Spring Security

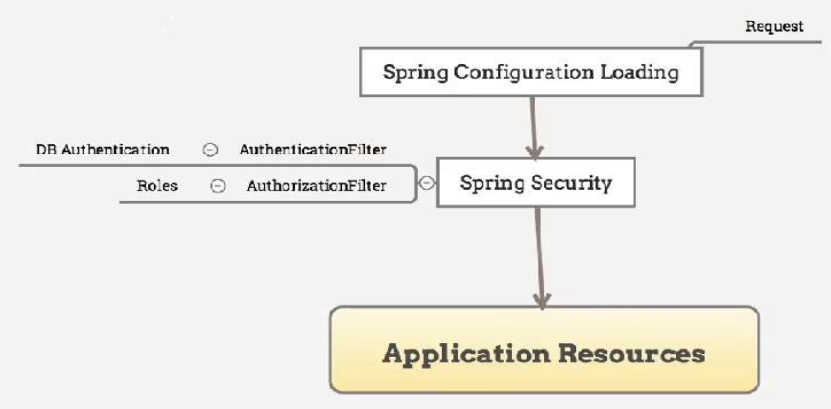
145

Spring Security Features

* LDAP (Lightweight Directory Access Protocol)
* Single sign-on
* JAAS (Java Authentication and Authorization Service) LoginModule
* Basic Access Authentication
* Digest Access Authentication
* Remember-me
* Web Form Authentication
* Authorization
* Software Localization
* HTTP Authorization
* OAuth 2.0 Login (Spring 5.0 onwards)
* Reactive Support
* Modernized Password Encoding

146

Spring security Architecture



147

Spring Security Java Based

```
@EnableWebSecurity
@Configuration
public class SecurityConfiguration extends WebSecurityConfigurerAdapter{}
```

The above class will have below methods

```
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
auth.inMemoryAuthentication()
.withUser("ameya").password("1234").roles("ADMIN").and()
.withUser("avani").password("1234").roles("USER");
}
protected void configure(HttpSecurity httpSecurity) throws Exception {
httpSecurity
.authorizeRequests() //all requests are authorized
.antMatchers("/").permitAll()
.antMatchers("/rest/hello*").access( "hasRole('USER')")
.and()
.httpBasic();
httpSecurity.csrf().disable();
}
```

148

Get userName in code

```
private String getPrincipal(){
String userName = null;
Object principal  =
SecurityContextHolder.getContext().getAuthentication().
        getPrincipal();
if (principal instanceof UserDetails) {
userName = ((UserDetails)principal).getUsername();
} else {
userName = principal.toString();
}
return userName;
}
```

149

Spring Boot-Spring Security

* Dependencies :

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-security </artifactId>
 </dependency>
```

150

MICROSERVICES

151

What is Microservices?

* Microservices is not a new term. It coined in 2005 by Dr Peter Rodgers
* It was then called micro web services based on SOAP. It became more popular since 2010.
* Micoservices allows us to break our large system into number of independent collaborating processes.

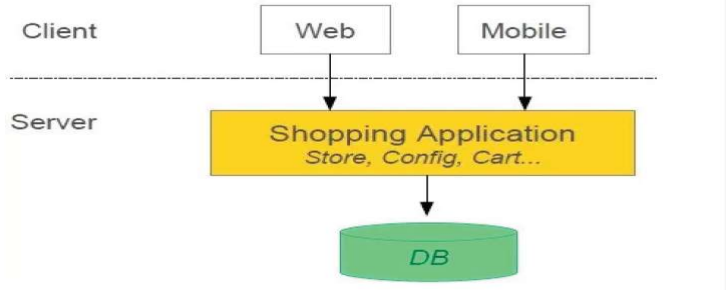
152

What is Microservices?

* Microservices architecture allows to avoid monolith application for large system.
* It provides loose coupling between collaborating processes running independently in different environments with tight cohesion.
* It is an architectural style that structures an application as a collection of loosely coupled services, which implement business capabilities. The microservice architecture enables the continuous delivery/deployment of large, complex applications. It also enables an organization to evolve its technology stack.

153

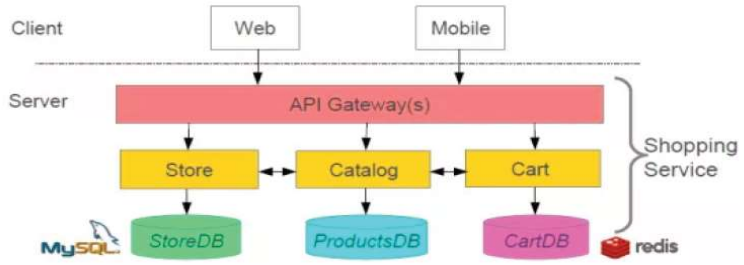
Example : Monolith Architecture



* This is Monolith architecture i.e. all collaborating components combine all in one application.

154

Example : Microservices Architecture



* This is Microservices architecture , One large Application divided into multiple collaborating processes

155

Microservices Challenges

* Difficult to achieve strong consistency across services
* ACID transactions do not span multiple processes
* In Distributed System it is hard to debug and trace the issues
* Greater need for end to end testing
* Platform as a Service like Pivotal Cloud Foundry help to deployment,easily run, scale, monitor etc.
* Continuous deployment, rolling upgrades of new versions of code, running multiple versions of same service at same time

156

Microservices – 12 Factor App

* In the modern era, software is commonly delivered as a service: called *web apps*, or *software-as-a-service*. The twelve-factor app is a methodology for building software-as-a-service apps that:
* Use **declarative** formats for setup automation, to minimize time and cost for new developers joining the project;
* Have a **clean contract** with the underlying operating system, offering **maximum portability** between execution environments;
* Are suitable for **deployment** on modern **cloud platforms**, obviating the need for servers and systems administration;
* **Minimize divergence** between development and production, enabling **continuous deployment** for maximum agility;
* And can **scale up** without significant changes to tooling, architecture, or development practices.
* The twelve-factor methodology can be applied to apps written in any programming language, and which use any combination of backing services (database, queue, memory cache, etc).

157

The Twelve Factors

* **I. Codebase**
  * One codebase tracked in revision control, many deploys
* **II. Dependencies**
  * Explicitly declare and isolate dependencies
* **III. Config**
  * Store config in the environment
* **IV. Backing services**
  * Treat backing services as attached resources
* **V. Build, release, run**
  * Strictly separate build and run stages
* **VI. Processes**
  * Execute the app as one or more stateless processes

158

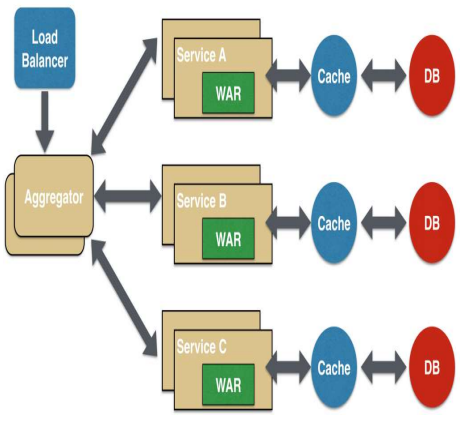
The Twelve Factors

- **VII. Port binding**
  - Export services via port binding
- **VIII. Concurrency**
  - Scale out via the process model
- **IX. Disposability**
  - Maximize robustness with fast startup and graceful shutdown
- **X. Dev/prod parity**
  - Keep development, staging, and production as similar as possible
- **XI. Logs**
  - Treat logs as event streams
- **XII. Admin processes**
  - Run admin/management tasks as one-off processes

159

Microservices – Design Patterns

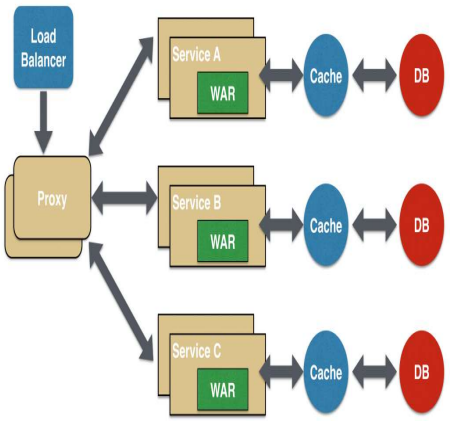
- **Aggregator Microservice Design Pattern**
- In its simplest form, Aggregator could be a simple web page that invokes multiple services to achieve the functionality required by the application. Since each service (Service A, Service B, and Service C) is exposed using a lightweight REST mechanism, the web page can retrieve the data and process/display it accordingly.



160

Microservices – Design Patterns

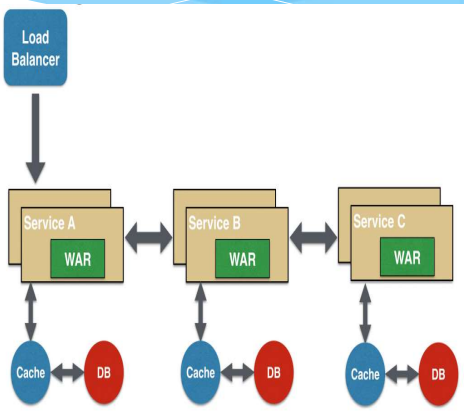
* **Proxy Microservice Design Pattern**
* Proxy microservice design pattern is a variation of Aggregator. In this case, no aggregation needs to happen on the client but a different microservice may be invoked based upon the business need.



161

Microservices – Design Patterns

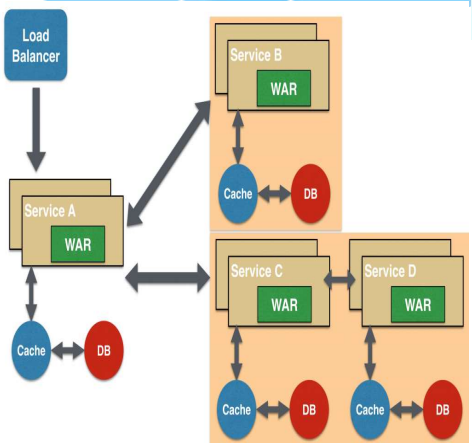
* **Chained Microservice Design Pattern**
* Chained microservice design pattern produce a single consolidated response to the request. In this case, the request from the client is received by Service A, which is then communicating with Service B, which in turn may be communicating with Service C. All the services are likely using a synchronous HTTP request/response messaging.



162

Microservices – Design Patterns

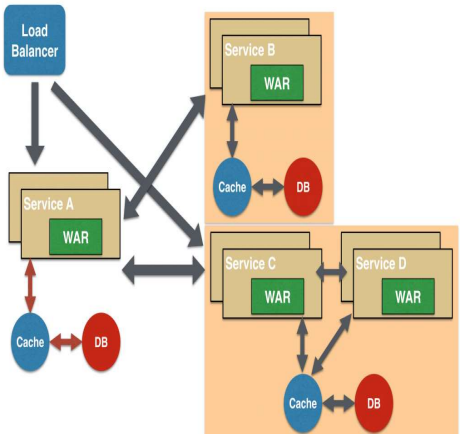
* **Branch Microservice Design Pattern**
* Branch microservice design pattern extends Aggregator design pattern and allows simultaneous response processing from two, likely mutually exclusive, chains of microservices. This pattern can also be used to call different chains, or a single chain, based upon the business needs.



163

Microservices – Design Patterns

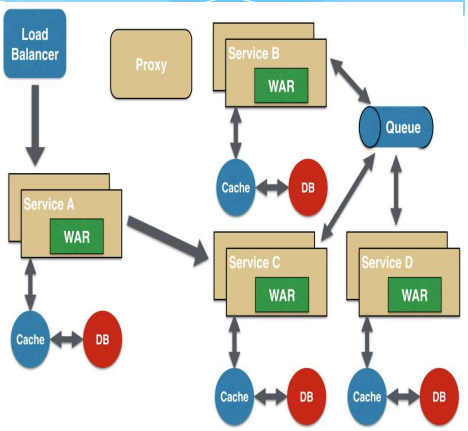
* **Shared Data Microservice Design Pattern**
* One of the design principles of microservice is autonomy. That means the service is full-stack and has control of all the components – UI, middleware, persistence, transaction. This allows the service to be polyglot, and use the right tool for the right job



164

Microservices – Design Patterns

* **Asynchronous Messaging Microservice Design Pattern**
* While REST design pattern is quite prevalent, and well understood, but it has the limitation of being synchronous, and thus blocking. Asynchrony can be achieved but that is done in an application specific way. Some microservice architectures may elect to use message queues instead of REST request/response because of that.



165

Microservices Benefits

* Smaller code base is easy to maintain.
* Easy to scale as individual component.
* Technology diversity i.e. we can mix libraries, databases, frameworks etc.
* Fault isolation i.e. a process failure should not bring whole system down.
* Better support for smaller and parallel team.
* Independent deployment of various components()
* Reduced Deployment time

166

Spring Cloud

167

Spring Cloud

* What is Spring Cloud?
  * It is building blocks for Cloud and Microservices
  * It provides microservices infrastructure like provide and use services such as Service Discovery, Configuration server and Monitoring etc.
  * It provides several other open source projects like Netflix OSS.
  * It provides PaaS like Cloud Foundry, AWS and Heroku.
  * It uses Spring Boot style starters

168

Spring Boot Spring Cloud

* Dependencies :

Your Microservices, will depend on the below parent project. This will enable them to work with various spring cloud features.

```
<parent>
  <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-parent</artifactId>
      <version>Brixton.RELEASE</version>
 </parent>
```

169

Spring Cloud Config

170

Config Server

* Config server is where all configurable parameters of microservices are written and maintained.
* It is more like externalizing properties / resource file out of project codebase to an external service altogether, so that any changes to that property does not necessitate the deployment of service which is using the property.
* All such property changes will be reflected without redeploying the microservice.

171

Spring Cloud Config Server

* The idea of *config server* has come from the 12-factor app manifesto related to best practice guideline of developing modern cloud native application.
* It suggests **to keep properties / resources in the environment of the server** where the values of those resources vary during run time – usually different configurations that will differ in each environment.

172

Spring Boot-
Spring Cloud Config Server

* Dependencies :

```
<dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-config-server</artifactId>
</dependency>
<dependencyManagement>
        <dependencies>
                <dependency>
                        <groupId>org.springframework.cloud</groupId>
                        <artifactId>spring-cloud-dependencies</artifactId>
                        <version>Finchley.M8</version>
                        <type>pom</type>
                        <scope>import</scope>
                </dependency>
        </dependencies>
</dependencyManagement>
```

173

Spring Boot-
Spring Cloud Config Server

* @EnableConfigServer : This will standup a config server that other applications can talk to.
* We also need a Config Service to act as a sort of intermediary between your Spring applications and a typically version-controlled repository of configuration files.

* bootstrap.properties/bootstrap.yml :

Server port

server.port = 9088

spring.cloud.config.server.git.uri=file://E:/practice/spring-boot/config-server-repo

management.security.enabled=false

174

Enable git

- Install Git.
- Create the folder
- E:/practice/spring-boot/config-server-repo
- Create config-server-client-*.properties
- Populate the msg key value in each
- Run git init
- Run git add .
- Git commit –m "initial checkin"
- After modifying the files again run git add .
  and git commit –m "test"

175

Spring Boot-
Spring Cloud Config Client

- Dependencies :

```
<dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
<dependencyManagement>
        <dependencies>
                <dependency>
                        <groupId>org.springframework.cloud</groupId>
                        <artifactId>spring-cloud-dependencies</artifactId>
                        <version>${spring-cloud.version}</version>
                        <type>pom</type>
                        <scope>import</scope>
                </dependency>
        </dependencies>
</dependencyManagement>
```

176

Spring Boot-
Spring Cloud Config Client

```
@RefreshScope
@RestController
public class MessageController {

   @Value("${msg:Hello world - Config Server is not working..pelase check}")
   private String msg;

   @RequestMapping("/msg")
   String getMsg() {
      return this.msg;
   }
}
```

177

Spring Boot-
Spring Cloud Config Client

bootstrap.properties :

```
spring.application.name=config-server-client
spring.profiles.active=production
spring.cloud.config.uri=http://localhost:9088
management.security.enabled=false
```

178

Netflix Eureka

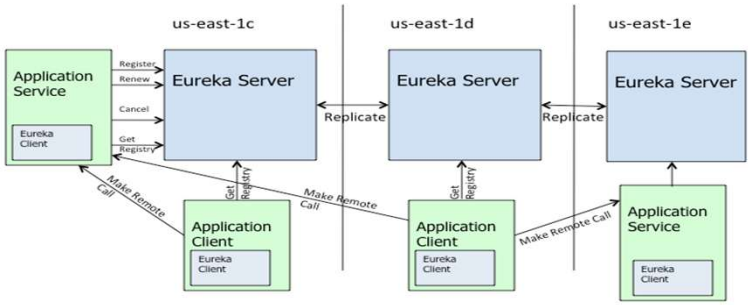
179

Netflix Eureka(Discovery Service)

* Eureka is a REST (Representational State Transfer) based service that is primarily used in the AWS cloud for locating services for the purpose of load balancing and failover of middle-tier servers.
* We call this service, the **Eureka Server**.
* Eureka also comes with a Java-based client component, the **Eureka Client**, which makes interactions with the service much easier.

180

High Level Architecture



181

Spring Boot-Netflix Eureka

* Dependencies :

```
<dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-eureka-
server</artifactId>
</dependency>
```

182

Spring Boot-Netflix Eureka

* @EnableEurekaServer : (Eureka Server Project)

You can use Spring Cloud's @EnableEurekaServer to standup a registry that other applications can talk to.

* @EnableEurekaClient : (Service Project)

Any Spring Boot application with @EnableEurekaClient will try to contact a Eureka server

* @EnableDiscoveryClient : (Service Project)

There are multiple implementations of "Discovery Service" (eureka, consul, zookeeper). @EnableDiscoveryClient picks the implementation on the classpath.

183

Spring Boot-Netflix Eureka

* Application.yml : (Eureka Server Project)

```
server:
  port: ${PORT:8761}
eureka:
  instance:
    hostname: localhost
  client:
    registerWithEureka: false
    fetchRegistry: false
  server:
    enableSelfPreservation: false
```

184

Spring Boot-Netflix Eureka

* Application.yml : (service Project – Eureka client)

```
eureka:
 client:
  serviceUrl:
   defaultZone: ${DISCOVERY_URL:http://localhost:8761}/eureka/
 instance:
  leaseRenewalIntervalInSeconds: 1
  leaseExpirationDurationInSeconds: 2
```

185

Netflix Zuul

186

Netflix Zuul(Gateway Service)

* A common problem, when building microservices, is to provide a unique gateway to the client applications.
* The fact that your services are split into small microservcies apps that shouldn't be visible to users otherwise it may result in substantial development/maintenance efforts.
* Also there are scenarios when whole ecosystem network traffic may be passing through a single point which could impact the performance of the cluster.
* To solve this problem, Netflix (a major adopter of microservices) created an open-sourced **Zuul proxy server** .
* Zuul is an edge service that proxies requests to multiple backing services.
* It provides a unified "front door" to your ecosystem, which allows any browser, mobile app or other user interface to consume services from multiple hosts

187

Spring Boot-Zuul

* Dependencies :

```
<dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-zuul</artifactId>
</dependency>
```

188

Netflix Hystrix

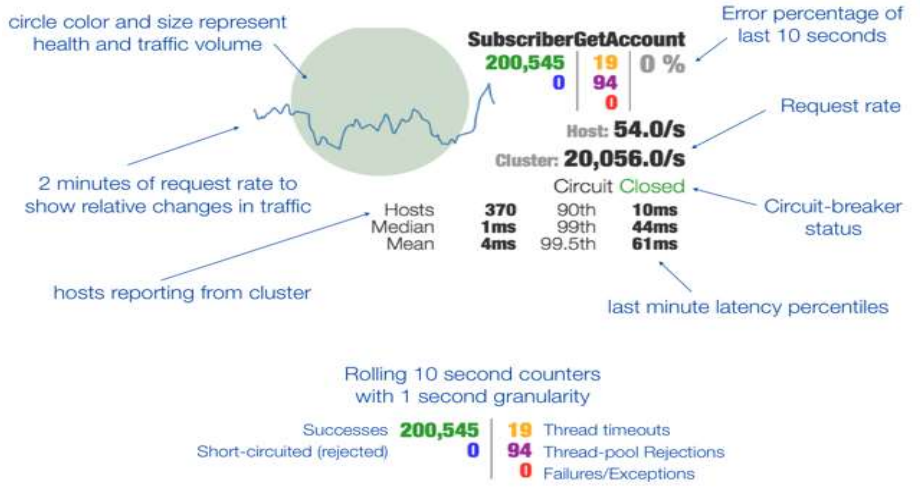
189

Netflix Hystrix (latency and fault Tolerance)

* Hystrix is a latency and fault tolerance library designed to isolate points of access to remote systems, services and 3rd party libraries, stop cascading failure and enable resilience in complex distributed systems where failure is inevitable.
* It is a circuit breaker pattern

190

Hystrix Dashboard



191

Spring Boot-Hystrix

* Dependencies :

```
<dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-hystrix</artifactId>
</dependency>
<dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-hystrix-dashboard
        </artifactId>
</dependency>
```

192

Spring Boot-Hystrix

* @EnableHystrixDashboard/@EnableCircuitBreaker :

This will **enable Hystrix circuit breaker** in the application and also will add one useful dashboard running on localhost provided by Hystrix.

* @HystrixCommand(fallbackMethod = “invokeStudentServiceAndGetData_Fallback") :

Add fallback method – invokeStudentServiceAndGetData_Fallback which will simply return some default value.

193

Spring-Boot Spring Reactive

194

194

Reactive Programming

* Reactive systems have certain characteristics that make them ideal for low-latency, high-throughput workloads.
* Project Reactor and the Spring portfolio work together to enable developers to build enterprise-grade reactive systems that are responsive, resilient, elastic, and message-driven.

195

What is reactive processing?

* Reactive processing is a paradigm that enables developers build
* non-blocking,
* asynchronous applications that can handle back-pressure (flow control).

196

Why use reactive processing?

* Reactive systems better utilize modern processors.
* The inclusion of back-pressure in reactive programming ensures better resilience between decoupled components.

197

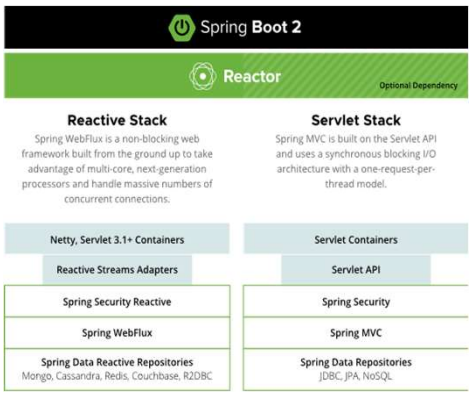
Reactive Microservices

* One of the main reason developers move from blocking to non-blocking code is efficiency.
* Reactive code does more work with fewer resources.
* Project Reactor and Spring WebFlux let developers take advantage of multi-core, next-generation processors,handling potentially massive numbers of concurrent connections.
* With reactive processing, you can satisfy more concurrent users with fewer microservice instances.

198

Springboot - Reactive Microservices

* The Spring portfolio provides two parallel stacks.
* One is based on a Servlet API with Spring MVC and Spring Data constructs.
* The other is a fully reactive stack that takes advantage of Spring WebFlux and Spring Data's reactive repositories.



199

Springboot Dependencies

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

* Reactive data types called **MONO** or **FLUX** is used to model the reactive stream.
* **MONO** or **FLUX** under the hood implement Publisher and is just a function which iterates over the data source and pushes the values to consumer.
* **Flux** is a stream which can emit **0..N** elements.
* **Mono** is a stream of **0..1** elements.

200

201