第六章

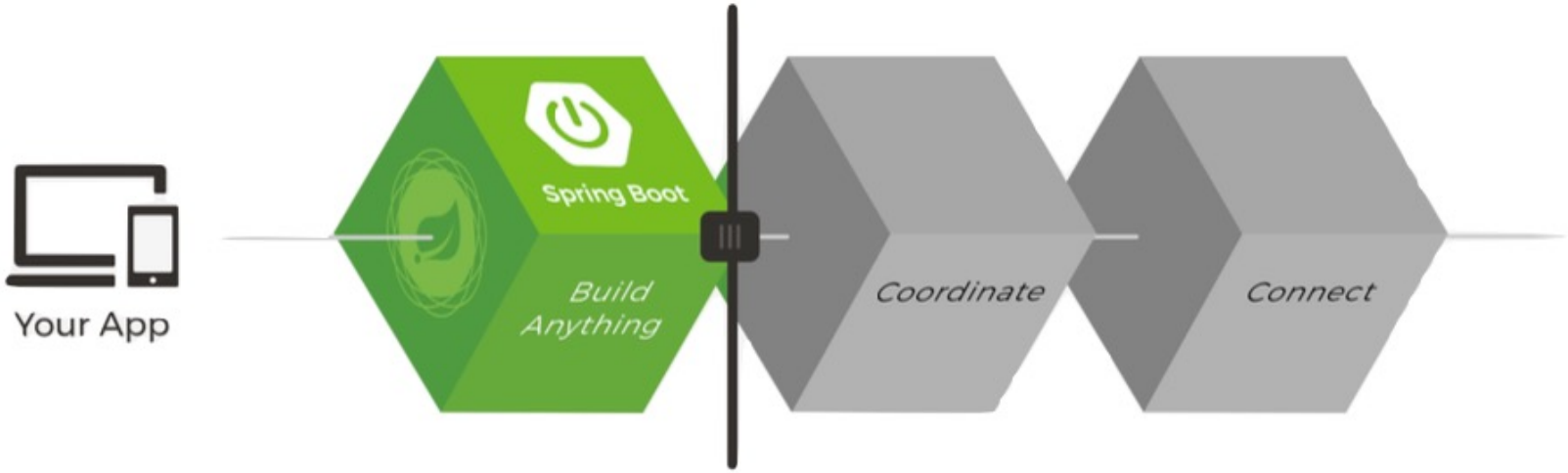
Spring Boot

—— Build Anything

内容提要

- **Spring Boot组成**
- **自动配置的实现原理**
- **起步依赖starter的实现原理**
- **配置加载机制**
- **Actuator Endpoint**
- **Spring Boot Admin监控**
- **定制 Web 容器**
- **打包 Docker 镜像**

Spring: the source for modern java



Spring Boot

BUILD ANYTHING

Spring Boot is designed to get you up and running as quickly as possible, with minimal upfront configuration of Spring. Spring Boot takes an opinionated view of building production-ready applications.

Spring Cloud

COORDINATE ANYTHING

Built directly on Spring Boot's innovative approach to enterprise Java, Spring Cloud simplifies distributed, microservice-style architecture by implementing proven patterns to bring resilience, reliability, and coordination to your microservices.

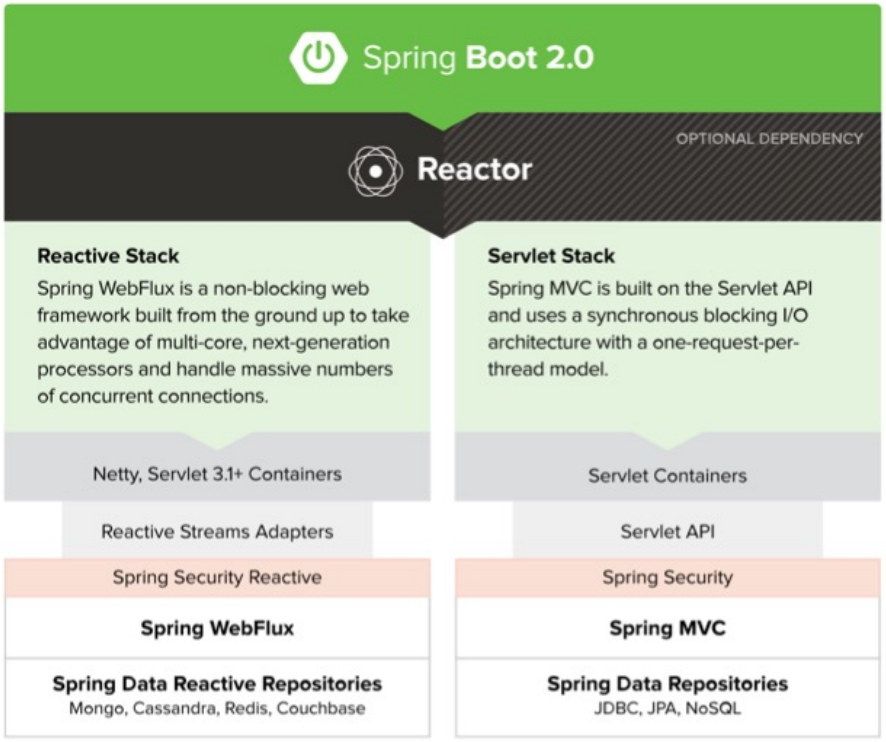
Spring Cloud Data Flow

CONNECT ANYTHING

Connect the Enterprise to the Internet of Anything—mobile devices, sensors, wearables, automobiles, and more. Spring Cloud Data Flow provides a unified service for creating composable data microservices that address streaming and ETL-based data processing patterns.

Spring Boot

- **快速构建基于Spring的应用程序**
  - 快、很快、非常快
  - 进可开箱即用，退可按需改动
  - 提供各种非功能特性
  - 不用生成代码，没有 XML 配置
- **在本课程中，你还会看到**
  - Spring Data、Spring MVC、Spring WebFlux……



Spring Boot不是什么

- 不是应用服务器
- 不是 Java EE 之类的规范
- 不是代码生成器
- 不是 Spring Framework 的升级版

Spring Boot的特性

- 方便地创建可独立运行的 Spring 应用程序
- 直接内嵌 Tomcat、Jetty 或 Undertow
- 简化了项目的构建配置
- 为 Spring 及第三方库提供自动配置
- 提供生产级特性
- 无需生成代码或进行 XML 配置

Spring Boot 的四大核心

- 自动配置 - Auto Configuration
- 起步依赖 - Starter Dependency
- 命令行界面 - Spring Boot CLI
- Actuator

内容提要

- Spring Boot组成
- 自动配置的实现原理
- 起步依赖starter的实现原理
- 配置加载机制
- Actuator Endpoint
- Spring Boot Admin监控
- 定制 Web 容器
- 打包 Docker 镜像

自动配置

Chapter_2/datasource-demo

- 自动配置
  - 基于添加的 **JAR** 依赖自动对 **Spring Boot** 应用程序进行配置
  - **spring-boot-autoconfiguration**
- 开启自动配置
  - **@EnableAutoConfiguration**
    - **exclude = Class<?>[]**
  - **@SpringBootApplication**

自动配置的实现原理

- @EnableAutoConfiguration
  - **AutoConfigurationImportSelector**
  - **META-INF/spring.factories**
    - **org.springframework.boot.autoconfigure.EnableAutoConfiguration**

自动配置的实现原理

条件注解

- @Conditional
- @ConditionalOnClass
- @ConditionalOnBean
- @ConditionalOnMissingBean
- @ConditionalOnProperty
- ……

了解自动配置的情况

- 观察自动配置的判断结果
  - --debug
- ConditionEvaluationReportLoggingListener
  - Positive matches
  - Negative matches
  - Exclusions
  - Unconditional classes

实现自己的自动配置

- 主要工作内容

`javaee-spring-boot-autoconfigure`

  - 编写**Java Config**
    - **@Configuration**
  - 添加条件
    - **@Conditional**
  - 定位自动配置
    - **META-INF/spring.factories**

内容提要

- Spring Boot组成
- 自动配置的实现原理
- 起步依赖starter的实现原理
- 配置加载机制
- Actuator Endpoint
- Spring Boot Admin监控
- 定制 Web 容器
- 打包 Docker 镜像

Spring Boot 的起步依赖

- Starter Dependencies
  - 直接面向功能
  - 一站获得所有相关依赖，不再复制粘贴
- 官方的 Starters
  - **spring-boot-starter-***

定制自己的起步依赖

- 主要内容
  - **autoconfigure** 模块，包含自动配置代码
  - **starter** 模块，包含指向自动配置模块的依赖及其他相关依赖
- 命名方式
  - **xxx-spring-boot-autoconfigure**
  - **xxx-spring-boot-starter**

`javaee-spring-boot-autoconfigure`

`javaee-spring-boot-starter`

`autoconfigure-starter-demo`

内容提要

- Spring Boot组成
- 自动配置的实现原理
- 起步依赖starter的实现原理
- 配置加载机制
- Actuator Endpoint
- Spring Boot Admin监控
- 定制 Web 容器
- 打包 Docker 镜像

外化配置加载

- 命令行参数（--server.port=9000)
- System.getProperties()
- 操作系统环境变量
- application.properties或yml
  - jar 包外部的 application-{profile}.properties 或 .yml
  - jar 包内部的 application-{profile}.properties 或 .yml
  - jar 包外部的 application.properties 或 .yml
  - jar 包内部的 application.properties 或 .yml

获取配置属性值

property-source-demo

- 程序中读取配置值
  - @Value
- Run/Debug Configurations
  - VM Options：-Dspring.profiles.active=prod
- 命令行参数
  - java -jar xxx.jar --spring.profiles.active=prod

application.properties默认位置

默认位置

- ./config
- ./
- CLASSPATH 中的 /config
- CLASSPATH 中的 /

内容提要

- Spring Boot组成
- 自动配置的实现原理
- 起步依赖starter的实现原理
- 配置加载机制
- Actuator Endpoint
- Spring Boot Admin监控
- 定制 Web 容器
- 打包 Docker 镜像

Actuator

- 目的
  - 监控并管理理应用用程序
- 访问方式
  - **HTTP**
  - **JMX**
- 依赖
  - **spring-boot-starter-actuator**

一些常用 Endpoint

ID	说明	默认开启	默认 HTTP	默认 JMX
beans	显示容器中的 Bean 列表	Y	N	Y
caches	显示应用中的缓存	Y	N	Y
conditions	显示配置条件的计算情况	Y	N	Y
configprops	显示 @ConfigurationProperties 的信息	Y	N	Y
env	显示 ConfigurableEnvironment 中的属性	Y	N	Y
health	显示健康检查信息	Y	Y	Y
httptrace	显示 HTTP Trace 信息	Y	N	Y
info	显示设置好的应用信息	Y	Y	Y

一些常用 Endpoint

ID	说明	默认开启	默认 HTTP	默认 JMX
loggers	显示并更新日志配置	Y	N	Y
metrics	显示应用的度量信息	Y	N	Y
mappings	显示所有的 @RequestMapping 信息	Y	N	Y
scheduledtasks	显示应用的调度任务信息	Y	N	Y
shutdown	优雅地关闭应用程序	N	N	Y
threaddump	执行 Thread Dump	Y	N	Y
heapdump	返回 Heap Dump 文件，格式为 HPROF	Y	N	N/A
prometheus	返回可供 Prometheus 抓取的信息	Y	N	N/A

如何访问 Actuator Endpoint

HTTP 访问

- /actuator/<id>

端口与路径

- management.server.address=
- management.server.port=
- management.endpoints.web.base-path=/actuator
- management.endpoints.web.path-mapping.<id>=路径

如何访问 Actuator Endpoint

开启 Endpoint

- management.endpoint.<id>.enabled=true
- management.endpoints.enabled-by-default=false

暴露 Endpoint

- management.endpoints.jmx.exposure.exclude=
- management.endpoints.jmx.exposure.include=*
- management.endpoints.web.exposure.exclude=
- management.endpoints.web.exposure.include=info, health

内容提要

- Spring Boot组成
- 自动配置的实现原理
- 起步依赖starter的实现原理
- 配置加载机制
- Actuator Endpoint
- Spring Boot Admin监控
- 定制 Web 容器
- 打包 Docker 镜像

Spring Boot Admin

- 目的
  - 为**Spring Boot**应用程序提供一套可视化管理界面
- 主要功能
  - 集中展示应用程序**Actuator**相关的内容
  - 变更通知

快速上手

服务端

- de.codecentric:spring-boot-admin-starter-server:2.1.3
- `@EnableAdminServer`

客户端

- de.codecentric:spring-boot-admin-starter-client:2.1.3
- 配置服务端及Endpoint
  - `spring.boot.admin.client.url=http://localhost:8080`
  - `management.endpoints.web.exposure.include=*`

安全控制

Spring_Bucks/spring-boot-admin

Spring_Bucks/waiter-service-0

安全相关依赖

- spring-boot-starter-security

服务端配置

- spring.security.user.name
- spring.security.user.password

客户端配置

- spring.boot.admin.client.username
- spring.boot.admin.client.password
- spring.boot.admin.client.instance.metadata.user.name
- spring.boot.admin.client.instance.metadata.user.password

内容提要

- Spring Boot组成
- 自动配置的实现原理
- 起步依赖starter的实现原理
- 配置加载机制
- Actuator Endpoint
- Spring Boot Admin监控
- 定制 Web 容器
- 打包 Docker 镜像

内嵌 Web 容器

- 可选容器列表
  - **spring-boot-starter-tomcat**
  - **spring-boot-starter-jetty**
  - **spring-boot-starter-undertow**
  - **spring-boot-starter-reactor-netty**

修改容器配置

- 端口
  - **server.port**
  - **server.address**
- 压缩
  - **server.compression.enabled**
  - **server.compression.min-response-size**
  - **server.compression.mime-types**

Tomcat 特定配置

- server.tomcat.max-connections=10000
- server.tomcat.max-http-post-size=2MB
- server.tomcat.max-swallow-size=2MB
- server.tomcat.max-threads=200
- server.tomcat.min-spare-threads=10

修改容器配置

- 错误处理理
  - server.error.path=/error
  - server.error.include-exception=false
  - server.error.include-stacktrace=never
  - server.error.whitelabel.enabled=true
- 其他
  - server.use-forward-headers
  - server.servlet.session.timeout

内容提要

- Spring Boot组成
- 自动配置的实现原理
- 起步依赖starter的实现原理
- 配置加载机制
- Actuator Endpoint
- Spring Boot Admin监控
- 定制 Web 容器
- 打包 Docker 镜像

什么是 Docker 镜像

- 镜像是静态的只读模板
- 镜像中包含构建 Docker 容器的指令
- 镜像是分层的
- 通过 Dockerfile 来创建镜像

Dockerfile

指令	作用	格式举例
FROM	基于哪个镜像	FROM <image>[:<tag>] [AS <name>]
LABEL	设置标签	LABEL maintainer=“Geektime"
RUN	运行安装命令	RUN ["executable", "param1", "param2"]
CMD	容器启动时的命令	CMD ["executable","param1","param2"]
ENTRYPOINT	容器启动后的命令	ENTRYPOINT ["executable", "param1", "param2"]
VOLUME	挂载目录	VOLUME ["/data"]
EXPOSE	容器要监听的端口	EXPOSE <port> [<port>/<protocol>...]
ENV	设置环境变量	ENV <key> <value>
ADD	添加文件	ADD [--chown=<user>:<group>] <src>... <dest>
WORKDIR	设置运行的工作目录	WORKDIR /path/to/workdir
USER	设置运行的用户	USER <user>[:<group>]

通过 Maven 构建 Docker 镜像

- 准备工作
  - 提供一个 **Dockerfile**
  - 配置 **dockerfile-maven-plugin** 插件
- 执行构建
  - **mvn package**
  - **mvn dockerfile:build**
- 检查结果
  - **docker images**

dockerfile-maven-plugin

```
<plugin>
  <groupId>com.spotify</groupId>
  <artifactId>dockerfile-maven-plugin</artifactId>
  <version>${dockerfile-maven-version}</version>
  <executions>
    <execution>
      <id>default</id>
      <goals>
        <goal>build</goal>
        <goal>push</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <repository>[illegible]</repository>
    <tag>${project.version}</tag>
    <buildArgs>
      <JAR_FILE>${project.build.finalName}.jar</JAR_FILE>
    </buildArgs>
  </configuration>
</plugin>
```