Microservice Architecture Blueprint Episode 4 “Finding Nemo“

Kim Horn

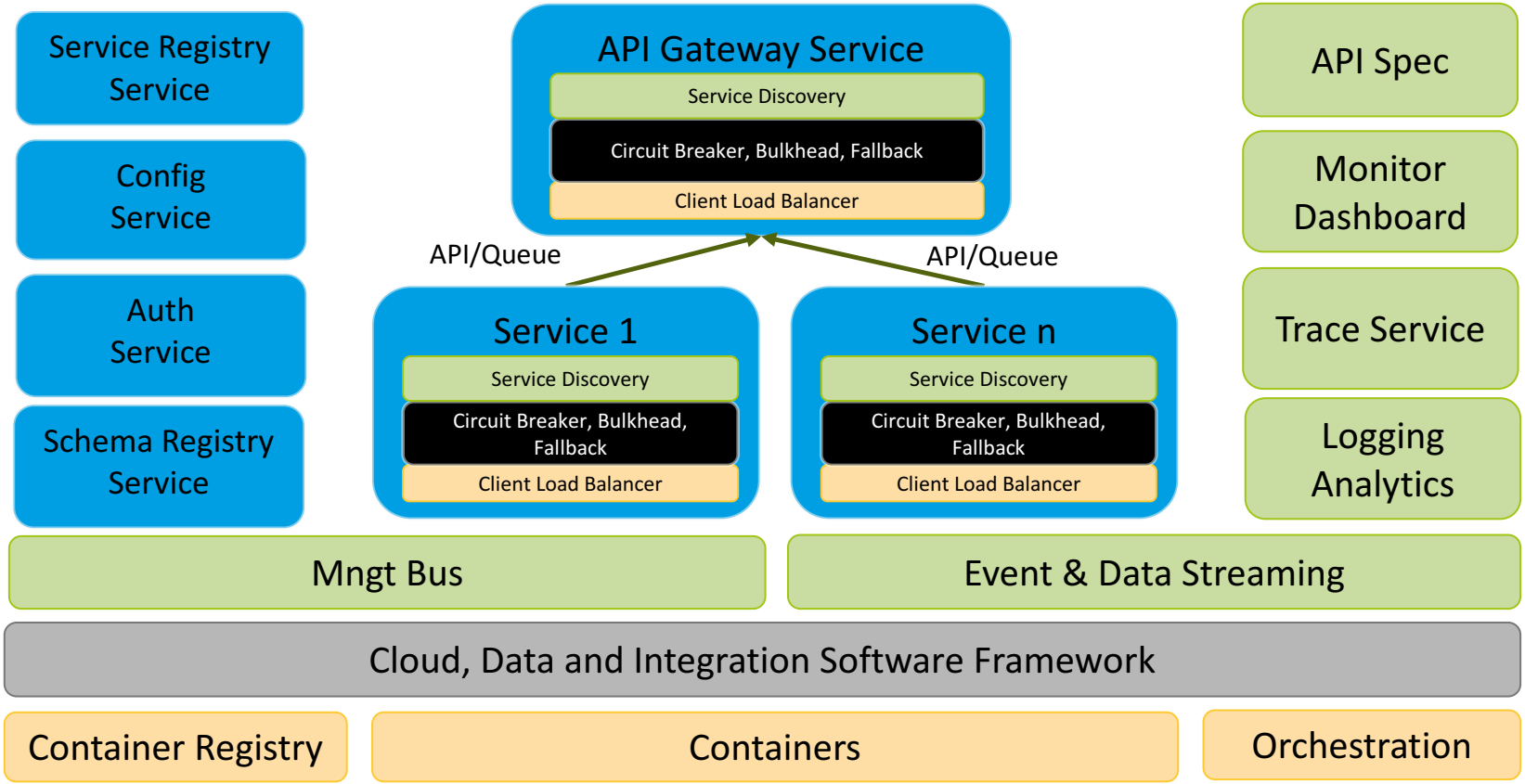
Version 1.3 **1 December 2017**

Agenda

- Configuration Server - Spring
- API Gateway Server - Zuul
- Registry Server - Eureka
- Service Discovery - Feign
- API Specs - Swagger
- Appendix
  - Zuul 2, supports Push Messaging
  - Eureka 2

Working Blueprint

Distributed Cloud Blueprint – NFRs, use what you need



Common Tech For Patterns and Stack

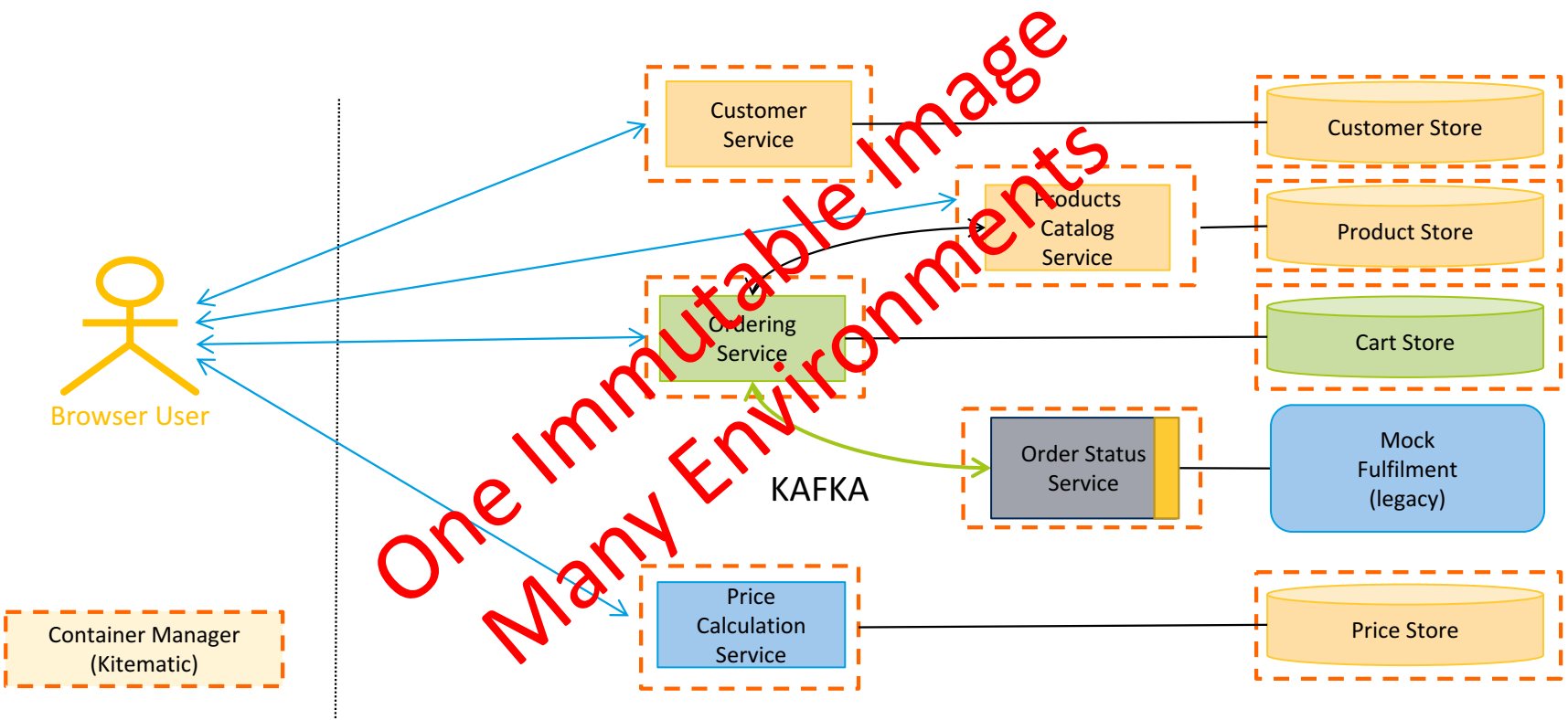
Pattern / Capability	Tech
Gateway (reverse proxy)	Zuul 1&2 (Netflix) – wrapped by SpringCloud, Spring Cloud Gateway
Circuit Breaker, Bulkhead, Fallback	Hysterix (Netflix) – wrapped by SpringCloud
Client Side Load Balancer	Ribbon (Netflix) – wrapped by SpringCloud
Service Discovery Registry	Eureka (Netflix) – wrapped by SpringCloud / Consul
REST HTTP and Service Discovery Client	Feign (Netflix) – wrapped by SpringCloud
Configuration Service	Spring Cloud Configuration
Authorisation Service	Spring Cloud Security, SAML, OATH, OpenID
Event Driven Architecture, Mngt Bus	Spring Cloud Stream wraps Kafka (also supports Rabbit MQ), Spring Cloud Bus
Data Analysis and Flow Pipelines	Spring Cloud Data, Spring Cloud Data Flow, Sparks
Core Software Cloud Frameworks	Spring Boot, Spring Actuator, Spring Cloud, Spring Data, Spring Integration
IaaS, Container, Orchestration	Docker, Compose [Kubernites, Swarm]
Logging, Tracing & Analysis	SL4J, ELK[Elastisearch, Logstash, Kibana], Sprint Cloud Sleuth, Spring Cloud Zipkin.
In memory Cache	Spring Data Redis
Analytics, Monitoring Dashboard, Orchestration(bpm)	Atlas, Turbine, Visceral, Camuda, Conductor (lightweight)
API/ Service Specs, Schema Registry,Testing	Spring Hateoas (HAL), Spring Fox (Swagger), Avro, Protobuf, Spring Cloud Contract, WireMock

Microservice Principles

A Microservice is:

1. Modelled around a bounded context; the business domain.
2. Responsible for a single capability;
3. Based on a culture of automation and continuous delivery;
4. The owner of its data;
5. Consumer Driven;
6. Not open for inspection; Encapsulates and hides all its detail;
7. Easily observed;
8. Easily built, operated, managed and replaced by a small autonomous team;
9. A good citizen within their ecosystem;
10. Based on Decentralisation and Isolation of failure;

RECAP - 9 Containers



Pattern: Isolation, Build and Deployment Pipeline, Immutable Servers

Configuration

“Store Configurations in the Environment”

- **Strict separation of Configuration from Code.**
  - Don’t use Constants in Code or Config Files that are part of source tree or repo – Bad Practice, but common.
  - Scope includes app’s configuration that is likely to vary between deploys and on the fly in all our environments.
- **One Immutable code image for all Environments;**
- **Store configuration in environment variables;**
  - These are language and OS agnostic.
  - Environment variables are easy to change between deploys without changing any code;
  - Originate configuration in a separate Configuration Repo (backend).

Configuration Principles

1. **Segregate.** Completely separate the services configuration information from the actual physical deployment of a service. Application configuration should not be deployed with the service instance. Instead configuration information should either be passed to the starting service as environment variables or read from a centralized repository when the service starts.
2. **Abstract**. Abstract the access of the configuration data behind a service interface. Rather than writing code that directly accesses the service repository (e.g. read the data out of a file or a database using JDBC), have the application leverage a JSON service to retrieve the configuration data.
3. **Centralise.** Centralise your application configuration into as few repositories as possible.
4. **Harden.** As configuration information is completely segregated from a deployed service and centralised it is critical that the configuration server be highly available and redundant.
5. **Version Managed**. Application configuration data needs to be tracked and version-controlled.
6. **Encryption.** Encrypt sensitive configuration information.

Configuration Server

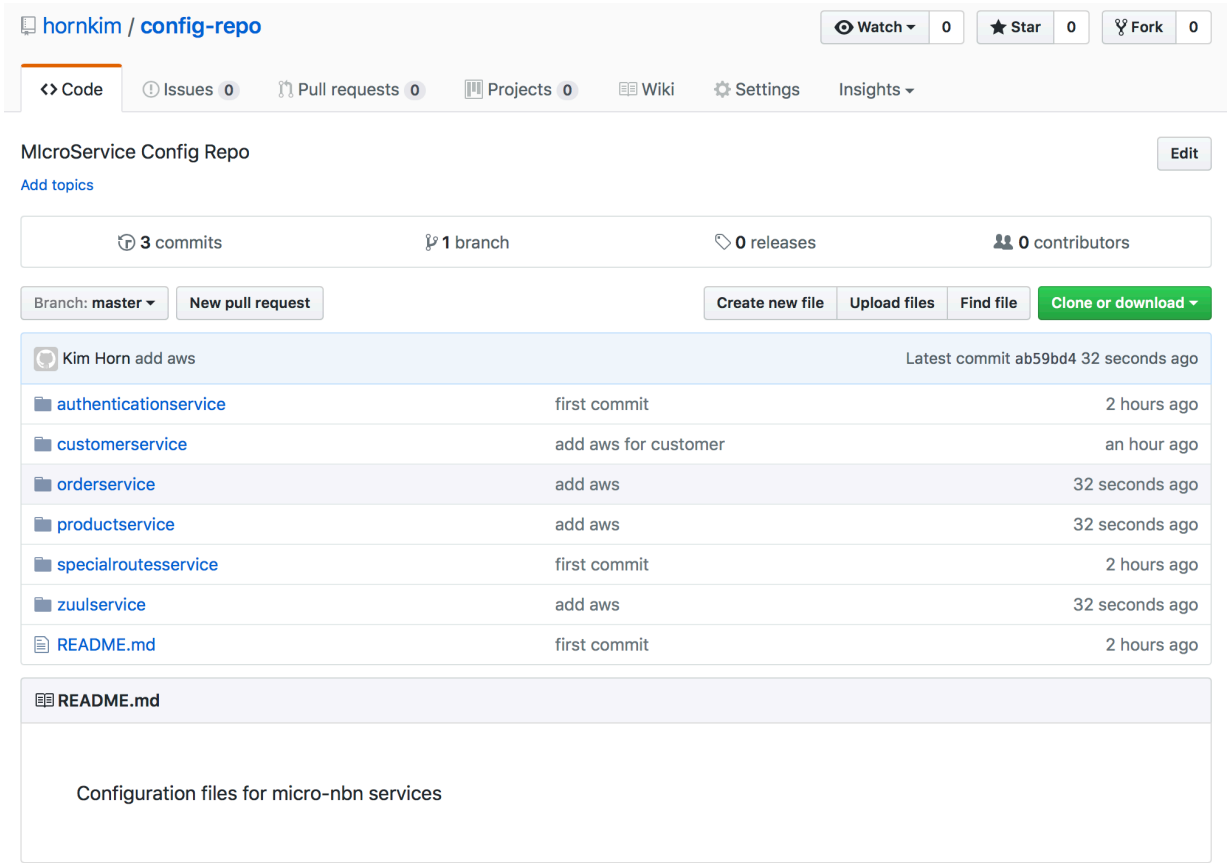
Spring Cloud Config is Spring's client/server approach for storing and serving distributed configurations across multiple applications and environments.

- Horizontally scalable.
- Support encryption and decryption of property values, so they can use public repositories as storage for sensitive data like usernames and passwords.

Pluggable repository backend choices:

- Git
  - Push Notifications (simply annotate you code to poll for changes)
- Filesystem – used for demo
- Vault – a tool for securely accessing secrets.
- Composite of these.

Configuration Git Repository



hornkim / config-repo

Watch 0 | Star 0 | Fork 0

Code | Issues 0 | Pull requests 0 | Projects 0 | Wiki | Settings | Insights

MIcroService Config Repo

Edit

Add topics

3 commits | 1 branch | 0 releases | 0 contributors

Branch: master | New pull request | Create new file | Upload files | Find file | Clone or download

Kim Horn add aws — Latest commit ab59bd4 32 seconds ago

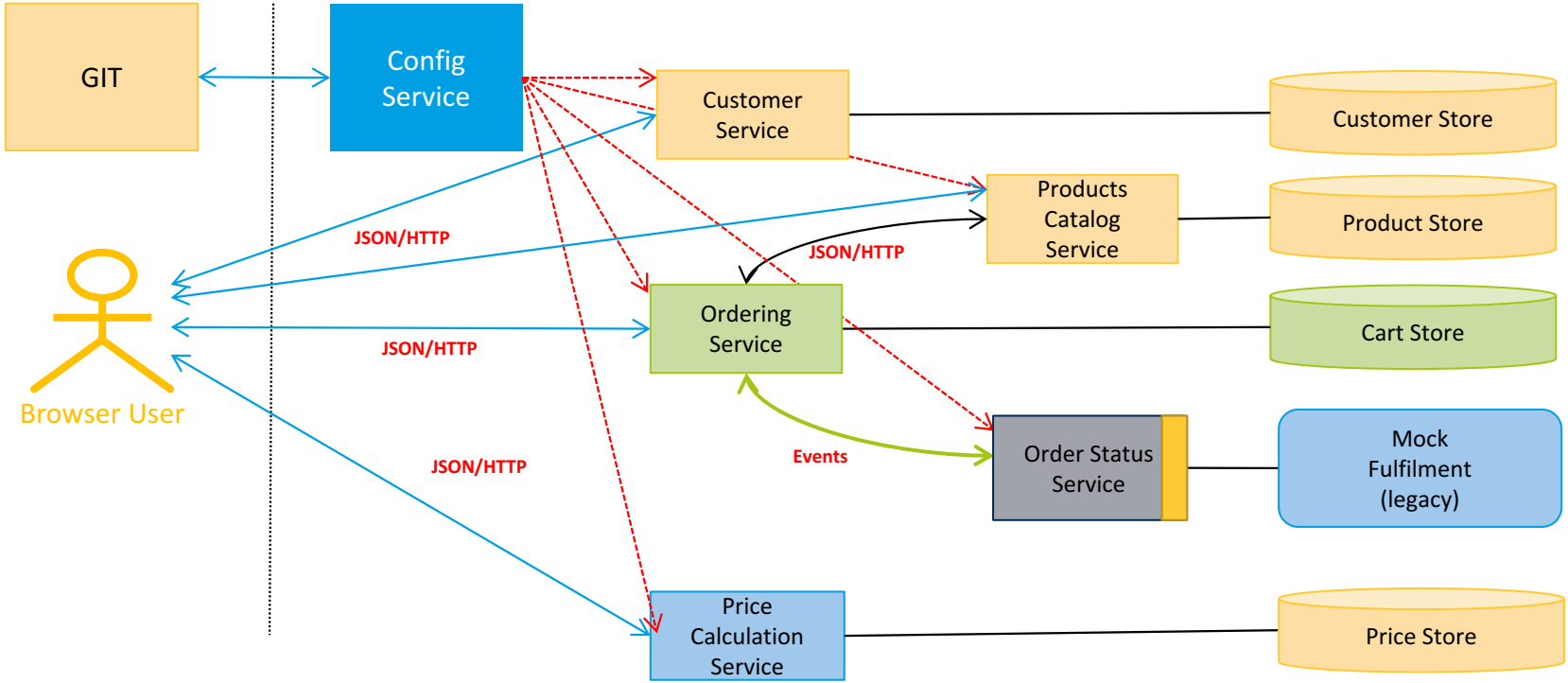
authenticationservice	first commit	2 hours ago
customerservice	add aws for customer	an hour ago
orderservice	add aws	32 seconds ago
productservice	add aws	32 seconds ago
specialroutesservice	first commit	2 hours ago
zuulservice	add aws	32 seconds ago
README.md	first commit	2 hours ago

README.md

Configuration files for micro-nbn services

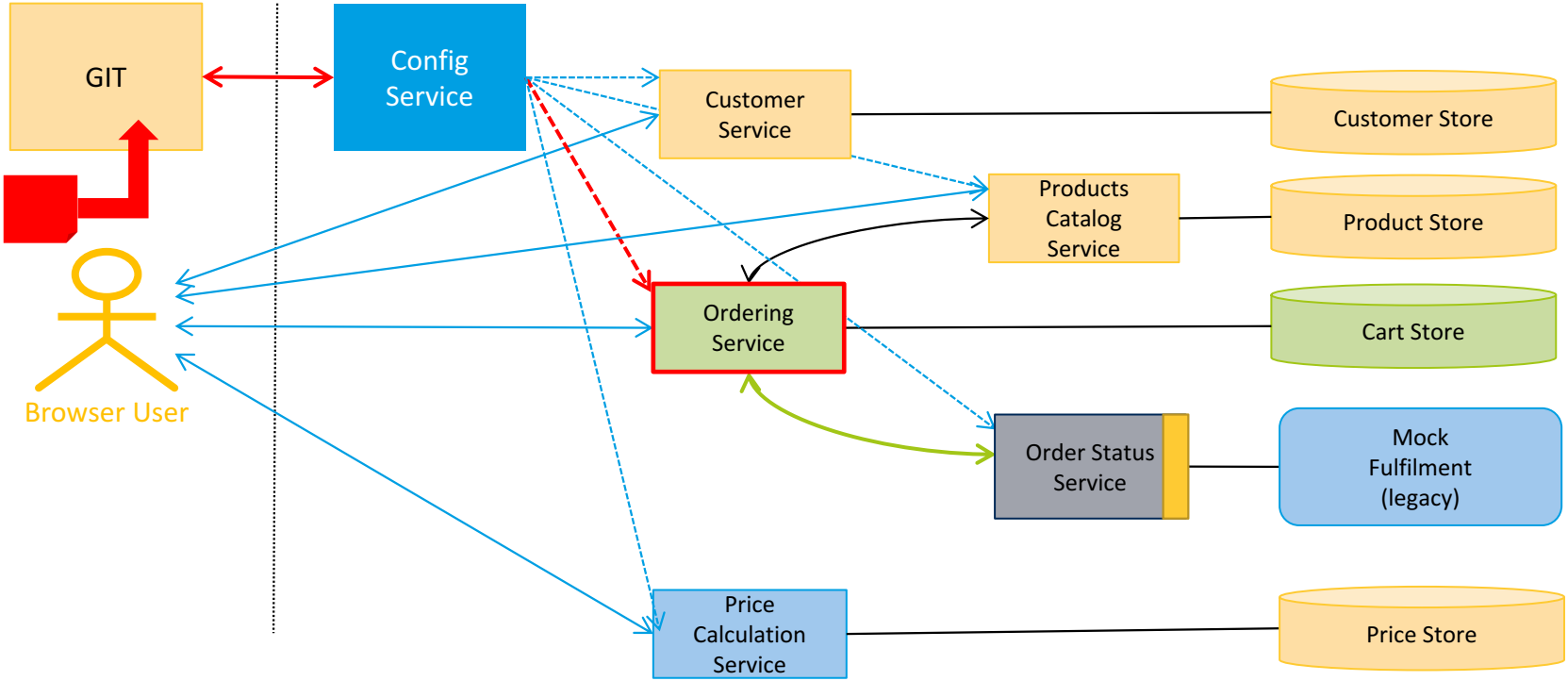
Add Configuration: 6 Services + 1 Repo

DP



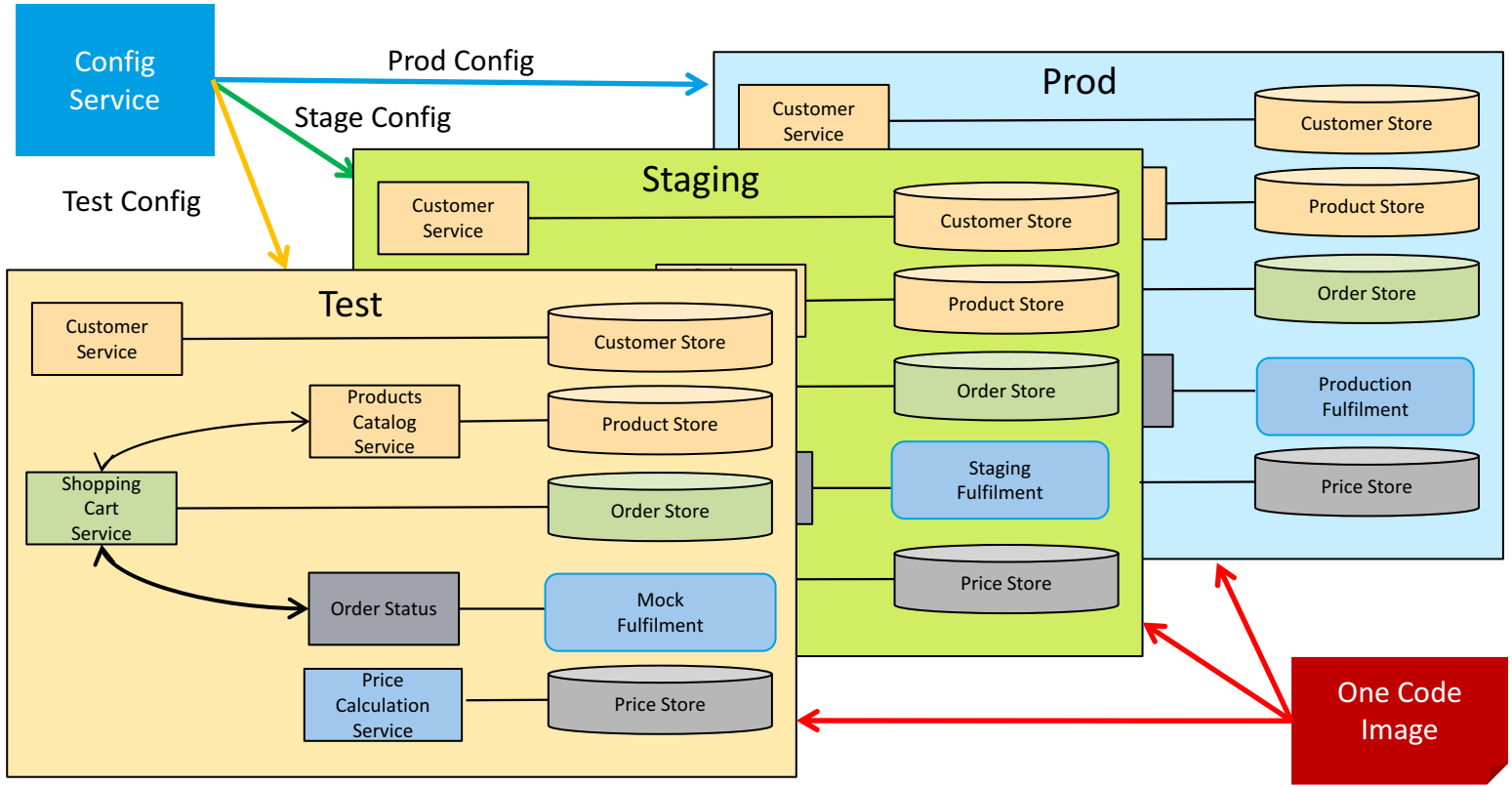
Pattern: Configuration, Protocol, Event Driven

Live Changes can be Pushed Out



Configure all Environments

DP



Configuration API

The Git-backed configuration API provided by the server can be queried using the following paths:

/{application}/{profile}[/{label}]

/{application}-{profile}.yml

/{label}/{application}-{profile}.yml

/{application}-{profile}.properties

/{label}/{application}-{profile}.properties

Spring Cloud Bus, can be used, to propagate an **EnvironmentChangeEvent** to all applications connected, when GIT is updated.

Get Configurations 'Default'

http://localhost:8888/orderservice/default

```
{
  "name": "orderservice",
  "profiles": [
    "default"
  ],

  "label": null,
  "version": null,
  "state": null,
"propertySources" :
```

```
[
   {
     "name": "classpath:config/orderservice/orderservice.yml",
     "source": {
       "example.property": "I AM THE DEFAULT",
       "spring.jpa.database": "POSTGRESQL",
       "spring.datasource.platform": "postgres",
       "spring.jpa.show-sql": "true",
       "spring.database.driverClassName": "org.postgresql.Driver",
       "spring.datasource.url": "jdbc:postgresql://database:5432/widget_store_local",
       "spring.datasource.username": "postgres",
       "spring.datasource.password": "p0stgr@s",
       "spring.datasource.testWhileIdle": "true",
       "spring.datasource.validationQuery": "SELECT 1",
       "spring.jpa.properties.hibernate.dialect": "org.hibernate.dialect.PostgreSQLDialect"
     }
   }
  ]
}
```

Get Configurations 'Dev'

http://localhost:8888/orderservice/dev

```
{
  "name": "orderservice",
  "profiles": [
    "dev"
  ],

  "label": null,
  "version": null,
  "state": null,
"propertySources" :
```

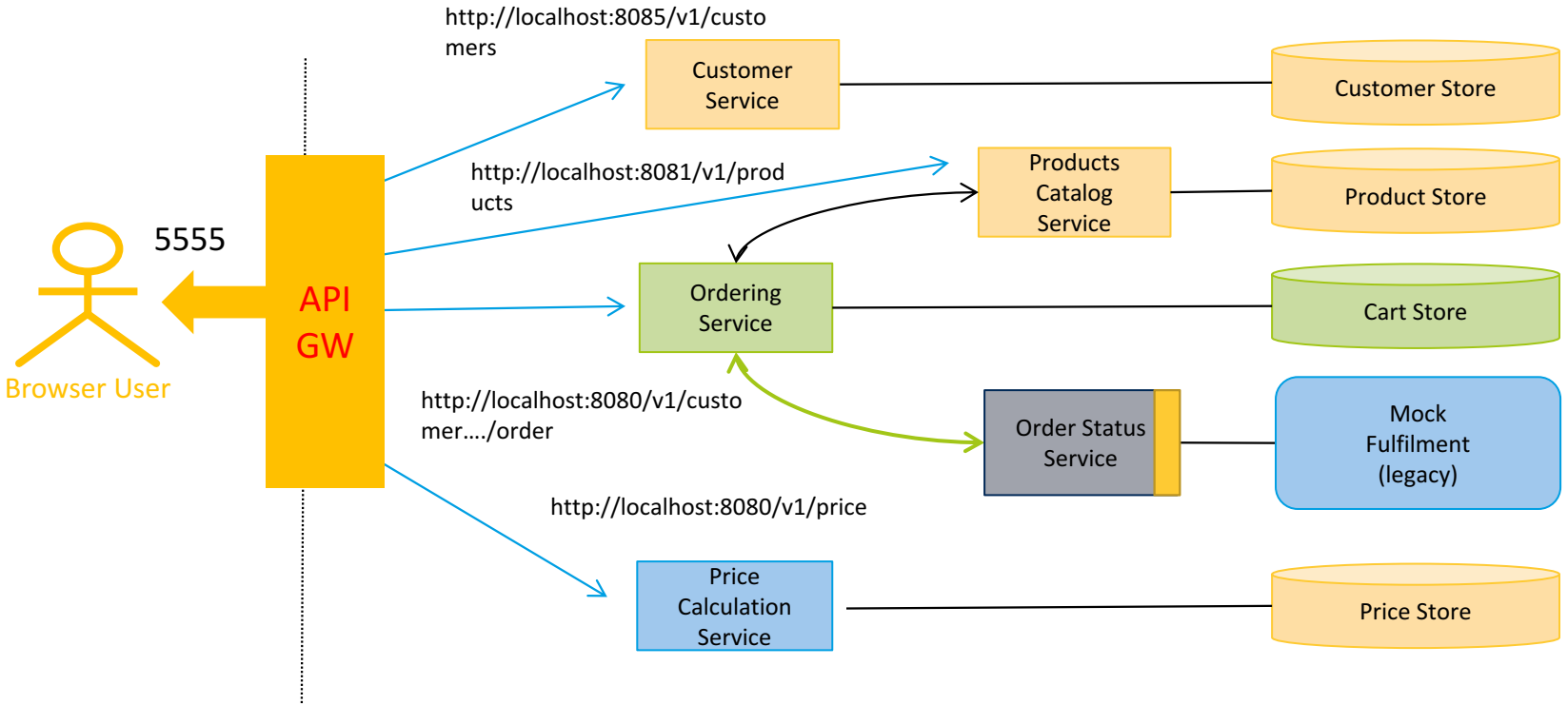
```
[
   {
     "name": "classpath:config/orderservice/orderservice-dev.yml",
     "source": {
       "spring.jpa.database": "POSTGRESQL",
       "spring.datasource.platform": "postgres",
       "spring.jpa.show-sql": "true",
       "spring.database.driverClassName": "org.postgresql.Driver",
       "spring.datasource.url": "jdbc:postgresql://database:5432/widget_store_dev",
       "spring.datasource.username": "postgres_dev",
       "spring.datasource.password": "p0stgr@s_dev",
       "spring.datasource.testWhileIdle": "true",
       "spring.datasource.validationQuery": "SELECT 1",
       "spring.jpa.properties.hibernate.dialect": "org.hibernate.dialect.PostgreSQLDialect"
     }
   },
   {
     "name": "classpath:config/orderservice/orderservice.yml",
     "source": {
       "example.property": "I AM THE DEFAULT",
           ..............
           ..............
  ]
}
```

Dev Config

Default Config

API Gateway Centralised

Can we make life easier for Client ?



API Gateway Features

- Encapsulate details of organisation or subdomain µservices – Protect Bounded Contexts;
- Static, Dynamic, Creative and Surgical Routing;
- Traffic Shaping – Throttling, load shedding;
- PEP:
  - Logging;
  - Monitoring;
  - Security (Authentication, Authorisation);
- Insights – Analytics;
- Provide vanity urls, via configuration: “/api/orderservice/….”
- Pre/Post Filter, e.g. remove sensitive information, add Correlation IDs, custom;
- Client Load Balancing, Health and Error Handling;
- Cache certain responses, e.g. default.
- Granularity - collect fine grained calls to coarse grained (orchestrate fined grained µservices);
- Protocol Abstraction (XML, JSON, TEXT, GraphQL)
- Applied at organisation Edge and Internally;
- Easily and cheaply deploy multiple instances, whenever and wherever needed, e.g. Back Ends for Front Ends.

Creative and Surgical Routing

Route to Subclusters for:

- Blue/Green (Red/Green) deployments;
- A/B Testing;
- Dev/Test code branches - In Vivo Vs In Vitro testing;
- Environment specific endpoints: prod, dev, test, etc..
- Instrumented Clusters (trickling traffic);
- Debug Routing;
- Service Canarying;
- Squeeze Testing;
- Failure Injection;
- Degraded Experience Testing;
- Surgical Routing – target and isolate a specific customer.
- Strangler Pattern –route old and new traffic appropriately.


NETFLIX ZUUL
github.com/Netflix

ZUUL Netflix's Goals

Use filters to quickly and nimbly apply functionality to edge service. These filters perform the following functions:

- **Authentication and Security** - identifying authentication requirements for each resource and rejecting requests that do not satisfy them.
- **Insights and Monitoring** - tracking meaningful data and statistics at the edge in order to give us an accurate view of production.
- **Dynamic Routing** - dynamically routing requests to different backend clusters as needed.
- **Stress Testing** - gradually increasing the traffic to a cluster in order to gauge performance.
- **Load Shedding** - allocating capacity for each type of request and dropping requests that go over the limit.
- **Static Response handling** - building some responses directly at the edge instead of forwarding them to an internal cluster.
- **Multiregion Resiliency** - routing requests across AWS regions in order to diversify ELB usage and move our edge closer to members.

Integrated and Cohesive Framework

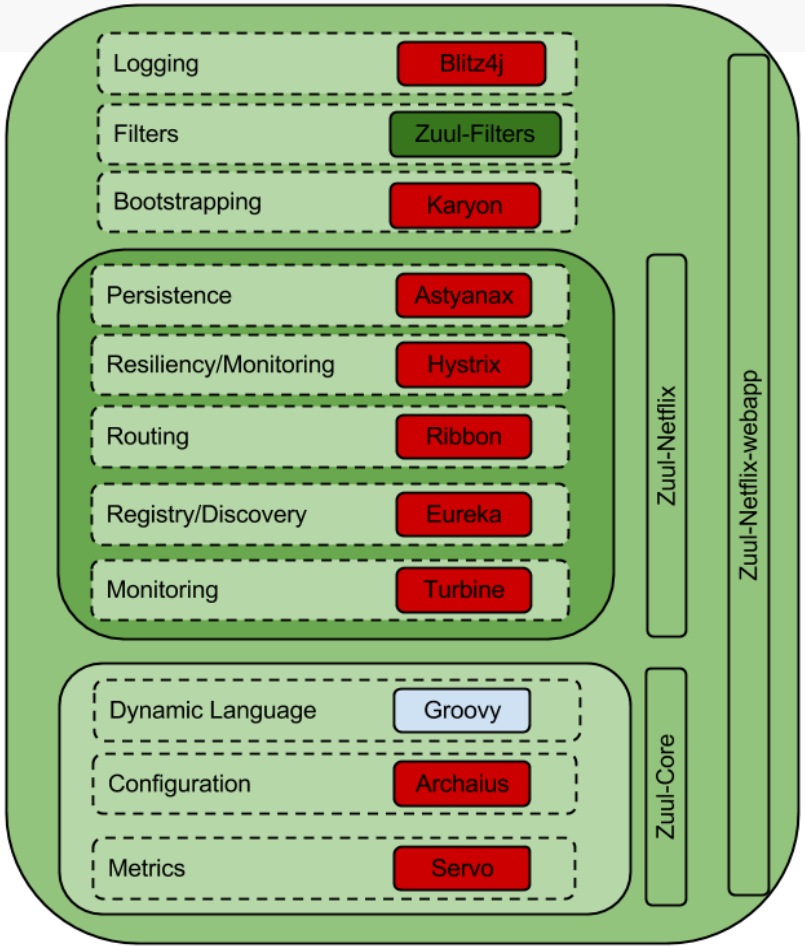
Zuul gives us a lot of **insight**, flexibility, and resiliency, in part by making use of other Netflix OSS components*:

- **Hystrix** is used to wrap calls to our origins, which allows us to shed and prioritize traffic when issues occur;
- **Ribbon** is the client for all outbound requests from Zuul, which provides detailed information into network performance and errors, as well as handles software load balancing for even load distribution;
- **Turbine** aggregates fine grained metrics in realtime so that we can quickly observe and react to problems;

NETFLIX
OSS

*Note these components are used for all services, covered in the next talk, but are mentioned now to introduce the conceptual integrity of the framework;

Zuul OSS



zuul-core — A library containing a set of core features.

zuul-netflix — An extension library using many Netflix OSS components:

Servo for insights, metrics, monitoring

Hystrix for real time metrics with Turbine

Eureka for instance discovery

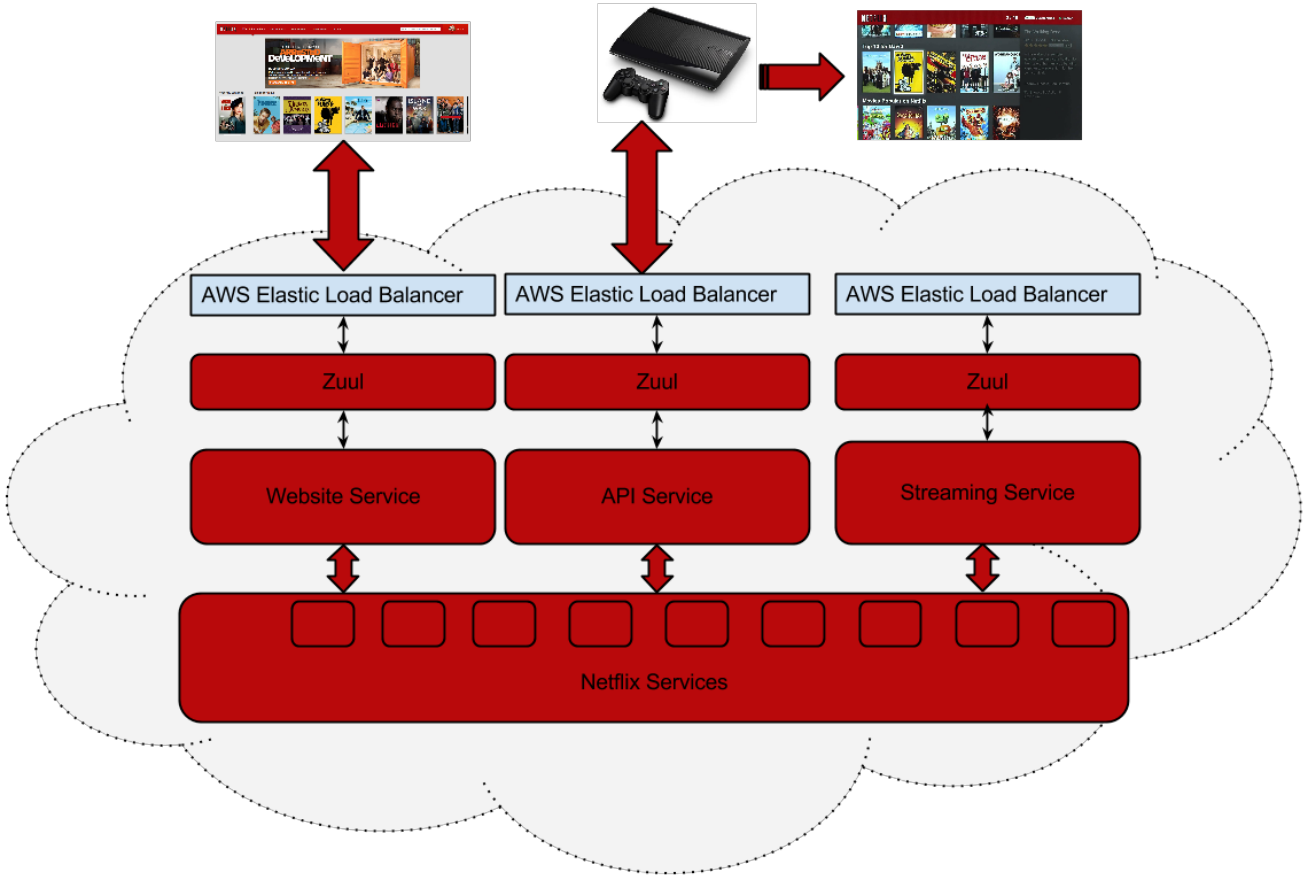
Ribbon for routing

Archaius for real-time configuration

Astyanax for and filter persistence in Cassandra

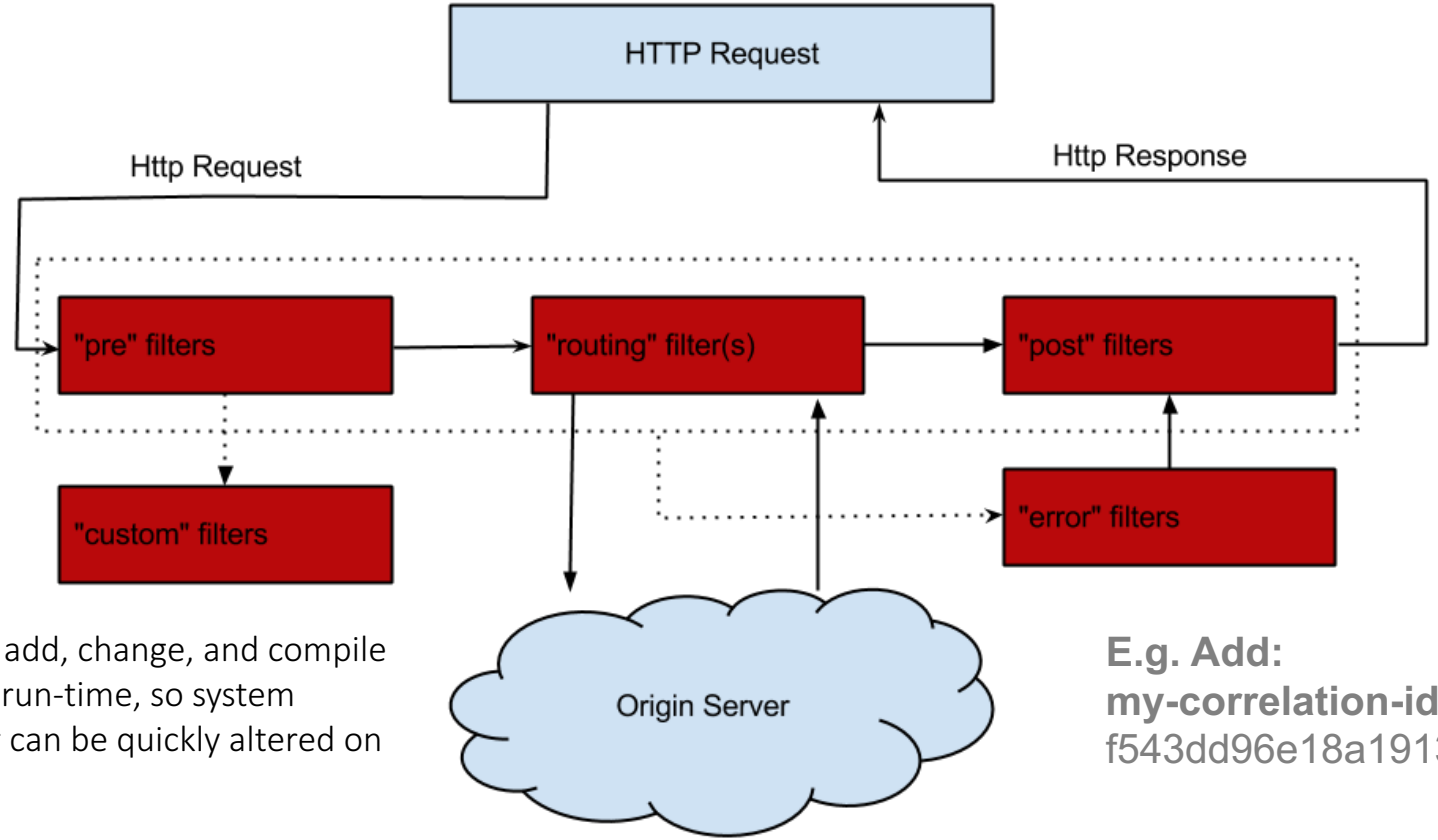
zuul-filters — Filters to work with zuul-core and zuul-netflix libraries

Netflix Zuul used within AWS



Note its not just API services.

ZUUL Request Lifecycle



Zuul can add, change, and compile filters at run-time, so system behavior can be quickly altered on the fly.

E.g. Add:
my-correlation-id →
f543dd96e18a1913

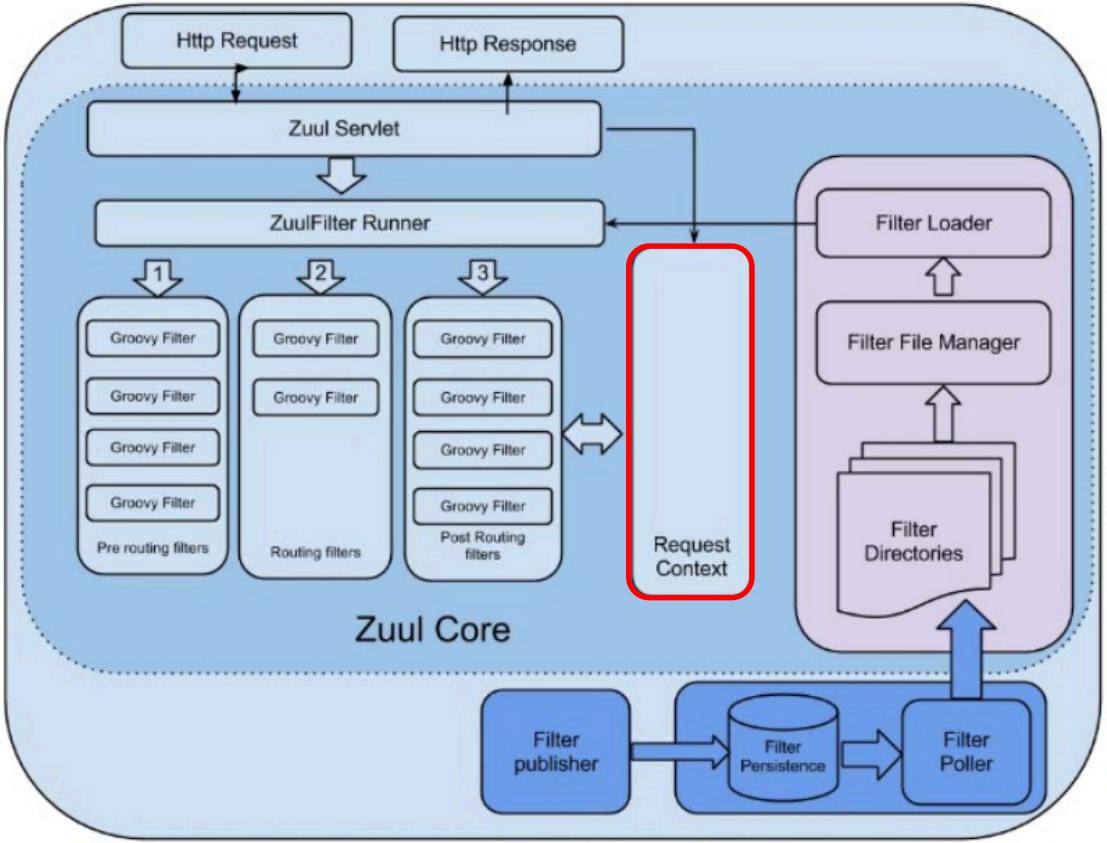
ZUUL Gateway Framework

https://github.com/Netflix/zuul/wiki

Zuul provides a framework to dynamically read, compile, and run these filters.

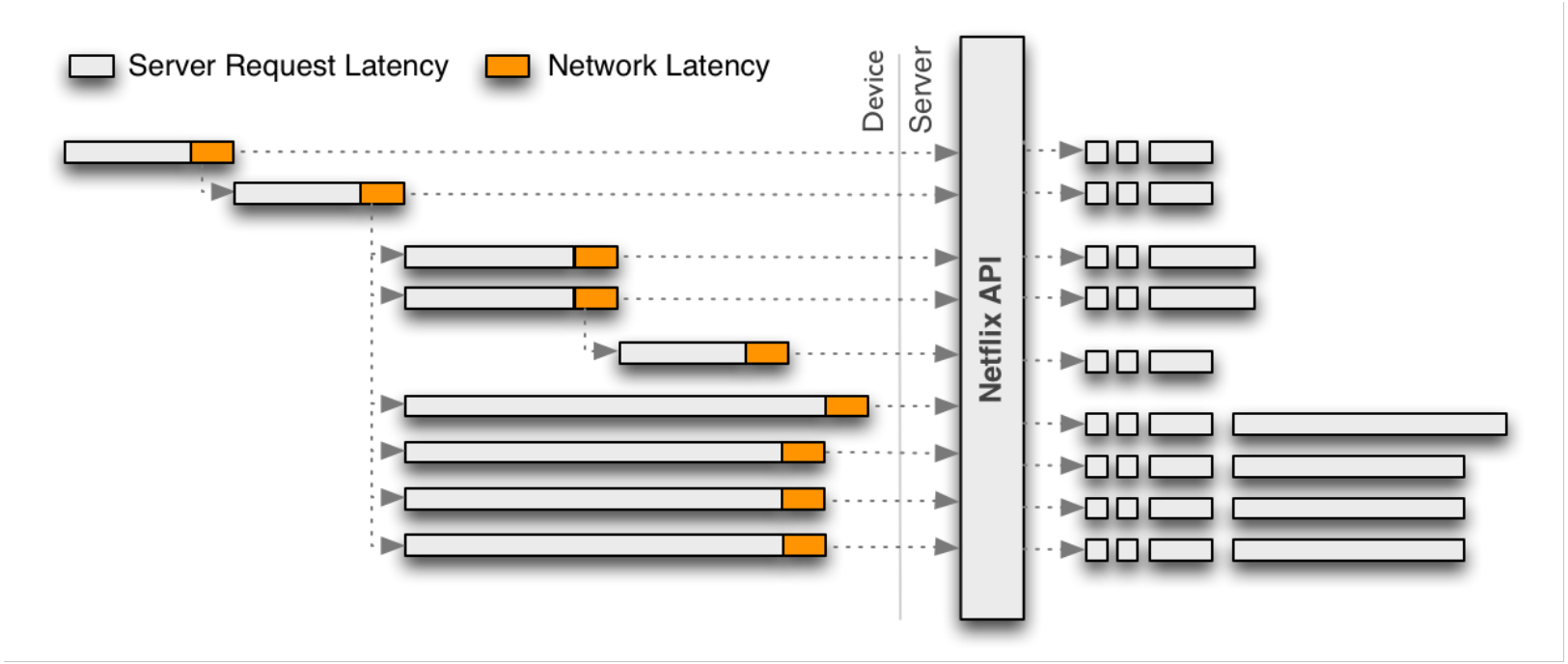
Filters do not communicate with each other directly — instead they share state through a RequestContext which is unique to each request.

Filters written in Groovy or any JVM language, Java.

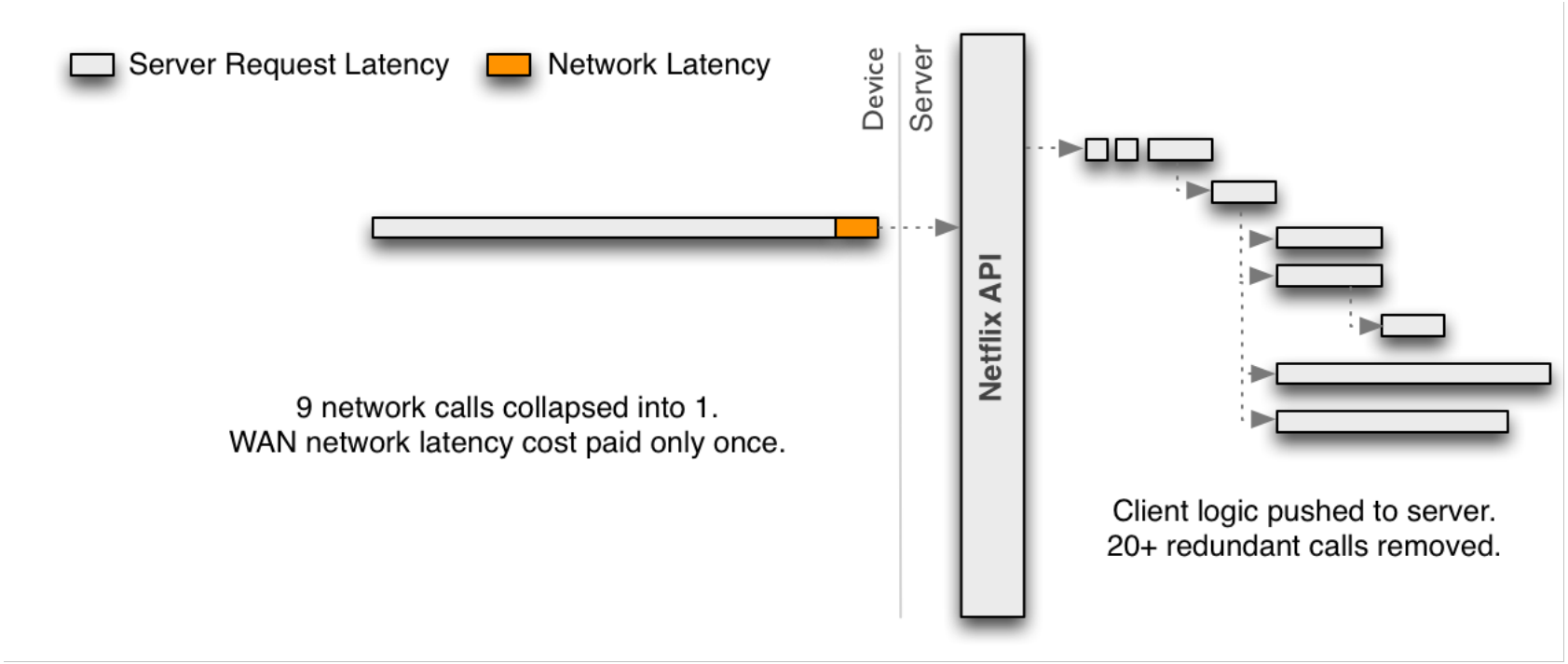


Netflix 'Chatty' REST APIs

Need to improve WAN latency

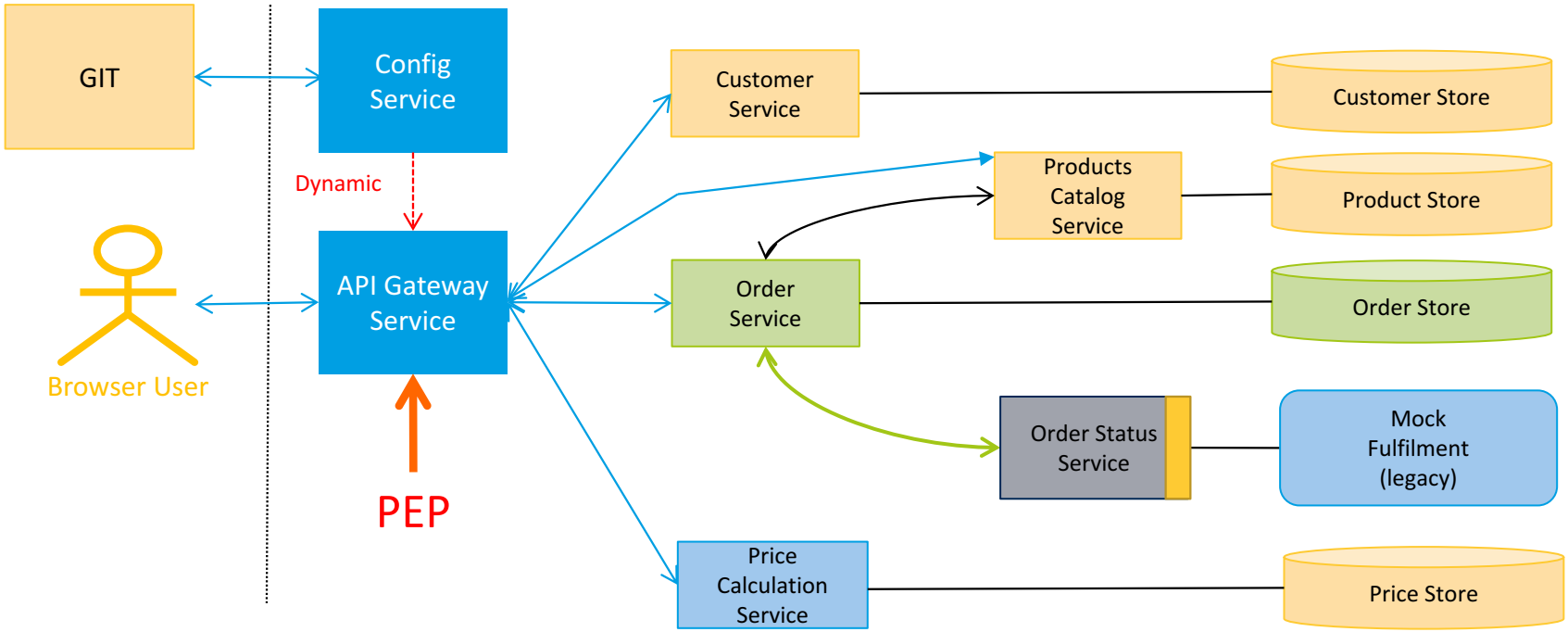


Reducing Chattiness



ZUUL API Gateway : 7 Services

RP



Pattern: Routing, Gateway

Netflix Issue, they said:

“Our REST API, while very capable of handling the requests from our devices in a generic way, is optimized for none of them. This is the case because our REST API focuses on resources that are meant to be granular representations of the data, from the perspective of the data. The granularity is exactly what allows the API to support a large number of known and unknown developers. Because it sets the rules for how to interface with the data, it also forces all of the developers to adhere to those rules. That means that each device potentially has to work a little harder (or sometimes a lot harder) to get the data needed to create great user experiences because devices are different from each other.”

The solution to this issues ……

Solutions, Netflix says:

“To solve this issue, it is becoming increasingly common for companies (including Netflix) to think about the interaction model in a different way. Rather than having the API create a set of rather rigid rules and forcing the various devices to follow them, companies are now thinking about ways to let the UI have more control in dictating what is needed from a service in support of their needs. Some are creating custom REST-based APIs to support a specific device or category of devices. Others are thinking about greater granularity in REST resources with more batching of calls. Some are creating orchestration layers, such as ql.io, in their API system to customize the interaction.

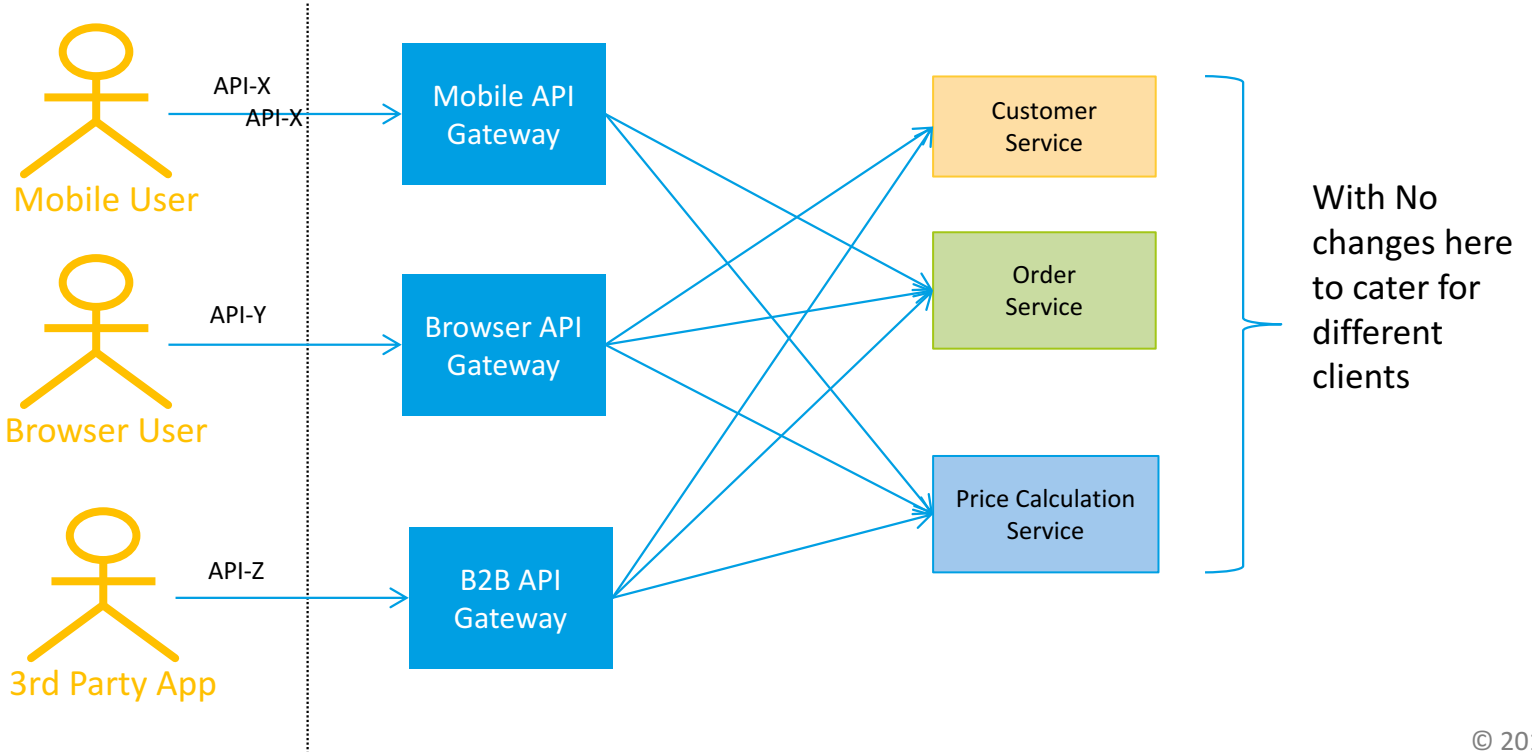
For Netflix, **our goal is to take advantage of the differences of these devices rather than treating them generically**. As a result, we are handing over the creation of the rules to the developers who consume the API rather than forcing them to adhere to a generic set of rules applied by the API team. In other words, we have created a **platform for API development**.”

Note: this leads to a discussion of “Consumer Driven Contract”.

BFFs : BackEnds for FrontEnds - 9 Services

RP

A variation of Gateway is the Backend for Front-End pattern. It defines separate APIs and gateways for each kind of client and device. Vary the content, granularity, detail of API resources for each context.



Moving from one size fits all !

How do you know if your company is ready to consider alternatives to the one-size-fits-all API model?

Here are the ingredients needed to help you make that decision:

- Small number of targeted API consumers is the top priority;
- Very close relationships between these API consumers and the API team;
- An increasing divergence of needs across the top priority API consumers;
- Strong desire by the API consumers for more optimized interactions with the API;
- High value proposition for the company providing the API to make these API consumers as effective as possible;
- If these ingredients are met, then you have the recipe for needing a new kind of API.

Spring Cloud Zuul RateLimit

There are five built-in rate limit approaches:

- **Authenticated User:** Uses the authenticated username or 'anonymous';
- **Request Origin:** Uses the user origin request;
- **URL:** Uses the request path of the upstream service
  - Can be combined with Authenticated User, Request Origin or both;
- **Authenticated User and Request Origin:** Combines the authenticated user and the Request Origin;
- **Global configuration per service:** This one doesn't validate the request Origin or the Authenticated User.

API Gateway Pros and Cons

Pros:

- Provides an easier interface to clients.
- Can be used to prevent exposing the internal microservices structure to clients.
- Allows refactoring of microservices without forcing the clients to refactor the consuming logic.
- Can centralize cross-cutting concerns like security, monitoring, rate limiting, etc.

Cons:

- Can become a single point of failure if the proper measures are not taken to make it highly available.
- Knowledge of various microservice APIs may creep into the API Gateway.
- Business functionaility creeps into Gateway

Service Discovery The Registry

Discovering Services

Given:

- Services may fail, or become unhealthy, be replaced;
- Service may be upgraded;
- Services may have different versions running in parallel;
- Services may be added or removed at any time; autoscale;

and

- Service Instances have IP addresses and Ports, not names;
- IP addresses are allocated dynamically;

How does a client, that may be the gateway, find a service ?

Solution is a **Service Discovery Agent Architecture**.

5 Aspects of Service Discovery

1. **Service Registration** – A service needs to register its details with the Service Discovery agent.
2. **Client Lookup of service address** – Needs to be a mechanism for a client service to look up a producer service information, such as location of a particular version.
   - A robust approach is to allow clients to cache these locations, and load balance them to reduce calls to and dependency on the Service Discovery agent.
3. **Health Monitoring** – Services need to communicate their health back to the Service Discovery agent.
4. **Information Sharing** – To be available, service information (knowledge) including service health, needs to be shared across nodes, across the application.
5. **Consistent Architecture -** Same architecture, approach and components for providers, clients and gateways;

Finding Services - A Registry Service

Employ a **Registry Service** to record meta data:

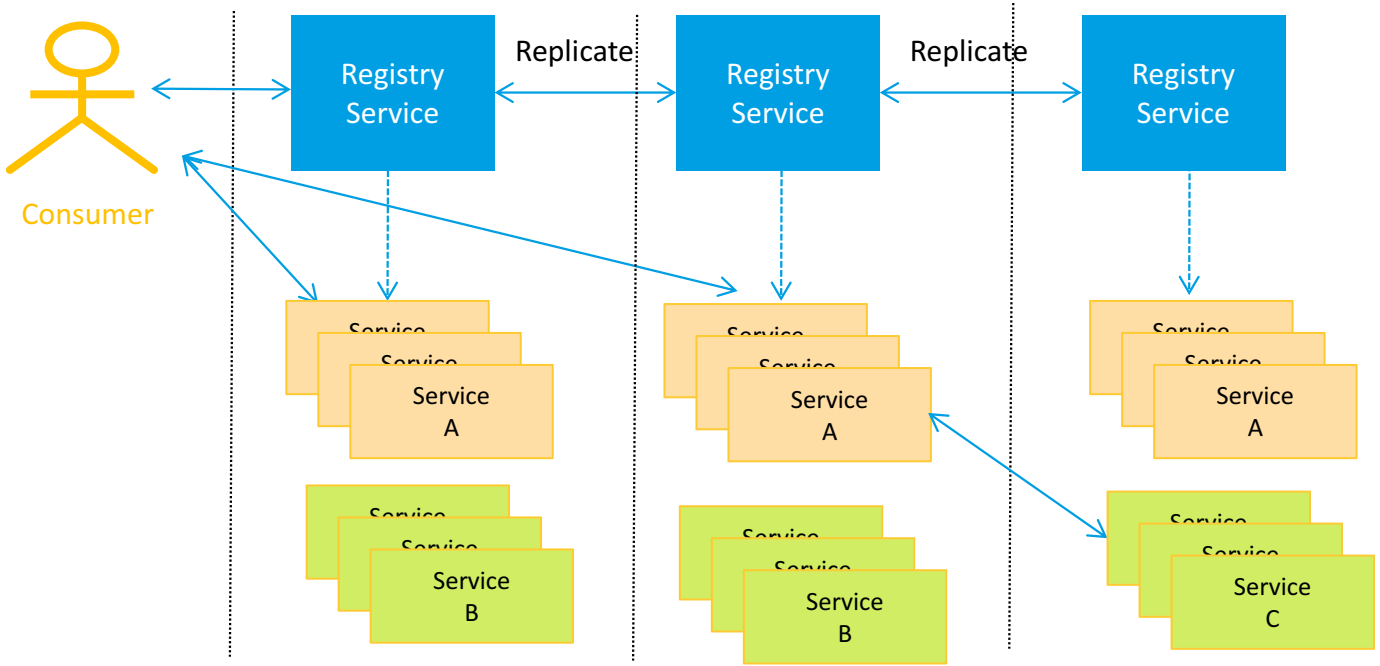
- **Startup:** On startup, the application or service will register with the Registry Server and provide meta-data, such as host and port, health indicator URL, home page, version, etc.
- **Heart Beat:** The registry receives heartbeat messages from each instance belonging to a service. If the heartbeat fails (over a configurable time limit) the instance will be marked as sick, not for use.
- **Cache:** The registry will communicate with clients to help them maintain a cached list of service instances that are available for work within a cluster.
- **ZUUL:** Gateway can now ask what can I expose, and how to reach Instance;

Service Discovery in the Cloud.

- **Scalable.** Quickly scale up or down the number of instances. As the consumers are abstracted away from the physical location of services, via discovery, new instances can be added or removed as required. Supports Autoscaling;
- **Highly Available**. Support "hot" clustering, where service lookups can be shared across multiple nodes in a service discovery cluster. If a node becomes unavailable, other nodes in the cluster can take over.
- **Peer-to-Peer Sharing**. Each node in the service discovery cluster shares the state of a service instance.
- **Load Balanced**. Needs to be able to dynamically load balance requests across all service instances to spread load:
  - Service discovery replaces the more static, manually managed load balancers;
- **Resilient**. The service discovery's client should be able to "cache" service information locally. Allows for gradual degradation of the service discovery feature so that if service discovery service does become unavailable, applications can still function and locate the services based on the local cache.
- **Fault Tolerant**. Needs to detect when a service instance is not healthy and remove the instance from the list of available services automatically.

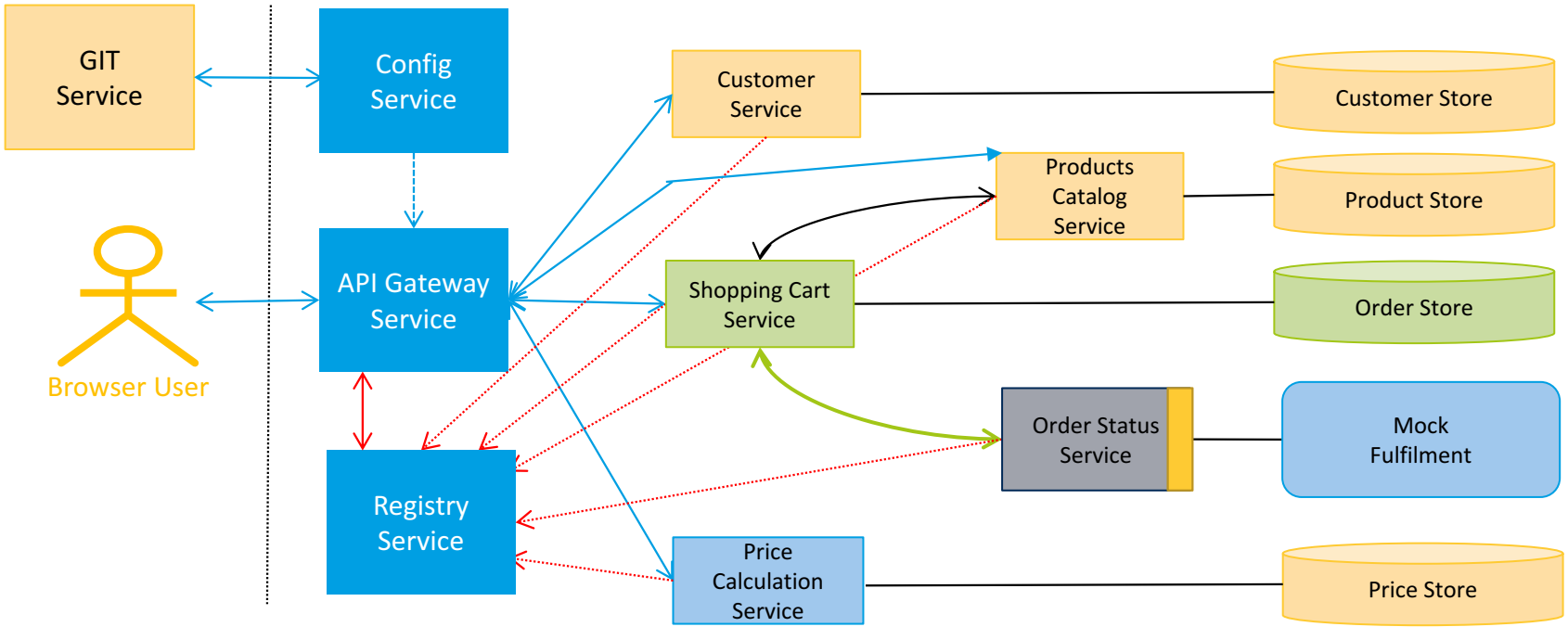
High Availability

When the Eureka server comes up, it tries to get all of the instance registry information from a neighboring node. If there is a problem getting the information from a node, the server tries all of the peers before it gives up.



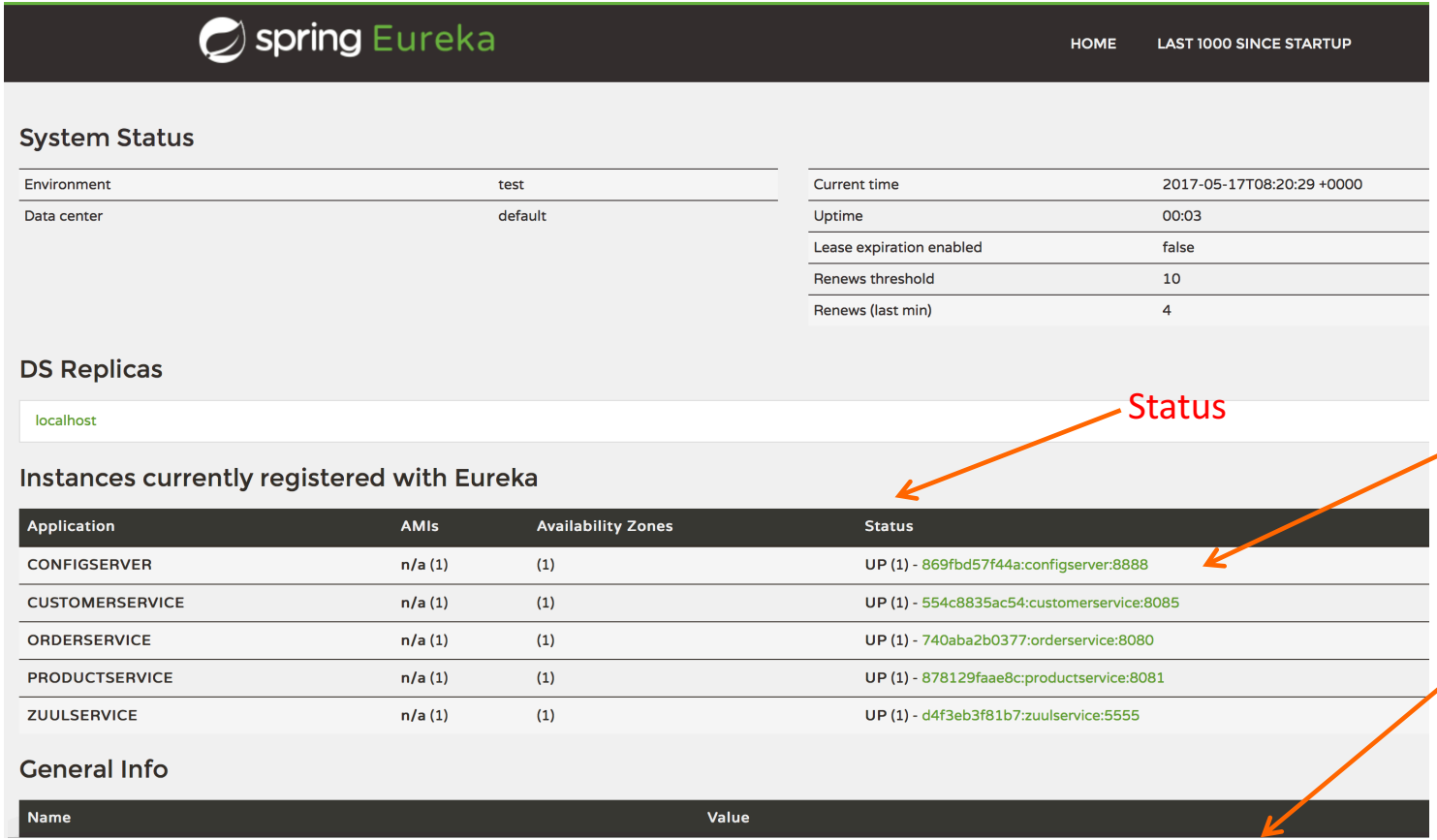
Service Registry : Eureka ! 8 Services

RP



Pattern: Service Registry

Eureka - Registry User Interface



Sample - Eureka JSON API

Operation	HTTP action
Register new application instance	POST /eureka/v2/apps/**appID**
De-register application instance	DELETE /eureka/v2/apps/**appID**/**instanceID**
Send application instance heartbeat	PUT /eureka/v2/apps/**appID**/**instanceID**
Query for all instances	GET /eureka/v2/apps
Query for all **appID** instances	GET /eureka/v2/apps/**appID**
Take instance out of service	PUT /eureka/v2/apps/**appID**/**instanceID**/status?value=OUT_OF_SERVICE
Put instance back into service	DELETE /eureka/v2/apps/**appID**/**instanceID**/status?value=UP

Registry /apps API - list all the apps

http://localhost:8761/eureka/apps/

```
<applications>
  .....
  <application>
      <name>PRODUCTSERVICE</name>
  .......
  </application>
  ......
    <name>ORDERSERVICE</name>
  ......
    <name>CUSTOMERSERVICE</name>
 ......
    <name>CONFIGSERVER</name>
 ......
    <name>ZUULSERVICE</name>
  </application>
</applications>
```

```
<name>PRODUCTSERVICE</name>
    <instance>
      <instanceId>336c99b0236a:productservice:8081</instanceId>
      <hostName>172.18.0.3</hostName>
      <app>PRODUCTSERVICE</app>
      <ipAddr>172.18.0.3</ipAddr>
      <status>UP</status>
...
...
```

To get JSON rather than XML
consumes="application/json"

Default Gateway Routing from Registry

http://localhost:5555/actuator/routes

```
{
 "/configserver/**": "configserver",
 "/customerservice/**": "customerservice",
 "/productservice/**": "productservice",
 "/orderservice/**": "orderservice"
}
```

These routes are provided automatically from the Registry, out of the box.
The registry by default uses the 'app' name.

New Vanity URL patterns can be provided via configurations, on the fly.

```
zuul:
   prefix:  /api
   routes:
      orderservice: /order/**
      customerservice: /customer/**
      productservice: /product/**
```

Services now via Gateway

http://localhost:8081/v1/products

http://localhost:8085/v1/customers/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a

http://localhost:8080/v1/customers/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a/orders/

Via Gateway:

http://localhost:5555/productservice/v1/products

http://localhost:5555/customerservice/v1/customers/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a

http://localhost:5555/orderservice/v1/customers/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a/orders/

my-correlation-id → f543dd96e18a1913

Services now via Vanity URL

http://localhost:5555/productservice/v1/products

http://localhost:5555/customerservice/v1/customers/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a

http://localhost:5555/orderservice/v1/customers/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a/orders/

Configured to Vanity:

http://localhost:5555/api/product/v1/products

http://localhost:5555/api/customer/v1/customers/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a

http://localhost:5555/api/order/v1/customers/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a/orders/

See Route Configurations

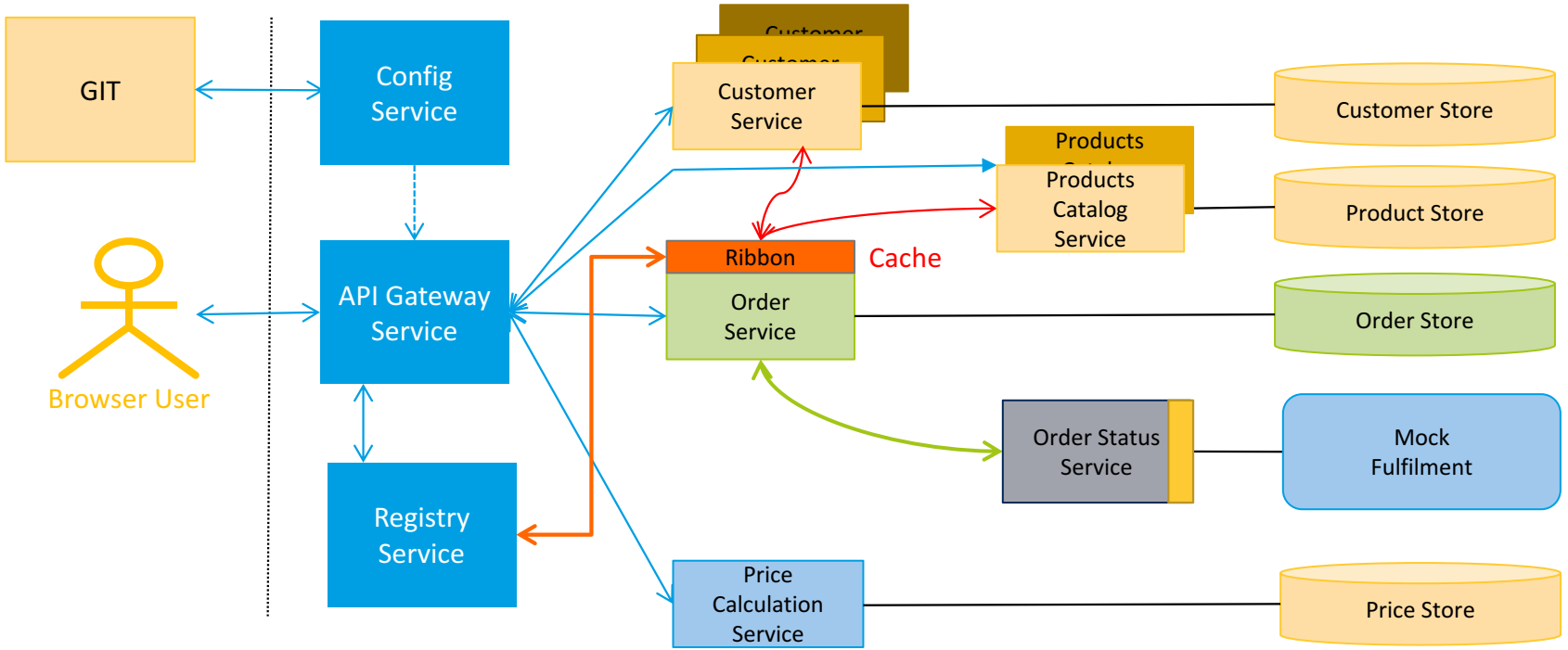
http://localhost:8888/zuulservice/dev


```
{
    "zuul.ignored-services": "*",
    "zuul.prefix": "/api",
    "zuul.routes.customerservice": "/customer/**",
    "zuul.routes.orderservice": "/order/**",
    "zuul.routes.productservice": "/product/**",
    "zuul.routes.authenticationservice": "/auth/**",
    "zuul.sensitiveHeaders": "Cookie,Set-Cookie",
    "zuul.debug.request": true,
    "hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds": 2500
}
```

Service Discovery The Client

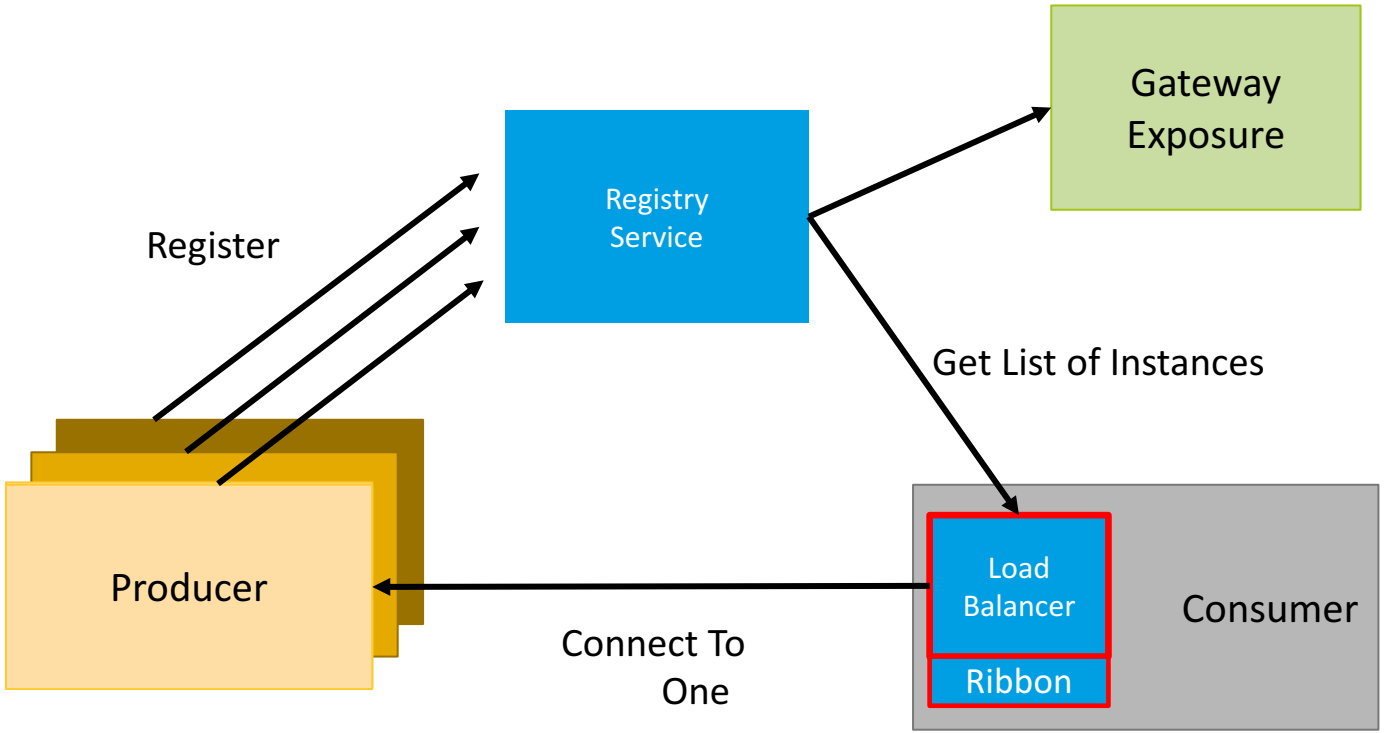
Client Discovery – Find an Instance

RP



Pattern: Client Discovery

Producer – Registry - Consumer



Ribbon

Ribbon provides a client with the ability to look up a service, via the registry; and to manage a **cached** physical list of instance locations.

There are 3 Spring/Netflix client libraries that move from the lowest level of abstraction to the highest, they are:

- **Spring Discovery Client** – provides a list of all the physical instances; manual coding to use the list, as you like, to call an instance;
- **Spring Discovery Client enabled RestTemplate** – does the load balancing, no need to use the list; code a class;
- **Netflix Feign Client** - as above, but is declarative, requires no code, just JAX-RS annotations and an Interface;
  - Feign dynamically generates clients from the interface.
  - Automatically wraps calls with Hysterix.

Customer REST Template

Example: Order service looks up a Customer via Customer Service

```
package com.widget.order.clients;

import com. widget.order.model.Customer;
…
…

@Component
public class CustomerRestTemplateClient {
   @Autowired
   RestTemplate restTemplate;

   public Customer getCustomer (String customerId){
      ResponseEntity<Customer> restExchange =
            restTemplate.exchange(
                  http://customerservice/v1/customers/{customerId}, HttpMethod.GET, null, Customer.class, customerId);
      return restExchange.getBody();
   }
}
```

Templated call Using Eureka ID
Not Physical URL. Uses load balanced rest template bean.

Customer Feign Client

Example: Order service looks up a Customer via Customer Service

```
package com.widget.order.clients;

import com.widget.order.model.Customer;
…
…

@FeignClient("customerservice")
public interface CustomerFeignClient {
   @RequestMapping(
         method= RequestMethod.GET,
         value="/v1/customers/{customerId}",
         consumes="application/json")
   Customer getCustomer(@PathVariable("customerId") String customerId);
}
```

Service autodiscovery via Eureka

Interface – used to create code.

Templated URL

Retries

In Spring Cloud Auto Retries can be configured for:

- REST Templates
- Feign
- Zuul – Gateway retries on behalf of the consumer;

- Ribbon also has retries and timeouts.

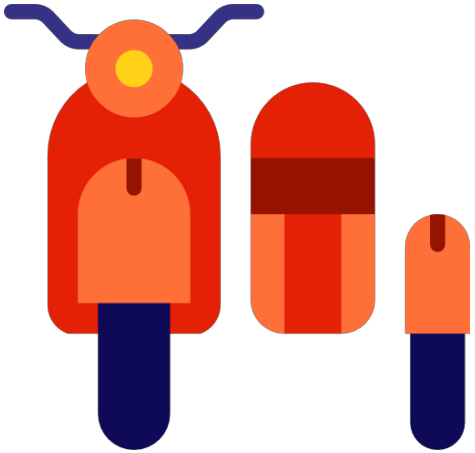
This is a massive productivity bonus, as this code is very complex;

Non-Java, non Spring Services and Clients

For non-Java services, you have a choice of:

- implementing the client part of Eureka in the language of the service OR
- run a "side car" - a Java application with an embedded Eureka client that handles the registrations and heartbeats.

- REST API based endpoints are also exposed for all operations that are supported by the Eureka client.
- Non-Java clients can use the REST end points to query for information about other services.



Netflix SideCar

Demo Service Locations – Containers

Containers configured on my LocalHost:

- Customer Service – Port 8085
- Product Service – Port 8081
- Order Service – Port 8080
- DBMS – Port 5432

Via Zuul Now !

Added in this session:

- Configuration Server – Port 8888
- Eureka Registry Server – Port 8761
- Zuul API Gateway – Port 5555

Spring Actuator - Management Endpoints

GET: http://localhost:8085/mappings

/env
/health
/metrics
/info
/features
/v1/customers],methods=[POST]
/v1/customers/{customerId}],methods=[PUT]
/v1/customers/{customerId}],methods=[GET]
/v1/customers/{customerId}],methods=[DELETE]
/error
/pause
/resume

Customer Service APIs

/channels -if streams
/hystrix.stream/**
/restart
/autoconfig
/beans
/refresh
/dump
/heapdump
/trace
/info
/archaius
/mappings
/configprops
/webjars/**

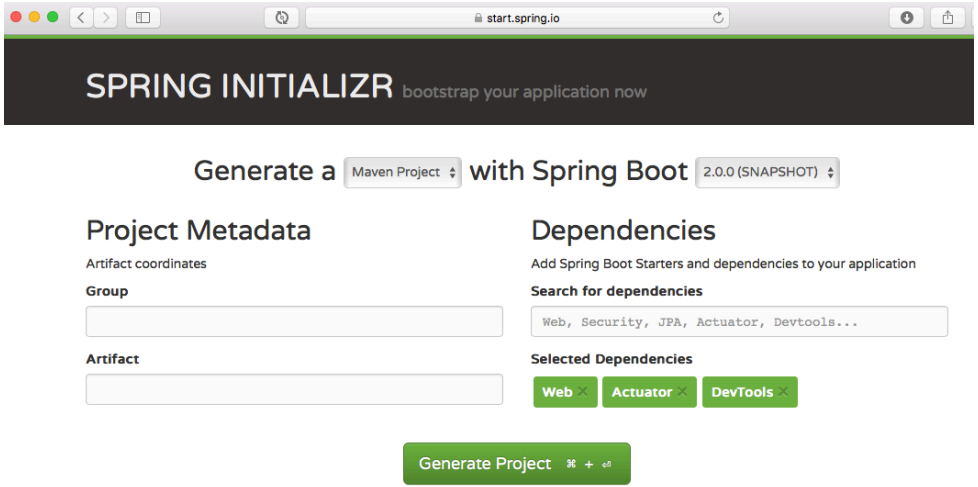
Note: can define a separate management port for these, for security, separate from the Business APIs.

Service Discovery Admin

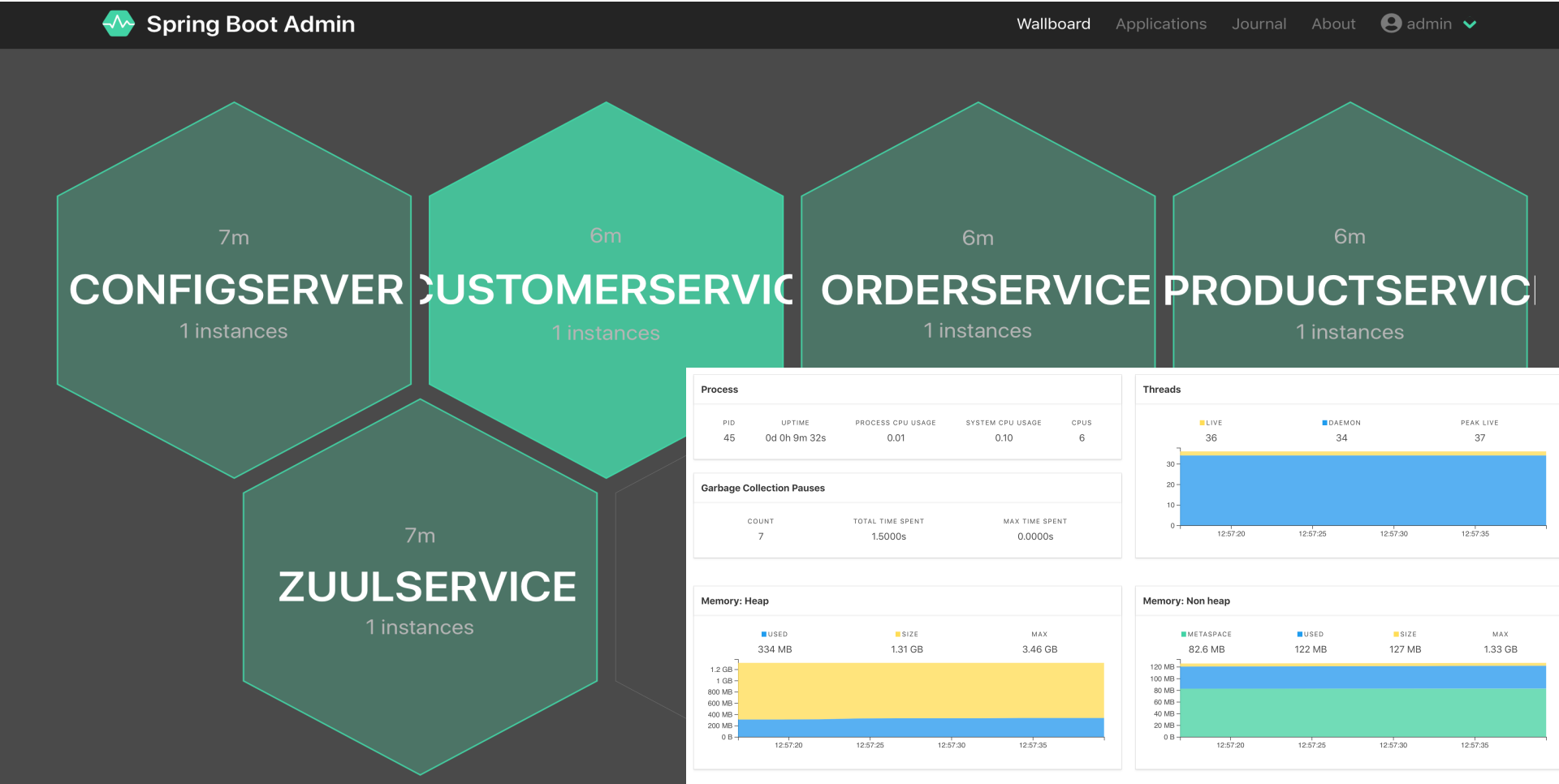
Spring Boot + Spring Cloud

Spring Boot is an opinionated framework built on top of the Spring Framework.

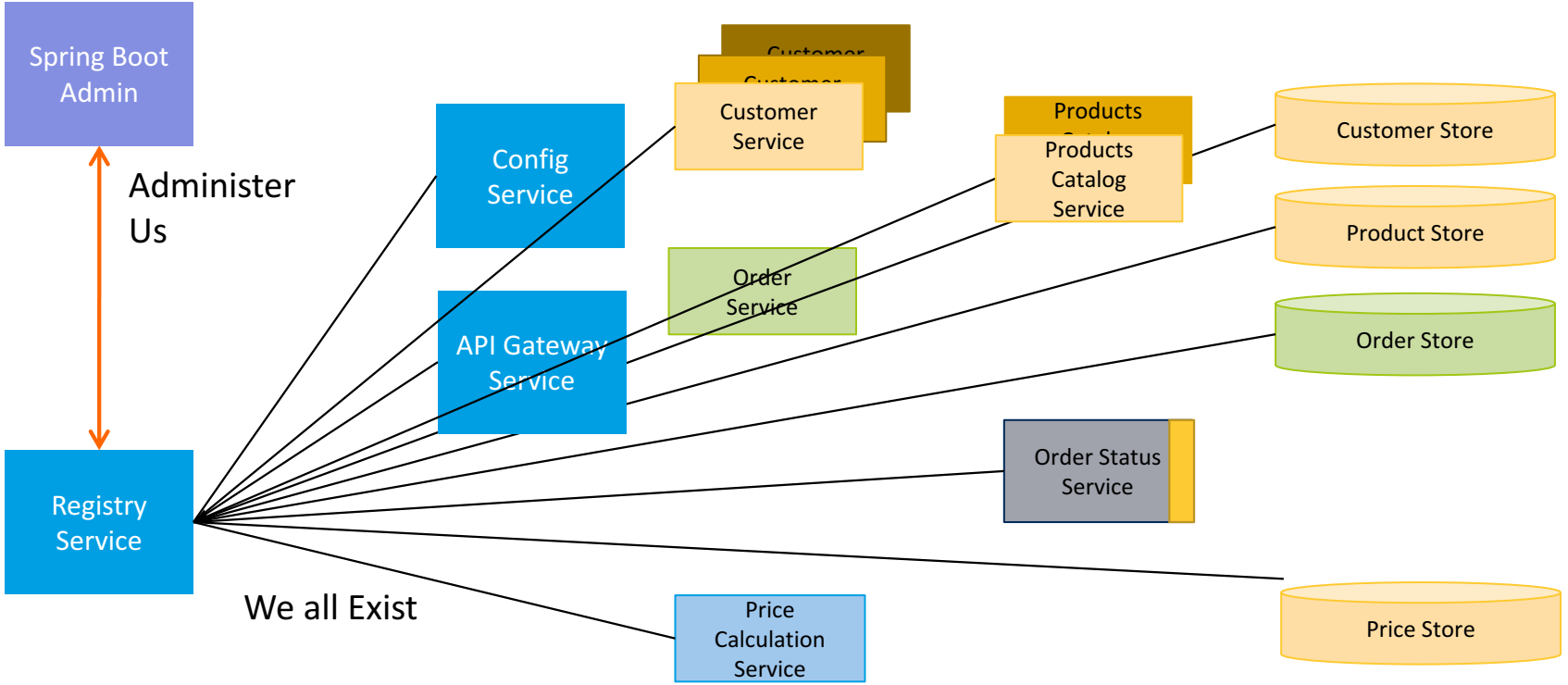
- The fundamental functionality provided by the Spring Container is dependency injection.
- Spring is very powerful and gives you many many features and choices, it can also require a lot of manual configuration.
- Spring Boot improves productivity by providing default 'auto' configuration for most features but giving you the ability to easily change the behavior according to your needs.
- Mostly used to create web applications but can also be used for command line applications.
- A Spring Boot web application can be built to a stand-alone jar with an embedded web server that can be started with java -jar.
- Makes it easy to integrate the various Spring modules into your application through starter POMs that contain all necessary dependencies.
- There are many Spring Components that are Ideal for Microservices, e.g. Spring Cloud, Spring Data, Spring Cloud Gateway, Spring Stream Spring Netflix…



Spring Boot Admin



Service Discovery and Admin



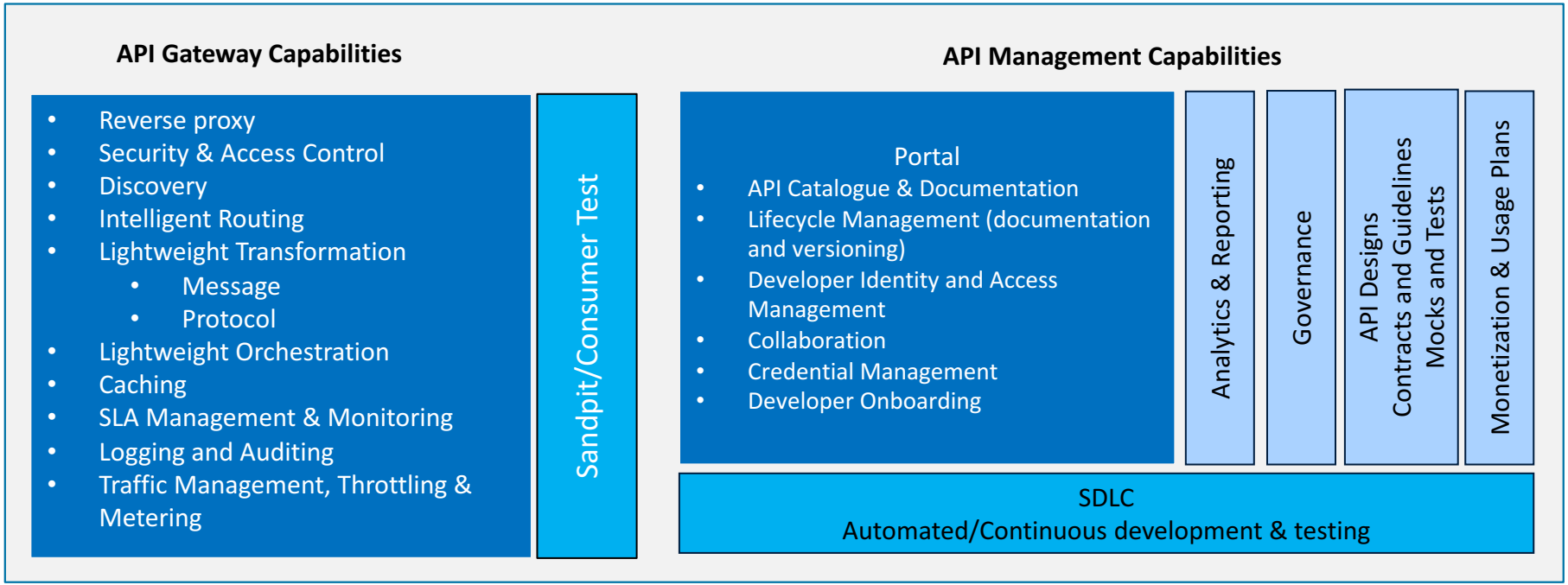
Pattern: Admin Discovery via registry

API Management

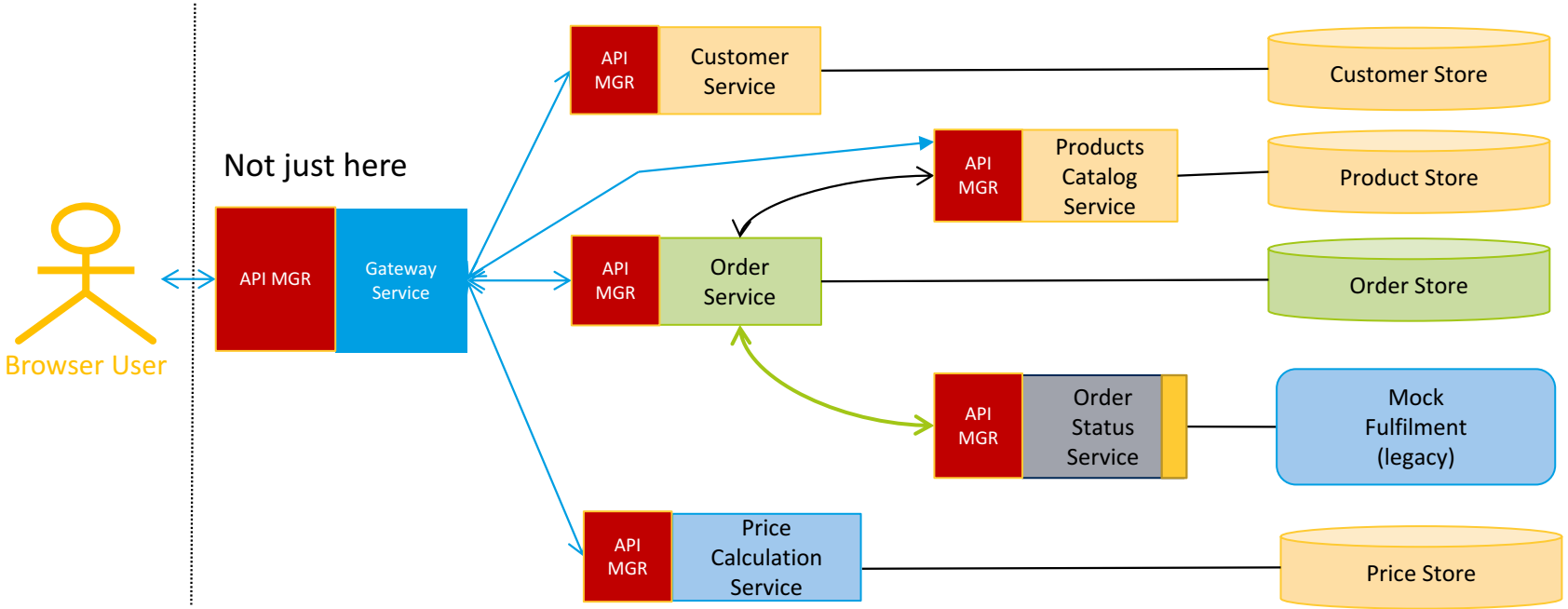
API ‘Management’ Vs API Gateway

- API Discovery (Catalog, Search and Provisioning)
- API Security (SSL, PKI, threat protection, schema validation, encryption, signatures, etc)
- API Identity (AuthN & AuthZ, API key, OAuth, SAML, LDAP, proprietary IAM, multifactor, token translation & management)
- API Orchestration (adaptation of multiple services, workflow operations, branching policies, etc.)
- Uniform interface/proxy to multiple backend messaging protocols (JMS, RMI etc)
- Developer and App OnBoarding (Client ID/App Key generation, Interactive API console)
- Community Management (Blogs, Forums, Social features etc)
- API Lifecycle governance (Versioning)
- Traffic Mediation (SOAP to REST mediation, data format transformation, legacy application integration)
- Traffic Monitoring and Shaping(Rate limitation, Caching etc)
- Analytics and Reporting
- API metering, Billing and Monetization
- Data Protection(Data encryption, Data masking etc for PCI/PII compliance)
- Mobile Optimization (Pagination, Compression, JSON etc)
- Deployment Flexibility (on-premise, cloud, managed service, SaaS, hybrid)
- Operational Integration (System Monitoring, Clustering, Scalability, Migration)
- Mobile Integration (support for push notifications, geolocation, streaming protocols)
- Cloud Integration (SSO to SaaS providers, IaaS integration, SaaS data connectors, hybrid cloud support)
- Distributed API Management(Manage at each service, not just at a Gateway)

API Gateway Vs API Management



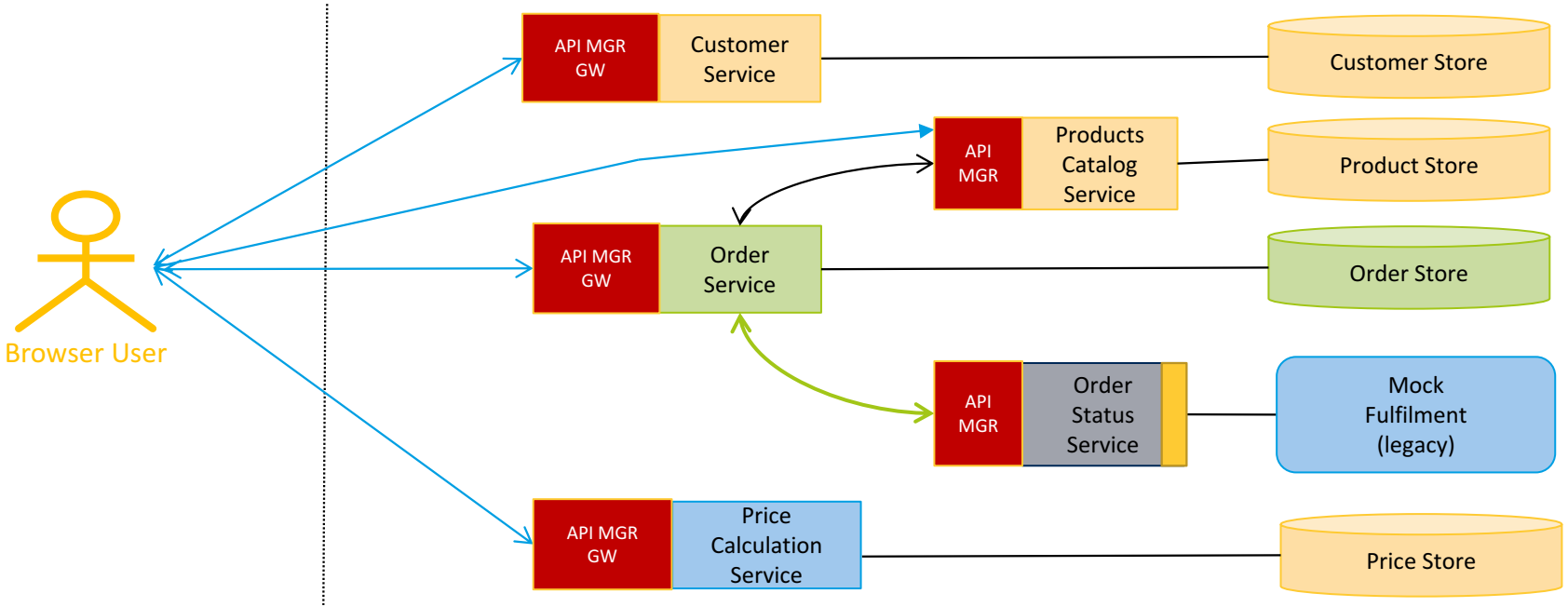
Distributed Lightweight API Management



Pattern: Distributed API Management, API Gateway

Distributed Lightweight API Management

Option: Move services closer to edge



Pattern: Distributed API Management – No Gateway

Swagger

Publishing the API Contract

To include swagger just add Springfox to POM:

```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.6.1</version>
</dependency>
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.6.1</version>
</dependency>
```

Add To Application

```
@EnableSwagger2

@Bean
public Docket api() throws XmlPullParserException {
	return new Docket(DocumentationType.SWAGGER_2)
		.apiInfo(apiInfoX())
		.select()
		.apis(RequestHandlerSelectors.basePackage("com.widget.product.controllers"))
		.paths(PathSelectors.any()).build();
}

 private ApiInfo apiInfoX() {
	return new ApiInfoBuilder()
		.title("Product API")
		.description("Product API reference for developers")
		.contact("kimhorn@gmail.com")
		.license("The License")
		.licenseUrl("kimhorn@gmail.com")
		.version("1.0")
		.build();
}
```

http://localhost:8085/swagger-ui.html

Order API

Order API reference for developers

Created by kimhorn@gmail.com
The License

order-service-controller : Order Service Controller — Show/Hide | List Operations | Expand Operations

GET	/v1/customers/{customerId}/orders	getOrders
POST	/v1/customers/{customerId}/orders	saveOrders
DELETE	/v1/customers/{customerId}/orders/{orderId}	deleteOrders
GET	/v1/customers/{customerId}/orders/{orderId}	getOrders
PUT	/v1/customers/{customerId}/orders/{orderId}	updateOrders
GET	/v1/customers/{customerId}/orders/{orderId}/items	getOrderItems
POST	/v1/customers/{customerId}/orders/{orderId}/items	saveOrderItem
GET	/v1/customers/{customerId}/orders/{orderId}/items/{itemId}	getOrderItem
PATCH	/v1/customers/{customerId}/orders/{orderId}/items/{itemId}	patchOrderItem
PUT	/v1/customers/{customerId}/orders/{orderId}/items/{itemId}	updateOrderItem

tools-controller : Tools Controller — Show/Hide | List Operations | Expand Operations

GET	/v1/tools/eureka/services	getEurekaServices

[BASE URL: / , API VERSION: 1.0]

Summary

Summary

- Need a catalog of services; ask the registry, that provides a source of knowledge across the whole application;
- Clients including the Gateway use registry to discover healthy instances and then load balance their requests directly to providers;
- Instances are cached by client;
- No physical or application load balancers in the path;
- Cohesive Framework and Architecture – same approach for clients as the Gateway, later is not a special product;
  - Netflix components used everywhere;
- ZUUL, EUREKA, FEIGN, RIBBON – all designed to work within AWS, as Netflix runs on AWS
- Highly available architecture;

Microservice Lifecycle

1. Service assembly:
   - build container image;
   - deploy it
   - start it.
2. Service bootstrapping – gets its configuration;
3. Service registration/discovery
   - register myself;
4. Service monitoring:
   - heart beats - health
   - logs
   - manage my cache of destinations
5. Listen for new configuration changes, and apply;

Next Episode

Not the whole 'Service Discovery' story … More Next Time:

- Health
- Client/Server Side Load Balancer
- Achieving resiliency with Netflix Hystrix:
  - Circuit Breaker
  - Bulkhead
  - Fallback
  - Turbine - Monitoring
- Sleuth & Zipkin - Distributed Tracing and Logging

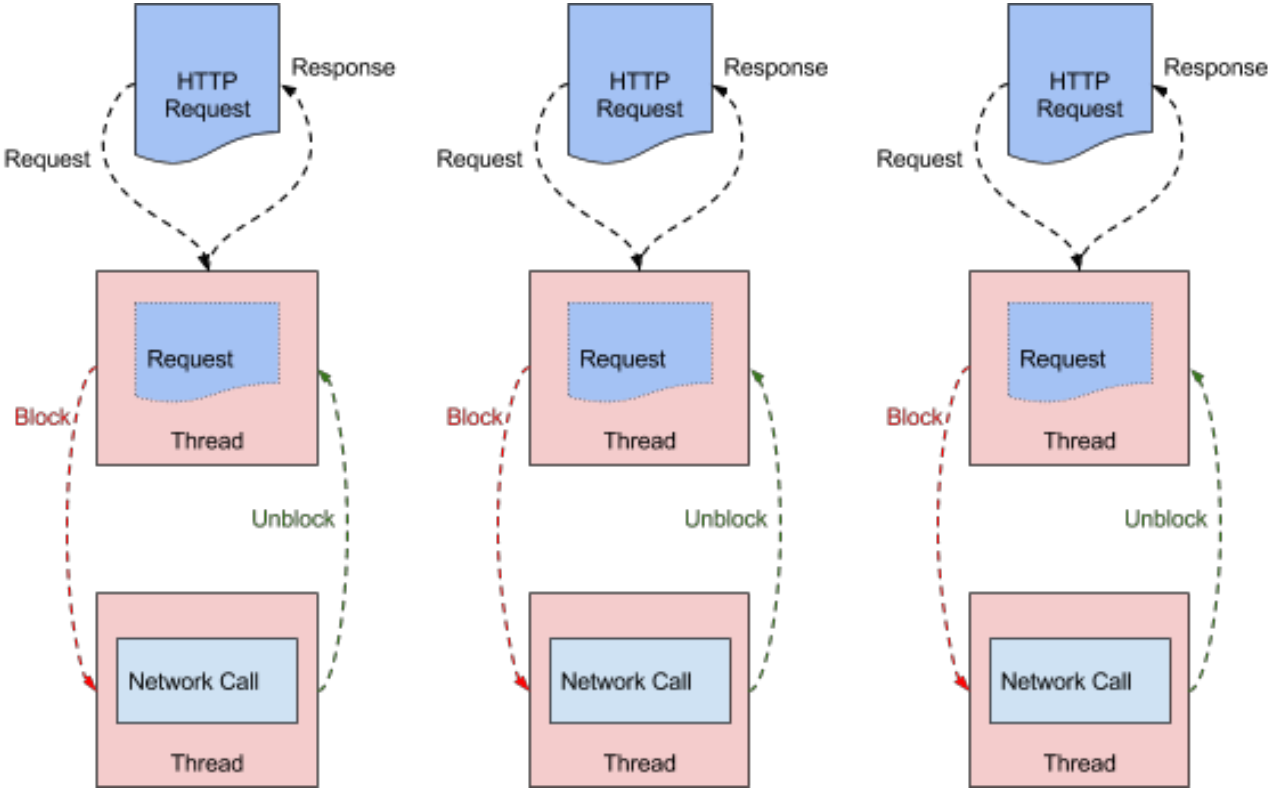
Appendix Zuul 2 and Eureka 2

Zuul 1 – Servlet Framework Based

Zuul 1 was built on the Servlet framework.

Such systems are blocking and multithreaded.

They process requests by using one thread per connection.

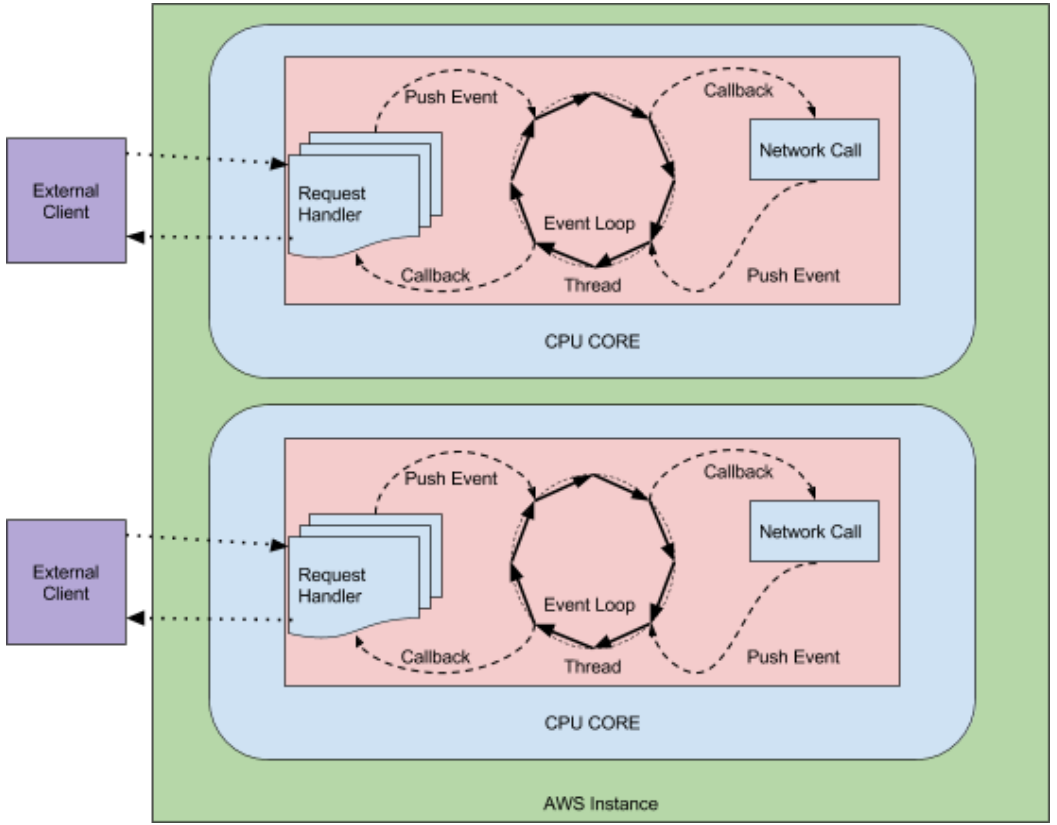


Zuul 2 – Non Blocking I/O

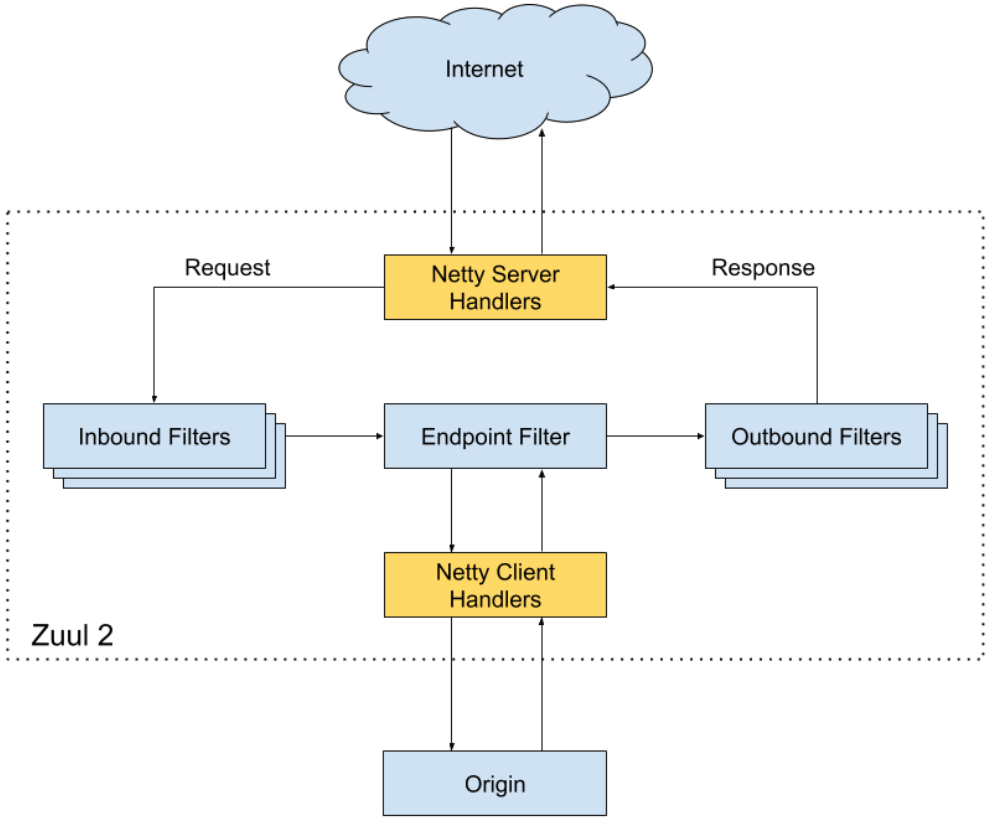
Netflix has re-architected Zuul around Netty Non-blocking I/O.

Async generally have one thread per CPU core handling all requests and responses. The lifecycle of the request and response is handled through events and callbacks.

As there isn't a thread per request, the cost of connections is cheap.



Zuul 2 – Netty and Filter Core



The filters are where the core of the business logic happens for Zuul. They have the power to do a very large range of actions and can run at different parts of the request-response lifecycle as shown in the diagram above.

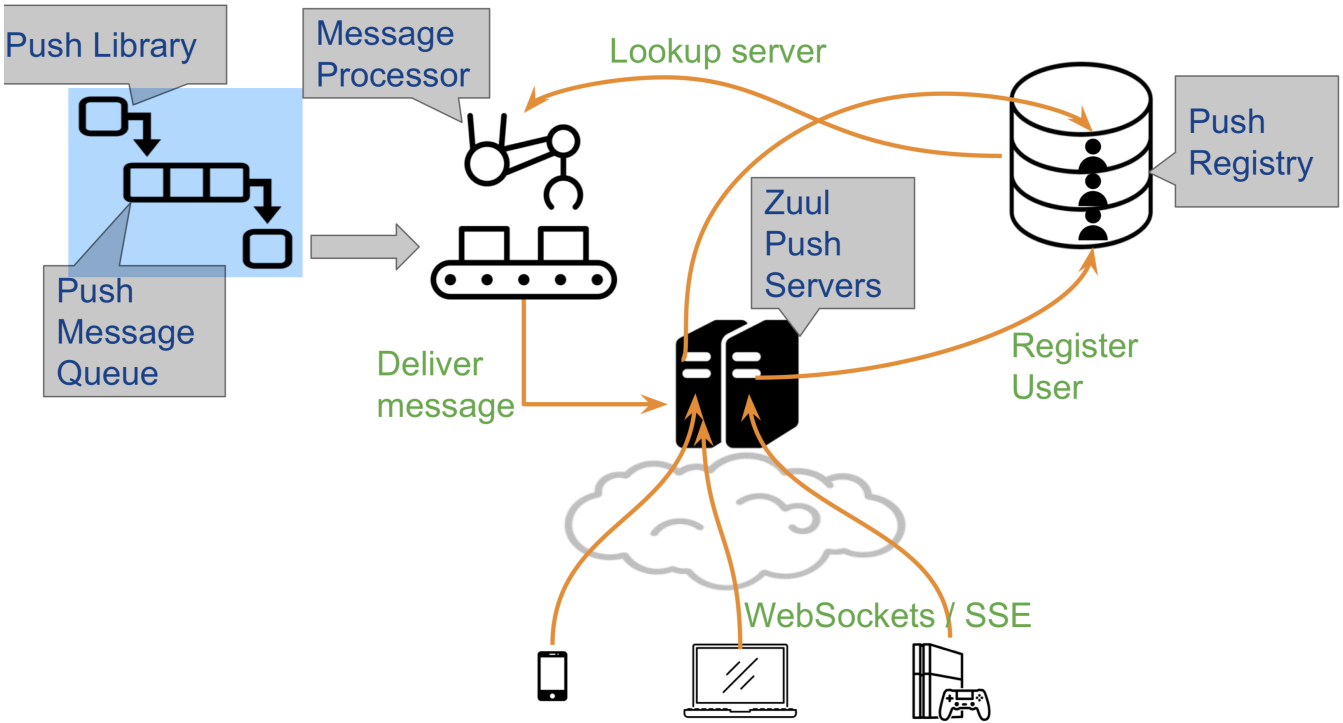
Inbound Filters execute before routing to the origin and can be used for things like authentication, routing and decorating the request.

Endpoint Filters can be used to return static responses, otherwise the built-in ProxyEndpoint filter will route the request to the origin.

Outbound Filters execute after getting the response from the origin and can be used for metrics, decorating the response to the user or adding custom headers.

Zuul Push

Zulu 2.0 supports push messaging - i.e. sending messages from server to client. It supports two protocols, WebSockets and Server Sent Events (SSE) for sending push messages.



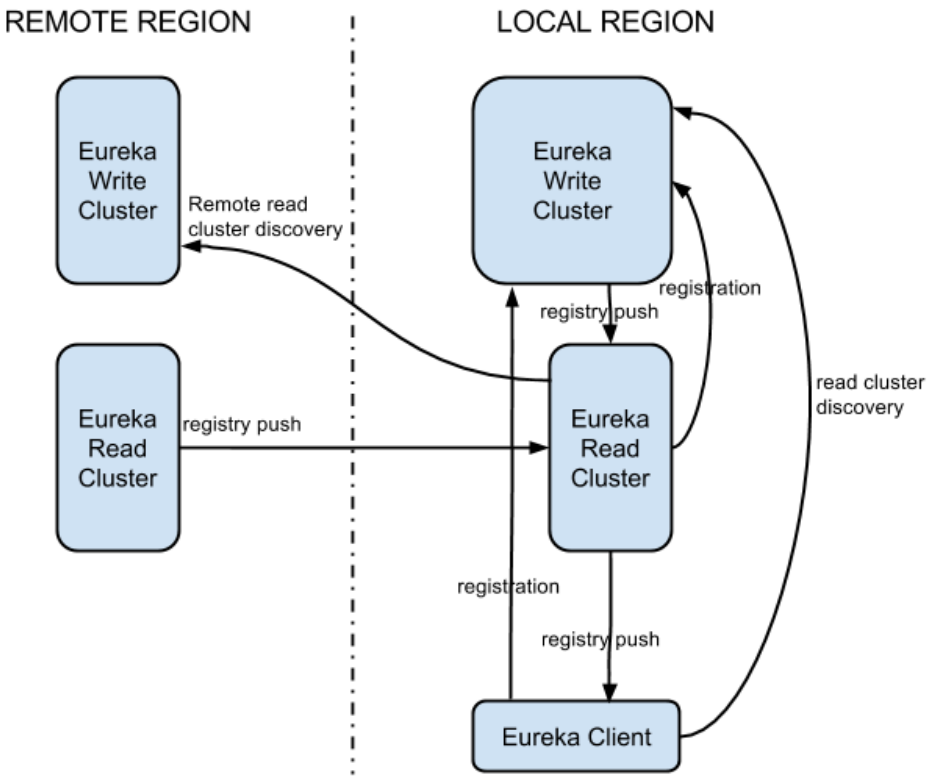
Eureka 2.0 for Cloud Deployment

Eureka system itself consists of a write and read cluster.

The write cluster is a stateful subsystem, which handles client registrations and maintains internal service registry. The registry content is replicated between the all write server nodes in an eventually consistent manner. The write cluster's registry content is read by the read cluster, that ultimately is used by the Eureka clients.

As the read cluster is effectively a cache layer, it can be easily, and rapidly scaled up and down depending on the volume of the traffic. The write cluster should be pre-scaled with a capacity enough to handle the peek/busy hour traffic.

Although it can be scaled up and down dynamically, it needs to be done with more coordination. Scaling up will require traffic re-balancing, which will eventually happen, but not immediately. Scaling down will force the clients from the shutdown node(s) to re-register.



Eureka 2.0

Eureka 2.0 is designed to work with different cloud providers and data centers, thus its underlying data model must be extendable to accommodate current and future deployments.

Out of the box a basic data center model, and Amazon AWS/VPC clouds are supported.