**Microservices, Spring Boot and Spring Cloud**

**1. Introduction to Microservices, Spring Boot and Spring Cloud**

**What are Microservices?**

"Loosely coupled service-oriented architecture with bounded context".

Adrian Cockcroft, Battery Ventures

**Why are Microservices Architectures popular?**

* Desire for faster changes
* Need for greater availability
* Motivated for fine-grained scaling
* Compatible with a DevOps mindset

**Core Characteristics of Microservices**

* Components exposed as services
* Tied to specific Domain
* Loosely coupled
* Built to tolerate failure
* Delivered continuously via automation
* Built and run by independent teams

Should every app be turned into a set of Microservices?

No, some application should not turn into a set of Microservices

if they are not having multiple team dependency, continuous delivery etc...

**What is Spring Cloud?**

* Microservices Scaffolding with Spring Cloud
  + Released on March 2015
  + Build Common Distributed System Pattern
  + Open Source Software
  + Optimized for Spring Apps
  + Run Anywhere
  + Includes NetflixOSS Technology

* Catalog of Spring Cloud Projects

|  |  |
| --- | --- |
| Spring Cloud Config | Git-backed configuration server |
| Spring Cloud Netflix | Suite for service discovery, routing, availability |
| Spring Cloud Consul | Service discovery with Consul |
| Spring Cloud Security | Simplify OAuth 2.0 flows |
| Spring Cloud Sleuth | Distributed tracing |
| Spring Cloud Stream | Message bus abstraction |
| Spring Cloud Task | Short-lived, Single-task microservices |
| Spring Cloud Dataflow | Orchestration of data microservices |
| Spring Cloud Zookeeper | Service discovery and configuration with Zookeeper |
| Spring Cloud for AWS | Exposes core AWS service to Spring Developers |
| Spring Cloud Spinnaker | Multi-Cloud deployment |
| Sprint cloud Contract | Stubs for service contracts |

**What is Spring Boot?**

* Offers opinionated runtime for Spring
* Convention, Not configuration
* "Opinions" can be overridden
* Handles boilerplate setup
* Simple dependency management
* Embeds app server in executable JAR
* Built in endpoints for health metrics

**DEMO: Building a Spring Boot Apps**

* Reviewing the Spring INITIALIZR Site
  + https://spring.io
  + https://start.spring.io
* Looking at Spring Tool Suite
* Creating a Spring Boot Project
* Reviewing the POM File
* Editing the application.properties file
* Adding a REST Endpoint
* Starting and running an application
* Viewing the Actuator endpoints

application.properties configuration

server.port=8080

management.endpoints.web.exposure.include=\*

management.trace.http.enabled=true

Useful Endpoints

http://localhost:8080/actuator/health

http://localhost:8080/actuator/beans

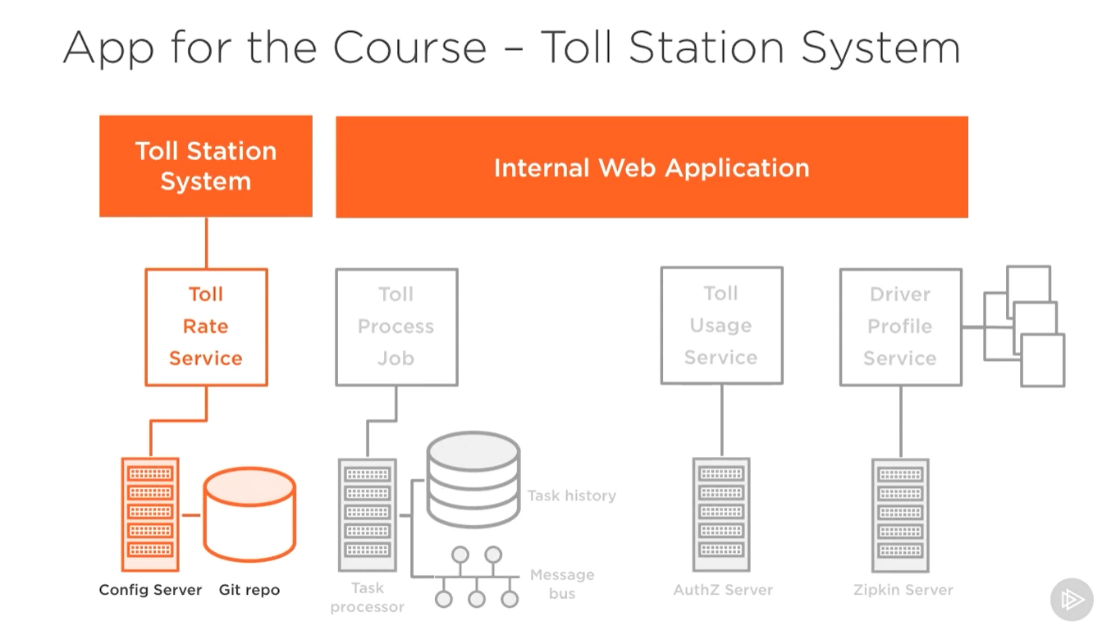
http://localhost:8080/actuator/env

http://localhost:8080/actuator/metrics

http://localhost:8080/actuator/mappings

http://localhost:8080/actuator/trace

**Core Application and Prerequisites**



* Java and OOP
* Spring Framework
* Java-friendly IDE
* MYSQL and RabbitMQ
* Goals for this Course
  + Understand modern Microservices pattern
  + Get comfortable starting Spring Boot and Spring Cloud projects
  + Explore the core capabilities of Spring Cloud
  + Lean how to configure and extend Spring Cloud
* Workstation
  + Spring Tool Suite
  + Maven build manager
  + Vagrant
  + Postman
  + GitHub Account

**2. Simplifying Environment Management with Centralized Configuration**

**The Role of Configuration in Microservices**

* Removing **“settings”** from compiled code
* Change runtime behaviors
* Enforce consistency across elastic services
* Cache values to reduce load on databases

**Problems with the Status Quo**

* Local configuration files fall out of sync
* No history of changes with env variables
* Configuration changes required restart
* Challenges with sensitive information
* Inconsistent usage across teams

**Creating Config Server**

* Spring Cloud Config
  + HTTP access to git or file-based configurations
* Choose a config source
  + Local Files
    - Points to classpath or file system
    - Multiple search locations possible
    - No audit trails
    - Supports labelling
    - Support for placeholders in URI
    - Relies on “native” profile
    - Dev/Test only, unless set up in reliable, shared fashion
  + Git-based Repository
    - Points to Git repository
    - Multiple search locations possible
    - Full change history
    - Supports labelling
    - Support for placeholders in URI
    - Multiple profiles possible
    - Local Git for Dev/Test highly available file system or service for production
* Add config files
  + Setting up Configuration Files
    - Native support for YML, properties file
    - Can serve out any text file
    - File name contains app, optionally profile
    - Nested folder supported
    - All matching files returned
* Build the Spring project
  + Use <https://spring.io> or Spring Tool Suite to generate scaffolding.
  + See POM dependency spring-cloud-config-server and spring-boot-starter-actuator.
  + Add @EnableConfigServer annotation to class.
  + Create application properties (or YAML) with server port, app name and profile.
* Secure the configurations

**DEMO: Creating a Config Server for Local Files**

* Create a Spring Starter project
* Annotate the main class
  + @EnableConfigServer
* Set the application properties
  + server.port=8080
  + spring.profiles.active=native
  + management.endpoints.web.exposure.include=mappings
* Add **local** configuration files
  + Create config folder under resources and add properties files
    - app1.properties with greeting=Hello
    - app2.properties with greeting=Hi
    - app3.properties with greeting=Howdy
* Run as a Spring Boot app
* Query for configurations
  + <http://localhost:8080/app1/default>
    - We will get all the properties from app1.properties
  + <http://localhost:8080/app2/default>
    - We will get all the properties from app2.properties
  + <http://localhost:8080/app3/default>
    - We will get all the properties from app3.properties

**Working with Config Server URIs**

---

spring:

cloud:

config:

server:

git:

uri: <https://github.com/wa-tolls/rates> 🡨 Point to Git Location

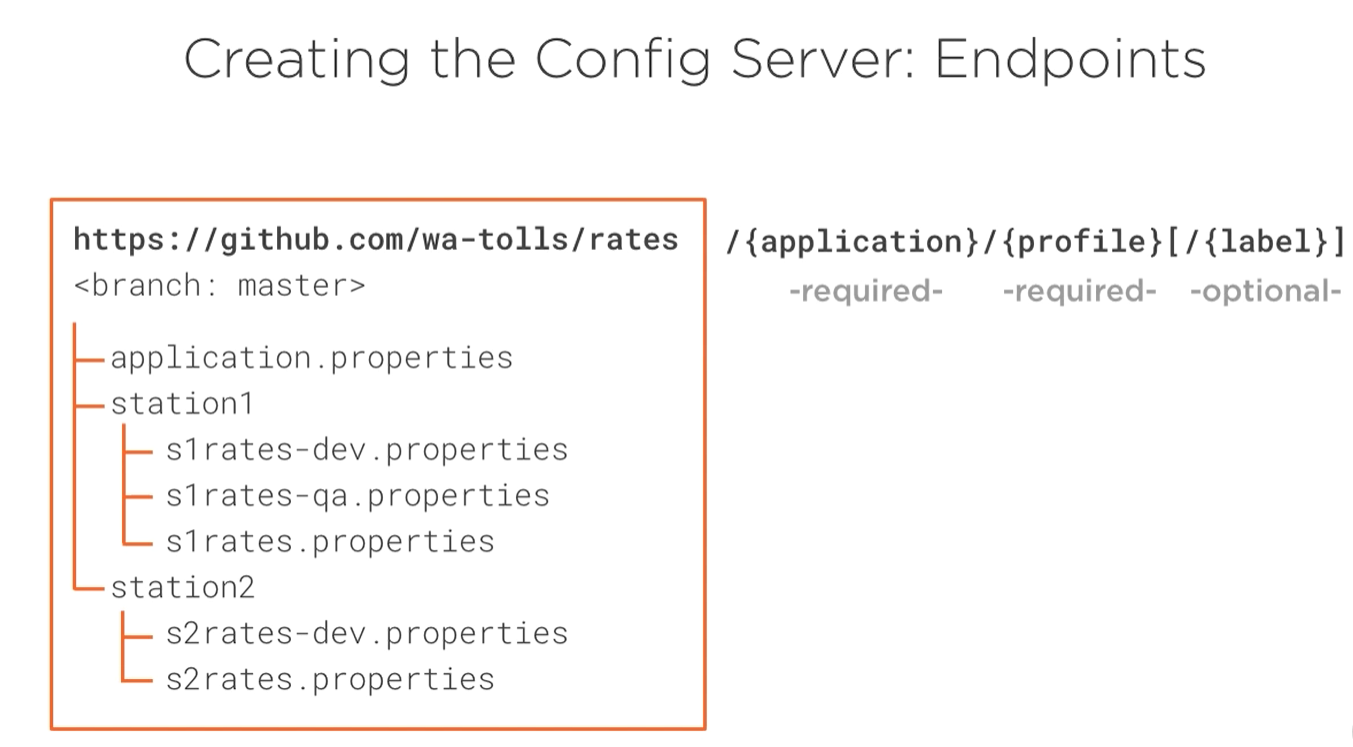
searchPaths: ‘station\*’ 🡨 Pattern to search sub directories

repos: 🡨 Points to alternate repositories

prod:

pattern: ‘\*/prod’ 🡨 Pattern to go to alternate repositories

uri: https://github.com/wa-tolls/rates-prod



**DEMO: Creating a Config Server for GitHub Files**

* Create GitHub Repository with Files
  + <https://github.com/Raj-Tomar/spring-cloud-config-wa-tolls>
  + <https://github.com/Raj-Tomar/spring-cloud-config-wa-tolls-performace>
* Create a Spring Starter Project
  + <https://github.com/Raj-Tomar/spring-cloud-m2-config-server-git>
* Annotate the main class
  + @EnableConfigServer
* Set Git URL in application YAML
  + Configure below properties in application.yml file

---

server:

port: 8080

spring:

cloud:

config:

server:

git:

uri: https://github.com/Raj-Tomar/spring-cloud-config-wa-tolls

#username: Raj

#password: RajTomar

search-paths:

- 'station\*'

repos:

perfs:

pattern:

- "\*/performance"

uri: https://github.com/Raj-Tomar/spring-cloud-config-wa-tolls-performance

search-paths:

- 'station\*'

* Run as a Spring Boot App
* Experiments with search paths, queries
  + <http://localhost:8080/s1rates/default>
  + <http://localhost:8080/s1rates/master>
  + <http://localhost:8080/s1rates/dev>
  + <http://localhost:8080/s1rates-default.properties>
  + <http://localhost:8080/s1rates-default.yml>
  + <http://localhost:8080/s1rates-default.json>
  + <http://localhost:8080/s1rates/performace>
  + <http://localhost:8080/s3rates/performace>

**Consuming Configurations from Spring Boot Apps**

* Spring apps are use Config Servers as a property source
* Loads values based on app name, Spring profile, label
* Annotate code with ***@Value*** attribute
* Can consume from non-spring apps via URL

**DEMO: Consuming Configuration from Spring Boot App**

* Create a Spring Starter Project
  + <https://github.com/Raj-Tomar/spring-cloud-m2-config-client>
* Add application and bootstrap files
  + bootstrap.properties for loading configuration quickly
* Create Controller with annotations
* Return values derived from properties
* Experiment with different profile, profiles

**Applying Access Security to Configuration**

* Integrated Security via Spring Security
* Default HTTP Basic, but other options like OAuth2
* Configured in properties, YAML files
* Could be unique per profile
* Look to also secure with network security, API gateways

**DEMO: Applying Access Security to Configuration**

* Add POM dependency for springboot-starter-security

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

* Test project and get authentication error
  + http://localhost:8888/s1rates/default
* Add Basic Auth credentials

spring:

security:

user:

name: <UserName>

password: <Password>

* Call API with valid credentials
  + Add Authorization as Basic Auth with UserName and Password
  + <http://localhost:8888/s1rates/default>
* Update Client app with credentials
  + Add Security Configuration in bootstrap.properties of Client App
    - spring.cloud.config.username=<UserName>
    - spring.cloud.config.password=<Password>

**Encrypting and Decrypting Configurations**

* Property values not stored in plain text
* Symmetric or Asymmetric options
  + Symmetric option share key
  + Asymmetric share key value pair and it is more secure
* Server offers /encrypt and /decrypt endpoint
* Can decrypt on Server not in the Client

**DEMO: Encrypting and Decrypting Configurations**

* Download full-strength JCE (Java Cryptography Extension)
  + Required only for JDK 8 updates earlier than **8u161**
* Add key to bootstrap file
  + create bootstrap.properties and add encrypt.key=ABCDEFGHIJKLMNOPQRSTUVWXYZ
* Generate encrypted value and add to properties file
  + <http://localhost:8888/encrypt> (POST)
  + Verify generated encrypted values using below endpoint
    - <http://localhost:8888/decrypt> (POST)
    - Pass encrypted String in Body
  + Pass in Body connectionstring=sever123;user=root;password=password1
  + Copy the encrypted value and add in application.properties file of spring-cloud-config-wa-tolls
    - <https://github.com/Raj-Tomar/spring-cloud-config-wa-tolls/blob/master/application.properties>
    - constring={cipher}<Encrypted String>
* Retrieve configuration via API
  + <http://localhost:8888/s1rates/default> (GET)
* Test Client app with **Server-Side** decrypt value
  + @Value("${constring}") add in Controller and Model
  + <http://localhost:8080/rate>
* Update Server to require **Client-Side** decryption
  + Add in bootstrap.properties spring.cloud.config.server.encrypt.enabled=false
  + Test <http://localhost:8888/s1rates/default>
* Change client to decrypt
  + Add in bootstrap.properties encrypt.key=ABCDEFGHIJKLMNOPQRSTUVWXYZ
  + Test <http://localhost:8080/rate>

**Advanced Settings and Property Refresh**

* Configure for “fail fast” to fail service if can not connect to config server
* We Can add Client retry if Config server occasionally unavailable
* Refresh Clients individually or in bulk

**DEMO: Advanced Settings and Property Refresh**

* Add RefreshScope to the Controller
  + @RefreshScope Annotation on Controller
* Start Server and Client Apps
* Change a Property in GitHub
  + <https://github.com/Raj-Tomar/spring-cloud-config-wa-tolls/blob/master/station1/s1rates.properties>
* Trigger Client Refresh
  + Add in bootstrap.properties file management.endpoints.web.exposure.include=\*
  + <http://localhost:8080/actuator/refresh>
* See new value without a restart
  + <http://localhost:8080/rate>

**3. Offloading Asynchronous Activities with Lightweight, Short-lived Tasks**

**Introduction**

* The Role of asynchronous processing in Microservices
* Problem with the status quo
* Defining **“serverless”** computing
* Describing Spring Cloud Task
* Creating a Task
* Reviewing storage options for results
* Options for invoking Tasks

**The Role of Asynchronous Processing in Microservices**

* Reduce dependencies between services
* Support low latency, high throughput
* Facilitate **event-driven** computing

**Problem with the Status Quo**

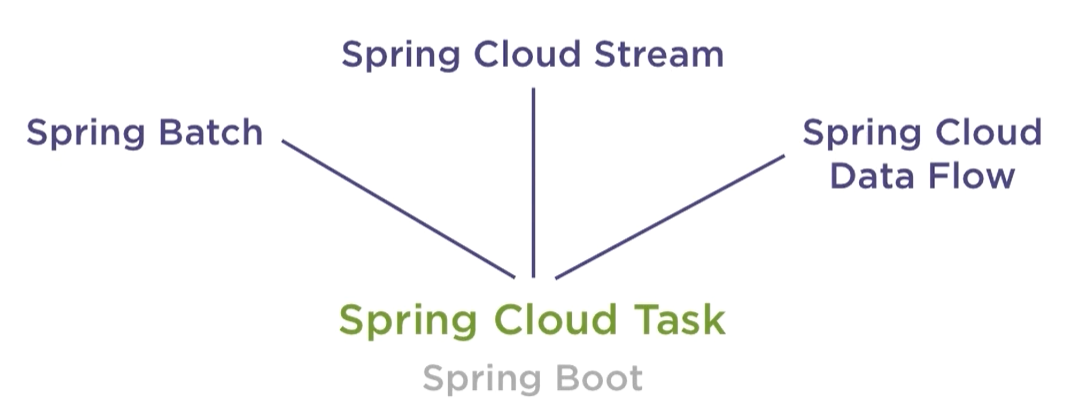
* Consuming resources even when services aren’t in use
* Services baked into monolithic deployments
* Challenges scaling services on demand
* Difficulty tracing service calls

**What Exactly Is “Serverless” Computing?**

* Deploy **“Function”** instead of **“Application”**
* Run code without knowledge of infrastructure
* Elastic, Automatic Horizontal scaling
* Start Fast, Run Short

**What is Spring Cloud Task, and Integration with Spring Projects**

* Short-lived, Asynchronous Microservices
* How this fit into the Spring Ecosystem?



**Creating a Task**

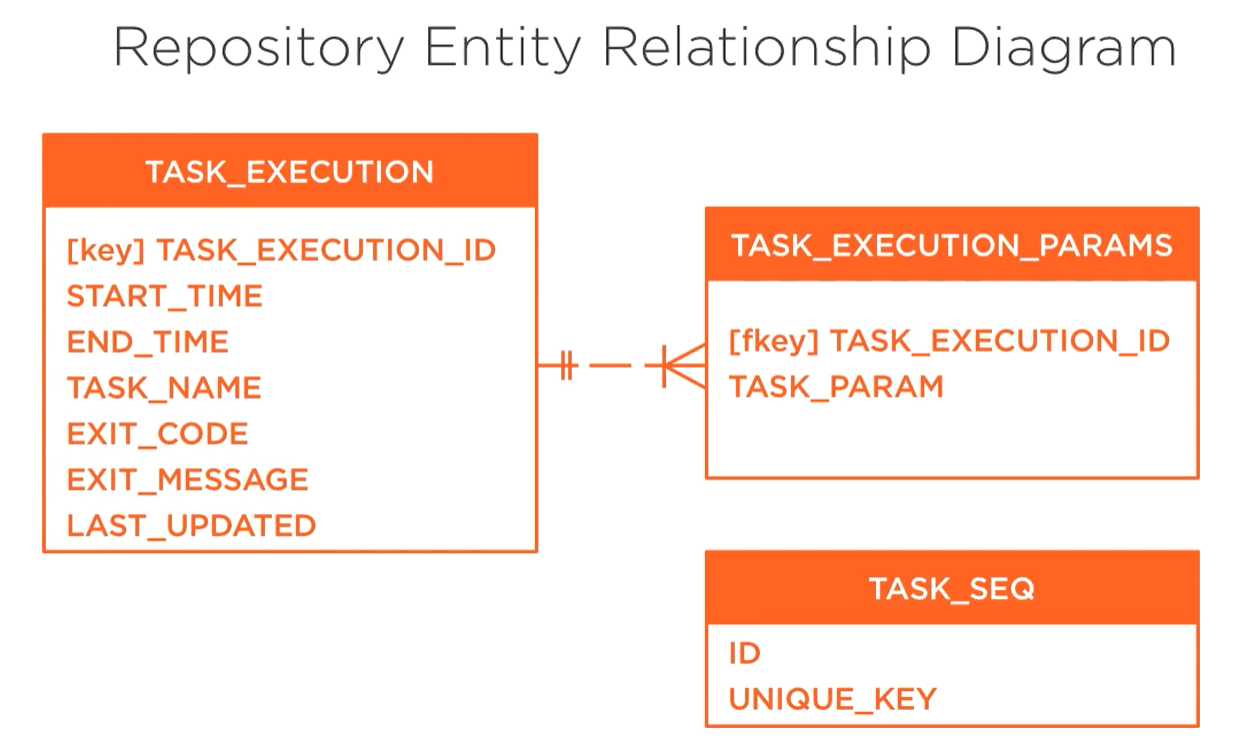
* Add dependency to POM
* Annotate the Class with **@EnableTask**
* Add Business logic to run the Task
* Deploy to the Maven Repository
* How does Task’s logic work?
  + Spring (Boot) app with access to beans
  + Task is stateless
  + Bootstrap logic with Runner
  + Can Subscribe to lifecycle events

**DEMO: Creating a Task**

* Create a new Spring Boot Project for Toll Processing Task
* Annotate Primary Class
  + @EnableTask
* Add Task logic as CommandLineRunner
  + **public** **class** TollProcessingTask **implements** CommandLineRunner
  + Add below properties to application.properties file
    - spring.application.name=Toll Processor
    - logging.level.org.springframework.cloud.task=DEBUG
* Execute Task and observe results

**Storing Task Execution History**

* Multiple Task Result Storage Options
  + H2 In-Memory Database
  + HSQLDB
  + MySQL
  + Oracle
  + PostgreSQL
  + SQL Server

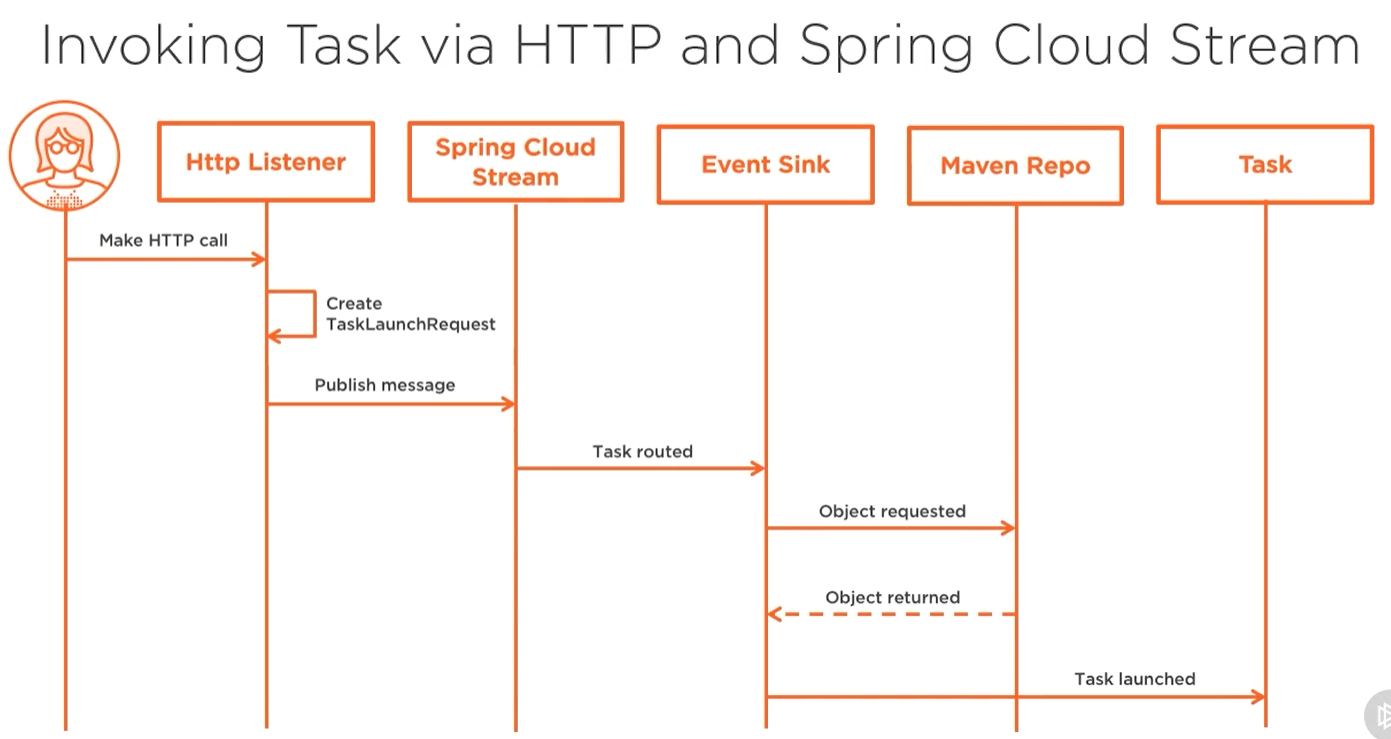


**DEMO: Storing Task Execution History**

* Create MySQL database
* Add MySQL dependencies in POM
* Update application properties
* Call Task and observe stored results
  + <https://github.com/Raj-Tomar/spring-cloud-m3-task>

**Options for Invoking Tasks**

* Run jar as Cron (scheduled) job
* Include as part of data pipeline
* Subscribe to event bus
* Directly invoked via custom Launcher



**DEMO: Invoking a Task Through HTTP and Spring Cloud Stream**

* Create RabbitMQ environment
* Build event sink
* Build launcher
* Submit request and see execution