SPRING BOOT and MICROSERVICES

Ameya J. Joshi
Email : ameya.joshi@vinsys.com
Cell No : +91 98 506 76160

AGENDA

* Spring framework Concept and Limitations
* Introduction To Spring Boot
* Spring Boot features
* Setup Spring Boot
* Writing First App

AGENDA

* Various ways to create Spring Boot App
* Spring Boot Actuators
* Spring Boot Spring Security
* JMS with Spring Boot
* SOAP with Spring Boot
* Microservices
* Microservices Using Spring Boot

AGENDA

* Spring Cloud
* Spring Cloud Config Server
* Netflix Eureka with Spring Boot
* Netflix ZUUL with Spring Boot
* Netflix Hystrix with Spring Boot
* ZipKin with Spring Boot

AGENDA

* Netflix Ribbon with Spring Boot
* Pivotal Cloud Foundry(PCF)
* Deploying Spring Boot App on PCF

SPRING Framework

What is spring?

* Spring framework is an open source Java platform and it was initially written by Rod Johnson and was first released under the Apache 2.0 license in June 2003.
* Spring is lightweight when it comes to size and transparency.
* Spring framework helps to simplify the development of Java based Enterprise Applications.

Spring features

* Lightweight
* Inversion of control (IOC)
* Aspect oriented (AOP)
* Container
* MVC Framework
* uses Plain Old Java Objects(POJO)

Benefits of Using Spring Framework

* Spring enables developers to develop enterprise-class applications using POJOs.
* Spring's web framework is a well-designed web MVC framework
* Lightweight IoC containers

Beans

* The objects that form the backbone of application and that are managed by the Spring IoC container are called beans.
* A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container.

Different Forms of Dependency Injection

- Setter Injection
- Constructor Injection

Scope of Beans

* When defining a <bean> in Spring, a scope for that bean needs to be specified.
* There are five scopes for a bean in Spring
  * Singleton
  * Prototype
  * Request
  * Session
  * Global-session

Spring MVC

* **@Controller** : Exposes the class as Spring Controller for MVC
* **@RestController :** this annotation marks the class as a Resource, it defines implicitly both @Controller and @ResponseBody MVC annotations, when annotating a class with @RestController, it's not necessary to write @ResponseBody beside the POJO classes returned from your methods.
* **@RequestMapping :** this annotation defines the url of the resource in addition to the method type: GET/POST, in our example we expose the payment service as POST which is accessed through /payment/pay.
* **@RequestParam :** this annotation represents a specific request parameter, in our example, we map a request parameter called key to an argument key of type String.
* **@RequestBody :** this annotation represents the body of the request, in our example, we map the body of the request to a POJO class of type PaymentRequest(Jackson handles the JSON/POJO conversion).

Spring MVC

* @RequestMapping(value="/viewAll")
* @RequestMapping(value = "/saveEmployee", method = RequestMethod.POST)
* @GetMapping(path="/topics")
* @GetMapping(path="/topics/{id}")

Public Topic getTopic(@PathVariable("id") String id){}

* @PostMapping(path="/topics")
* @PutMapping(path="/topics/{id}")
* @DeleteMapping(path="/topics/{id}")

Spring Framework Limitations

* Huge framework
* Multiple setup steps
* Multiple configuration steps
* Multiple Build and Deploy steps
* **Can We abstract these all steps?**

SPRING BOOT

What is Spring Boot?

* Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run".

What is Spring Boot?

* Opinionated (It makes certain assumptions)
* Convention Over Configuration
* Stand alone
* Production ready
* Spring module which provides RAD (Rapid Application Development) feature to Spring framework.

Features

* Create stand-alone Spring applications
* Embed Tomcat, Jetty or Undertow directly (no need to deploy WAR files)
* Provide opinionated 'starter' POMs to simplify your Maven configuration
* Automatically configure Spring whenever possible
* Provide production-ready features such as metrics, health checks and externalized configuration
* Absolutely **no code generation** and **no requirement for XML** configuration

Setup Spring Boot

* Pre-requisites
  * Hardware
    * Core i5 machine
    * 4 gb ram
  * Software
    * 64 bit Windows 7/10
    * Java 1.8
    * Spring Tools Suite (We use sts 3.9.2)

Setup Spring Boot

* Install Maven
  * Set MAVEN_HOME
  * Add it to PATH Environment Variable
  * Run mvn –version from command prompt to ensure maven is installed

Setup Spring Boot

* Start STS
* Create new Maven Project
  * In the STS UI
    * Check Create a simple project
    * Click on next
    * Enter Group Id (com.ameya)
    * Enter Artifact Id (course-api)
    * Enter Version (Keep default)
    * Enter Name (Ameya Joshi Course Api)

Setup Spring Boot

* Add following in pom.xml ,
* save the file and update the Maven Project

```
<parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.7.RELEASE</version>
</parent>
<dependencies>
        <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
</dependencies>
<properties>
        <java.version>1.8</java.version>
</properties>
```

Few more dependencies.

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

<dependency>
     <groupId>org.apache.derby</groupId>
      <artifactId>derby</artifactId>
       <scope>runtime</scope>
</dependency>
```

Few more dependencies.

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

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

Writing First App

* **Type following code and run as java application**

```
package com.ameya;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class CourseApiApp {

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

        }

}
```

Behind the Scenes

SpringApplication.*run(CourseApiApp.**class, args**);*

This runs the CourseApiApp class

This class is annotated with @SpringBootApplication

The @SpringBootApplication annotation is equivalent to using @Configuration, @EnableAutoConfiguration, and @ComponentScan

Behind the Scenes

* As a result Spring Boot :
* Sets up the default configuration
* Starts Spring application context
* Performs classpath scan
* Starts tomcat server

Few Other ways to start Spring Boot Application

* Spring Initializr
* Spring Boot CLI
* STS IDE

Spring Initializr

* Browse to :
  * http://start.spring.io/
* Fill in the necessary and required fields.
* Add the required dependencies
* Click on the Generate Project.
* Extract the downloaded zip file/
* Go through the pom.xml

Spring Boot CLI(Command Line Interface)

* It is a command line tool that can be used if you want to quickly prototype with spring. It allows you to run Groovy scripts, which means that you have familiar Java-like syntax, without much boilerplate code.
* You can download The Spring CLI distribution from the spring software repository
* Once downloaded the zip file, follow the instructions from the install.txt from unpacked archive.
* You could also use SDKMAN(Windows)/Homebrew(Osx) for installation.

STS IDE

* Create new Spring starter Project
* In the Guided UI fill in the appropriate values
* Select the required dependencies

Demos

* REST USING SPRING BOOT
* MVC USING SPRING BOOT

Spring Security

Introduction

* Spring Security is a framework which provides various security features like: authentication, authorization to create secure Java Enterprise Applications.
* It is a sub-project of Spring framework which was started in 2003 by Ben Alex. Later on, in 2004, It was released under the Apache License as Spring Security 2.0.0.
* It overcomes all the problems that come during creating non spring security applications and manage new server environment for the application.
* This framework targets two major areas of application are authentication and authorization. Authentication is the process of knowing and identifying the user that wants to access.

Advantages

* Spring Security has numerous advantages. Some of that are given below.
* Comprehensive support for authentication and authorization.
* Protection against common tasks
* Servlet API integration
* Integration with Spring MVC
* Portability
* CSRF protection (Cross Site Request Forgery)
* Java Configuration support

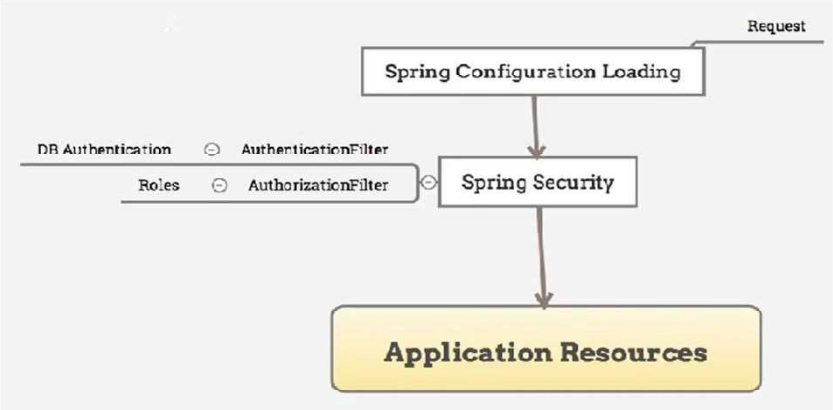
Spring Security Features

* 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

Spring security Module

* spring-security-core.jar
* spring-security-remoting.jar
* spring-security-web.jar
* spring-security-config.jar
* spring-security-ldap.jar
* spring-security-oauth2-core.jar
* spring-security-oauth2-client.jar
* spring-security-oauth2-jose.jar
* spring-security-acl.jar
* spring-security-cas.jar
* spring-security-openid.jar
* spring-security-test.jar

Spring security Architecture



Spring Security XML Based

* **Sample Extract :**

```
<http auto-config="true">
      <intercept-url pattern="/admin" access="hasRole('ROLE_ADMIN')" />
   </http>
   <authentication-manager>
    <authentication-provider>
      <user-service>
      <user name="admin" password="1234" authorities="hasRole(ROLE_ADMIN
)"/>
      </user-service>
    </authentication-provider>
   </authentication-manager>
```

Spring Security XML Based

```
<listener>
      <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
  </listener>
   <filter>
    <filter-name>springSecurityFilterChain</filter-name>
    <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
  </filter>
  <filter-mapping>
    <filter-name>springSecurityFilterChain</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
    <context-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>
        /WEB-INF/spring-servlet.xml
        /WEB-INF/spring-security.xml
      </param-value>
    </context-param>
```

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();
}
```

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;
}
```

Spring Boot-Spring Security

* Dependencies :

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

Demo

* Spring-Boot Security Basic
* Spring-Boot Security Custom Forms (Inmemory authentication)
* Spring-Boot Security Custom forms (Jdbc Authentication)

Swagger

Introduction

* Swagger (now the "Open API Initiative") is a specification and framework for describing REST APIs using a common language that everyone can understand. There are other available frameworks that have gained some popularity, such as RAML, Summation etc. but Swagger is most popular at this point of time considering its features and acceptance among the developer community.

* It offers both human readable and machine readable format of documentation. It provides both JSON and UI support. JSON can be used as machine readable format and Swagger-UI is for visual display which is easy for humans to understand by just browsing the api documentation.

Spring Boot-Swagger2

* Dependencies :

```
<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>
```

Enable Swagger2

```
@Configuration
@EnableSwagger2
public class SwaggerConfig {
   @Bean
   public Docket employeeCRUDApi() {
      return new Docket(DocumentationType.SWAGGER_2)
            .select()
            .apis(RequestHandlerSelectors.basePackage("com.ameya.employee.controllers"))
            .paths(regex("/employees.*"))
            .build()
            .apiInfo(metaData());
   }
   private ApiInfo metaData() {
      ApiInfo apiInfo = new ApiInfo(
            "Spring Boot REST API",
            "Spring Boot REST API for EmployeeCRUDService",
            "1.0",
            "Terms of service",
            new Contact("Ameya Joshi", "https://ameyajoshi.com/about/", "ameya@ameyajoshi.com"),
           "Apache License Version 2.0",
            "https://www.apache.org/licenses/LICENSE-2.0");
      return apiInfo;
   }
}
```

Swagger 2 Annotations

@Api(value="EmployeeCRUDService“)

@ApiOperation(value = "View a list of Employees", response = Iterable.**class)**

@ApiResponses(value = {

@ApiResponse(code = 200, message = "Successfully retrieved list"),

@ApiResponse(code = 401, message = "You are not authorized to view the resource"),

@ApiResponse(code = 403, message = "Accessing the resource you were trying to reach is forbidden"),

@ApiResponse(code = 404, message = "The resource you were trying to reach is not found") })

@ApiModelProperty(notes = "The Primary Key for Employee empId",required = true)

The annotations are used on top of classes/methods /attributes and swagger creates the docs respectively.

Demo

* Enabling and Using Swagger2 API

Actuators

Actuators

* Spring Boot provides actuator to monitor and manage your application.
* Actuator is a tool which has HTTP endpoints. when application is pushed to production, you can choose to manage and monitor your application using HTTP endpoints.
* To get production-ready features, we should use spring-boot-actuator module.

Actuators

* Dependencies :

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

Sample Actuator Endpoints

ID	Description	Enabled by default
auditevents	Exposes audit events information for the current application.	Yes
beans	Displays a complete list of all the Spring beans	Yes
conditions	Shows the conditions that were evaluated	Yes
configprops	Displays a collated list of all @Configuration Properties.	Yes
env	Exposes properties from Spring's Configurable Environment.	Yes
flyway	Shows any Flyway database migrations that have been applied.(Flyway is for DB version control)	Yes
health	Shows application health information.	Yes
httptrace	Displays HTTP trace information (by default, the last 100 HTTP request-response exchanges).	Yes

Demo

* Some Commonly used Actuator endpoints using REST Client

Spring Boot JMS-Support

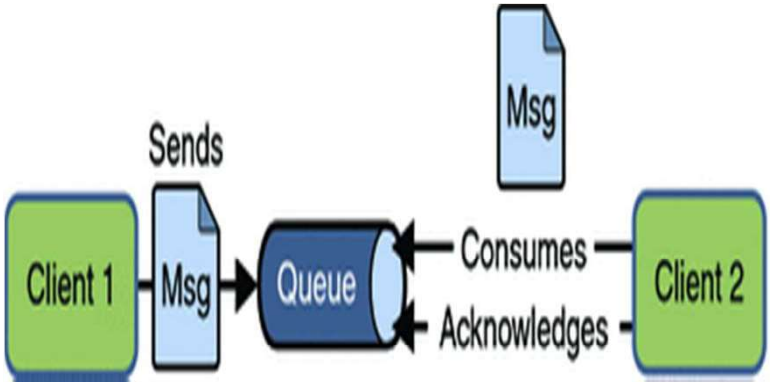
JMS (Java Message Service)

* JMS short for Java Message Service provides a mechanism for integrating applications in a loosely coupled, flexible manner.
* JMS delivers data asynchronously across applications on a store and forward basis.
* Applications communicate through MOM (Message Oriented Middleware) which acts as an intermediary without communicating directly.

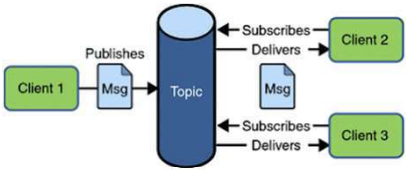
JMS Architecture

* Main components of JMS are:
* JMS Provider: A messaging system that implements the JMS interfaces and provides administrative and control features
* Clients: Java applications that send or receive JMS messages. A message sender is called the Producer, and the recipient is called a Consumer
* Messages: Objects that communicate information between JMS clients
* Administered objects: Preconfigured JMS objects created by an administrator for the use of clients.
* There are several JMS providers available like Apache ActiveMQ and OpenMQ.

JMS Queues (Point To Point)



JMS Topics (Publisher-Subscriber)



Spring Boot-JMS

* Dependencies :

```
<dependency> <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-activemq</artifactId>
</dependency>
 <dependency>
        <groupId>org.apache.activemq</groupId>
        <artifactId>activemq-broker</artifactId>
</dependency>
<dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
 </dependency>
```

Spring Boot-JMS

* @JmsListener(destination = "mailbox", containerFactory = "myFactory")
* The JmsListener annotation defines the name of the Destination that this method should listen to and the reference to the JmsListenerContainerFactory to use to create the underlying message listener container.
* Strictly speaking that last attribute is not necessary unless you need to customize the way the container is built as Spring Boot registers a default factory if necessary.

Spring Boot-JMS

* @EnableJms :
* This triggers the discovery of methods annotated with @JmsListener, creating the message listener container under the covers.

Spring Boot-JMS

```
@Bean
 public JmsListenerContainerFactory<?>
myFactory(ConnectionFactory connectionFactory,
DefaultJmsListenerContainerFactoryConfigurer configurer)
{
        DefaultJmsListenerContainerFactory factory = new
        DefaultJmsListenerContainerFactory();
        configurer.configure(factory, connectionFactory);
        return factory;
 }
```

Spring Boot-JMS

```
@Bean // Serialize message content to json using TextMessage
public MessageConverter  jacksonJmsMessageConverter()
{
        MappingJackson2MessageConverter converter = new
        MappingJackson2MessageConverter();
        converter.setTargetType(MessageType.TEXT);
        converter.setTypeIdPropertyName("_type");
        return converter;
}

JmsTemplate jmsTemplate = context.getBean(JmsTemplate.class);
jmsTemplate.convertAndSend("mailbox", new
Email("info@ameya.com", "Hello"));
```

Demo

* Spring-Boot JMS Queue Application
  Using ActiveMQ

Spring Boot SOAP-Support

Spring Boot SOAP service

* Dependencies :

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web-
                services</artifactId>
</dependency>
<dependency>
        <groupId>wsdl4j</groupId>
        <artifactId>wsdl4j</artifactId>
</dependency>
```

* Plugins :

```
<plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
<plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>jaxb2-maven-plugin</artifactId>
        <version>1.5</version>
        <executions>
                <execution>
                        <id>xjc</id>
                        <goals>
                                <goal>xjc</goal>
                        </goals>
                </execution>
        </executions>
        <configuration>
                <schemaDirectory>${project.basedir}/src/main/resources/</schemaDirectory>
                <outputDirectory>${project.basedir}/src/main/java</outputDirectory>
                <clearOutputDir>false</clearOutputDir>
        </configuration>
</plugin>
```

Spring Boot SOAP Service

Sample XSD :

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" targetNamespace="http://soap.ameya.com/student"
xmlns:tns="http://soap.ameya.com/student" elementFormDefault="qualified">
 <xs:element name="StudentDetailsRequest">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="StudentDetailsResponse">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Student" type="tns:Student"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="Student">
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="standard" type="xs:int"/>
      <xs:element name="address" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Spring Boot SOAP Service

* @Endpoint registers the class with Spring WS as a potential candidate for processing incoming SOAP messages.
* @PayloadRoot is then used by Spring WS to pick the handler method based on the message's namespace and localPart. Please note the Namespace URL and Request Payload root request mentioned in this annotation.
* @RequestPayload indicates that the incoming message will be mapped to the method's request parameter.
* @ResponsePayload annotation makes Spring WS map the returned value to the response payload.

Spring Boot SOAP Service

* WsConfigurerAdapter which configures annotation driven Spring-WS programming model.
* MessageDispatcherServlet – Spring-WS uses it for handling SOAP requests. We need to inject ApplicationContext to this servlet so that Spring-WS find other beans. It also declares the URL mapping for the requests.
* DefaultWsdl11Definition exposes a standard WSDL 1.1 using XsdSchema

```
<wsdl:definitions targetNamespace="http://soap.ameya.com/student"
xmlns:tns="http://soap.ameya.com/student" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:sch="http://soap.ameya.com/student" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
<wsdl:types>
<xs:schema targetNamespace=http://soap.ameya.com/student elementFormDefault="qualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
        <xs:element name="StudentDetailsRequest">
                <xs:complexType>
                        <xs:sequence>
                                <xs:element name="name" type="xs:string" />
                        </xs:sequence>
                </xs:complexType>
        </xs:element>
        <xs:element name="StudentDetailsResponse">
                <xs:complexType>
                        <xs:sequence>
                                <xs:element name="Student" type="tns:Student" />
                        </xs:sequence>
                </xs:complexType>
        </xs:element>
        <xs:complexType name="Student">
                <xs:sequence>
                        <xs:element name="name" type="xs:string" />
                        <xs:element name="standard" type="xs:int" />
                        <xs:element name="address" type="xs:string" />
                </xs:sequence>
        </xs:complexType>
</xs:schema>
</wsdltypes>
```

```
<wsdl:message name="StudentDetailsResponse">
        <wsdl:part name="StudentDetailsResponse"
        element="tns:StudentDetailsResponse">
        </wsdl:part>
</wsdl:message>
<wsdl:message name="StudentDetailsRequest">
        <wsdl:part name="StudentDetailsRequest"
        element="tns:StudentDetailsRequest">
        </wsdl:part>
</wsdl:message>
<wsdl:portType name="StudentDetailsPort">
        <wsdl:operation name="StudentDetails">
                <wsdl:input name="StudentDetailsRequest"
                message="tns:StudentDetailsRequest">
                </wsdl:input>
                <wsdl:output name="StudentDetailsResponse"
                message="tns:StudentDetailsResponse">
                </wsdl:output>
        </wsdl:operation>
</wsdl:portType>
```

```
<wsdl:binding name="StudentDetailsPortSoap11" type="tns:StudentDetailsPort">
        <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
        style="document" />
        <wsdl:operation name="StudentDetails">
                <soap:operation soapAction="" />
                <wsdl:input name="StudentDetailsRequest">
                        <soap:body use="literal" />
                </wsdl:input>
                <wsdl:output name="StudentDetailsResponse">
                        <soap:body use="literal" />
                </wsdl:output>
        </wsdl:operation>
</wsdl:binding>
<wsdl:service name="StudentDetailsPortService">
        <wsdl:port name="StudentDetailsPortSoap11"
        binding="tns:StudentDetailsPortSoap11">
                <soap:address location="http://localhost:8080/service/student-
                details" />
        </wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

Spring Boot SOAP Client

* Dependencies :

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

Spring Boot SOAP Client

Plugins :

```
<plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
<plugin>
        <groupId>org.jvnet.jaxb2.maven2</groupId>
        <artifactId>maven-jaxb2-plugin</artifactId>
        <version>0.13.2</version>
        <executions>
                <execution>
                        <goals>
                                <goal>generate</goal>
                        </goals>
                </execution>
        </executions>
        <configuration>
                <generatePackage>com.ameya.soap.student</generatePackage>
                <generateDirectory>${project.basedir}/src/main/java</generateDirectory>
                <schemaDirectory>${project.basedir}/src/main/resources/wsdl</schemaDirectory>
                <schemaIncludes>
                <include>*.wsdl</include>
                </schemaIncludes>
        </configuration>
</plugin>
```

Spring Boot SOAP Client

```
public class SOAPConnector extends WebServiceGatewaySupport
{
    public Object callWebService(String url, Object request)
        {
            return getWebServiceTemplate().
                marshalSendAndReceive(url, request);
        }
}
```

Spring Boot SOAP Client

```
@Bean
  public Jaxb2Marshaller marshaller() {
     Jaxb2Marshaller marshaller = new Jaxb2Marshaller();
      marshaller.setContextPath("com.ameya.soap.student");
     return marshaller;
  }

@Bean
public SOAPConnector client(Jaxb2Marshaller marshaller) {
        SOAPConnector client = new SOAPConnector();
        client.setDefaultUri("http://localhost:8080/service/student-details");
        client.setMarshaller(marshaller);
        client.setUnmarshaller(marshaller);
        return client;
}
```

Spring Boot SOAP Client

```
StudentDetailsRequest request = new StudentDetailsRequest();
request.setName("Kshiti");
StudentDetailsResponse response =(StudentDetailsResponse)
client.callWebService("http://localhost:8080/service/student-details",
request);
Student student=response.getStudent();
System.out.println("Name : "+response.getStudent().getName());
System.out.println("Standard : "+response.getStudent().getStandard());
System.out.println("Address : "+response.getStudent().getAddress());
```

Spring Boot SOAP Client

SOAP ENV (Request) :

```
<SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Header/>
<SOAP-ENV:Body>
<ns2:StudentDetailsRequest xmlns:ns2="http://soap.ameya.com/student">
<ns2:name>Kshiti</ns2:name>
</ns2:StudentDetailsRequest>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Spring Boot SOAP Client

SOAP ENV (Response)

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Header/>
<SOAP-ENV:Body>
<ns2:StudentDetailsResponse xmlns:ns2="http://soap.ameya.com/student">
<ns2:Student>
<ns2:name>Kshiti</ns2:name>
<ns2:standard>2</ns2:standard>
<ns2:address>Pune</ns2:address>
</ns2:Student>
</ns2:StudentDetailsResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Demo

* Spring-Boot SOAP Server
* Spring-Boot SOAP Client

MICROSERVICES

What is Microservices?

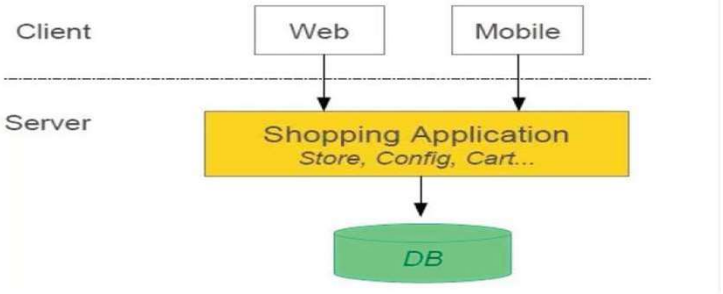
* Microservices is not a new term. It coined in 2005 by Dr Peter Rodgers
* It was then called micro web services based on SOAP. It became more popular since 2010.

* Micoservices allows us to break our large system into number of independent collaborating processes.

What is Microservices?

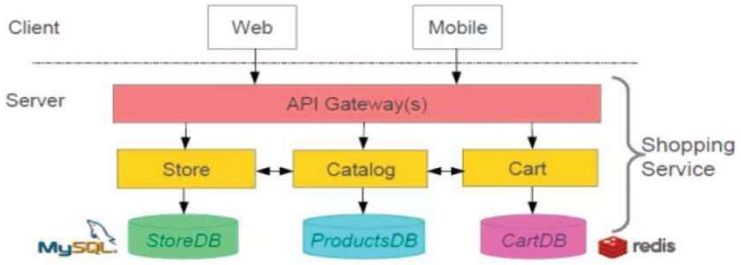
* Microservices architecture allows to avoid monolith application for large system.
* It provides loose coupling between collaborating processes running independently in different environments with tight cohesion.
* It is an architectural style that structures an application as a collection of loosely 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.

Example : Monolith Architecture



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

Example : Microservices Architecture



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

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

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

MICROSERVICES using SPRING BOOT

Spring for creating Microservices

* Setup new service by using Spring Boot
* Expose resources via a RestController
* Consume remote services using RestTemplate

Spring for creating Microservices

* Create the Service Producer
* Create the Service consumer
* Configure the services using various Microservices related features.

Spring Cloud

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

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>
```

Spring Cloud Config

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.

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.

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>
```

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

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”

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>
```

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;
   }
}
```

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

Demo

* Spring-Boot Config Server
* Spring-Boot Config Client

Netflix Eureka

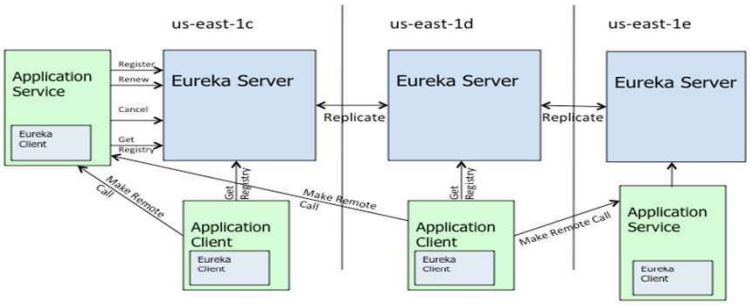
Netflix Eureka(Discovery Service)

* 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.

Need For Eureka

* In AWS cloud, because of its inherent nature, servers come and go. Unlike the traditional load balancers which work with servers with well known IP addresses and host names, in AWS, load balancing requires much more sophistication in registering and de-registering servers with load balancer on the fly. Since AWS does not yet provide a middle tier load balancer, Eureka fills a big gap in the area of mid-tier load balancing.

High Level Architecture



Spring Boot-Netflix Eureka

* Dependencies :

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

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.

* @EnableEurekClient : (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.

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
```

Spring Boot-Netflix Eureka

* Application.yml : (service Project)

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

Demo

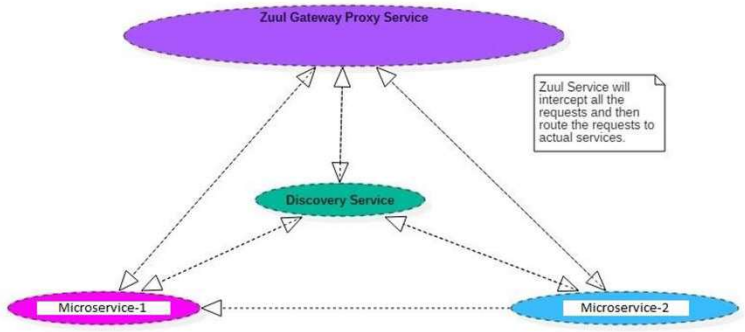
* Enabling Eureka Server
* Enabling Eureka Client

Netflix Zuul

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

Netflix Zuul



Zuul Components

* **pre filters –** are invoked before the request is routed.
* **post filters –** are invoked after the request has been routed.
* **route filters –** are used to route the request.
* **error filters –** are invoked when an error occurs while handling the request.

Zuul Components

* **Authentication and Security:** identifying authentication requirements for each resource.
* **Insights and Monitoring**: tracking meaningful data and statistics.
* **Dynamic Routing:** dynamically routing requests to different backend..
* **Stress Testing:** gradually increasing the traffic.
* **Load Shedding:** allocating capacity for each type of request and dropping requests.
* **Static Response handling:** building some responses directly.
* **Multiregion Resiliency:** routing requests across regions.

Spring Boot-Zuul

* Dependencies :

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

Spring Boot-Zuul

Application.yml :

```
info:
  component: Edge Server
endpoints:
  restart:
    enabled: true
  shutdown:
    enabled: true
  health:
    sensitive: false
zuul:
  routes:
    account:
      path: /account/**
      serviceId: account-service
    customer:
      path: /customer/**
      serviceId: customer-service
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
    registerWithEureka: false
server:
  port: 8765
```

Spring Boot Zuul

* @EnableZuulProxy :

Spring Cloud Netflix includes an embedded Zuul proxy, which we can enable with this annotation.

This will turn the Gateway application into a reverse proxy that forwards relevant calls to other services

* defaultSampler() method :Decides How the calls will be spanned.This is optional it is used for tracing.Internally Zuul uses Spring Cloud Sleuth Api which in turn will use this information.

Demo

* Enable ZUUL Proxy
* Invoking the REST Service using ZUUL

Netflix Hystrix

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

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>
```

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.

* Hystrix Enabled REST Services

SchoolService and StudentService

Zipkin

Zipkin(tracing service)

* Zipkin is a distributed tracing system. It helps gather timing data needed to troubleshoot latency problems in microservice architectures. It manages both the collection and lookup of this data. Zipkin's design is based on the Google Dapperpaper.
* Applications are instrumented to report timing data to Zipkin. The Zipkin UI also presents a Dependency diagram showing how many traced requests went through each application.
* This project includes a dependency-free library and a spring-boot server. Storage options include in-memory, JDBC (mysql), Cassandra, and Elasticsearch.

zipkin

* **Collector** – Once any component sends the trace data arrives to Zipkin collector daemon, it is validated, stored, and indexed for lookups by the Zipkin collector.
* **Storage** – This module store and index the lookup data in backend. Cassandra, ElasticSearch and MySQL are supported. Default is inMemory
* **Search** – This module provides a simple JSON API for finding and retrieving traces stored in backend. The primary consumer of this API is the Web UI.
* **Web UI** – A very nice UI interface for viewing traces.

Spring Boot Zipkin

* Dependencies :

```
<dependency>
        <groupId>io.zipkin.java</groupId>
        <artifactId>zipkin-server</artifactId>
        <version>1.18.0</version>
</dependency>

<dependency>
        <groupId>io.zipkin.java</groupId>
        <artifactId>zipkin-autoconfigure-ui</artifactId>
        <scope>runtime</scope>
        <version>1.18.0</version>
</dependency>
```

Spring Boot-Zipkin

* Application.yml :

```
server:
  port: ${PORT:9411}
eureka:
  client:
   serviceUrl:
     defaultZone: ${DISCOVERY_URL:http://localhost:8761}/eureka/
```

* @EnableZipkinServer :

will set up this server to listen for incoming spans and act as our UI for querying

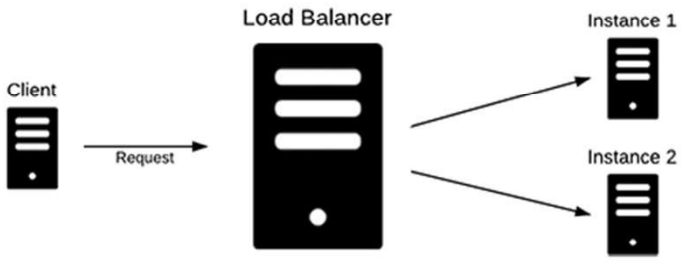
Demo

* Enabling Tracing and Monitoring using Zipkin

Netflix Ribbon

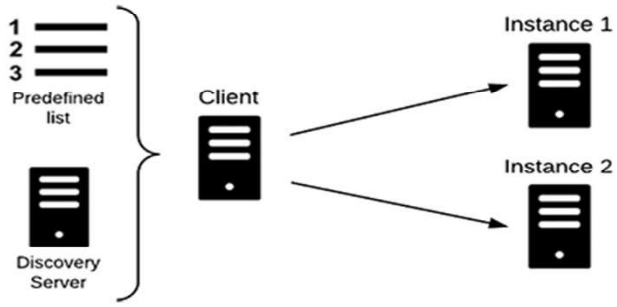
Load Balancing Client side Vs Server Side

* Server Side Load Balancing



Load Balancing Client side Vs Server Side

* Client Side Load Balancing



Netflix Ribbon (Client side Load Balancer)

* Ribbon plays a critical role in supporting inter-process communication in the cloud.
* The library includes the Netflix client side load balancers and clients for middle tier services.
* Ribbon provides the following features:
  * Multiple and pluggable load balancing rules
  * Integration with service discovery
  * Built-in failure resiliency
  * Cloud enabled
  * Clients integrated with load balancers

Spring Boot-Ribbon

* Dependencies :

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

With Following Within <dependencyManagement><dependencies>

```
<dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-dependencies</artifactId>
        <version>Edgware.SR2</version>        <type>pom</type>
            <scope>import</scope>
</dependency>
```

Spring Boot-Ribbon

* Application.yml :

```
spring:
  application:
    name:user
server:
  port: 8770
customer-service:
  ribbon:
    eureka:
      enabled: false
    listOfServers: localhost:3333,localhost:4444,localhost:5555
ServerListRefreshInterval: 15000
```

Spring Boot-Ribbon

* This configures properties on a Ribbon *client*. Spring Cloud Netflix creates an ApplicationContext for each Ribbon client name in our application.
* This is used to give the client a set of beans for instances of Ribbon components, including:
  * an IClientConfig, which stores client configuration for a client or load balancer,
  * an ILoadBalancer, which represents a software load balancer,
  * a ServerList, which defines how to get a list of servers to choose from,
  * an IRule, which describes a load balancing strategy, and
  * an IPing, which says how periodic pings of a server are performed

Spring Boot Ribbon

* This configuration is optional and will be used to override default one
* RibbonConfig.java extract :

```
        @Autowired  IClientConfig ribbonClientConfig;

@Bean
public IPing ribbonPing(IClientConfig config) {
        return new PingUrl();
 }
 @Bean
public IRule ribbonRule(IClientConfig config) {
  return new AvailabilityFilteringRule();
 }
```

Spring Boot-Ribbon

* In the restclient :

* @RibbonClient(name = “customer-service", configuration = RibbonConfig.class)
* @LoadBalanced @Bean RestTemplate restTemplate(){ return new RestTemplate(); }
* restTemplate.getForObject("http://customer-service/greeting", String.class);

Demo

* Demonstrate the Client side Load Balancing Using Ribbon on invoking a REST service.

Pivotal Cloud Foundry

Cloud Foundry

* Cloud Foundry is an open source cloud platform as a service (PaaS) .
* On which developers can build, deploy, run and scale applications.
* VMware originally created Cloud Foundry, and it is now part of Pivotal Software, whose parent company is Dell Technologies.
* Similar platforms and competitors to Cloud Foundry include OpenShift, Google App Engine and Heroku.

Cloud Foundry Components

* **Routing.** Directs traffic coming into the Cloud Foundry platform to the appropriate component.
* **Authentication.** Contains an OAuth2 server and login server for user identity management.
* **Application Lifecycle.** Provides application deployment and management services. It includes the Cloud Controller, a service that pushes or deploys an application to Cloud Foundry; pieces of Cloud Foundry's Diego architecture for the management of containerized applications; and nsync, which monitors the state of an application.

Cloud Foundry Components

* **Application Storage and Execution.** Contains a Blobstore repository for large files, as well as Diego Cell, another component of the Diego architecture. Every virtual machine (VM) has a Diego Cell that manages when an application starts and stops, as well as the VM's containers.
* **Messaging**. Provides the ability for VMs to communicate through HTTP or HTTPS protocols. It includes the Consul server, which stores long-term control data like component IP addresses, as well as the Bulletin Board System (BBS), which stores data that is updated more frequently, such as application status.

Cloud Foundry Components

* **Service Brokers.** Helps link applications to certain services, such as databases.
* **Metrics and Logging.** Provides Loggregator, which streams application logs to developers, and the Metrics Collector, which provides data to help organizations monitor their Cloud Foundry environment.

Cloud Foundry Usage

* Signup : https://account.run.pivotal.io/z/uaa/sign-up
* Login
* Click on Pivotal Web Services
* Fill in the requested details
* Create an Organization and space
* You are ready to push applications to Cloud Foundry

CF-Command Line Interface(CLI)

* **Windows Installation**
* To use the cf CLI installer for Windows, perform the following steps:
* Download the Windows installer.
* Unpack the zip file.
* Double click the cf CLI executable.
* When prompted, click **Install**, then **Close**.
* To verify your installation, open a terminal window and type cf. If your installation was successful, the cf CLI help listing appears.

CF-Command Line Interface(CLI)

* cf login –a api.run.pivotal.io
* cf push (next slide)
* cf logs [app-name] –recent
* cf logout

CF-Command Line Interface(CLI)

* **cf push** APP_NAME [-b BUILDPACK_NAME] [-c COMMAND] [-f MANIFEST_PATH | --no-manifest] [--no-start] [-i NUM_INSTANCES] [-k DISK] [-m MEMORY] [-p PATH] [-s STACK] [-t HEALTH_TIMEOUT] [-u (process | port | http)] [--no-route | --random-route | --hostname HOST | --no-hostname] [-d DOMAIN] [--route-path ROUTE_PATH]
* **cf push** APP_NAME --docker-image [REGISTRY_HOST:PORT/]IMAGE[:TAG] [--docker-username USERNAME] [-c COMMAND] [-f MANIFEST_PATH | --no-manifest] [--no-start] [-i NUM_INSTANCES] [-k DISK] [-m MEMORY] [-t HEALTH_TIMEOUT] [-u (process | port | http)] [--no-route | --random-route | --hostname HOST | --no-hostname] [-d DOMAIN] [--route-path ROUTE_PATH]
* **cf push** -f MANIFEST_WITH_MULTIPLE_APPS_PATH [APP_NAME] [--no-start]

Demo

* Creating a sample RESTful Service application
* Deploying it on PCF using PWS