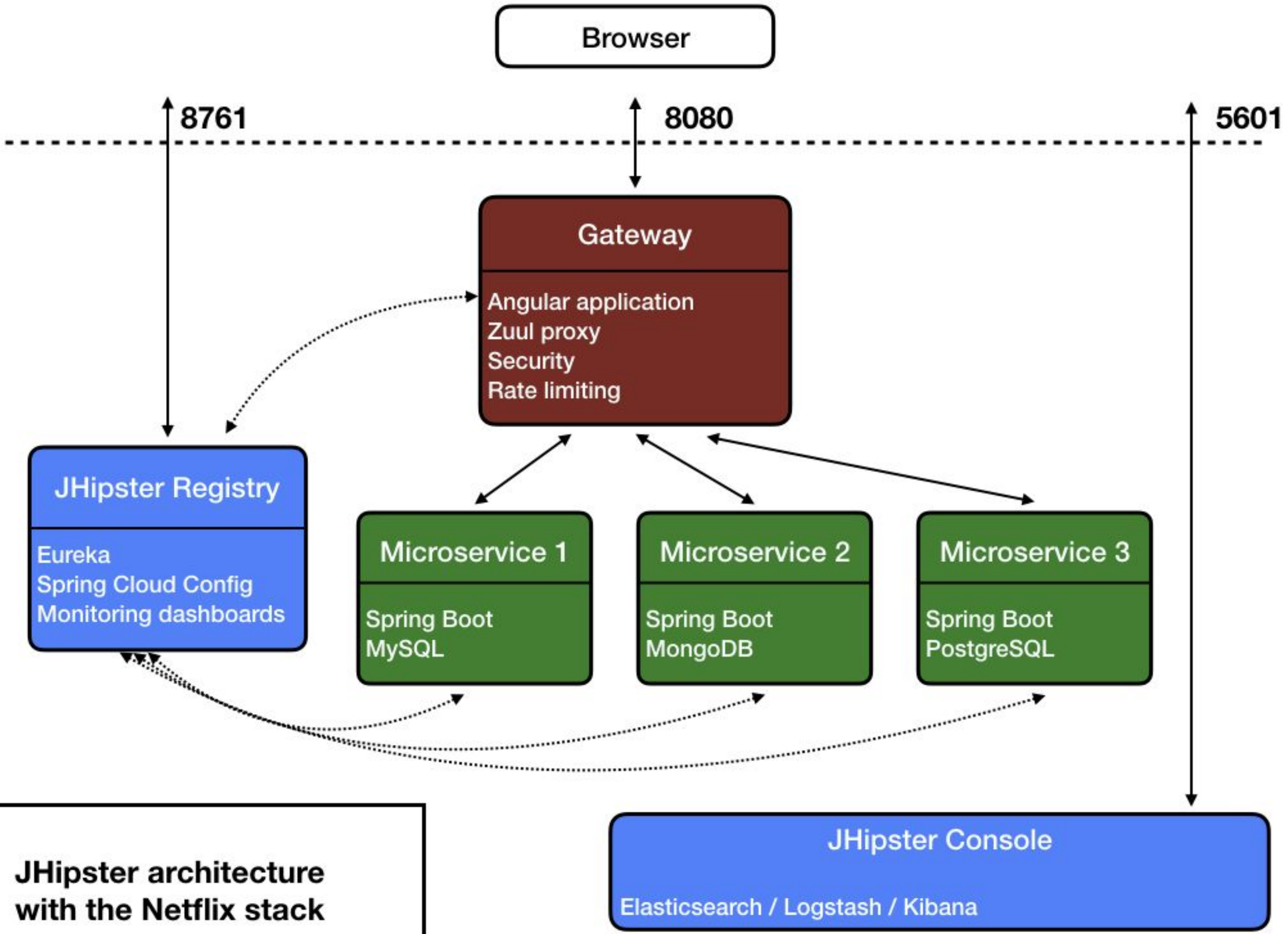
- Aggregator pattern
- Proxy pattern
- Chained pattern
- Branch pattern
- Shared Data pattern
- Asynchronous messaging pattern

DS

Aggregator/Proxy pattern

- One of the most commonly used pattern made famous by Netflix
- Also known as Iceberg pattern
- Characterized by an API gateway hiding several microservices under it
- Gateway can act as Aggregator and/or Proxy
- JHipster uses the Proxy pattern OOB

DS



JHipster architecture with the Netflix stack

JD

Good reasons for choosing microservices

- The application scope is large & not well defined and you are sure that the application 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 Microservice patterns.
- Time to market is not critical.
- You are ready to spend more on infrastructure, monitoring, and so on, in order to improve the product quality.
- Your user base is huge and you expect them to grow. For example, a social media application targeting users all over the world.

DS

Bad reasons for choosing microservices

- You thought it was cool
- You wanted to impress someone
- Peer pressure
- You thought microservices perform better than Monoliths automatically

PS: You are not Netflix, facebook or Google you probably do not need microservices.

DS

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 (e.g; Netflix Ribbon)
    - Client is responsible for discovery and load balancing
  - Server side discovery pattern (e.g; AWS ELB)
    - 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

Load balancing

- Load balancing in JHipster is done with Netflix Ribbon
  - Supports Fault tolerance
  - Supports Multiple protocol (HTTP, TCP, UDP) support in an asynchronous and reactive model
  - Supports Caching and batching

Circuit breaking

- Circuit breaking in JHipster is done using Netflix Hystrix
    - Stops 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.

The 12 factors

JHipster closely follows the 12 factors methodology for web apps and SaaS as detailed in https://12factor.net/

1. One codebase tracked in revision control, many deploys
2. Explicitly declare and isolate dependencies
3. Store config in the environment
4. Treat backing services as attached resources
5. Strictly separate build and run stages
6. Execute the app as one or more stateless processes
7. Export services via port binding
8. Scale out via the process model
9. Maximize robustness with fast startup and graceful shutdown
10. Keep development, staging, and production as similar as possible
11. Treat logs as event streams
12. Run admin/management tasks as one-off processes

DS

The gateway

- “Edge service” or “gateway”, this is the entry to our microservices application
- Acts as a proxy
  - Protects the microservices
  - Routes the requests
  - Serves the front-end (Angular)
- There are often several gateways
  - One for a client-facing front-office application
  - One for the internal back-office
  - One for a specific mobile application
  - This is sometimes used with the “backend for frontend” pattern, see https://www.thoughtworks.com/radar/techniques/bff-backend-for-frontends

JD

API management

- The gateway can be (or work with) an API management solution
- API management solutions provide
  - Quality of service (rate limiting)
  - Security (OpenID Connect, JWT)
  - Automatic documentation (Swagger)
- As the number of microservices grow, they become a very important part of an API strategy

Configuration management

- Spring Boot can be configured in many different ways
- Spring Cloud Config offers centralized configuration
  - All microservices can be automatically configured from one central location
  - Using Git, configurations can be tagged and rollbacked
  - The JHipster Registry adds an UI layer and a security layer on top of it

JD

Let’s code a complete microservices architecture

Building several microservices

- For the first microservice let us use the one from the previous step
- Second microservice is more complex
  - with MySQL DB & hibernate 2nd level cache
  - Several entities generated using the JDL Studio
  - JDL is at https://github.com/jhipster/jdl-samples/blob/master/simple-online-shop.jh
- On the gateway we generate 2 entities from the second microservice
  - Product and Category

DS

Application generation using the command line

- We will use the jhipster CLI to generate the second microservice
- Run `jhipster`

DS

About JDL

- JHipster Domain Language is a specific DSL for working with JHipster applications.
- It allows creation of entities and relationships using a simple DSL in a single file
- Recommended for real world use cases where entity model is complex

The JDL Studio

- We will use http://www.jhipster.tech/jdl-studio/ to create the JDL
- Use the `jhipster import-jdl` sub generator to create the entities

Generating a gateway

- We will use http://start.jhipster.tech to generate the gateway
- Use the `jhipster entity` sub-generator to create the front-end on the Gateway for the Product and Category entities from the second microservice

The Netflix OSS Stack

DS

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

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 requests made with Feign.

DS

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.

API management

Security

- An API management solution, like a JHipster gateway, should secure the access to the back-end microservices
- JHipster supports 3 security mechanisms
  - JWT
  - JHipster UAA
  - OpenID Connect
- Requests are secured by default
  - The JHipster gateway adds the necessary security tokens to the HTTP requests
  - Microservices 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

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

Swagger aggregation

- A JHipster gateway can also aggregates Swagger configuration from all microservices
  - 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

JD

20 minutes break

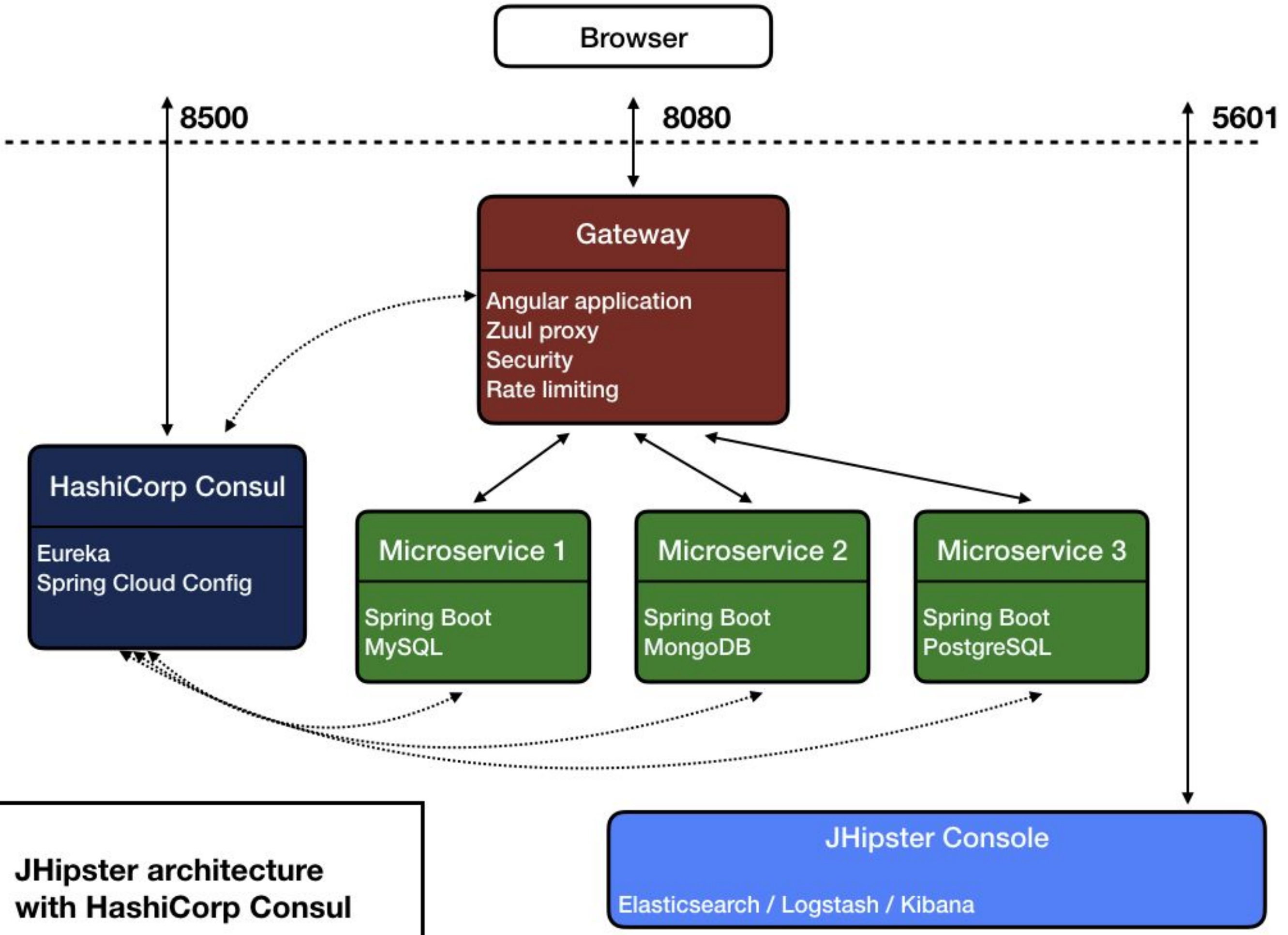
Alternatives to Netflix OSS

JD

Consul

- Service discovery system from HashiCorp
- Open Source
- Written in Go
- https://www.consul.io/
- 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

DS



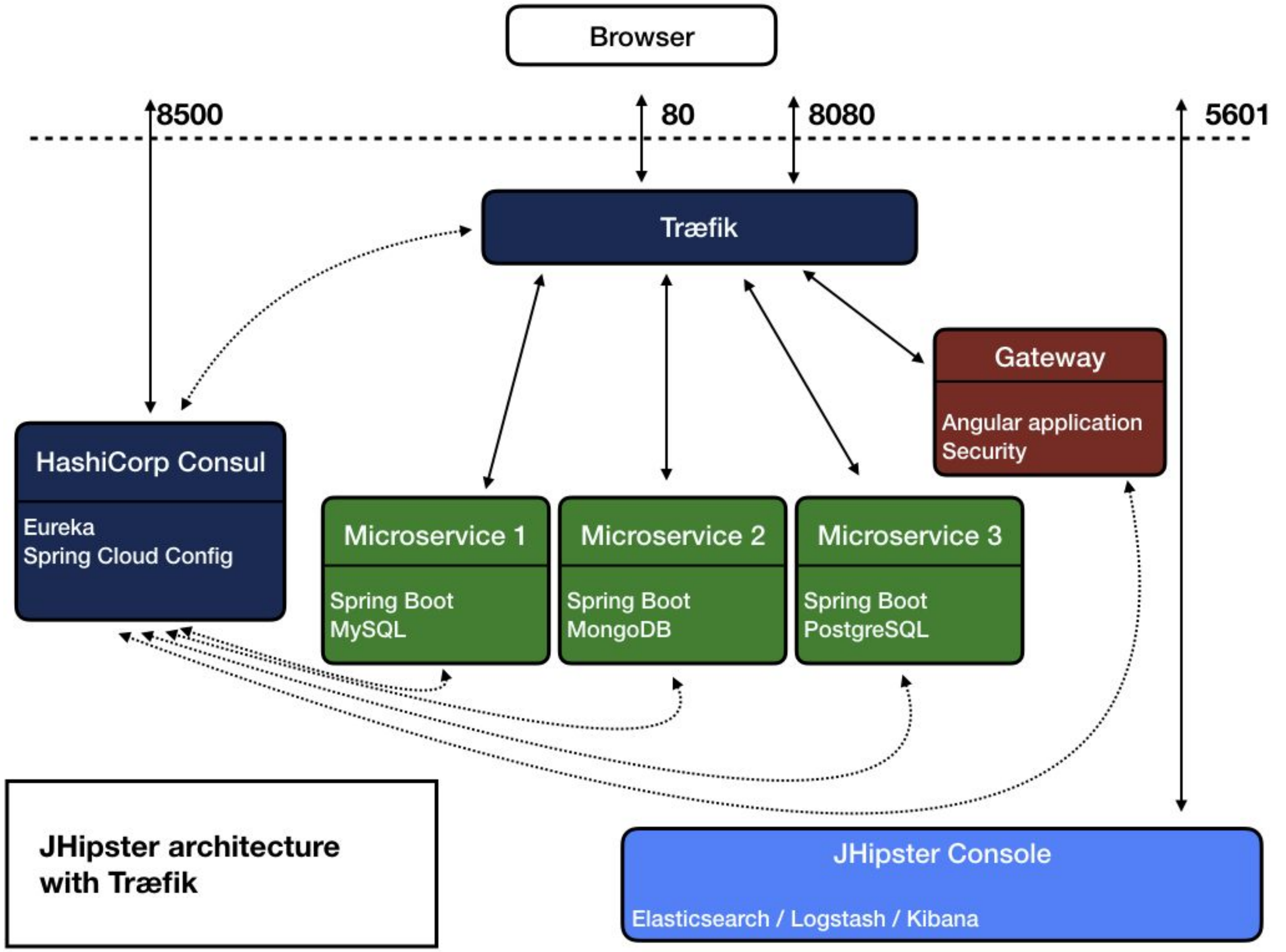
JHipster architecture with HashiCorp Consul

DS

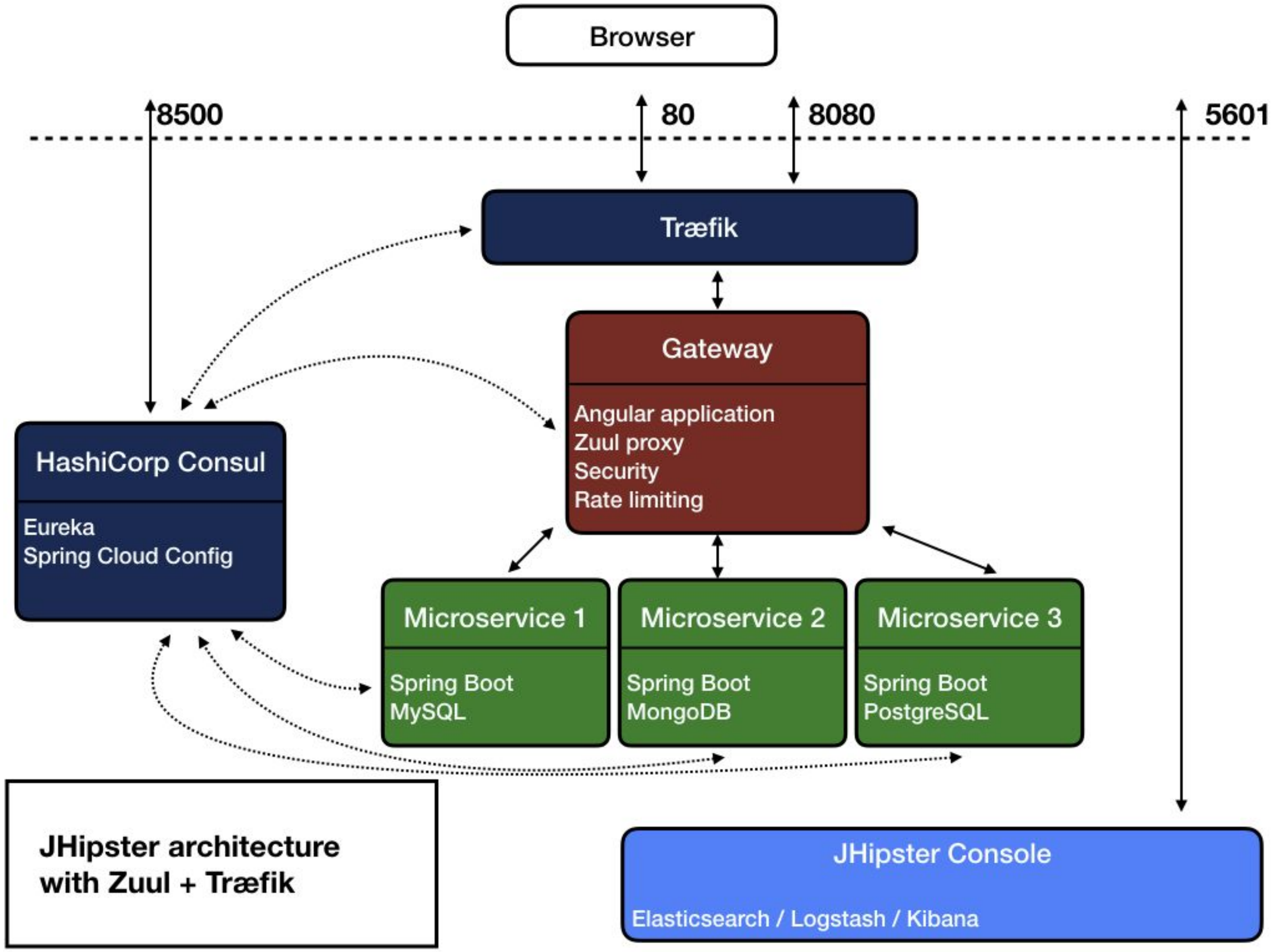
Træfik

- HTTP reverse proxy and load balancer
- Open Source
- Written in Go
- https://traefik.io/
- 2 patterns are possible
  - Replace Zuul completely by Træfik
  - Use Zuul and Træfik together

JD



JHipster architecture with Træfik


Browser
8500
80
8080
5601
Træfik
Gateway
Angular application
Zuul proxy
Security
Rate limiting
HashiCorp Consul
Eureka
Spring Cloud Config
Microservice 1
Spring Boot
MySQL
Microservice 2
Spring Boot
MongoDB
Microservice 3
Spring Boot
PostgreSQL
JHipster architecture
with Zuul + Træfik
JHipster Console
Elasticsearch / Logstash / Kibana

JD

Consul and Træfik demo

- Generate a simple gateway and a simple microservice
  - By default you have the Zuul+Træfik pattern, as the gateway uses relative URLs
  - If you want absolute URLs and use Træfik directly, just configure the URL constant in the gateway’s Webpack configuration
- Use Consul and Træfik
- Run everything!

Security with microservices

JD

HTTPS

- HTTPS support comes built-in with a JHipster application
  - 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
- Træfik 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 an official Let’s Encrypt support

JD

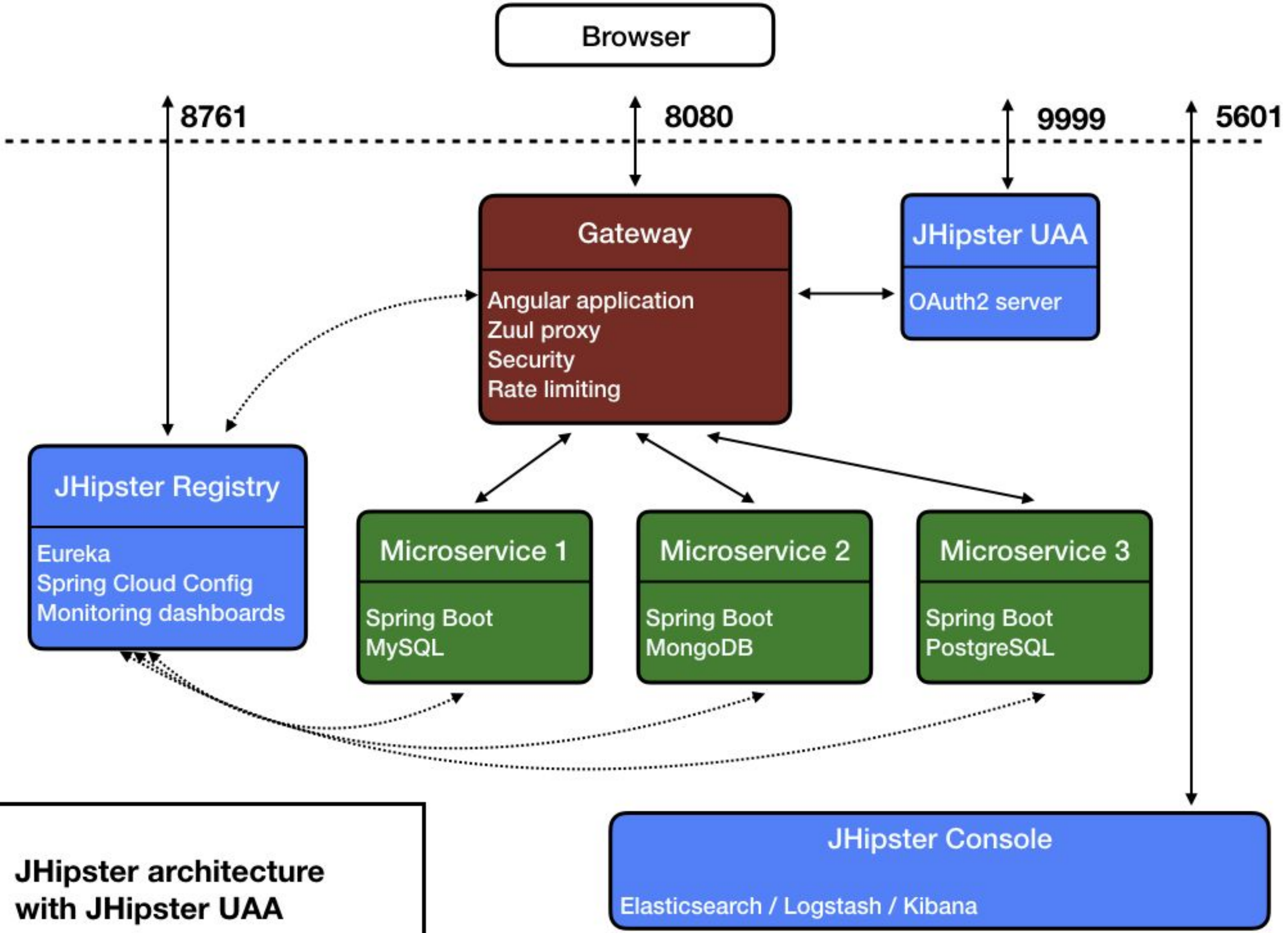
JWT

- Our most popular and easy-to-use option
- Stateless, signed token that all microservices can share and trust
- By default, the JHipster gateway generates a JWT
  - It sends it to the various microservices
  - 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

JHipster UAA

- A mix of a JHipster application 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 microservices architectures
- Has to be generated for your microservices architecture
    - Can easily be tuned and customized
- Provides OAuth2 tokens to all applications

DS



JHipster architecture with JHipster UAA

OpenID Connect

- Provides an identity layer on top of OAuth2
  - Standard with many implementations
  - Starts to be widely used across enterprises
- Great for microservices architecture
  - User management, authentication and authorizations are handled by a third-party OpenID Connect implementation
- JHipster support is very new (latest release!)
  - Support two major OpenID Connect implementations: Okta and Red Hat Keycloak

OpenID Connect demo

- Generate a simple gateway and a simple microservice
- Use Keycloak as OpenID Connect provider
- Run everything!

Monitoring

JD

JHipster Registry

- 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

JHipster Console

- Based on the Elastic Stack
  - Logstash, 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

Prometheus

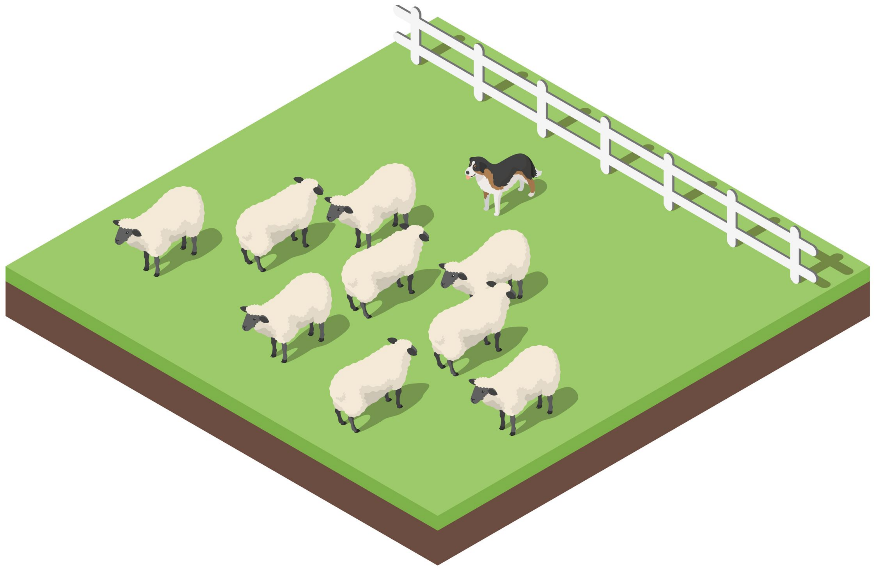
- Open Source monitoring system, alternative to the JHipster Console
- Multi dimensional data model
- Great for time series
- Flexible Queries and Grafana based visualizations
- Alerting
- Written in Go
- https://prometheus.io/

DS

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
- Zipkin helps to troubleshoot latency problems in microservice architectures
- Supports in-memory, JDBC (mysql), Cassandra, and Elasticsearch as Storage options

Scaling microservices



JD

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 applications, but it doesn’t work well in a microservices architecture

JD

Scaling caches

- No cache
  - The application scales easily :-)
  - But sends all the load to the database, which doesn’t scale easily :-(
- Ehcache
  - Adds 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 (not recommended)
  - Default option for JHipster microservices
- Infinispan
  - Adds nodes on-the-fly using the JHipster Registry
  - Great alternative to Hazelcast

JD

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”

JD

Failover

- When a node fails, it is managed by the underlying cloud infrastructure
  - Hopefully it will be re-started
- The service discovery mechanism should alert other services
  - Eureka can take 30-60 seconds to remove an old node
  - Consul should be much quicker
- HTTP request routing should handle failure
  - Zuul is specifically tuned by JHipster, so a failing node is quickly ignored (before being removed by Eureka)
  - Feign allows to configure a fallback

Monitoring + Scaling + Failover demo

- Use the gateway and the 2 microservices from the first demo
- Configure everything with Docker Compose
  - Add JHipster Console monitoring
- Run the stack
  - Use the JHipster Console dashboards to monitor the application
  - Scale the microservice
  - Crash some of the microservice nodes

JD

Continuous delivery

DS

Testing options - server side - JUnit

- De Facto standard for unit testing in Java
- Junit tests are generated out of the box for most of the code
- Run using ./`mvnw test` or ./`gradlew test`

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`

DS

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`

DS

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

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`

DS

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`

DS

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.

DS

Going to production

JD

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

JD

Q & A