CS544: Enterprise Architecture

Lecture 14: Microservices

Maharishi University of Management

Computer Science Department

Orlando Arrocha

December 2017

CS544: Enterprise Architecture

Wholeness of the Lesson

Lesson 14
MICROSERVICES WITH SPRING
Knowledge is structured in consciousness

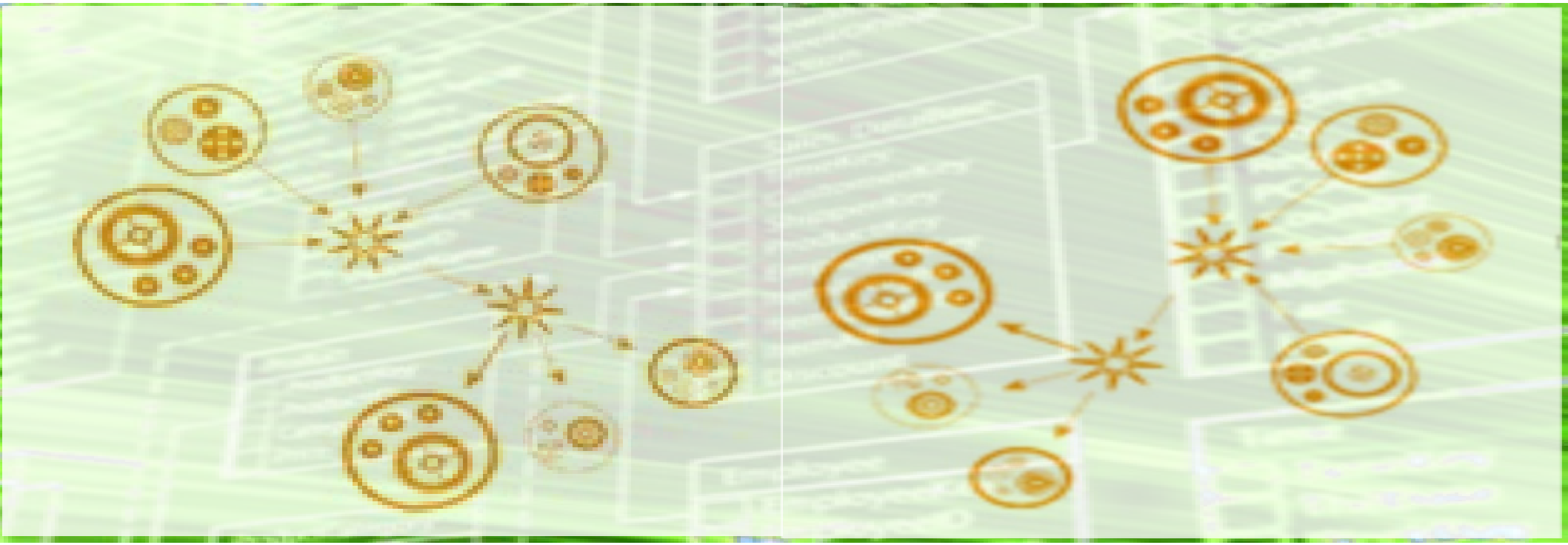
Structuring applications based on their services creates a more scalable architecture, that includes fault-tolerance.

Science of Consciousness: *Nature discriminative qualities spring from the pure field of creative intelligence, so we say creative intelligence is holistic; it contains the wholeness of life.*

Summary

- Microservices
  - Architecture
  - Spring Cloud
  - Creating microservices

Spring Microservices

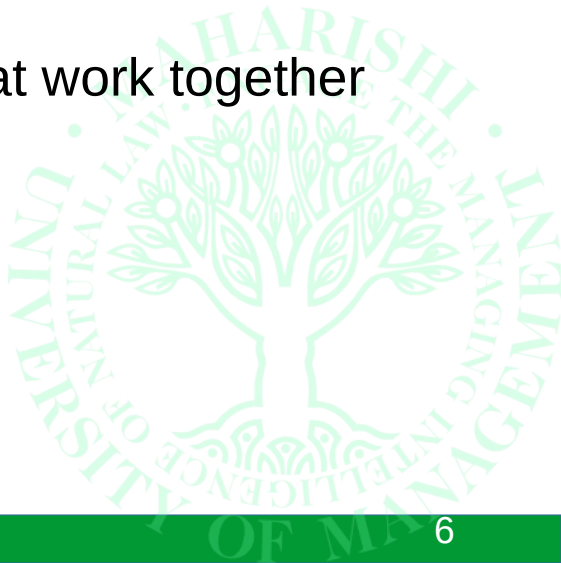


“The golden rule: can you make a change to a service and deploy it by itself without changing anything else?”.

– Sam Newman, Building Microservices

What are Microservices?

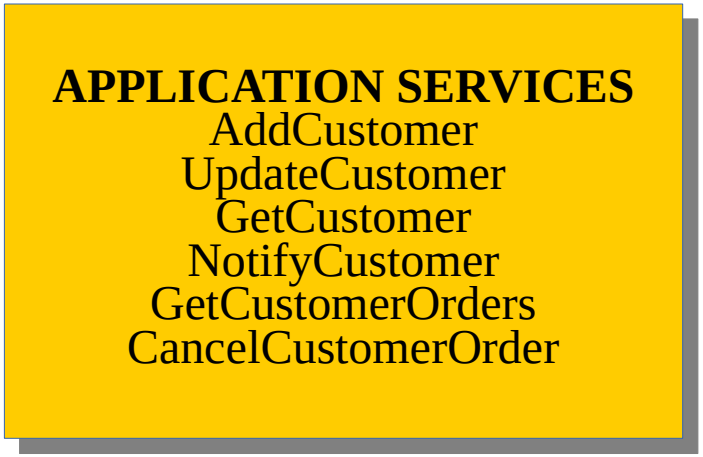
- Is an architecture style (an implementation of SOA)
- Applications are composed of independent services
  - Highly decoupled
  - Easy to replace (continuous delivery)
  - Always available
  - Publish/Subscriber architecture
  - High availability / Redundancy
  - Multiple copies by process type
  - Develop expecting failures
  - Distributed
- Communications must be language agnostic
- In other words: a group of small, autonomous services that work together



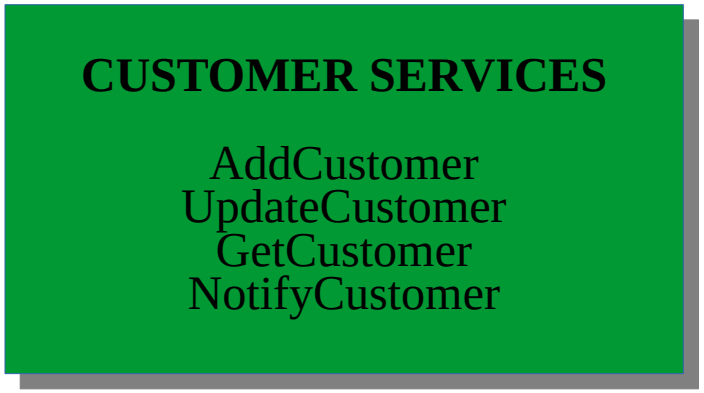
How Small?

- The smaller the service, the more you maximize the benefits of microservices.
- Follow the Single Responsibility Principle
  - "Gather together those things that can change for some reason, and separate those that change for different reasons" - Robert C Martin
  - http://programmer.97things.oreilly.com/wiki/index.php/The_Single_Responsibility_Principle
- The right size for your system:
  - Every service must be able to be deployed independently

Cohesion



- **Cohesion** refers to the degree to which the elements of a module belong together.

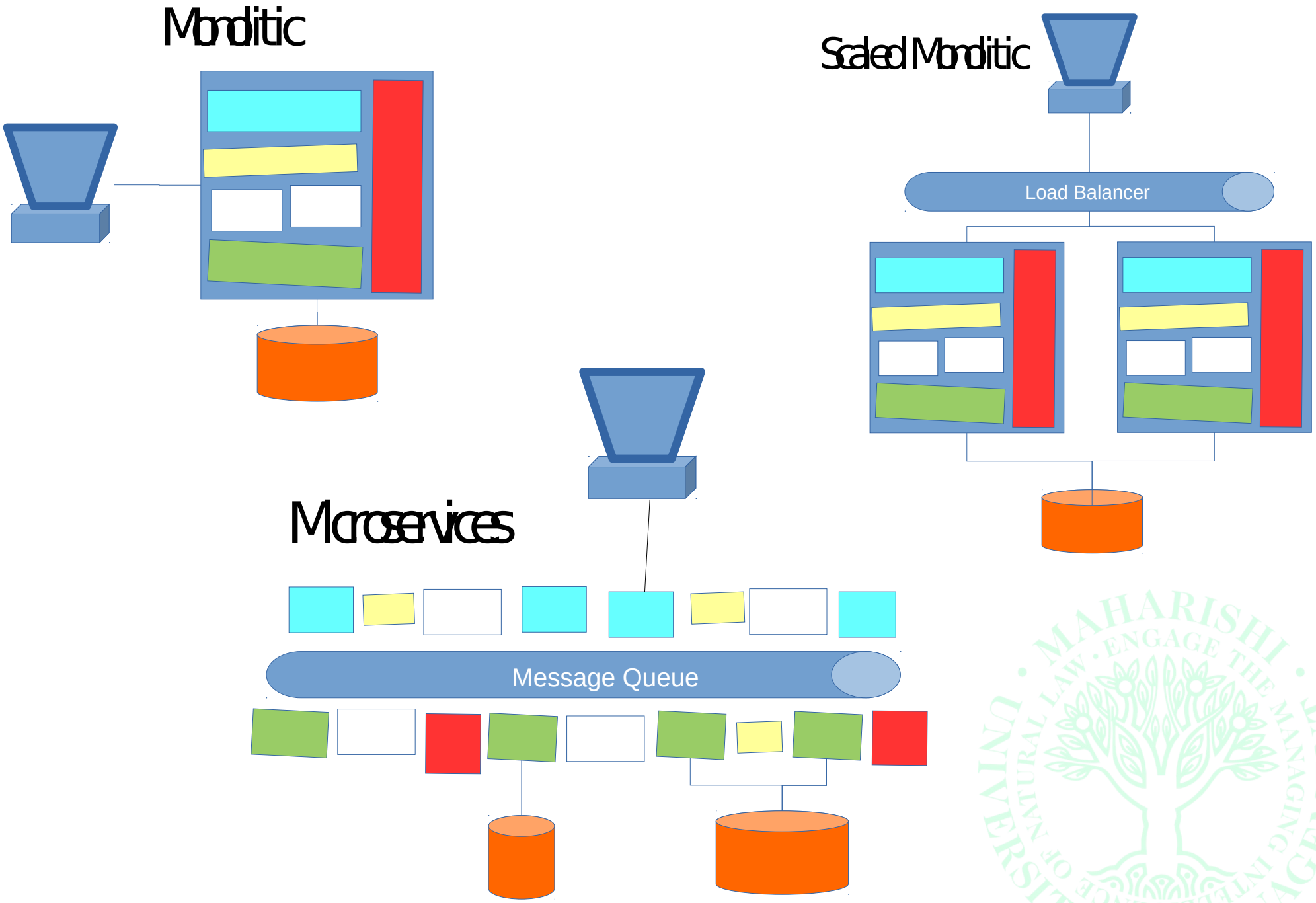


- **Command Query Responsibility Segregation (CQRS) Pattern** promotes further separation. Split the conceptual model into separate models for update and reading.

Benefits

- Technology Heterogeneity: best technology (DB / language) to accomplish the task.
- Resilience : if one system component fails, the failure doesn't cascade, you can isolate the problem and the rest of the system can continue working.
- Scaling: code is smaller than monolithic systems, allowing us to deploy in smaller machines. On demand provisioning allows us scale on demand for those pieces that need it. This allows us to control cost more effectively.
- Ease of deployment: we can make a change on a single service and deploy it independently. Faster updates, easier rollbacks.
- Organizational alignment: minimizes the number of people involved on a particular service. Ownership of a service codebase is set to a small team and can easily be shifted to another.
- Composability: micro services can be rearranged and reused for different customer engagement as these change on the organization.

Architecture



Infrastructure vs Platform

- Manage virtual machines (IaaS)
    - Amazon EC2, Google Compute Engine
  - Docker
    - Software container
    - Lighter and faster than a VM
- Manage applications (PaaS)
    - Heroku, Cloud Foundry, Google App Engine
  - Offers a complete managed platform
  - Allows you to push applications (not Vms)
  - Offers ways to bind other services to your application (i.e. databases)

Platform as a Service (Paas)

- Heroku
  - One of the first cloud providers
  - Push application binaries directly
  - Or link to a git repository and push source code

    https://www.heroku.com/
- Cloud Foundry
  - Open source PaaS
  - Push jars from the command line

- Sign up at **http://run.pivotal.io/**
- Get the **cf** CLI.
- **cf login**
- Select the target organization and environment
- You may create shared services like databases
- **cf create-service p-mysql 100mb** *logical-name-db*
- **cf push -p** your.jar *logical-name* **--random-route --no-start**
- **cf bind-service** *logical-name* logical-name-db
- **cf start logical-name**

Microservice Development

- Each microservice must have its own codebase (build, deploy, defect tracking, etc.)
- Since each microservice should have a very specific functional
  - Code foot print is smaller – easier to learn and maintain
  - Defects on one microservice don't block release of other microservices
- Follow the 12 factor application

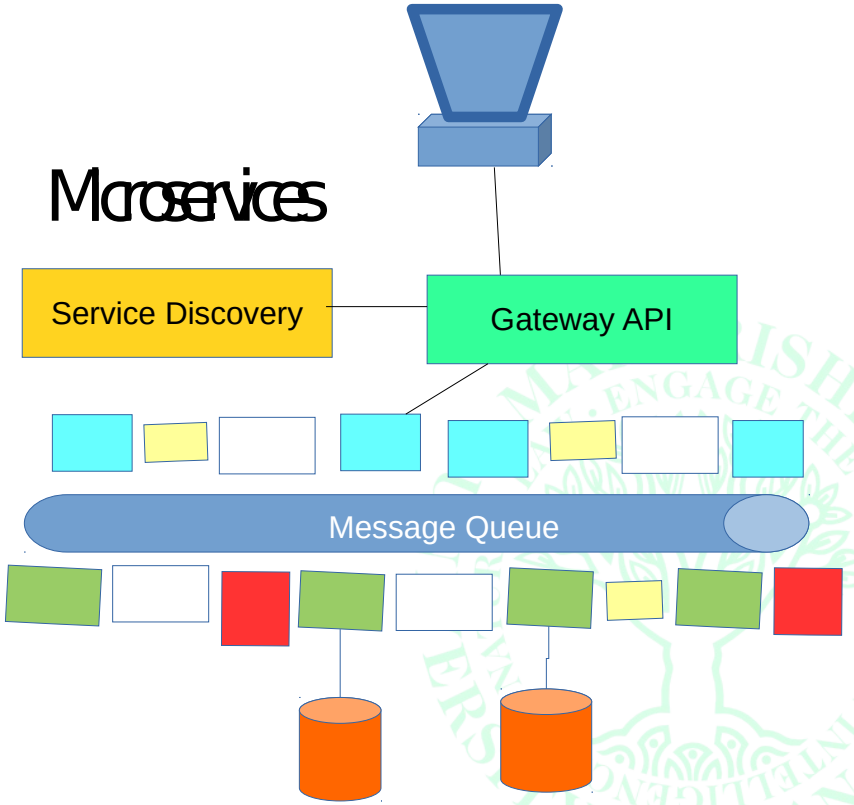
12 Factor Application

1) Codebase – Once tracked in revision control, deploy
2) Dependencies – Explicitly declare and isolate dependencies
3) Config - Store in the environment
4) Backing Services – Treat as attached resources
5) Build, Release and Run – Separate the stages
6) Processes – Execute the app as one or more stateless processes
7) Export services – via port binding
8) Concurrency – Scale out via process type of work
9) Disposability – Maximize robustness with fast startup and graceful shutdown
10) Dev/Prod Parity – Keep development, staging, and production as similar as possible
11) Logs – Treat as events streams
12) Run admin/management tasks as one-off processes – same in all identical environments

http://12factor.net/

Robust Microservice Development

- Include an abstraction layer between the clients and the application – API_Gateway
  - Handle differences between client devices communication requirements
  - The API decouples the client from the application
- Include a Service Discovery
  - All deployed services register with the Service Discovery
  - Service dependencies are resolved querying the Service Directory
  - This process adds scalability and possibility of fault-tolerance

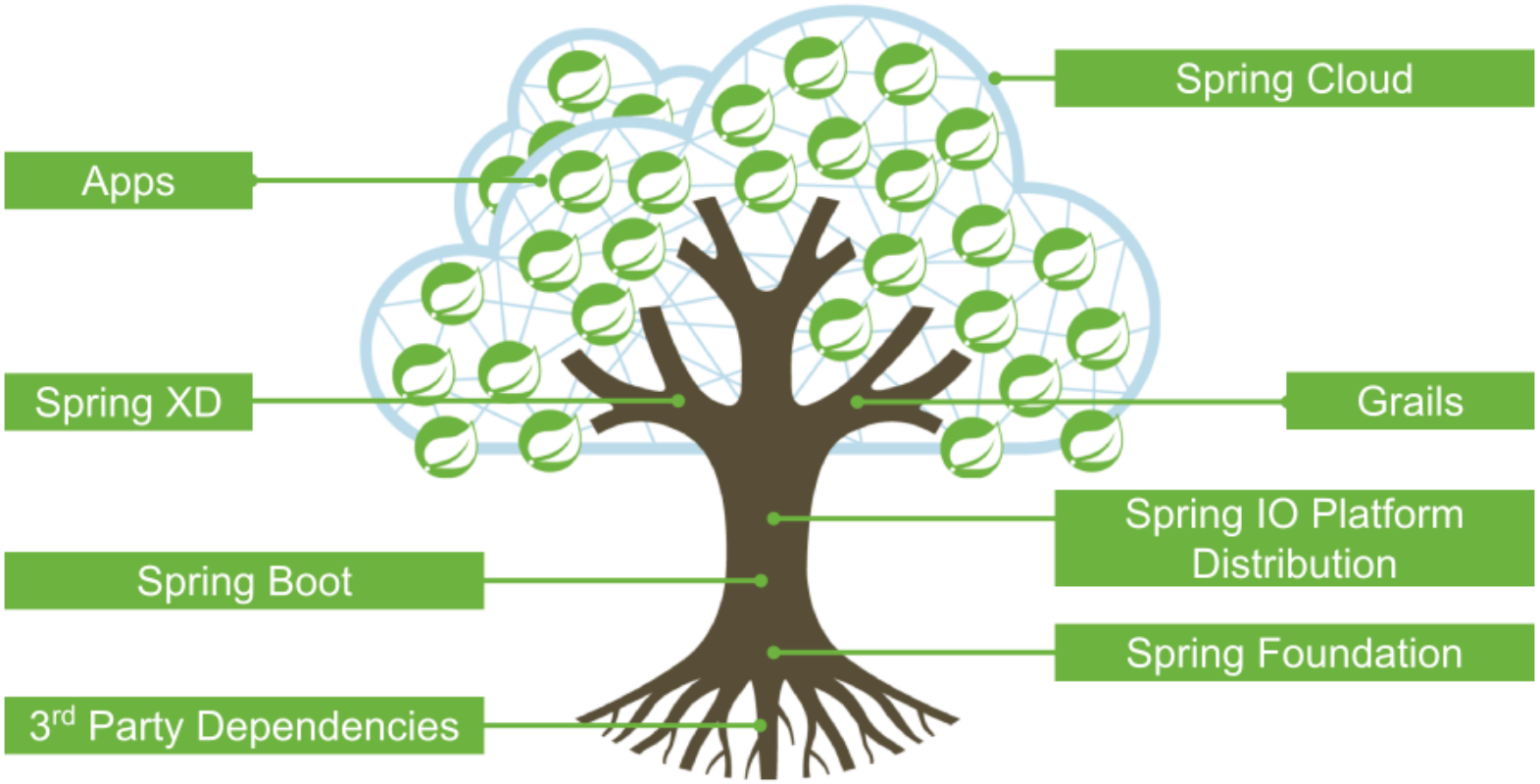


Spring Cloud

- Distributed/versioned configuration
- Service registration and discovery`
- Routing
- Service-to-service calls
- Load balancing
- Circuit Breakers

Global locks

- Leadership election and cluster state
- Distributed messaging
- Spring Cloud Netflix project



Netflix Open Source Software

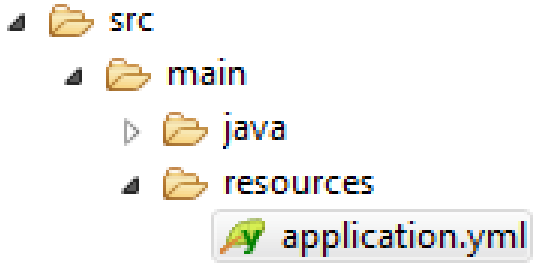
- Netflix is the biggest, most robust, microservices applications (37% of USA bandwidth usage), and they provided some of their code under the Netflix OSS project
  - **Eureka** – for service discovery
  - **Hystrix** – for all your circuit breaker needs
  - **Feign** – allows you to define declarative REST clients
  - **Ribbon** – client side load balancing
  - **Zuul** – for routing and filtering

Spring Boot

- Makes it easy to create stand-alone, production-grade Spring based Applications
- Standard Java App
- Sensible defaults and automatic configuration
- Embedded HTTP server (Tomcat, Jetty or Undertow)
- Maven or Graddle with required dependencies, depending on the selected type of project
- Generates self-contained runnable applications (JAR)
- Very cloud friendly
- Run and debug in your IDE

Understanding Auto-Configuration

- **@EnableAutoConfiguration** loads **/META-INF/spring.factories**
- **spring.factories** declares **@Configuration** classes
- Each **@Configuration** is **@Conditional** (it loads the configuration if finds the indicated class on the classpath)
- @ConfigurationProperties to map properties to POJOs




```
configuration:
  projectName : Spring
Boot
```

```
...
@SpringBootApplication
@EnableConfigurationProperties
public class Application {

    @Value("${configuration.projectName}")
    void setProjectName(String projectName) {
        System.out.println("setting project name: " +
projectName);
    }

    @Autowired
    void setEnvironment(Environment env) {
        System.out.println("setting environment: "
                +
env.getProperty("configuration.projectName"));
    }
...
```

Checking SpringBoot Configuration Properties

- On SpringBoot Web applications you can check the active configuration through the Spring Boot Actuator endpoint ***/configprops***

Cloud Config Client

- Pulls down configuration from server

pom.xml

```
<dependency>
  <groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-config-client</artifact
Id>
</dependency>
```

/src/main/resources/bootstrap.yml

```
spring:
  application:
    name: config-client
  cloud:
    config:
      uri:
http://localhost:8888/config
```

- The URI indicates where to find the configuration files
- Environment Changes
  - Cloud Configuration will listen to EnvironmentChangedEvent and will rebind any @ConfigurationProperties beans in the context
  - Change the logger level for any properties in logging.level.*
  - The default configuration detects filesystem changes in local git repositories (the webhook is not used in that case but as soon as you edit a config file a refresh will be broadcast).
  - To refresh beans that were initialized with configured values add the @RefreshScope

Session Data

- Instances of the service can come and go so you cannot rely on the standard sessions implementations
  - If the application goes down and the client access another instance on the cloud, it will be another session
- Bind your application to Redis (in memory database) for storing the session

manifest.yml

```
services:
- ll-redis-session
```

```
@SpringBootApplication
@EnableRedisHttpSession
public class CloudSessionApplication {

  public static void main(String[] args) {
     SpringApplication.run(CloudSessionApplication.class, args);
   }

}
```

http://projects.spring.io/spring-session

Other Storage Precautions

- **Files**
  - As the application can be removed from memory,it cannot rely on disk resources
  - Temporary storage is usually fine, but long term storage is not
  - Use:
    - Mongo GridFS
    - Amazon Simple Storage Service (S3)
- **Database in Microservices**
  - There are two common deployment models:
    - Apps can run databases on the cloud independently, using a virtual machine image
      - Database is not synchronized
      - The database is not shared across microservices
      - Services have to use the responsible microservice to obtain data from the database
    - Or they can purchase access to a database service, maintained by a cloud database provider (DBaaS)

Active Learning

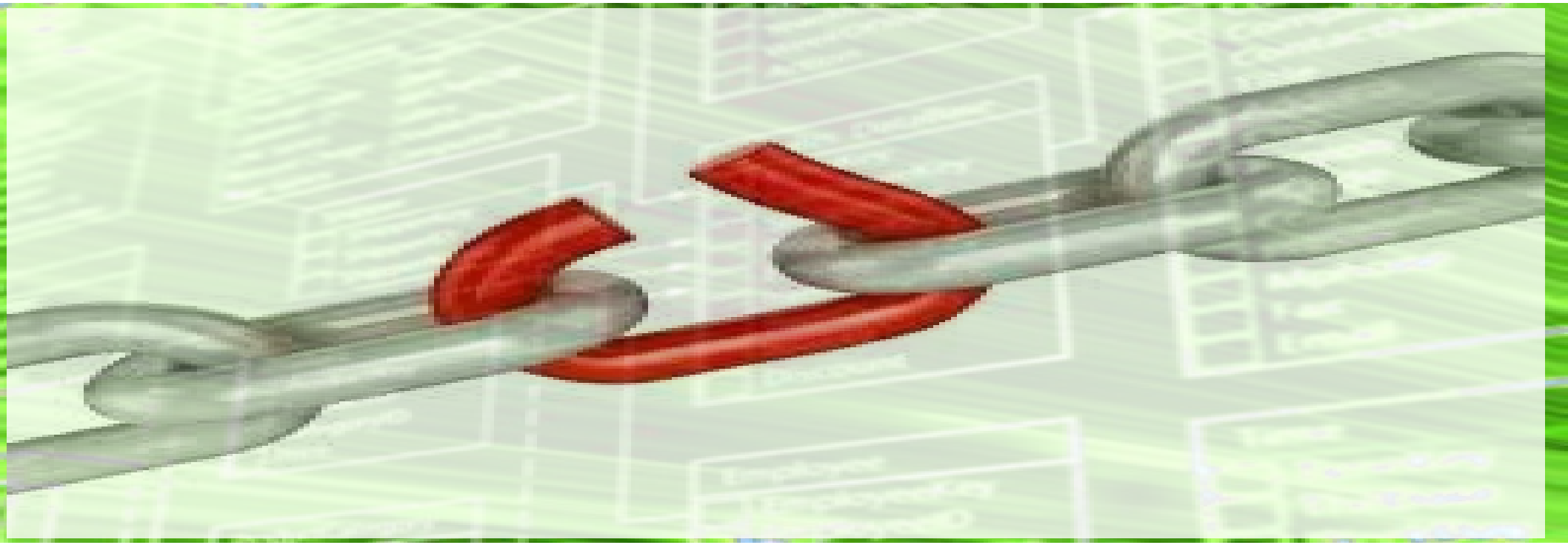
- How does the microservice architecture differs from the traditional SOA?

- What advantages do we gain using Spring Cloud Config Client?

Main Points

1) Dividing the application on its core simple components allows us to orchestrate those parts so together they compose the whole application. This provides the flexibility of scaling up the system with the appropriate amount of resources, in a more natural way. ***Science of Consciousness:*** *Before the unmanifest sap starts to green or red, it must begin to function within itself, the activity must start in unmanifest value, and then the next stage of that activity will be turning to be green.*

Microservices Fragility



“A chain is only as strong as the weakest link”

Fault Tolerance

- As you introduce different services, you add more points of failure into your system
- The Service Level Agreement is only as good as the weakest in the chain
- Build in fault-tolerance from the beginning
  - Service registration and discovery with Consul (HashiCorp) or Eureka
  - Circuit breakers or bulkheads with Hystrix and the Hystrix Dashboard to isolate points of access, stop cascading failure and enable resilience
  - Declarative REST service clients with Feign
  - Client-side load balancing with Ribbon
  - Integration with RestTemplate, Feign, Zuul (gateway that provides dynamic routing, monitoring, resiliency, security)

Service Registration & Discovery

- A Service Registry is like a phone-book
- Clients request if a particular service is available, and the service discovery responds with the host/port
- Automatic registration of services during initialization

```
@SpringBootApplication
@EnableEurekaServer
public class EurekaServerApplication {

  public static void main(String[] args) {
    SpringApplication.run(EurekaServerApplication.class, args);
  }

}
```

application.yml

```
server:
  port: ${PORT:8761}

eureka:
  client:
    registerWithEureka: false
    fetchRegistry: false
  server:
    waitTimeInMsWhenSyncEmpty: 0
```

Register Service with Eureka

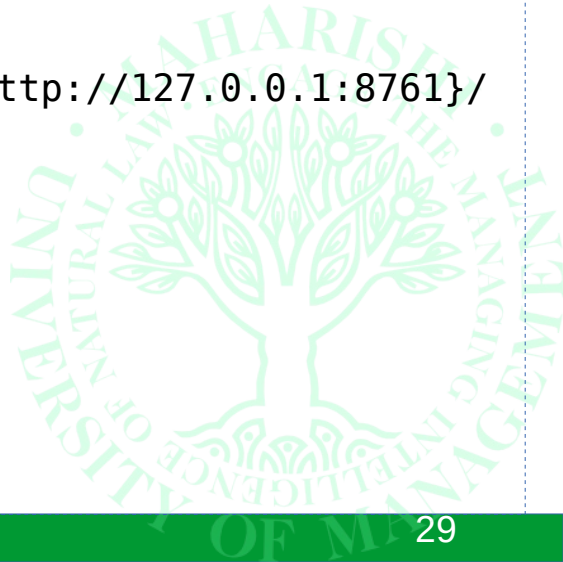
bootstrap.yml

```yaml
spring:
  application:
    name: bookmark-service
```

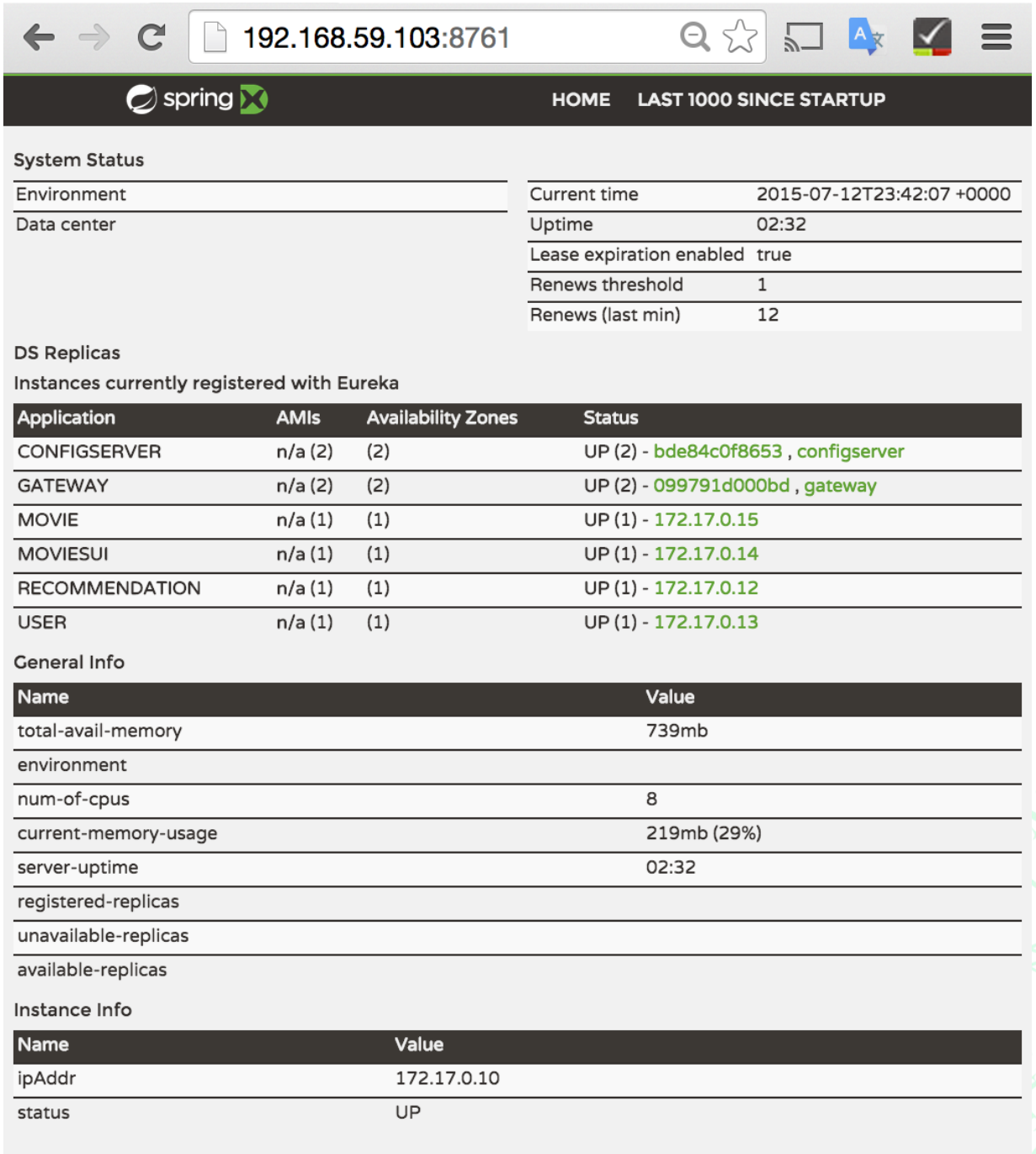
```
@SpringBootApplication
@EnableEurekaClient
public class BookmarkServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(
            BookmarkServiceApplication.class,
args);
    }
}
```

application.yml

```yaml
server:
  port: ${PORT:0}

spring:
  jpa:
    generate-ddl: true


eureka:
  client:
    serviceUrl:
      defaultZone: ${vcap.services.eureka-service.credentials.uri:http://127.0.0.1:8761}/
eureka/

---
spring:
  profiles: cloud
eureka:
  instance:
    hostname: ${APPLICATION_DOMAIN}
    nonSecurePort: 80
```



Eureka Dashboard



Client-Side Load Balancing

- Load-balance traffic in the mid-tier services along with service registry
  - Rely on DNS and Cloud Load Balancers at the edge services
- **Ribbon** provides many software load-balanccers out of the box
  - It is integrated with Feign, Spring RestTemplate, and Zuul automatically when using Spring Cloud

```
server:
  port: ${PORT:9023}
eureka:
  client:
    serviceUrl:
      defaultZone:
${vcap.services.eureka-service.credentials.uri:http://127.0.0.1:8761}/eureka/

---
spring:
  profiles: cloud
eureka:
  instance:
    hostname: ${APPLICATION_DOMAIN}
    nonSecurePort: 80
```

Feing Client Example (Continued)

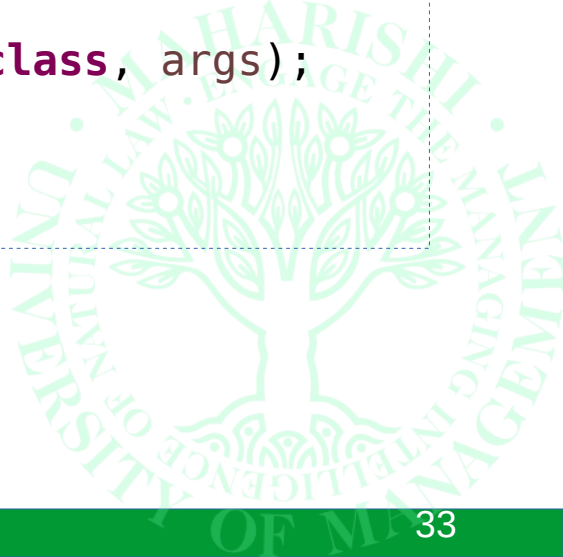
```
@FeignClient("bookmark-service")
public interface BookmarkClient {

  @RequestMapping(method = RequestMethod.GET, value =
"/{userId}/bookmarks")
  Collection<Bookmark> getBookmarks(@PathVariable("userId") String
userId);

}
```

```
@Component
public class FeignExample implements CommandLineRunner
{

@Autowired
private BookmarkClient bookmarkClient;

  @Override
  public void run(String... strings) throws Exception {
    this.bookmarkClient.getBookmarks("Joe")
        .forEach(System.out::println);
  }

}
```

Circuit Breaker

- Isolates service failure with basic state machine around service-to-service communication
- Hystrix is easy to plug in and easy to monitor

```
@Controller
@EnableHystrixDashboard
@SpringBootApplication
public class HystrixDashboardApplication {

  @RequestMapping("/")
  public String home() {
    return "forward:/hystrix/index.html";
  }

  public static void main(String[] args) {
    SpringApplication.run(HystrixDashboardApplication.class, args);
  }

}
```

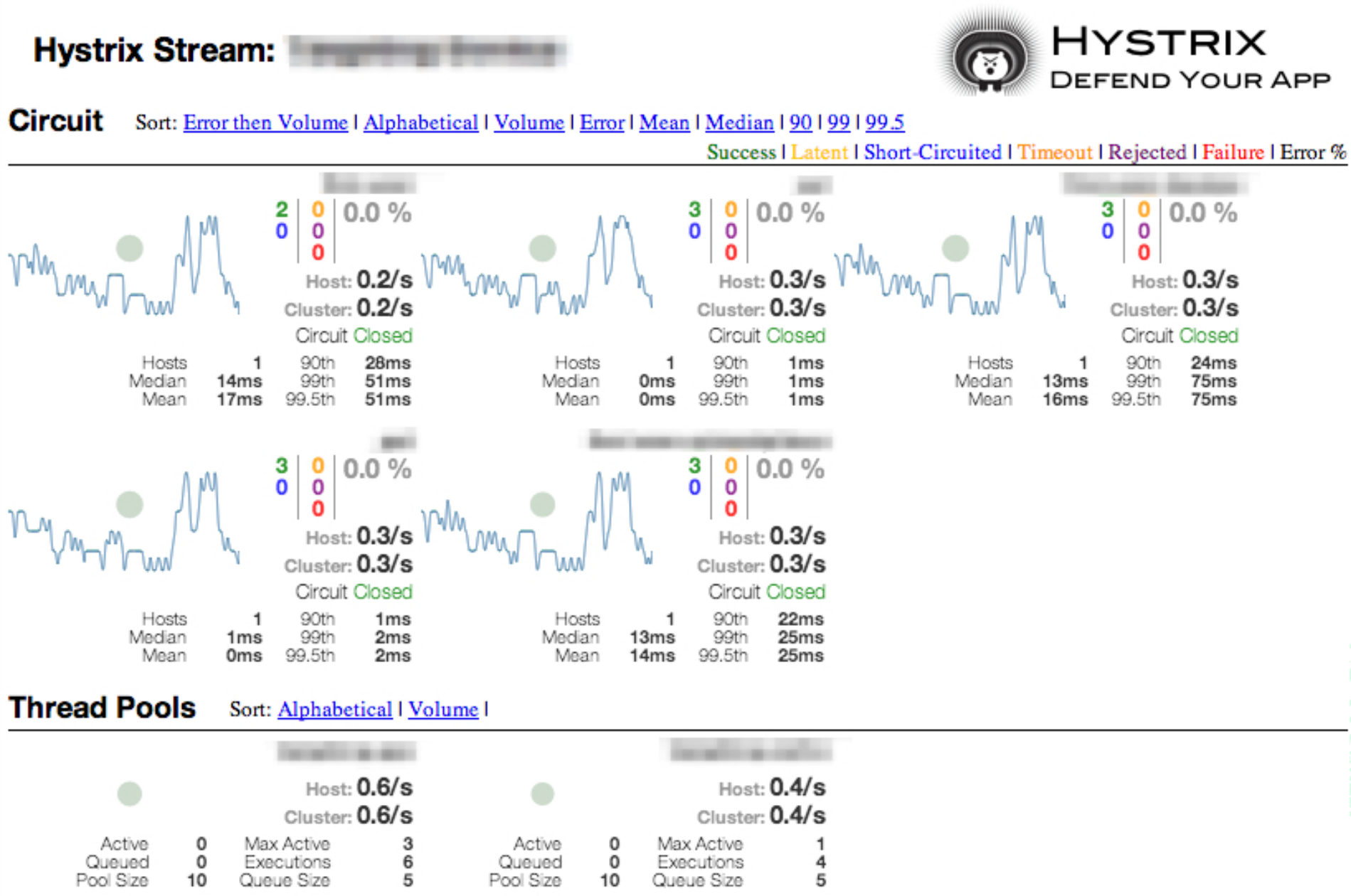


Circuit Breaker Application

```
@SpringBootApplication
@EnableEurekaClient
@EnableFeignClients
@EnableCircuitBreaker
public class CircuitBreakerApplication {

  public static void main(String[] args) {
    SpringApplication.run(CircuitBreakerApplication.class, args);
  }

}
```

```
@Component
class IntegrationClient {

  @Autowired
  private BookmarkClient bookmarkClient;

  public Collection<Bookmark> getBookmarksFallback(String userId) {
    System.out.println("getBookmarksFallback");
    return Arrays.asList();
  }

  @HystrixCommand(fallbackMethod = "getBookmarksFallback")
  public Collection<Bookmark> getBookmarks(String userId) {
    return this.bookmarkClient.getBookmarks(userId);
  }
...
```

Hysrtix Dashboard



Security

- Security is hard – don't reinvent the wheel – use Spring Security, it takes care of:
  - Identification, authorization and authentication
  - Encryption
  - Vulnerabilities like cross-site scripting, session fixation
  - and more

Main Points

2. The Discovery Services, together with load balancing and client templates, allows us to create a flexible fault tolerant environment which can provide service to many clients. ***Science of Consciousness****: A person established in Cosmic Consciousness is permanently happy, as if he is connected with a storehouse of infinite bliss.*

Summary

- Microservices architecture allows to build applications that are more scalable and fault-tolerant
- Microservices introduce complexity for the inter-service dependencies
- Spring Cloud provides a rich set of tools for dealing with that complexity
  - Supports patters like the circuit breaker
  - Declarative REST clients
  - Client-side load balancing
  - Gateway API

UNITY CHART

CONNECTING THE PARTS OF THE KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE

Configuring the services to provide high availability

1) Spring Cloud integrates a set of tools in a seamlessly way, so developing complex microservice applications is done through simple steps.
2) Microservices architecture provides a flexible structure that can be scaled up and maintained with ease.

3) **Transcendental Consciousness:** is the evolutionary power.

4) **Impulses in the Transcendental Field:** the wakeful state of pure potentiality – absolute aware of itself.
5) **Wholeness moving within itself:** All levels of the personality – sense perception, thought, intellect, feelings, ego, are fully developed, requiring little if any exertion to function.