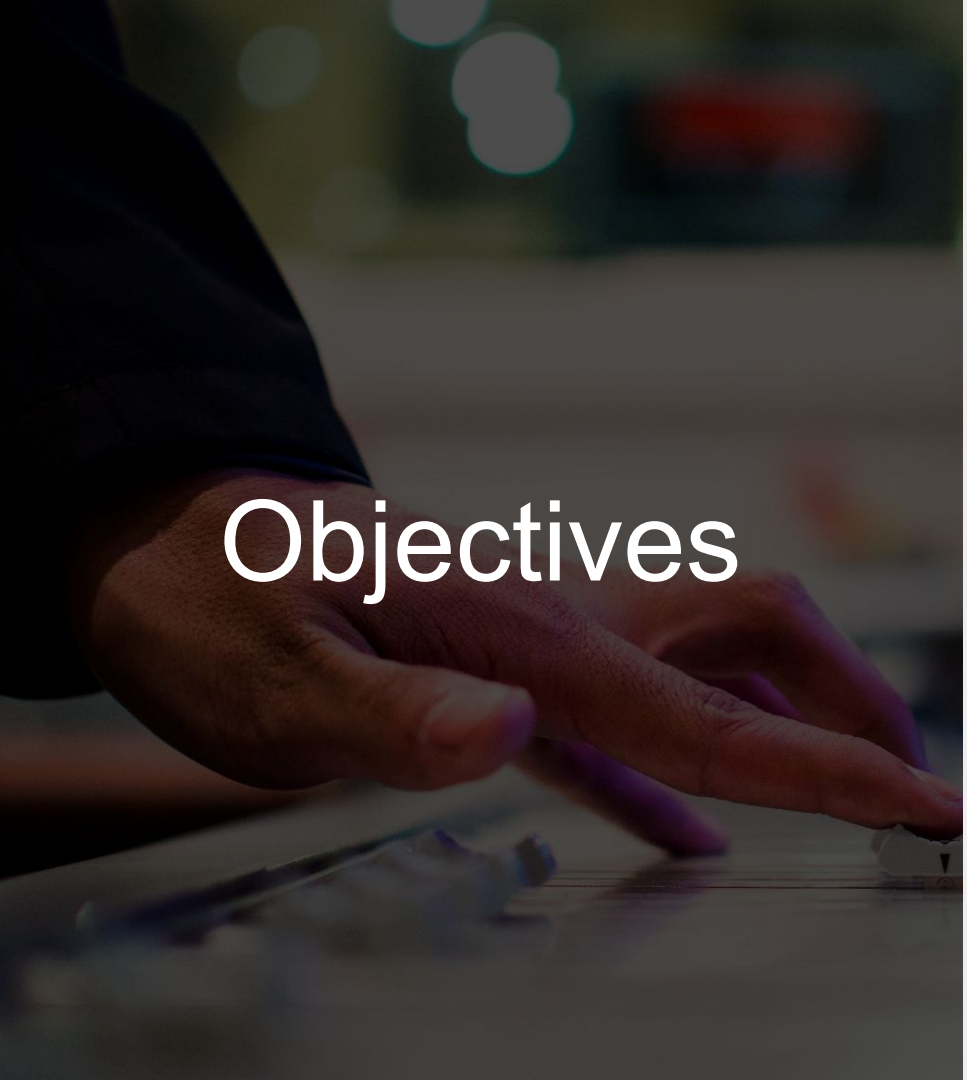
Microservices with SpringCloud

A workshop for
Learning the Microservice Architecture Style
And Implementing using Spring Technology

- M. Yauri Attamimi

What will you get from this workshop ?

- Articulate the Microservices architectural style; its advantages and disadvantages
- Brief Introduction to Spring Boot, Spring Data
- Build simple Spring-Boot applications utilizing web interfaces, REST interfaces, Spring Data, and HATEOAS
- Build Microservice applications utilizing the different Spring Cloud sub-projects, including Config Server, Eureka, Ribbon, Feign, etc
- Best Practices on SCM Branching Strategy, and CI/CD



Objectives

By the end of this workshop, you will be able to:

- Articulate the Microservices architectural style (advantages/disadvantages)
- Build simple Spring Boot applications (RESTful interfaces, Spring Data, Spring Security)
- Build microservice applications using Spring Cloud
    - Utilize centralized configuration using Spring Cloud Config and Bus
    - Utilize Service Discovery using Eureka
    - Implement Resilient Service clients with Ribbon, Feign, and Hystrix

About the Speaker

M. Yauri Attamimi

- 15 years (or so) in Software Development
- LOTS of project experience
- Java, NodeJS, Golang
- Microservices Provocateur
- AI & Blockchain Enthusiast
- TDD and Clean Code Evangelist
- VP Engineering & Co-Founder of Automate.id
- G2Lab Speaker

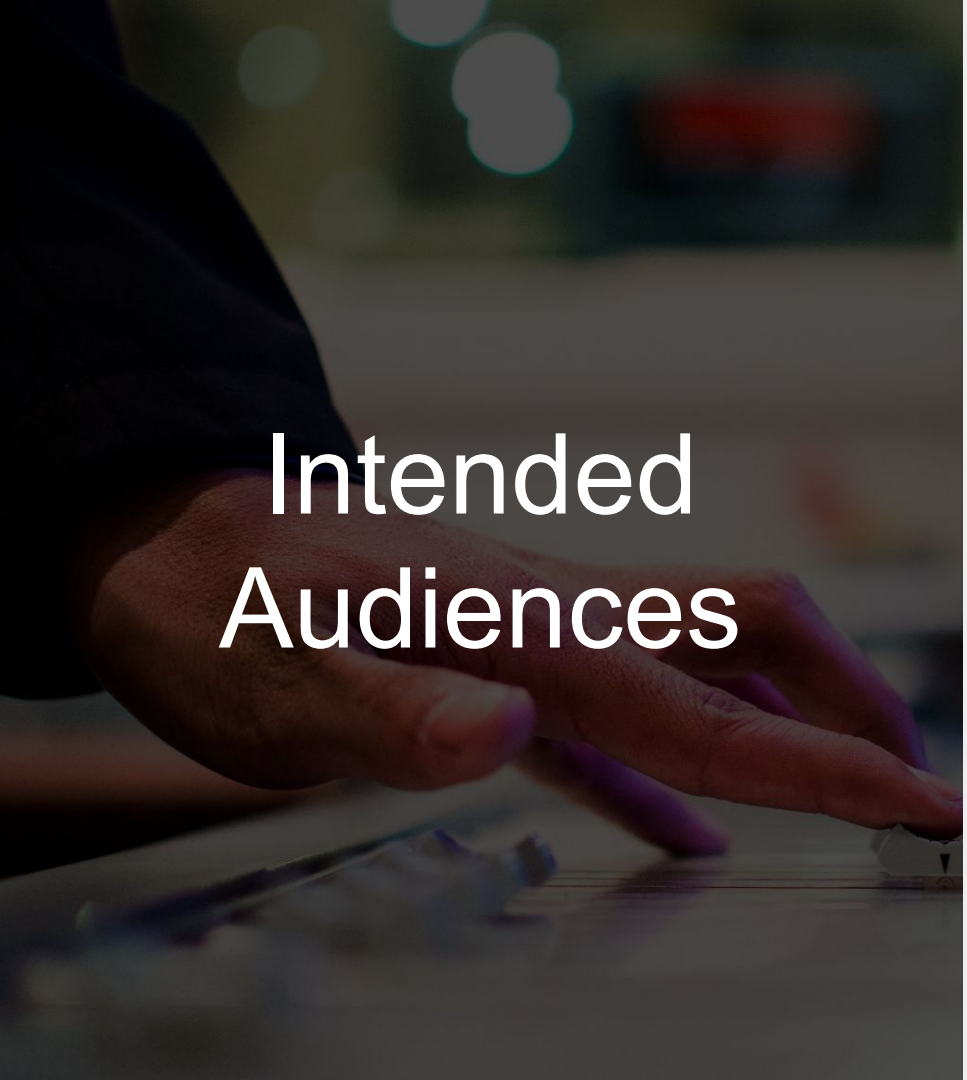
https://about.me/yauritux

My Professional Mission :

Guiding individuals and organizations to commercial success through the application of modern technologies

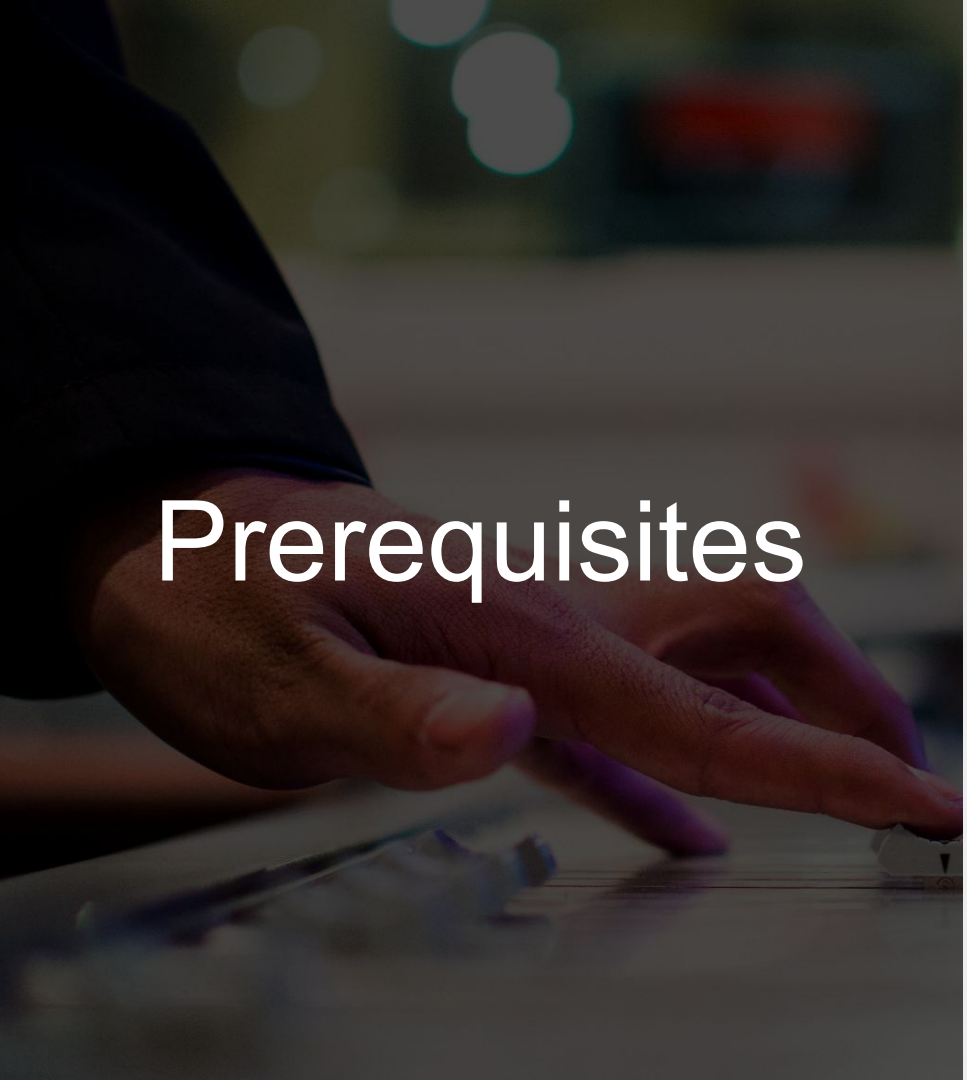
@yauritux

DZone https://dzone.com/users/366249/yauritux.html



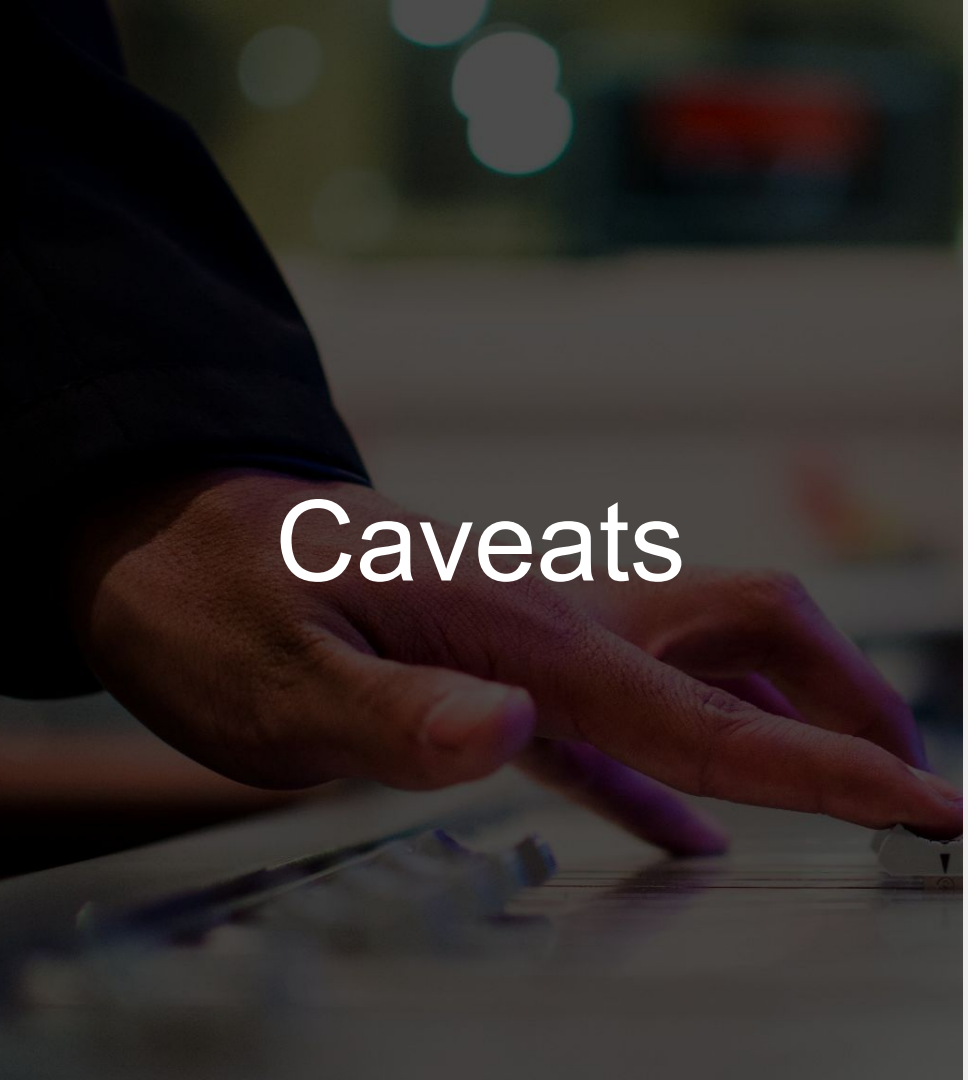
Intended Audiences

- Intermediate Java Developers
- NOT intended for beginners. However, depending on your technical experience in other technologies, and your learning style, you may find this course a fascinating deep-dive into Microservices, Spring Cloud, and Cloud-Native applications.



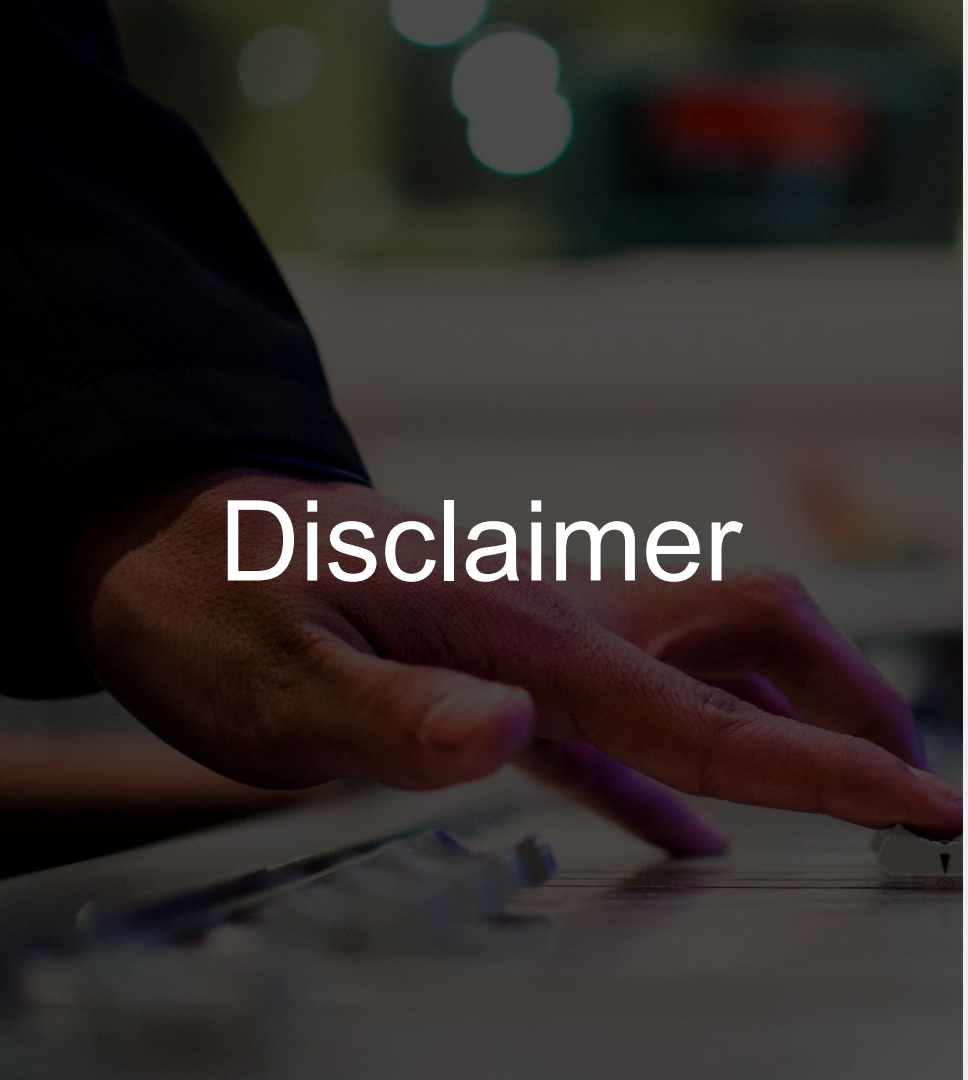
Prerequisites

- Knowledge of Java programming
- Familiarity with Spring concepts such as ApplicationContext, Profiles, RestTemplate, @Value, @Autowired, @Component, Java Configuration, etc. However, we'll be covering each of those in this course.
- Basic familiarity with Maven. How to specify dependencies, and how to do a "mvn clean package"
- Basic knowledge of Web Technologies (Web protocols, etc)
- Git - basic knowledge



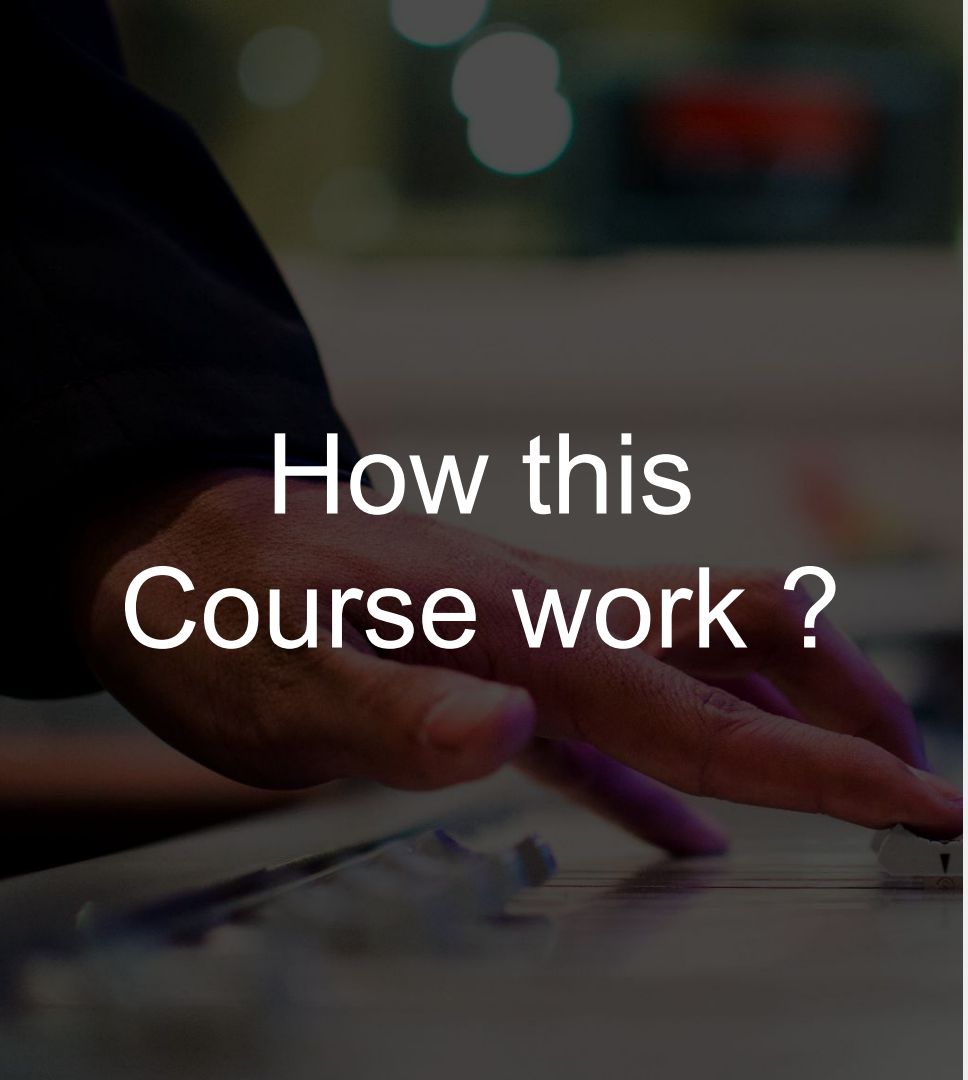
Caveats

- Java and Microservices are evolving topics and change rapidly
- Unlikely to cover all topics within short hours
  - Goal is to be reasonably comprehensive
  - Enable you to develop your own microservice applications
  - Enable you to fill in gaps yourself once you know what does it need to be a master



Disclaimer

The information provided here is designed to provide helpful information on the subjects discussed and just my own opinion based on my proven experiences (not represent any entities)



How this Course work ?

- Slide to explain concepts
- Exercises to reinforce concepts
- IDE recommended : SpringSource Tool Suite

Preparation

Please see the Lab instructions on:
https://github.com/yauritux/g2lab/blob/master/workshop/java/microservices-with-springcloud/lab-instructions/preparation.md

What are Microservices ?

Understanding the Microservice architectural style and its impact

Module Outline

- Defining Microservices
- Microservices Explanation
  - Understanding the Monolith
  - Understanding Microservices
- Practical Considerations

What are Microservices ?

- ➔ Presently a lot of hype!
- ➔ Definition from IBM:
  - An engineering approach focused on decomposing applications into single-function modules with well-defined interfaces which are independently deployed and operated by small teams who own the entire lifecycle of the service.
  - Microservices accelerate delivery by minimizing communication and coordination between people while reducing the scope and risk of change.
- ➔ Best described as:
  - An Architectural Style
  - An alternative to more traditional ‘monolithic’ applications
  - Decomposition of single system into a suite of small services, each running as independent processes and intercommunicating via open protocols with all the benefits/risks this implies.

Microservices : Definitions from the Experts

- Developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is bare minimum of centralize management of these service, which may be written in different programming languages and use different data storage technologies. (James Lewis and Martin Fowler)
- Fine-Grained SOA. (Adrian Cockcroft - Netflix)
- Small autonomous services that work together. (Sam Newman - https://samnewman.io/)

Microservices : Working Definition

- Composing a single application using a suite of small services (rather than a single, monolithic application)
- … each running as independent processes (not merely modules/components within a single executable)
- … intercommunicating via open protocols (like HTTP/REST, or messaging)
- Separately written, deployed, scaled, and maintained (potentially in different languages and/or different teams)
- Services encapsulate business capability (rather than language constructs <classes,packages> as primary way to encapsulate)
- Services are independently replaceable and upgradable

Microservices : Definitions from Me

- An architectural style which comprises of **small size** independent **web services** running in an isolated environment (*loosely coupled*) and designed to communicate with each other by using a standard data exchange format.
- The **size** is determined by something called **bounded-context**.

Basic definition of Web Services : *Services delivered over the Web.*
Web Service definition from W3C :
Software system designed to support interoperable machine-to-machine interaction over a network.

The key thing to understand here is:

- Web services are designed for machine-to-machine or application-to-application interaction
- Web services should be interoperable, not platform dependent
- Web services should allow communication over a network

Microservices are NOT:

- ➔ The same as SOA
  - SOA is about integrating various enterprise applications. Microservices are mainly about decomposing single applications.
  - SOA relies on orchestration, microservices rely on choreography
  - SOA relies on smart integration technology (dumb services), microservices rely on smart services (dumb integration technology)
- ➔ A silver bullet
  - The microservices approach involves drawbacks and risks
- ➔ New! You may be using microservices now and not know it !

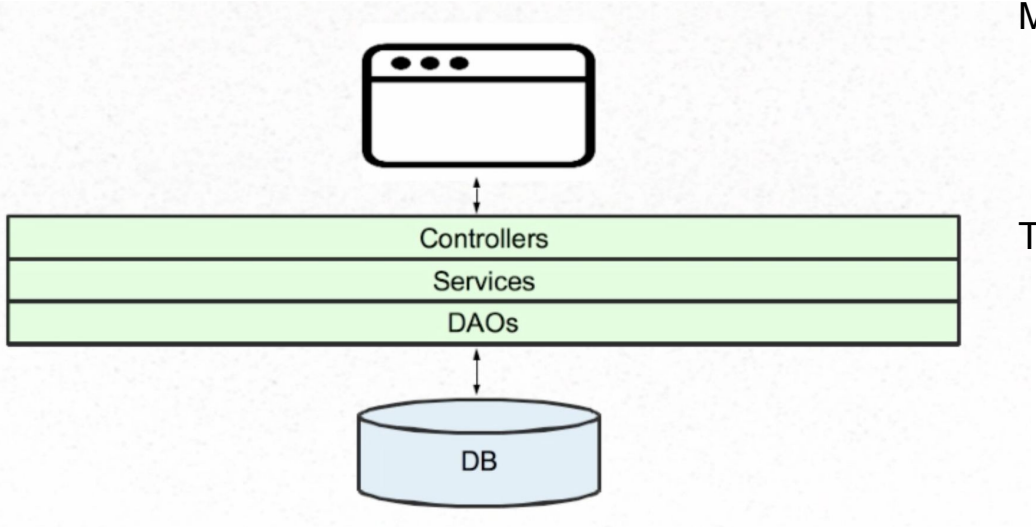
Some Microservices Trend

- Twitter moved from Ruby/Rails monolith to microservices
- Facebook moved from PHP monolith to microservices
- Netflix moved from Java monolith to microservices

Microservices Example

- Consider a monolithic shopping cart application:
  - Web / Mobile interfaces
  - Functions for :
    - Searching for products
    - Product catalog
    - Inventory management
    - Shopping cart
    - Checkout
    - Fulfillment
- How would this look with microservices ?

Shopping Cart App with Monolithic



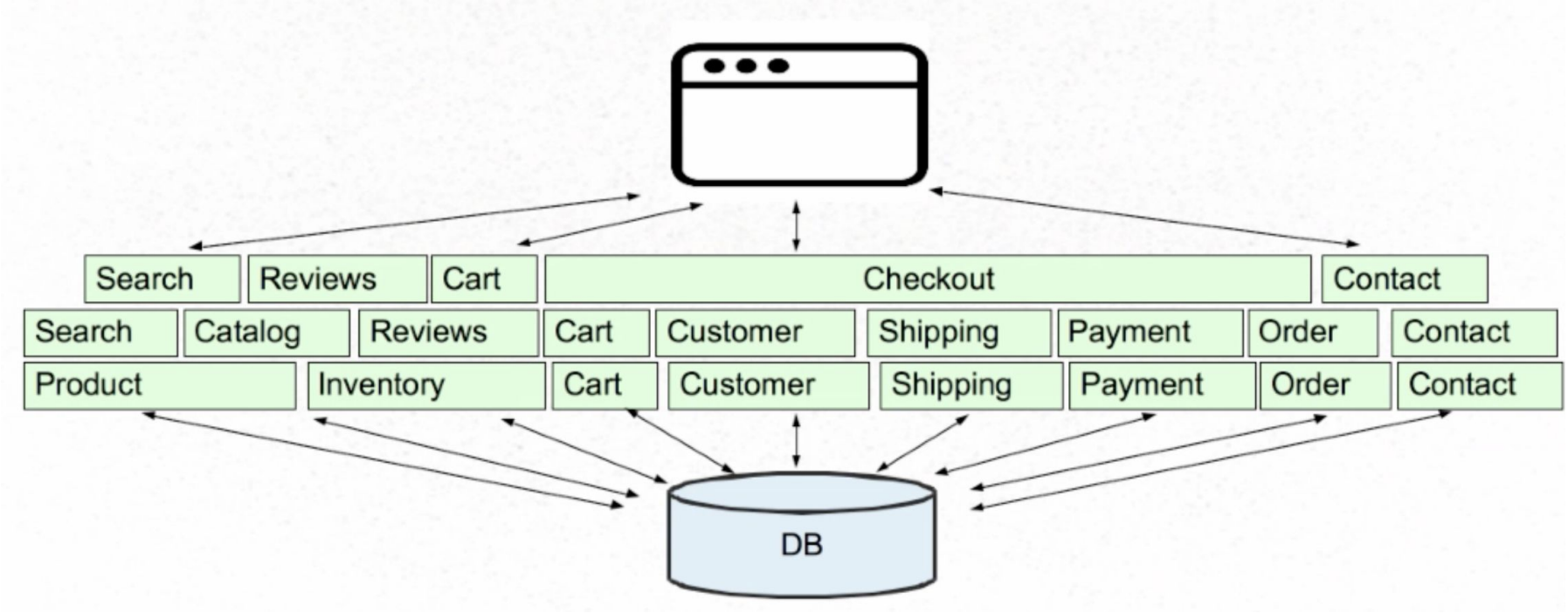
Monolithic apps can be characterized by :

- Large Application Size
- Long Release Cycles
- Large Teams

Typical challenge includes :

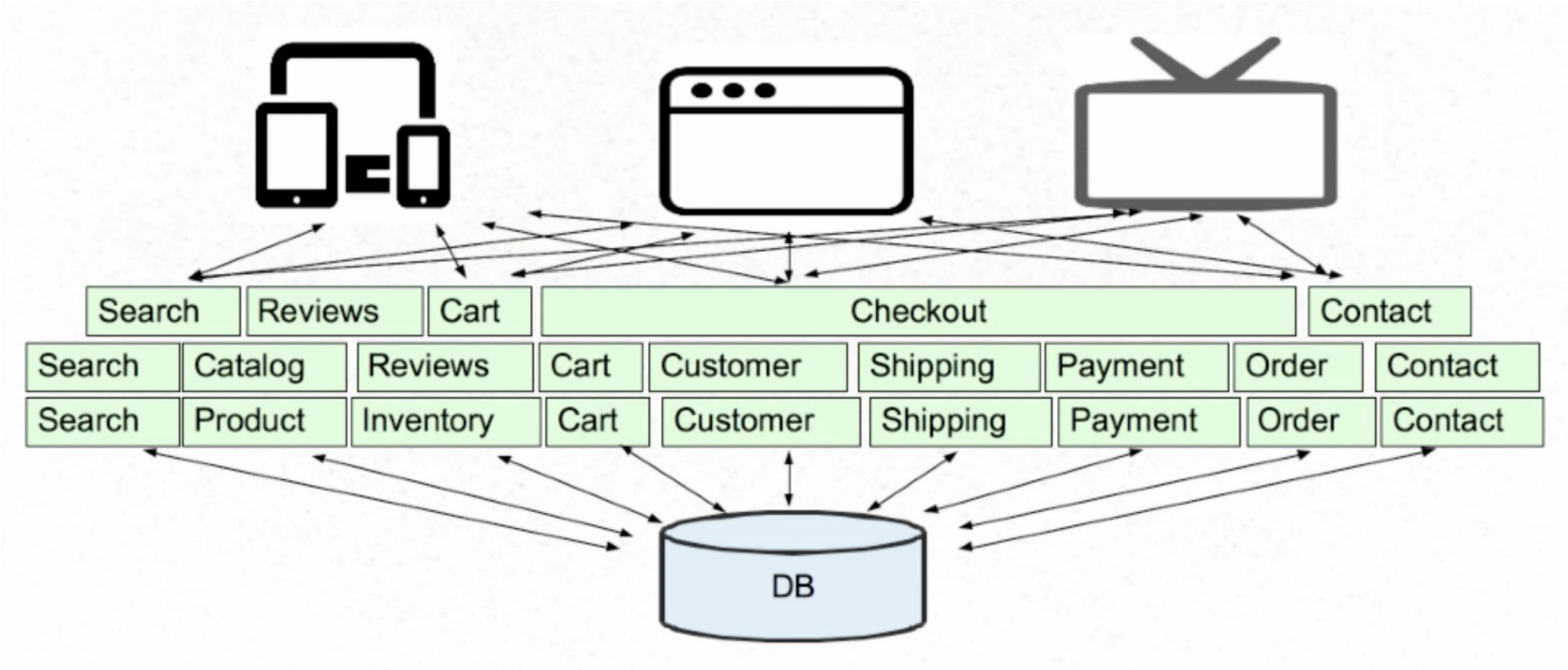
- Scalability Challenges
- New Technology Adoption
- New Processes - Agile ?
- Difficult to Automation Test
- Difficult to Adapt to Modern Development Practices
- Adapting to Device Explosion

Shopping Cart App with Monolithic



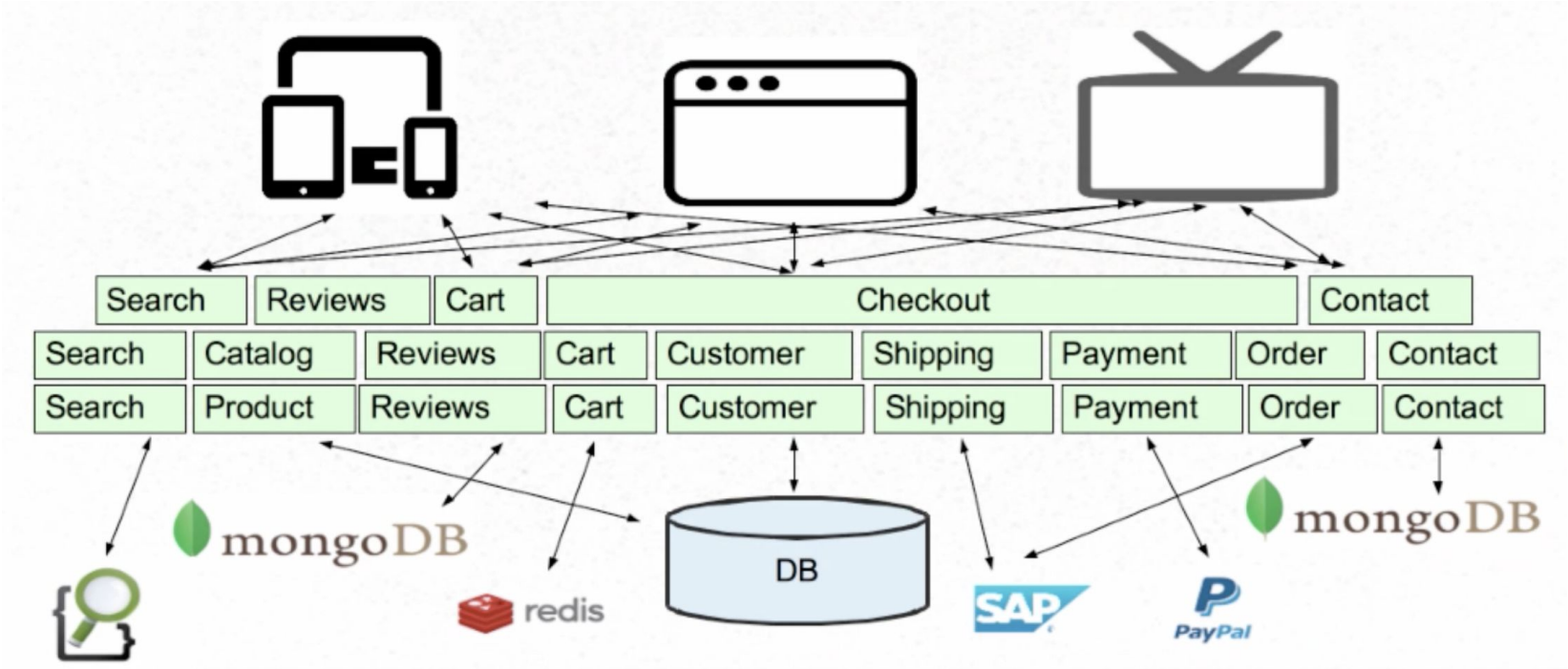
Monolithic Challenges:

New types of client applications



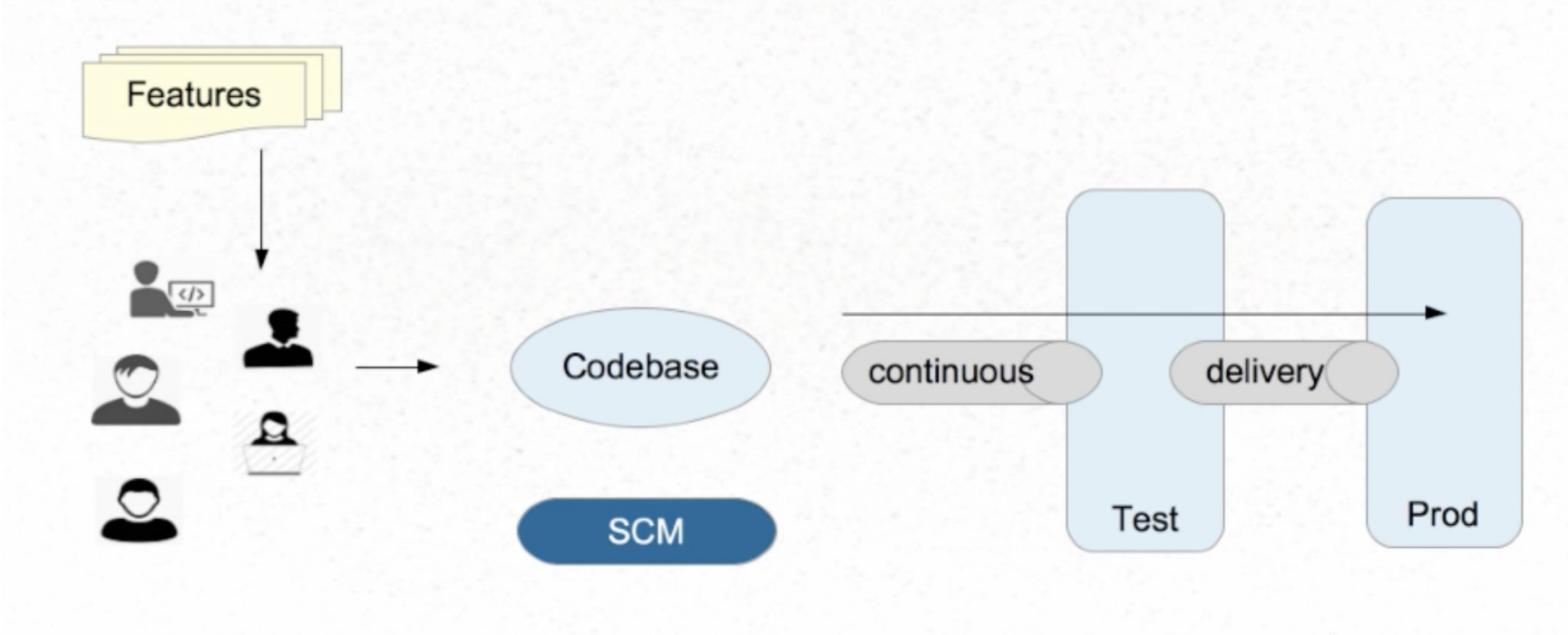
Monolithic Challenges:

New types of Persistence/Services



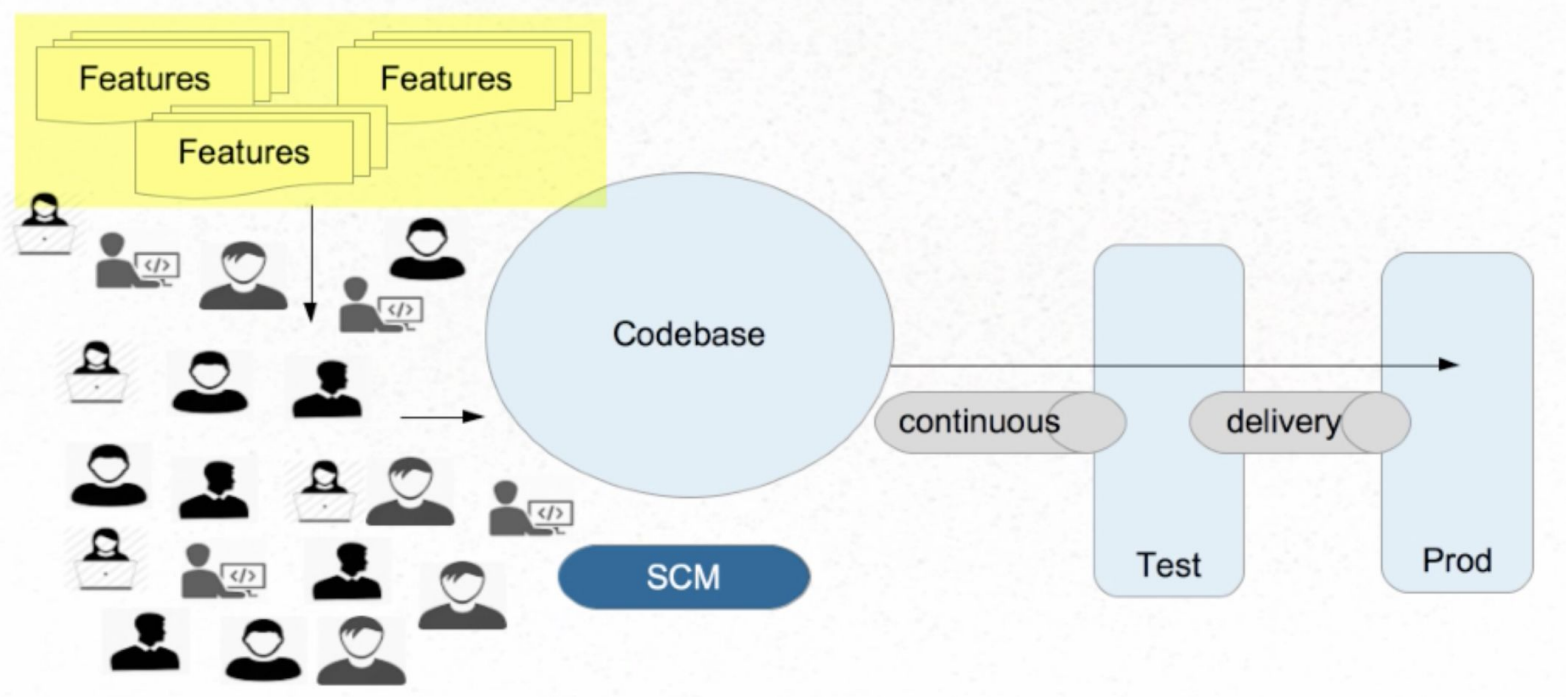
Monolithic Challenges:

Single Codebase, Deployment, Versioning, Team Size



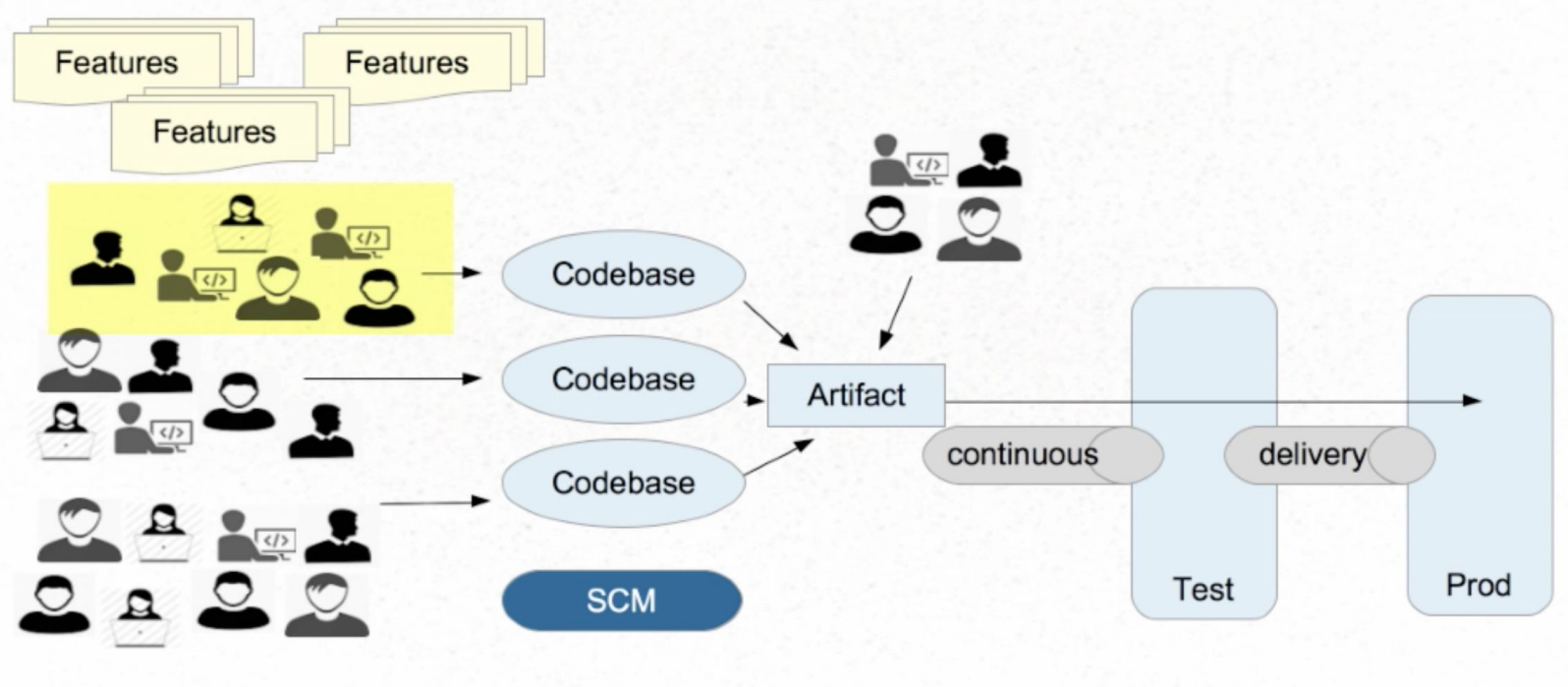
Monolithic Challenges:

Single Codebase, Deployment, Versioning, Team Size



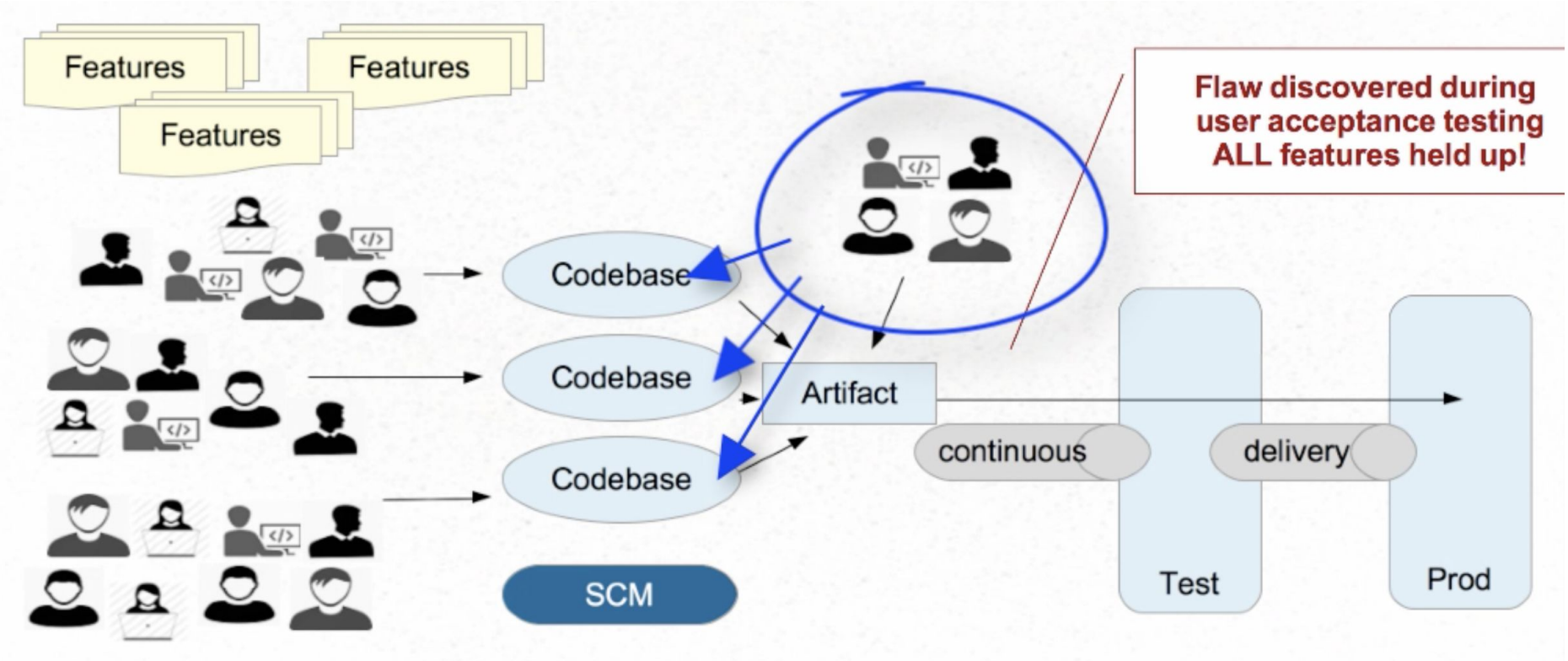
Monolithic Challenges:

Using Teams / Language Constructs



Monolithic Challenges:

Using Teams / Language Constructs



Understanding the Monolithic Implementation

- ➔ Single application executable
  - ◆ Easy to comprehend, but not to digest
  - ◆ Must be written in a single language
- ➔ Modularity based on program language
  - ◆ Using the constructs available in that language (packages, classes, functions, namespaces, frameworks)
  - ◆ Various storage / service technologies used
    - RDBMS, Messaging, Email, etc

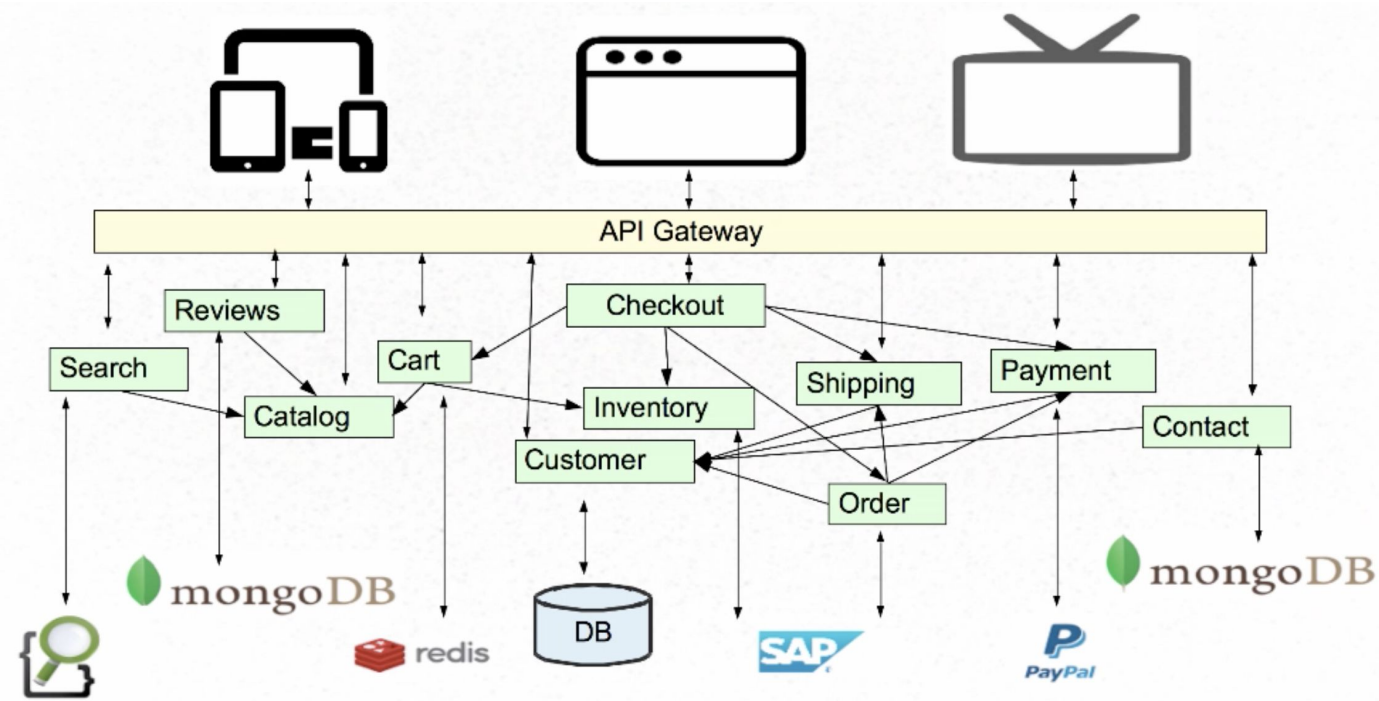
Monolithic Advantages

- Easy to comprehend (but not digest)
- Easy to test as a single unit (up to a size limit)
- Easy to deploy as a single unit
- Easy to manage (up to a size limit)
- Easy to manage changes (up to a point)
- Easy to scale (when care is taken)
- Complexity manage by language constructs

Monolithic Drawbacks

- ➔ Language / Framework lock
  - ◆ Entire app written with single technology stack. Cannot experiment / take advantage of emerging technologies
- ➔ Digestion
  - ◆ Single developer cannot digest a large codebase
  - ◆ Single team cannot manage a single large application
    - Amazon’s 2 pizza rule
- ➔ Deployment as single unit
  - ◆ Cannot independently deploy single change to single component
  - ◆ Changes are “held-hostage” by other changes

Enter Microservices Architecture



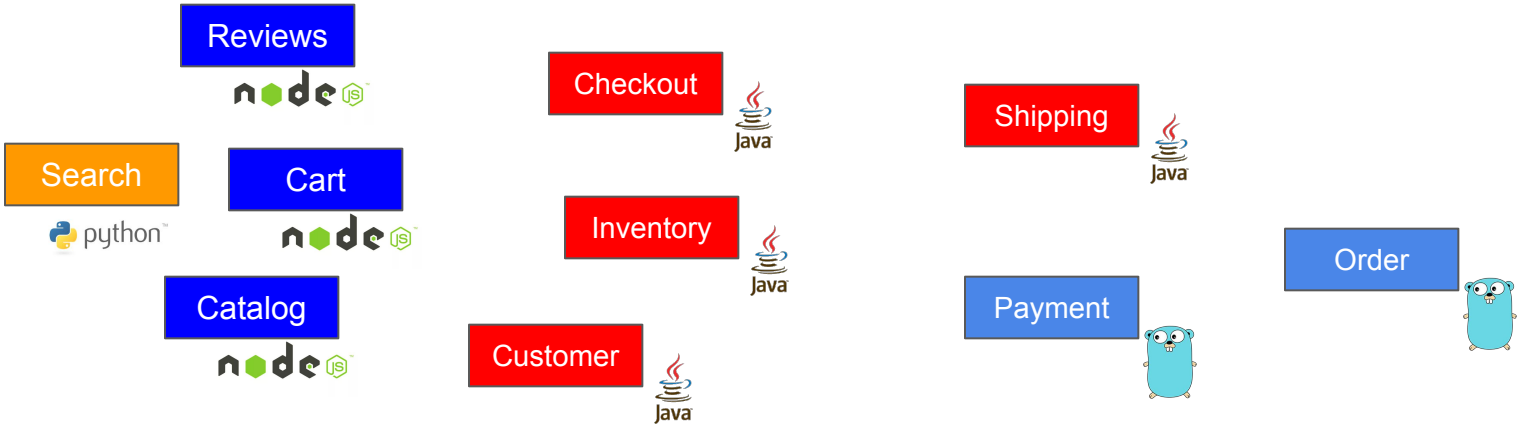
Componentization via Services

- NOT language constructs
- Where services are small, independently deployable applications
  - NOT a single codebase
  - NOT (necessarily) a single language / framework
  - Modularization not based on language / framework constructs
- Forces the design of clear interfaces
- Changes scoped to their affected service

Microservices:

Composed using suite of small services

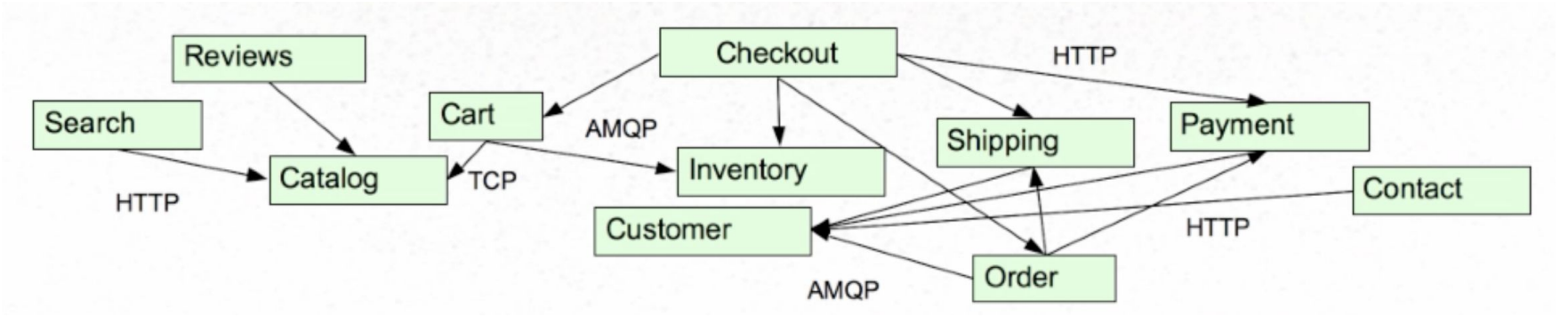
- Services are small, independently deployable applications
  - Not a single codebase
  - Not (necessarily) a single language / framework
  - Modularization not based on language / framework constructs



Microservices

Communication based on lightweights protocols

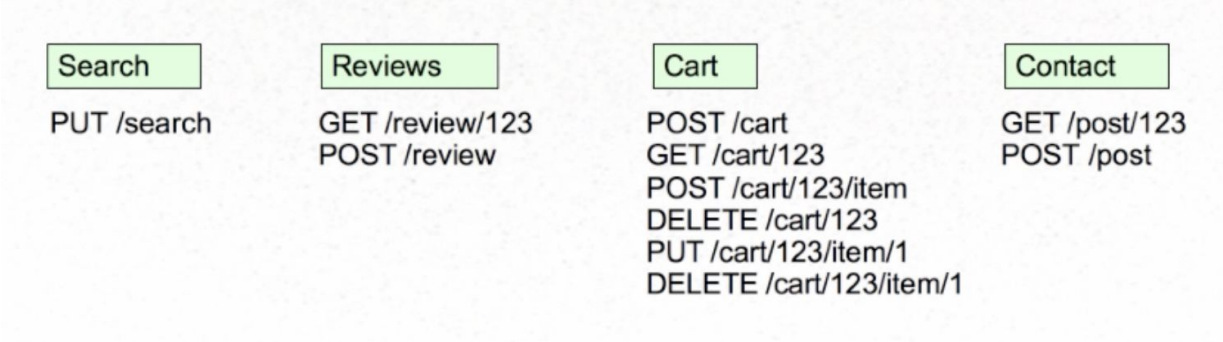
- HTTP, TCP, UDP, Messaging, etc
  - Payloads : JSON, BSON, XML, Protocol Buffers, etc
- Forces the design of clear interfaces
- Netflix’s cloud native architecture - communicate via APIs (not common database)



Microservices

Services encapsulate business capabilities

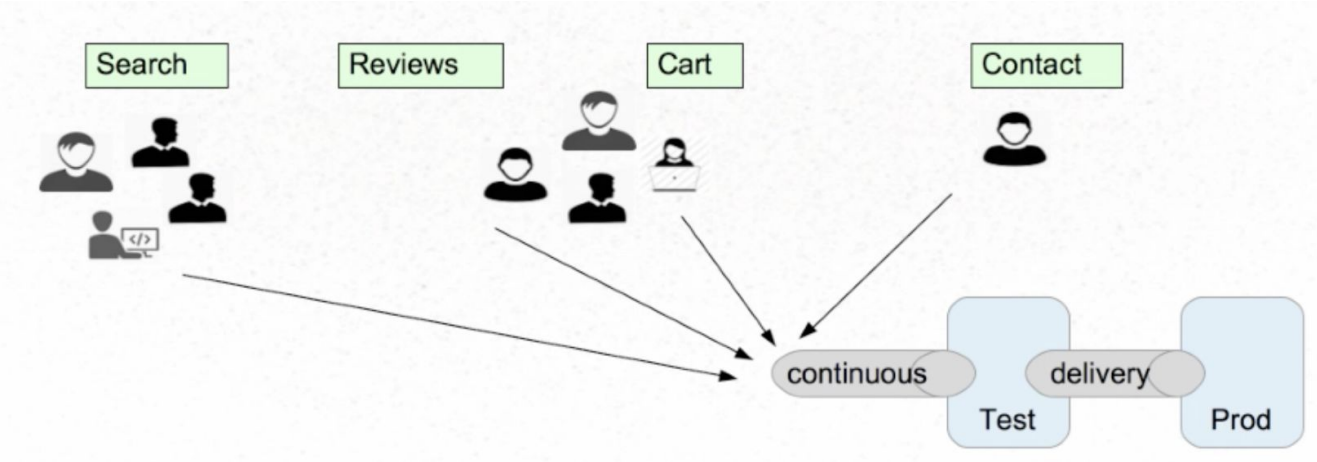
- NOT based on technology stacks
- Vertical slices by business function (e.g. cart, catalog, checkout)
- … Though technology chunk also practical (email service)
- Suitable for cross-functional teams



Microservices

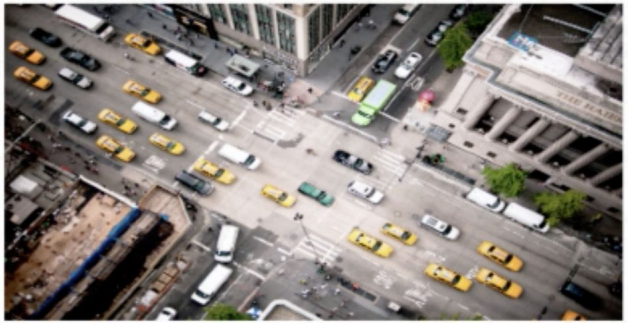
Services easily managed

- Easy to comprehend, alter, test, version, deploy, manage, overhaul, replace
  - By small, cross functional teams (or even individuals)



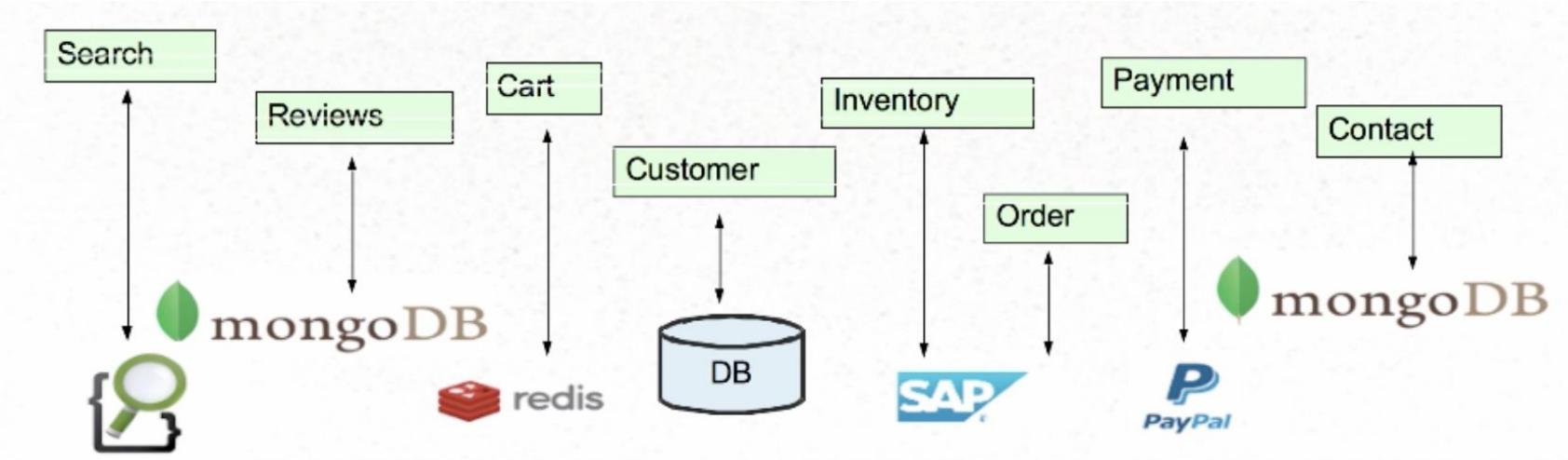
Decentralized Governance

- Use the right tool (language, framework) for the job
- Services evolve at different speeds, deployed and managed according to different needs
- Make services be “Tolerant Readers / Tolerant Consumers” (i.e. relatively resilient to changes made to the services)
- Consumer-Driven Contracts
- Services are not orchestrated, but choreographed



Polyglot Persistence

- Freedom to use the best technology for the job
  - Don’t assume single RDBMS is always best
  - Very controversial ! Many DBAs will not like this :-).
    - No pan-enterprise data model !
    - No transactions ! (nevertheless, we wouldn’t need transactions for everything)



Microservice Advantages

- Easy to digest each service (difficult to comprehend whole)
- VERY easy to test, deploy, manage, version, and scale single services
- Change cycle decoupled; fastest release cycle
- Easier to scale staff
- No language / framework lock; new technology adoption becomes easier

Challenges with Microservices

- Complexity has moved out of the application, but into the operations layer
  - Fallacies of distributed computing
- Services may be unavailable
  - Never needed to worry about this in a monolith
  - Design for failure, circuit breakers
  - Much more monitoring needed
- Remote calls more expensive than in-process calls
- Automation: because there are numbers of smaller components instead of a monolith, you need to automate everything - Builds, Deployment, Monitoring, etc.
- Bounded Context : deciding the boundaries of a microservice is not an easy task. Bounded Contexts from Domain Driven Design (DDD) is a good starting point. Your understanding of the domain evolves over a period of time. Therefore, you need to ensure the microservices boundaries also evolve.
- Configuration Management : Need to maintain configurations for hundreds of components across environments. You would need a configuration management solution.

Challenges with Microservices - (continued)

- Dynamic Scale Up and Scale Down : The advantages of microservices will only be realized once your applications can scaled up and scaled down easily in the cloud.
- Pack of Cards : if a microservices at the bottom of the call chain failed, it can have knock on effects on all other microservices. Microservices should be fault tolerant by design.
- Debugging : when there is a problem that needs investigation, you might need to look into multiple services across different components. Centralized logging and Dashboards are essential to make it easy to debug problems.
- Consistency : you cannot have wide range of tools solving the same problem. While it is important to foster innovation, it is also important to have some decentralized governance around the languages, platforms, technology and tools used for implementing / deploying / monitoring microservices.

Challenges with Microservices (continued)

- Transactions: must rely on eventual consistency over ACID
- Features span multiple services
- Change management becomes a different challenge
  - Need to consider the interaction of services
  - Dependency management / versions
- Refactoring module boundaries

Fallacies of Distributed Computing

- The Network is reliable
- Latency is Zero
- Bandwith is infinite
- The Network is secure
- Topology doesn’t change
- There is one Administrator
- Transport cost is zero
- The network is homogeneus

Microservices Practical Considerations

How do you break a Monolith into Microservices ?

- Primary Considerations : Business Functionality
  - Noun-based (catalog, cart, customer)
  - Verb-based (search, checkout, shipping)
  - Single-Responsibility Principle
  - Bounded-Context Principle from Domain-Driven Design (DDD)

How Micro is Micro ?

- Size is not the compelling factor
  - Small enough for an individual developer to digest
  - Small enough to be built and managed by small team
    - Amazon’s two pizzas rule
  - Documentation small enough to read and understand
  - Dozens of secrets, not hundreds
  - Predictable, easy to experiment with

Differences with SOA

- SOA addresses integration between systems.
  - Microservices address individual applications
- SOA relies on Orchestration.
  - Microservices rely on Choreography
- SOA relies on smart integration technology, dumb services
  - Microservices rely on smart services, dumb integration technology
  - Consider: Linux commands, pipes and filters

Discussion

Which of the pros / cons of microservices are most applicable in your situation ?

What are some potential services in your application ?

Spring Boot

Spring Boot is all about:

Getting Java / Spring Applications Up and Running Quickly

Module Outline

- Spring Boot
- Web Applications with Boot
  - Thymeleaf, JSP
- Spring Data JPA
- Spring Data REST

What is Spring Boot ?

- Radically faster getting started experience
- “Opinionated” approach to configuration / defaults (i.e. Convention over Configuration)
  - Intelligent defaults
  - Gets out of the way quickly
- What does it involve ?
  - Easier dependency management (it embraces tool like **maven** and **gradle**)
  - Automatic configuration / reasonable defaults : Spring will check what framework exists in the classpath and provide the basic/default needed configuration (e.g. : if spring-mvc exists within the classpath, then spring will provide default configuration for dispatcher servlet, view resolver, etc).
  - Different build / deployment options

Detail of Example Auto Configuration

We'll take `DataSourceAutoConfiguration` as example.
Typically all `AutoConfiguration` classes look at other classes available in the classpath. If specific classes are available in the classpath, then configuration for that functionality is enabled through auto configuration. Annotations like `@ConditionalOnClass`, `@ConditionalOnMissingBean` help in providing these features.

`@ConditionalOnClass({ DataSource.class, EmbeddedDatabaseType.class })` : this configuration is enabled only when these classes are available in the classpath.

```
@Configuration
@ConditionalOnClass({ DataSource.class, EmbeddedDatabaseType.class })
@EnableConfigurationProperties(DataSourceProperties.class)
@Import({ Registrar.class,
DataSourcePoolMetadataProvidersConfiguration.class })
public class DataSourceAutoConfiguration {
```

Detail of Example Auto Configuration - Continued

Embedded Database is configured only if there are no beans of type `DataSource.class` or `XADataSource.class` already configured.

```
@Conditional(EmbeddedDatabaseCondition.class)
@ConditionalOnMissingBean({ DataSource.class, XADataSource.class })
@Import(EmbeddedDataSourceConfiguration.class)
protected static class EmbeddedDatabaseConfiguration {
    …
}
```

What Spring Boot is not ?

- Plugins for IDEs
  - Use Boot with any IDE (or none at all)
- Code Generation

Demonstration - Spring Boot

Creating a new, bare-bones Spring application

- Create with Spring Tool Suite, or
- Start from Spring Initializr (i.e. http://start.spring.io/)

Spring Boot - What Just Happened ?

- Boilerplate project structure is created
  - Mostly folder structure
  - “Application” class + test
  - Maven POM (or Gradle if desired)
- Dependency management

Demonstration

Running a Spring Boot Project

Running the newly created project

Running Spring Boot - What Just Happened ?

- SpringApplication
  - Created Spring Application Context
- @SpringBootApplication
  - Combination of @Configuration
    - Marks a configuration file
    - Java equivalent of XML <beans> file
  - … and @ComponentScan
    - Looks for @Components (none at the moment)
  - … and @EnableAutoConfiguration
    - Master runtime switch for Spring Boot
    - Examines ApplicationContext and classpath
    - Creates missing beans based on intelligent defaults

Tips : Googling for “Spring Framework Reference Documentation”; do search for “Java-based Configuration”

Demonstration

Adding Web Capability

- Adding `spring-boot-starter-web` dependency
- Adding GreetController

Adding Web - What Just Happened ?

- **spring-boot-starter-web** dependency
  - Adds **spring-web**, **spring-mvc** jars
  - Adds embedded Tomcat jars
- When application starts …
  - Your beans are created
  - @EnableAutoConfiguration looks for “missing” beans
    - Based on your beans + classpath
    - Notices @Controller / Spring MVC jars
  - Automatically creates MVC beans
    - Dispatcher Servlet, HandlerMappings, Adapters, ViewResolvers
  - Launches embedded Tomcat instance

But Wait, I want a WAR ...

- To convert from JAR to WAR:
  - Change POM Packaging
  - Extend SpringBootServletInitializer

```
public class Application extends SpringBootServletInitializer {

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

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
        return application.sources(Config.class);
    }

}
```

  - Deploy to Web / App Server (e.g. Tomcat, Jetty, JBoss, etc)
    - URL becomes http://localhost:8080/<app>/

Demonstration

WAR Deployment

- WAR Packaging
- SpringBootServletInitializer

What about Web Pages ?

- SpringMVC supports a wide range of view options
- Easy to use JSP, Freemarker, Velocity,Thymeleaf
- Boot automatically establishes defaults
  - InternalResourceViewResolver for JSPs
  - ThymeleafViewResolver
    - If thymeleaf is in the classpath
- Spring-boot-starter-thymeleaf

Tips:

Googling for “Spring Boot Reference Guide”; and do a search for **Starter POM** from within.

Demonstration

Thymeleaf Web Pages

- spring-boot-starter-thymeleaf
- /templates folder
- Controller adjustments
- Web page

Thymeleaf Web Pages - What Just Happened ?

- spring-boot-starter-thymeleaf
  - Brought in required jars
  - Automatically configured ThymeleafViewResolver
- Controller returned a “logical view name”
- ViewResolver found a matching template
- Render

Tips :

furthermore.. Googling for “Spring Framework Reference”; and do a search about “**Web MVC framework**”

But Wait, I want JSPs ...

- We’ve already had lots of existing JSPs
- No Problem !
- Just as easy to use JSPs !
  - Place JSPs in desired web-relative location
  - Set spring.mvc.view.prefix / spring.mvc.view.suffix as needed
  - Remove thymeleaf starter pom

Demonstration
JSP Web Pages

- Place JSP in desired folder
- Set spring.mvc.view.prefix / spring.mvc.view.suffix
  (see Appendix A - Common Application Properties on Spring Boot Reference Guide)
- Remove spring-boot-starter-thymeleaf (if any)
- Exclude spring-boot-starter-tomcat

JSP Web Pages - What Just Happened ?

- No ThymeleafViewResolver configured
- Controller returned a “logical view name”
- InternalResourceViewResolver forwarded to JSP
- Render

Spring & REST
JSP Web Pages

- REST capability is built-in to Spring MVC
  - Simply use domain objects as parameters / return values
  - Mark with @RequestBody / @ResponseBody
  - Spring MVC automatically handles XML / JSON conversion
    - Based on converters available in classpath

Demonstration

REST Controllers in Spring MVC

- Additional domain objects
- Automatic HTTP Message Conversion

REST Controllers - What Just Happened ?

- Controller returned a domain object
  - Not a logical view name (page)
- Spring MVC noticed @ResponseBody
  - Or @RestController
- Invoked correct HttpMessageConverter
  - Based on
    - Requested format
    - JARs on classpath

What if I want XML ?

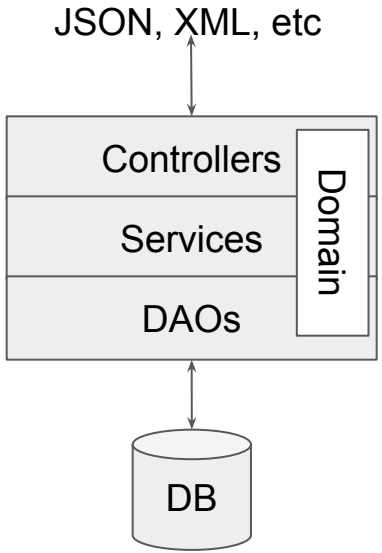
- Annotate domain classes with JAXB annotations
  - JAXB already part of Java SE
- When App starts
  - Spring creates HttpMessageConverter for JAXB
    - Based on classpath contents
- XML or JSON returned
  - Based on requested format

Adding JPA Capability

- Adding the spring-boot-starter-data-jpa dependency
  - Adds Spring JDBC / Transaction Management
  - Adds Spring ORM
  - Adds Hibernate / Entity Manager
  - Adds Spring Data JPA subproject
- Does NOT add a Database Driver
  - Add one manually (e.g. HSQL)

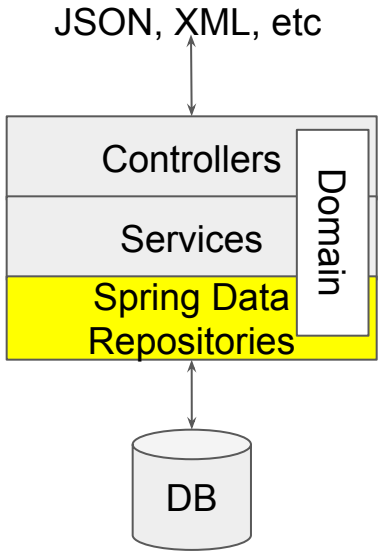
Spring Data JPA

- Typical Web Application Architecture
- REST controllers provide CRUD interface to clients
- DAO provide CRUD interface to database



Spring Data - Instant Repositories

- Spring Data provides Dynamic Repositories
- We provide the interface, Spring Data dynamically implements
  - JPA, MongoDB, Redis, GemFire, etc (see: https://projects.spring.io/spring-data/)
- Service Layers or Controllers have almost no logic



Demonstration

Adding Spring Data JPA

- spring-boot-starter-data-jpa
- org.postgresql / postgresql
- Annotate domain objects with JPA
- Extends CrudRepository
- Add postgresql connection info on application.properties
  - spring.datasource.url=jdbc:postgrelsql://localhost:5432/<db>
  - spring.datasource.username=<username>
  - spring.datasource.password=<password>
  - spring.jpa.hibernate.ddl-auto=create|create-drop|update|validate

Tips:

adds **spring.jpa.properties.hibernate.jdbc.lob.non_contextual_creation=true**

If maven complaints :

org.postgresql.jdbc.PgConnection.createClob() is not yet implemented as suggested on :

https://github.com/pgjdbc/pgjdbc/issues/1102

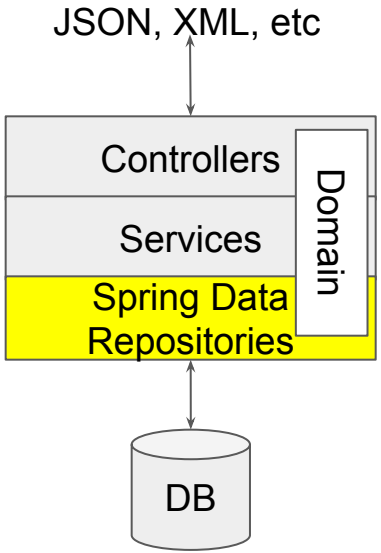
https://vkuzel.com/spring-boot-jpa-hibernate-atomikos-postgresql-exception

Adding Spring Data JPA - What Just Happened ?

- What we did:
  - Added dependencies for spring-boot-starter-data-jpa and postgresql
  - Annotated domain objects with plain JPA annotations
  - Added an interface for Spring Data JPA
  - Dependency injected into controller
- When application starts:
  - Spring Data dynamically implements repositories
    - find*(), delete(), save() methods implemented
  - DataSource, Transaction Management, all handled

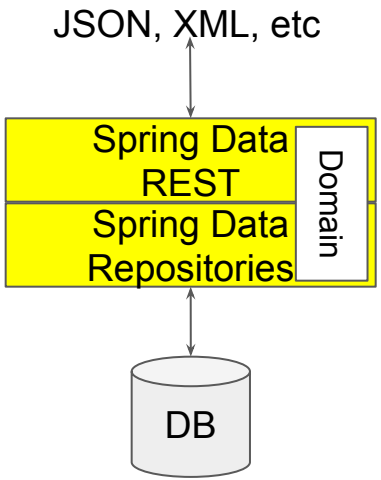
Spring Data REST

- Often, applications simply expose DAO methods as REST resources
- Spring Data REST handles this automatically



Adding Spring Data REST

- Plugs into dynamic repositories
- Generates RESTful interface
  - GET, PUT, POST, DELETE
- Code needed only to override defaults



Demonstration

Adding Spring Data REST

- Spring-boot-starter-data-rest
- Annotates Repository interface with @RestResource annotation
- Specify **path** and **rel** attributes on @RestResource annotation

Adding Spring Data REST - What Just Happened ?

- When application starts …
  - @RestResource annotations interpreted
  - @Controllers beans created
  - @RequestMappings created

Adding HATEOAS

- Spring Data REST simply returns RESTful resources
  - Conversion handled by Jackson, or JAXB
- Underlying data relationships used to build links
  - If matching repositories exist
- Consider the Team → Player relationship
- Player Repository needed to force link creation

Demonstration

Adding HATEOAS

- Creating a Player DAO

Adding HATEOAS - What Just Happened ?

- Spring Data REST noticed two repositories
  - The relationship between entities is known via JPA annotations
- Spring automatically represents the children as links
  - @RestResource determines names of links

Summary

- Spring Boot makes it easy to start (bootstrap) projects
  - And easy to add feature sets to projects
  - Opinionated approach
  - RUN as JAR or WAR
  - Web Applications (JSP, Thymeleaf, etc)
- REST
  - Automatic Resource Conversion
- Spring Data JPA
  - Automatic Repository Implementation
- Spring Data REST
  - Automatic REST Controllers

Spring Cloud

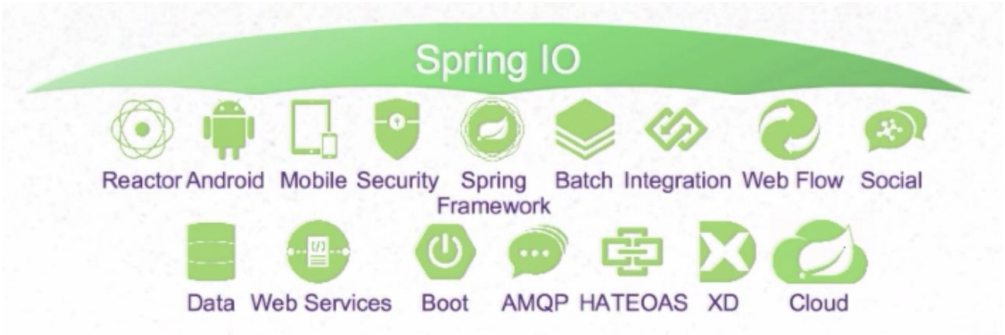
Addressing the issues found
in cloud-native applications

Module Outline

- Spring / Spring IO / Spring Cloud
- Spring Cloud Netflix
- Common Concepts

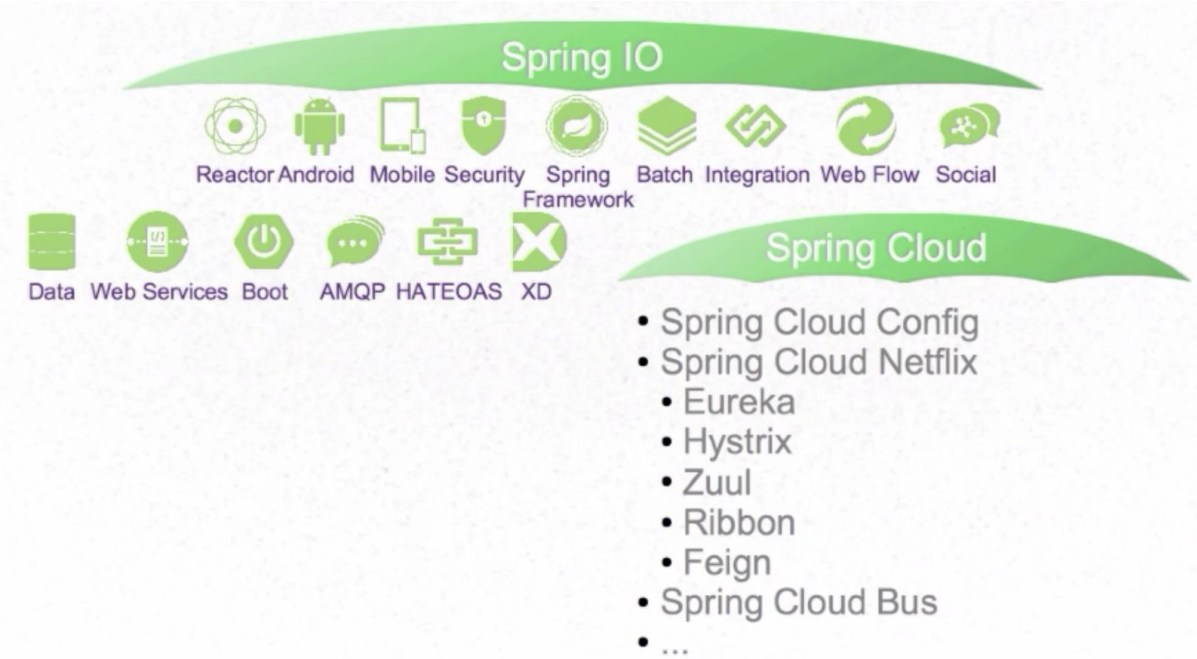
Spring Cloud Origins

- First, there was the Spring Framework (2004)
  - Alternative to low-level JEE approaches
- Next, Spring subprojects emerged (2006 - present)
  - Spring Security, Web Flow, Integration, Batch, Web Services, XD, Social, Data, Boot, Session, etc
  - Organized under Spring IO umbrella:



Spring Cloud Subproject

- “Sub-Umbrella” Project within Spring IO Platform



Goal of Spring Cloud

- Provide libraries to apply common patterns needed in distributed applications
  - Distributed / Versioned / Centralized Configuration Management
  - Service Registration and Discovery
  - Load Balancing
  - Service-to-Service Calls
  - Circuit Breakers
  - Routing
  - …

http://projects.spring.io/spring-cloud/docs/1.0.1/spring-cloud.html

Spring Cloud Important Modules

Dynamic Scale Up and Down, using the combination of :

- Naming Server (Eureka)
- Ribbon (Client-Side Load Balancing)
- Feign (Easier REST Clients)

Visibility and Monitoring with :

- Zipkin Distributed Tracing
- Netflix API Gateway

Configuration Management with :

- Spring Cloud Config Server

Fault Tolerance with :

- Hystrix

Where does NETFLIX Fit into All of This ?

- NETFLIX reinvented itself since early 2007
  - Moved from DVD mailing to video-on-demand
    - Once USPS largest first-class customer
    - Now biggest source of North American internet traffic in evenings.
- Became Trailblazers (pioneers) in Cloud Computing
  - All running on Amazon Web Services
- Choose to Publish many general-use technologies as Open Source projects
  - Proprietary video-streaming technologies are still secret

Spring and NETFLIX

- The Spring Team has always been forward looking
  - Trying to Focus on Applications of Tomorrow
- Netflix OSS Mature and Battle-Tested; Why Reinvent ?
- Netflix OSS Not Necessarily Easy and Convenient
  - Spring Cloud provides easy interaction
    - Dependencies
    - Annotations

Spring Cloud Setup

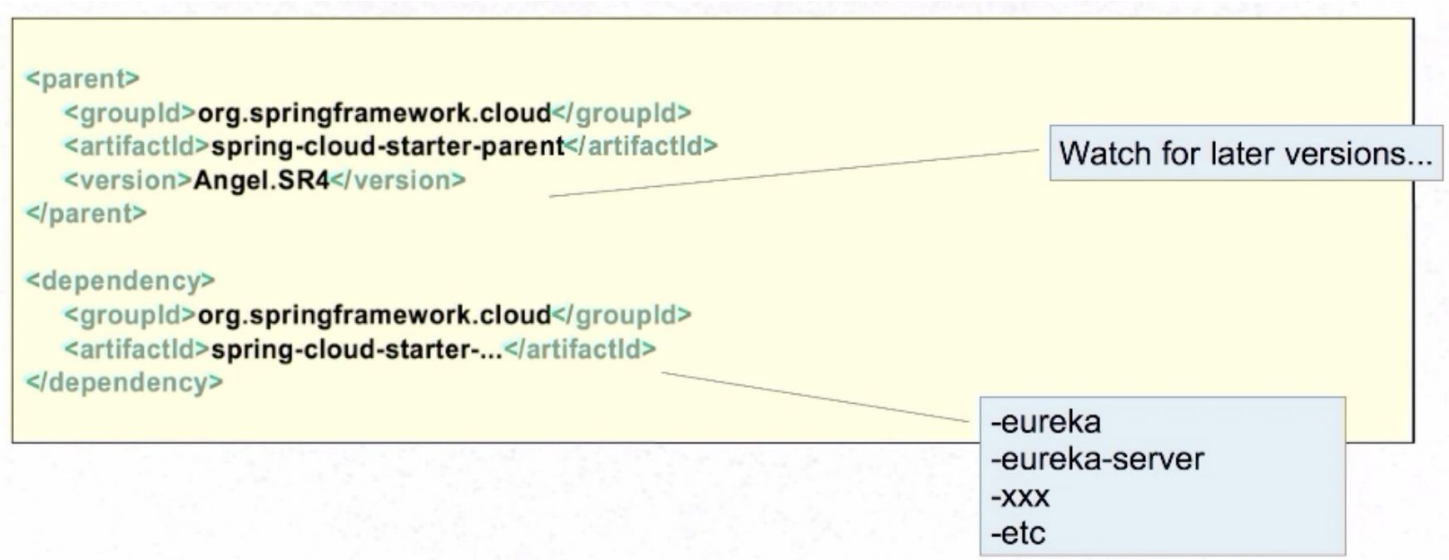
- Spring Cloud Projects are all based on Spring Boot
  - Difficult to employ using only core Spring Framework
  - Dependency management based on Boot
  - ApplicationContext startup process modified

Server vs Client

- “Client” and “Server” are relative terms
  - Based on the role in a relationship
  - A microservice is often a client and a server
- Don’t get lost on the terminology

Required Dependencies

- Replace Spring Boot Parent
  - Spring Cloud projects are based on Spring Boot



Required Dependencies

- … OR use Dependency Management Section

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-parent</artifactId>
      <version>Angel.SR4</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>

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

Summary

- Spring Cloud is a sub-project within Spring IO Umbrella
  - And is itself an umbrella project
- Spring Cloud addresses common patterns in distributed computing
- Spring Cloud enables easy use of Netflix libraries
- Spring Cloud is based on Spring Boot

Spring Cloud Config

Centralized, versioned configuration management for distributed applications

Objectives

- At the end of this module, you will be able to
  - Explain what Spring Cloud Config is
  - Build and Run a Spring Cloud Config Server
  - Establish a Repository
  - Build, Run, and Configure a Client

Module Outline

- Configuration Management
  - Challenges
  - Desired Solution
- Spring Cloud Config
  - Server Side
  - Client Side
- Repository Organization

What is Application Configuration ?

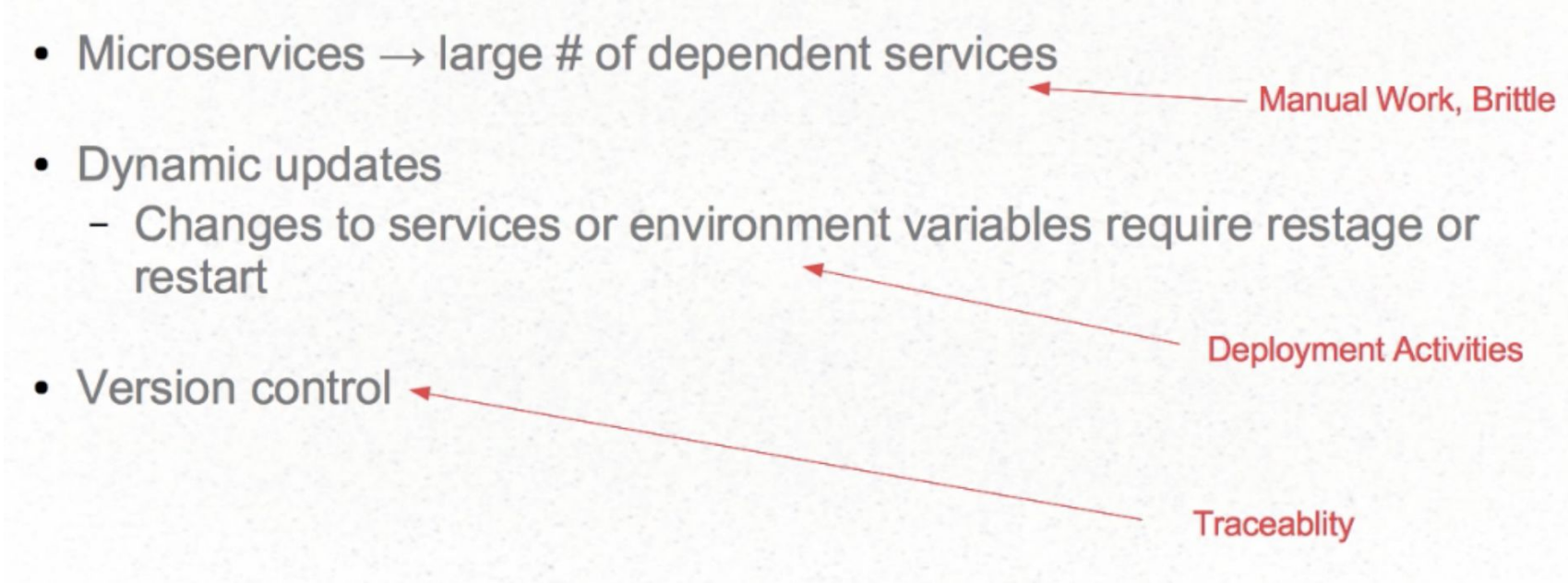
- Applications are more than just code
  - Connections to resources, other applications
- Usually use external configuration to adjust software behavior
  - Where resources are located
  - How to connect to the DB
  - etc

Configuration Options

- Package configuration files with application
  - Requires rebuild, restart
- Configuration files in common file system
  - Unavailable in cloud
- Use environment variables
  - Done differently on different platforms
  - Large # of individual variables to manage / duplicate
- Use a cloud-vendor specific solution
  - Coupling application to specific environment

Other Challenges

- Microservices → large # of dependent services
- Dynamic updates
  - Changes to services or environment variables require restage or restart
- Version control



Desired Solution for Configuration

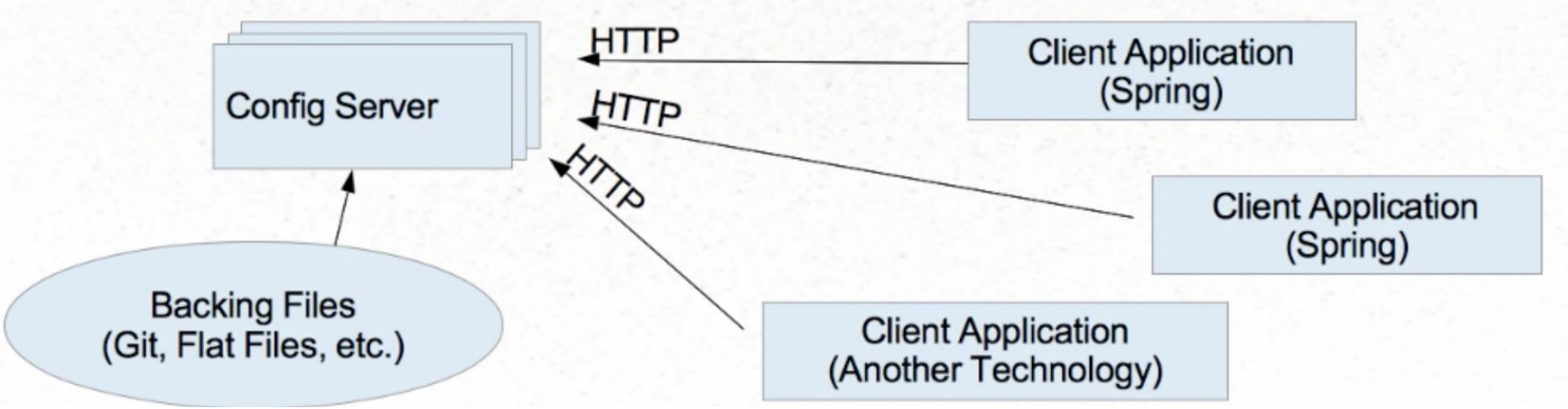
- Platform / Cloud-Independent Solution
  - Language Independent too
- Centralized
  - Or a few discrete sources of our choosing
- Dynamic
  - Ability to update settings while an application is running
- Controllable
  - Same SCM choices we use with software
- Passive
  - Services (Applications) should do most of the work themselves by self-registering

Solution

- Spring Cloud Config
  - Provides centralized, externalized, secured, easy-to-reach source of application configuration
- Spring Cloud Bus
  - Provides simple way to notify clients to config changes
- Spring Cloud Netflix Eureka
  - Service Discovery - Allow applications to register themselves as clients

Spring Cloud Config

- Designates a centralized server to serve-up configuration information
  - Configuration itself can be backed by source control
- Clients connect over HTTP and retrieve their configuration settings
  - In addition to their own, internal sources of configuration



Spring Cloud Config Server

- Source available at GitHub:
  - https://github.com/spring-cloud-samples/configserver
- Or, it is reasonably easy to build your own

Spring Cloud Config Server - Building, Part-1

- Include minimal dependencies in your POM (or Gradle)
  - Spring Cloud Starter Parent
  - Spring Cloud Config Server

```
<parent>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-parent</artifactId>
  <version>Angel.SR4</version>
</parent>

<dependencies>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-server</artifactId>
  </dependency>
</dependencies>
```

Spring Cloud Config Server - Building, Part-2

- application.yml - indicates location of configuration repository
- …, or application.properties

```
---
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/kennyk65/Microservices-With-Spring-Student-Files
          searchPaths: ConfigData
```

Spring Cloud Config Server - Building, Part-3

- Add @EnableConfigServer

```
@SpringBootApplication
@EnableConfigServer
public class Application {

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

}
```

The Client Side - Building, Part-1

- Use the Spring Cloud Starter Parent as Parent POM:

```
<parent>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-parent</artifactId>
   <version>Angel.SR4</version>
</parent>
```

- … OR use a Dependency Management Section:

```
<dependencyManagement>
   <dependencies>
      <dependency>
         <groupId>org.springframework.cloud</groupId>
         <artifactId>spring-cloud-starter-parent</artifactId>
         <version>Angel.SR4</version>
         <type>pom</type>
         <scope>import</scope>
      </dependency>
   </dependencies>
</dependencyManagement>
```

The Client Side - Building, Part-2

- Include the Spring Cloud Starter for config:

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

- Configure application name and server location in bootstrap.properties / .yml
  - So it is examined early in the startup process

```
# bootstrap.properties:
spring.application.name: lucky-word
spring.cloud.config.uri: http://localhost:8001
```

- That's it !
  - Client connects at startup for additional configuration settings.

EnvironmentRepository - Choices

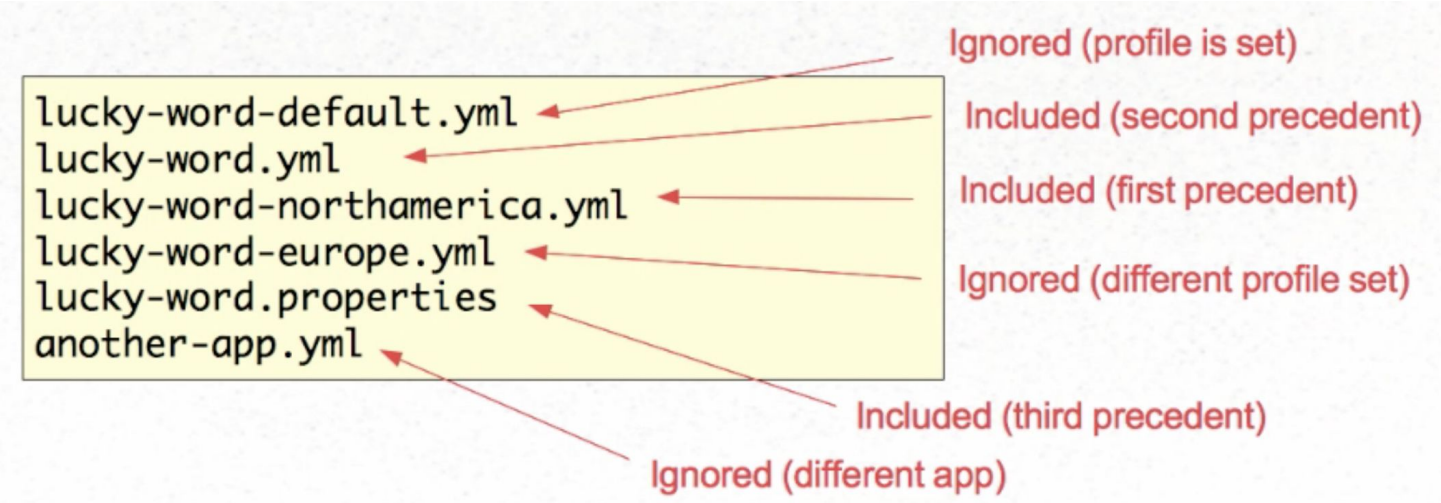
- Spring Cloud Config Server uses an `EnvironmentRepository`
  - Two implementations available: Git and Native (local files)
- Implement `EnvironmentRepository` to use other sources

EnvironmentRepository - Organization

- Configuration file naming convention:
  - <spring.application.name>-<profile>.yml
    - Or .properties (yml takes precedence)
  - Spring.application.name - set by client's application bootstrap.yml (or .properties)
  - Profile - Client's spring.profiles.active
    - (set various ways)
- Obtain settings from server:
  - http://<server>:<port>/<spring.application.name>/<profile>
  - Spring clients do this automatically on startup

EnvironmentRepository - Organization Example

- Assume client application named “lucky-word” and profile set to “northamerica”
  - Spring client (automatically) requests
    - /lucky-word/northamerica



.yml vs .properties

- Settings can be stored in either YAML or standard Java properties files
  - Both have advantages
  - Config server will favor .yml over .properties

```
# .properties file
spring.config.name=aaa
spring.config.location=bbb
spring.profiles.active=ccc
spring.profiles.include=ddd
```

```
# .yml file
---
spring:
  config:
    name: aaa
    location: bbb
  profiles:
    active: ccc
    include: ddd
```

Profiles

- YAML Format can hold multiple profiles in a single file

```
# lucky-word-east.properties
lucky-word: Clover
```

```
# lucky-word-west.properties
lucky-word: Rabbit's Foot
```

```
# luckyword.yml
---
spring:
  profiles: east
lucky-word: Clover

---
spring:
  profiles: west
lucky-word: Rabbit's Foot
```

The Client Side

- How Properties work in Spring Applications
  - Spring apps have an `Environment` object
  - `Environment` object contains multiple `PropertySources`
    - Typically populated from environment variables, system properties, JNDI, developer-specified property files, etc
  - Spring Cloud Config Client library simply adds another `PropertySource`
    - By connecting to server over HTTP
    - http://<server>:<port>/<spring.application.name>/<profile>
  - Result: Properties described by server become part of client application's environment

What about non-Java / non-Spring Clients ?

- Spring Cloud Server exposes properties over simple HTTP interface
  - http://<server>:<port>/<spring.application.name>/<profile>
- Reasonably easy to call server from any application
  - Just not as automated as Spring

What if the Config Server is Down ?

- Spring Cloud Config Server should typically run on several instances
  - So downtime should be a non-issue
- Client application can control policy of how to handle missing config server
  - spring.cloud.config.failFast=true
  - Default is false
- Config Server settings override local settings
  - Strategy: provide local fallback settings

Exercise

Setup your own Spring Cloud
Config Server, Client, and Repository

Instructions:
https://github.com/yauritux/g2lab/blob/master/workshop/java/microservices-with-springcloud/lab-instructions/lab-2.md

Service Discovery with Spring Cloud Eureka

Implementing Passive Service Discovery

What is it, and why would you use it ?

Objectives

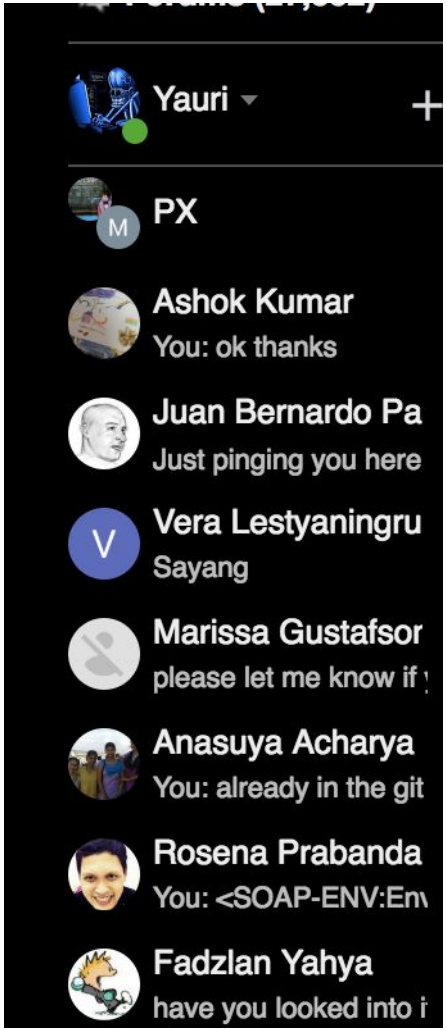
- At the end of this module, you will be able to
  - Explain what Passive Service Discovery is
  - Build, and Run a Spring Cloud Eureka Server
  - Build, Run, and Configure an Eureka Client

Module Outline

- Service Discovery
- Eureka Server
- Discovery Client
- Service Discovery Considerations

Service Discovery - Analogy

- When you sign into a chat client, what happens ?
  - Client ‘register’ itself with the server - server knows you are online
  - The server provides you with a list of all the other known clients
- In essence, you as a client has “discovered” the other clients
  - … and has itself been “discovered” by other clients



Service Discovery

- Microservice architectures result in large numbers of inter-service calls
  - Very challenging to configure
- How can one application easily find all of the other runtime dependencies?
  - Manual configuration - impractical, brittle
- Service Discovery provides a single “lookup” service
  - Clients register themselves, discover other registrants
  - Solutions: Eureka, Consul, Etcd, Zookeeper, SmartStack, etc

Eureka - Service Discovery Server and Client

- Part of Spring Cloud Netflix
  - Battle tested by Netflix
- Eureka provides a “lookup” server
  - Generally made highly available by running multiple copies
  - Copies replicate state of registered services
- “Client” services register with Eureka
  - Provide metadata on host, port, health indicator URL, etc
- Client Services send heartbeats to Eureka
  - Eureka removes services without heartbeats

Making a Eureka Server

- Just a regular Spring Boot web application with dependencies and @EnableEurekaServer:

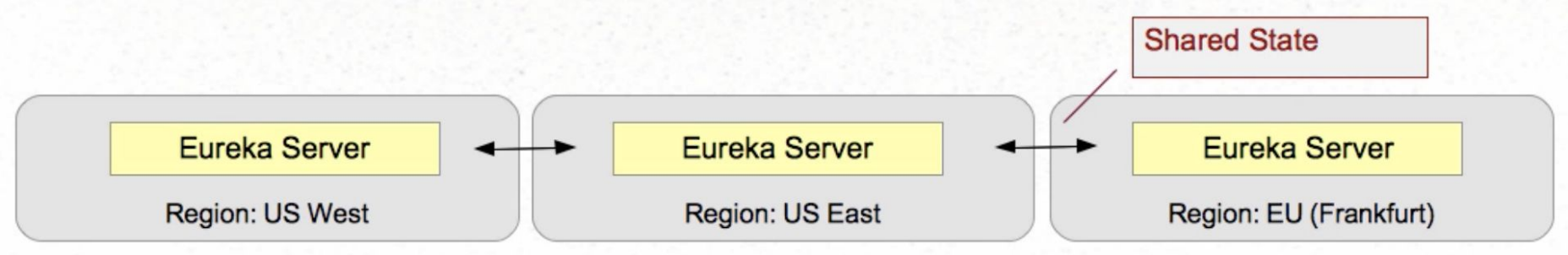
```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka-server</artifactId>
</dependency>
```

```
@SpringBootApplication
@EnableEurekaServer
public class Application {

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

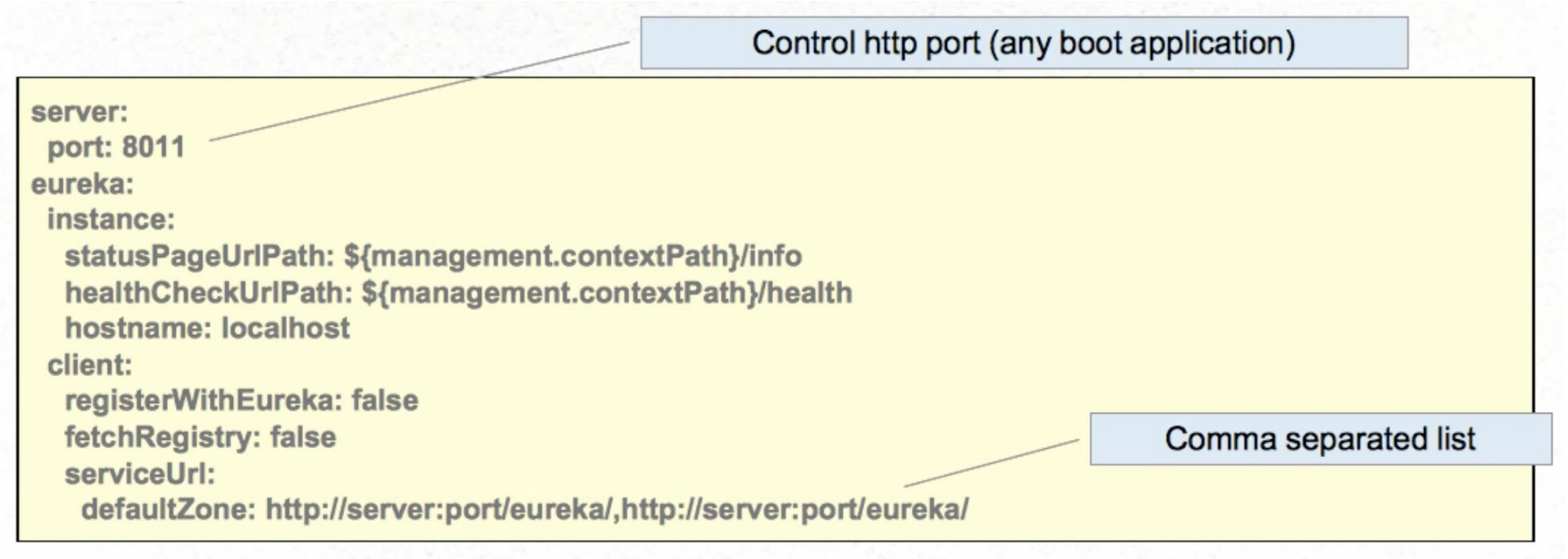
Multiple Servers

- Typically, multiple Eureka servers should be run simultaneously
  - Otherwise, you'll get many warnings in the log
  - Eureka servers communicate with each other to share state
  - Provides High Availability
- Each server should know URL to the others
  - Can be provided by Config Server
  - One server (JAR), multiple profiles



Multiple Servers - Configuration

- Common Configuration Options for Eureka Server:
  - See https://github.com/Netflix/eureka/wiki/Configuring-Eureka for full list.



Registering with Eureka

- From a SpringBoot application
  - Add Dependency:

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

  - Only the location of Eureka itself requires explicit configuration

```
@SpringBootApplication
@EnableDiscoveryClient
public class Application {


}
```

Default fallback.
Any Eureka instance will do
(we usually want several comma-separated URLs)

```
# application.properties
eureka.client.serviceUrl.defaultZone:  http://server:8761/eureka/
```

@EnableDiscoveryClient

- Automatically registers client with Eureka Server
  - Registers the application `name`, `host`, and `port`
    - Using values from the Spring `Environment`
    - But can be overridden
    - Give your application a spring.application.name
- Makes this app an “instance” and a “client”
  - It can locate other services

```
@Autowired DiscoveryClient client;
public URI storeServiceUrl() {
   List<ServiceInstance> list = client.getInstances("STORES");
   if (list != null && list.size() > 0 ) {
    return list.get(0).getUri();
   }
   return null;
}
```

Service ID (Eureka VIP)
Corresponds to
spring.application.name

What is a Zone ?

- Eureka server designed for multi-instance use
  - Standalone mode will actually warn you when it runs without any peers!
- Eureka server does not persist service registrations
  - Relies on client registrations; always up to date, always in memory
    - Stateful application
- Typical production usage - many Eureka server instances running in different availability zones / regions
  - Connected to each other as “peers”

Which comes first ? Eureka or Config Server ?

- **Config First Bootstrap** (default) - Use Config Server to configure location of Eureka Server
  - Implies spring.cloud.config.uri configured in each app
- **Eureka First Bootstrap** - Use Eureka to expose location to config server
  - Config server is just another client
  - Implies spring.cloud.config.discovery.enabled=true and eureka.client.serviceUrl.defaultZone configured in each app
  - Client makes two network trips to obtain configuration

References

- http://techblog.netflix.com/2012/09/eureka.html
- https://spring.io/blog/2015/01/20/microservice-registration-and-discovery-with-spring-cloud-and-netflix-s-eureka

Summary

- Passive Service Discovery
  - Having services register themselves and find others simultaneously
- Spring Cloud Eureka Server
  - Holds registrations, shares informations on other registrants
  - Synchronizes itself with other servers
- Spring Cloud Eureka Client
  - Connects to server to register, and obtain information on other clients

Exercise

Create and Run Several Applications coordinated by Spring Cloud Eureka

Instructions:

https://github.com/yauritux/g2lab/blob/master/workshop/java/microservices-with-springcloud/lab-instructions/lab-3.md

Spring Cloud Ribbon

Understanding and Using Ribbon,
The client side load balancer

Objectives

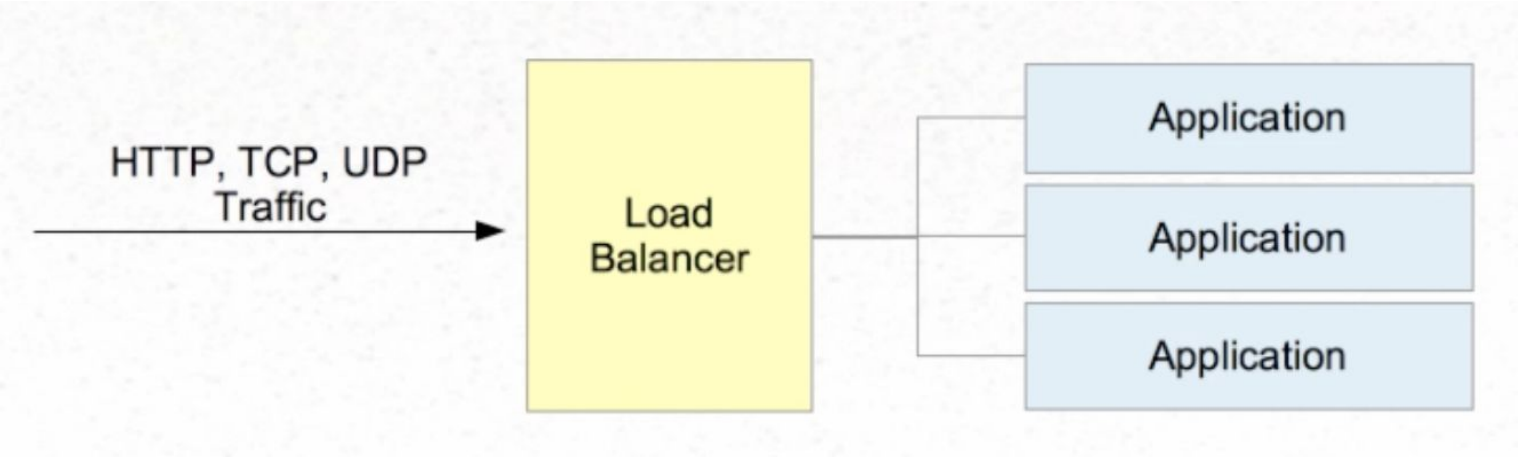
- At the end of this module, you will be able to
  - Understand the purpose of Client-Side Load Balancing
  - Use Spring Cloud Ribbon to implement Client-Side Load Balancing

Module Outline

- Client Side Load Balancing
- Spring Cloud Netflix Ribbon

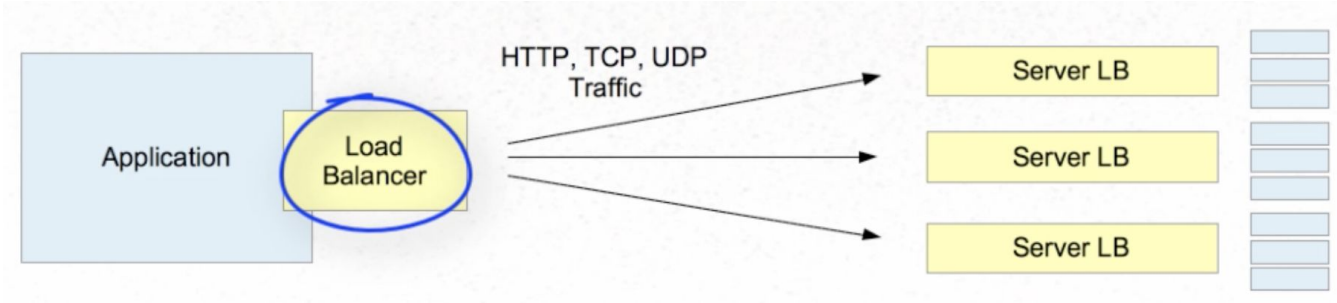
What is a Load Balancer ?

- Traditional Load Balancers are server-side components
  - Distribute incoming traffic among several servers
  - Software (Apache, Nginx, HA Proxy) or Hardware (F5, NSX, BigIP)



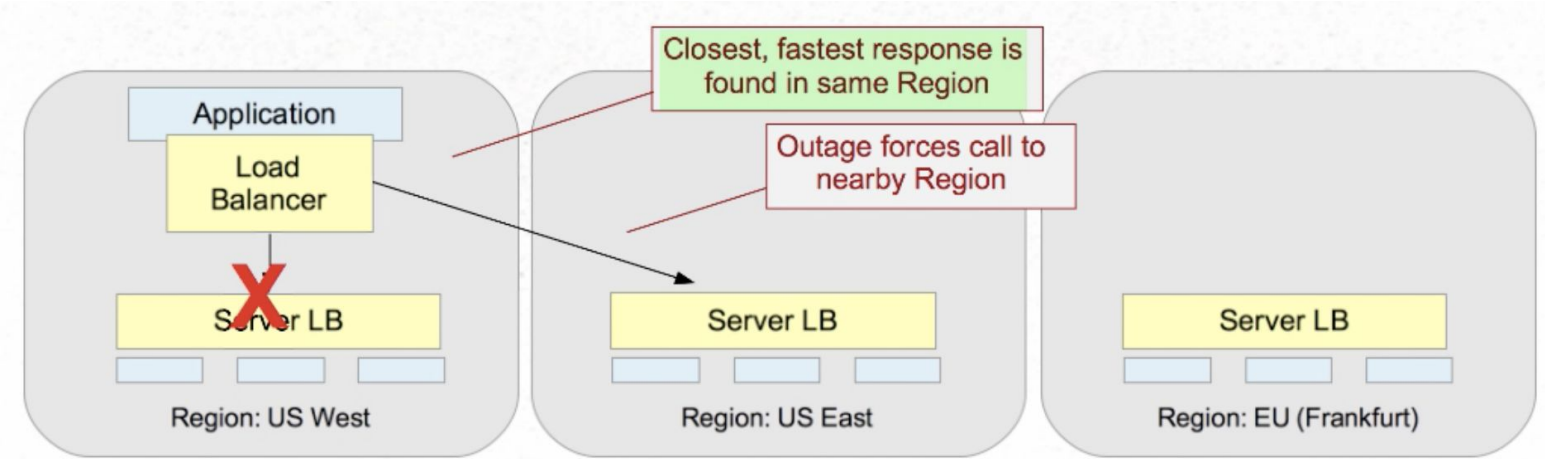
Client-Side Load Balancer

- Client-Side Load Balancer selects which servers to call
  - Based on some criteria
  - Part of client software
  - Server can still employ its own load balancer



Why is this necessary ?

- Not all servers are the same
  - Some may be unavailable (faults)
  - Some may be slower than others (performance)
  - Some may be further away than others (regions)



Spring Cloud Netflix Ribbon

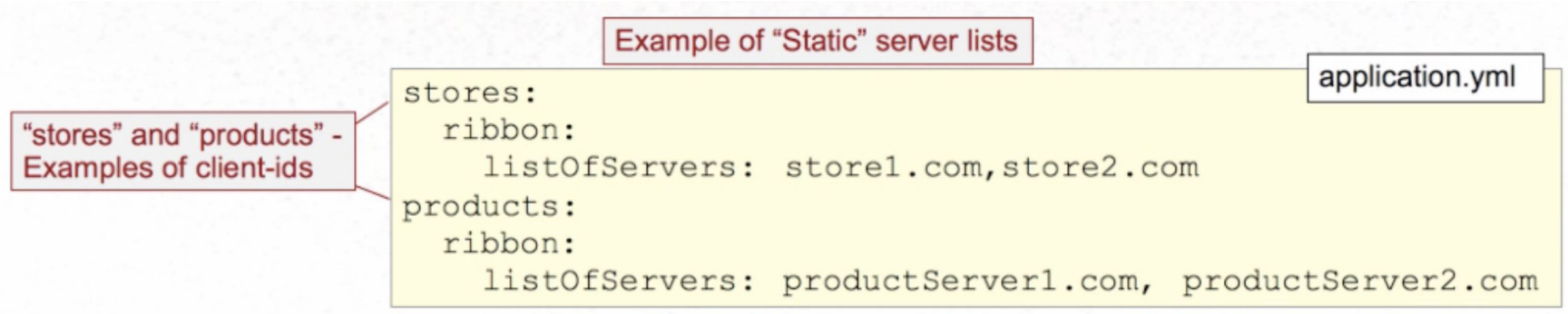
- Ribbon - Another part of the Netflix OSS family
  - Client side load balancer
  - Automatically integrates with service discovery (Eureka)
  - Built in failure resiliency (Hystrix)
  - Caching / Batching
  - Multiple protocols (HTTP, TCP, UDP)
- Spring Cloud provides an easy API wrapper for using Ribbon

Key Ribbon Concepts

- List of Servers
- Filtered List of Servers
- Load Balancer
- Ping

List of Servers

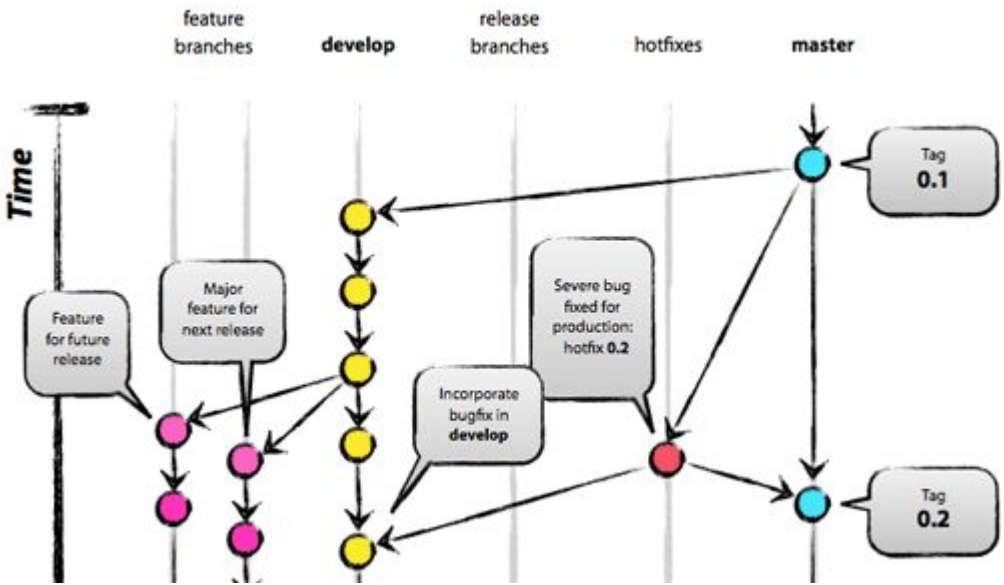
- Determines what the list of possible servers are (for a given service (client))
  - Static - populated via configuration
  - Dynamic - populated via Service Discovery (Eureka)
- Spring Cloud default - Use Eureka when present on the classpath



Filtered List of Servers

- Criteria by which you wish to limit the total list
- Spring Cloud default - Filter servers in the same *zone*

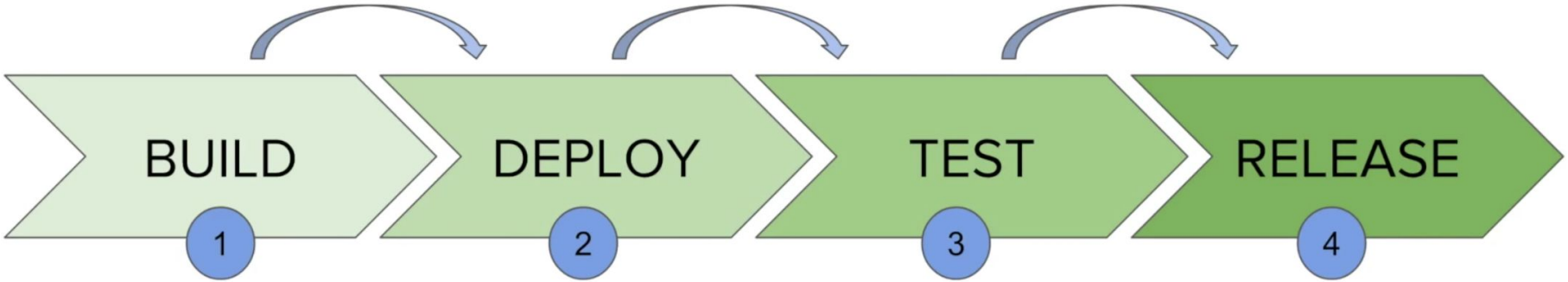
Git Best Practices (Branching Strategy)



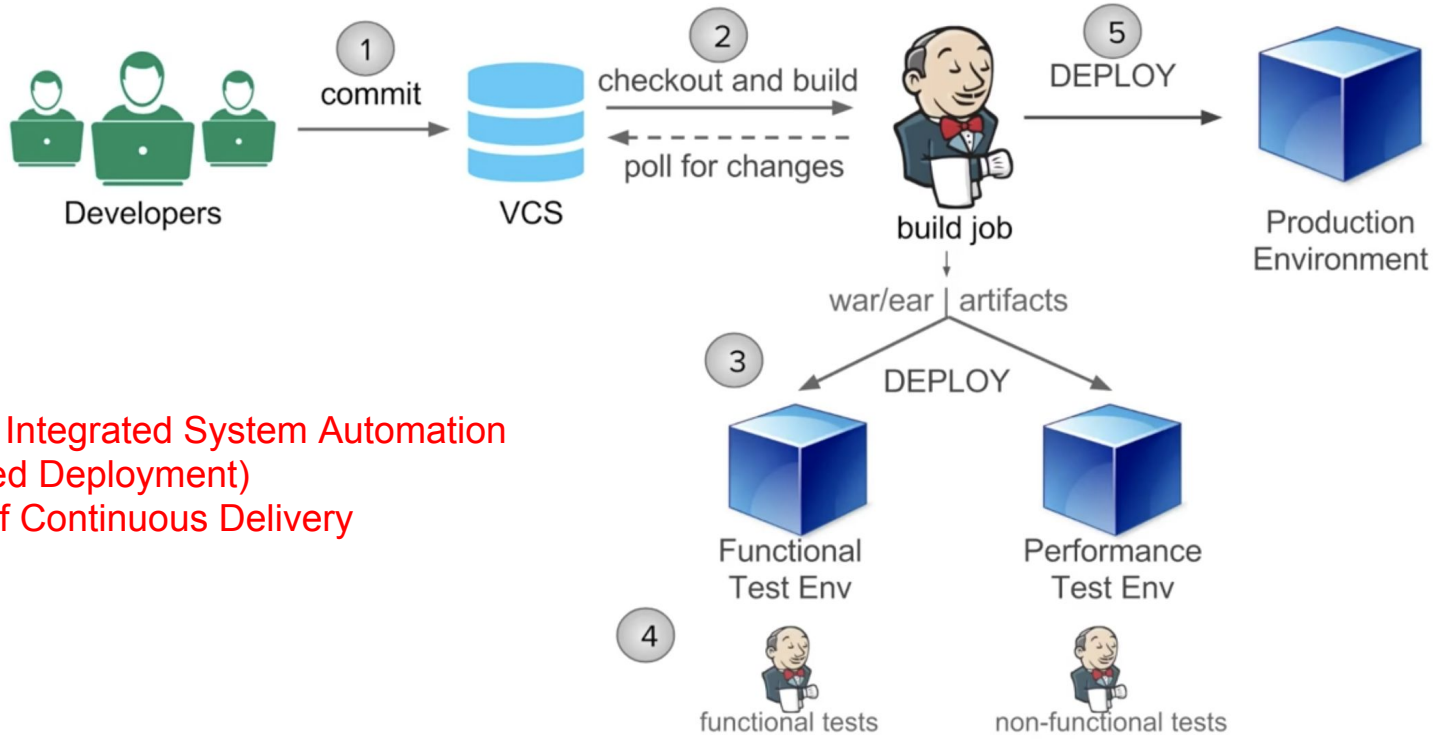
Jenkins

- Introduction and Getting Started
- Installing as “standalone” mode
- Installing on specific server (e.g. Tomcat)
- Change Jenkins Home directory
- How to use jenkins-cli
- Jenkins RBAC (manage users and roles)
- Basic Configurations
- Jenkins JOBS
- Integration with Git
- Catlight as Jenkins Build Monitor / Status Notifier
- Email Notifications

Continuous Delivery Pipeline



Real Scenario CI/CD Pipeline



Complete Integrated System Automation
(Automated Deployment)
is a part of Continuous Delivery

BUILD → DEPLOY → TEST → RELEASE

Conclusion

In General:

- ***Master path is a long journey***
- ***Master path needs passion and an amount of sacrifice***
- ***There's no short path to be a master***
- ***Master concerns with the best practices to solve something***
- ***Master knows the best learning path***
- ***Keep focus***

Today's Topic:

- ***Microservices is an architectural style***
- ***Microservices have advantages and disadvantages (no silver bullet)***
- ***It's easy to create microservice in Java (despite the fallacies of dist.computing)***


Q & A / Discussion