Deployment and Production Considerations

Development Tools

Spring Boot Devtools

Spring Boot Devtools

- `Add spring-boot-devtools` *like a starter* to your project
- Do a clean build of your project to enable it

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <optional>true</optional>
    <scope>runtime</scope>
</dependency>
```

Property Defaults

- Spring Boot enables a lot of defaults e.g. caching, but during development immediate feedback is needed
- Defaults could be overridden manually with properties e.g. `spring.thymeleaf.cache=false`
- `spring-boot-devtools` overrides many properties defaults automatically

Automatic Restarts

- File changes in classpath restart the application context
  - Not `main` but `restartedMain` Thread is restarted
- Changes on static files and view templates do not need a restart
- Reducing time considerably to verify changes
  - Code and configuration

Live Reload

Live Reload

- Activates *embedded LiveReload* server
- Triggers browser refresh when the resource is changed
- The browser needs a LiveReload plugin
- Only one LiveReload server can be started. If multiple Applications are running, only first has LiveReload support.

Chrome Plugin: https://chrome.google.com/webstore/detail/remotelivereload/jlppknnillhjgiengoigajegdpieppei/

Firefox Plugin: https://addons.mozilla.org/en-US/firefox/addon/livereload-web-extension/

Global Settings

- Add `.config/spring-boot/.spring-boot-devtools.properties` to `$HOME`
  - Note that the filename starts with "."
- All properties added to the dot file apply to all Spring Boot applications which use Devtools.

```
# application.properties
spring.devtools.restart.enabled=false
spring.devtools.restart.trigger-file=.reloadtrigger
spring.devtools.livereload.enabled=false
```

See also: https://docs.spring.io/spring-boot/docs/current/reference/html/using.html#using.devtools.globalsettings

Remote Applications

- Running remote client application
- Remote update via HTTP
- Remote Java debugging via HTTP
- Remote debug tunnel via HTTP

See also: https://docs.spring.io/spring-boot/docs/current/reference/html/using.html#using.devtools.remote-applications

Deployment Topics

Packaging

Packaging as a JAR

- Default artifact
- Maven (using `spring-boot-starter parent`)

```
<packaging>jar</packaging>
…
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
```

```
mvn package
java -jar demo.jar
```

Packaging as WAR

- Maven plugin (using `spring-boot-starter` parent)
- Deployable in Servlet Container (Jetty, Tomcat,...) and executable from the command line

```
<packaging>war</packaging>
…
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
```

```
mvn package
java -jar demo.war
```

Hybrid Apps – Running as a JAR or WAR

- Extend the application launcher class with `SpringBootServletInitializer`
- Runnable on CLI or with Tomcat/Jetty/UnderTow

```
@SpringBootApplication
public class DemoApplication extends SpringBootServletInitializer {

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

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

paketo

buildpacks

Buildpacks FTW!

- Spring Boot plugin uses "Build Packs" during the `build-image` task. It detects the Spring Boot App and optimizes created container:
- Optimizes the runtime by:
  - Extracting the fat jar into exploded form.
  - Calculates and applies resource runtime tuning at container startup.
- Optimized the container image:
  - Adds layer from build pack, spring boot, ...
  - Subsequent builds are faster, they only build and add layers for the changed code.

See also: https://buildpacks.io/

Packaging as Container

- The plugin can create an OCI image using Cloud Native Buildpacks. Images can be built using the `build-image` goal.

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

```
mvn spring-boot:build-image
```

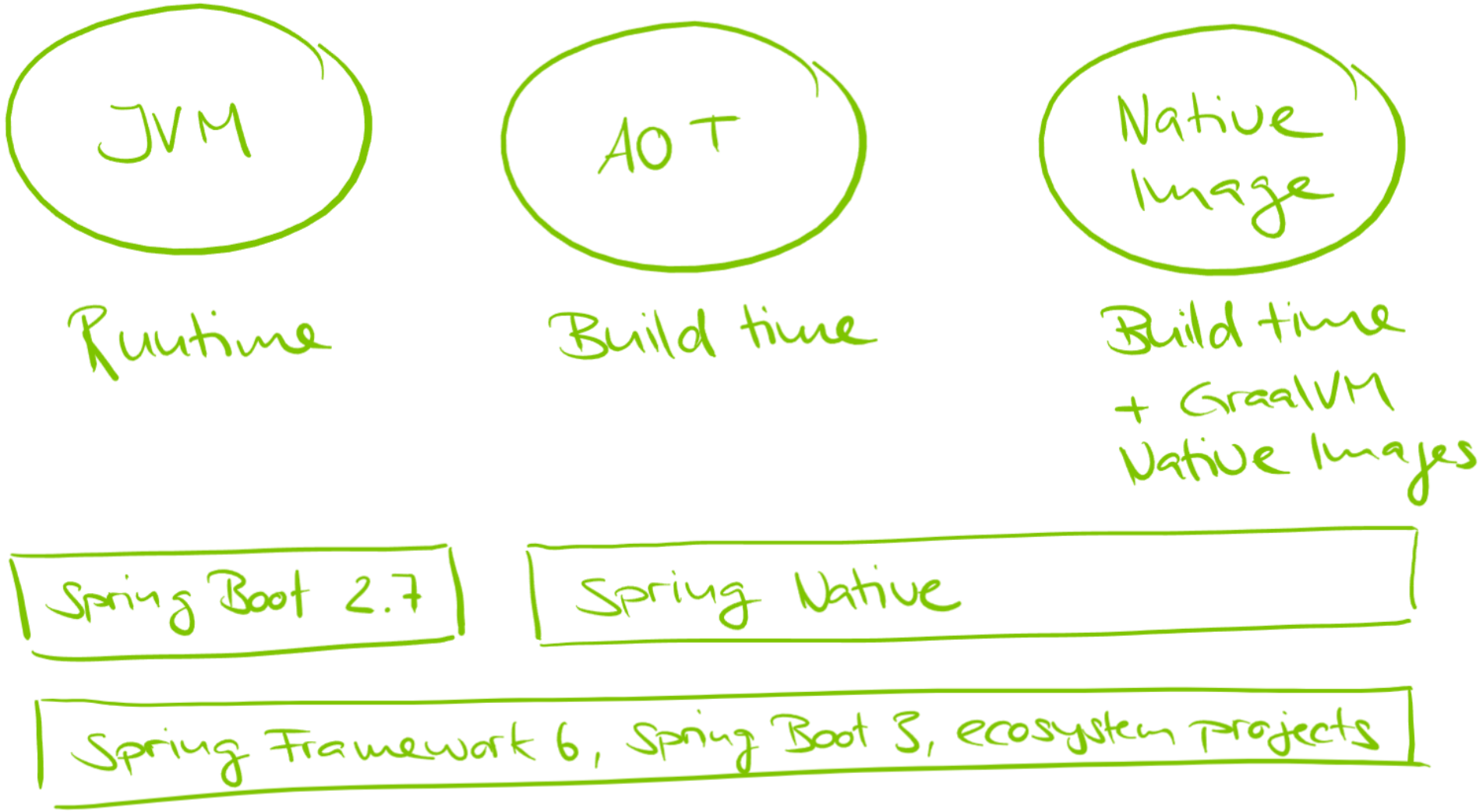
JLink

- The plugin can create a custom runtime image using the `jlink` tool.

```
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
    <configuration>
        <image>
            <env>
                <BP_JVM_JLINK_ENABLED>true</BP_JVM_JLINK_ENABLED>
            </env>
        </image>
    </configuration>
</plugin>
```

```
mvn spring-boot:build-image
```

Native Executables with GraalVM


JVM
AOT
Native Image
Runtime
Build time
Build time + GraalVM Native Images
Spring Boot 2.7
Spring Native
Spring Framework 6, Spring Boot 3, ecosystem projects

Maven Plugins

```
  <build>
    <plugins>
      <plugin>
        <groupId>org.graalvm.buildtools</groupId>
        <artifactId>native-maven-plugin</artifactId>
      </plugin>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
```

```
mvn -Pnative spring-boot:build-image
```

Resource Hints

```
public class ApplicationRuntimeHints implements RuntimeHintsRegistrar {

  @Override
  public void registerHints(RuntimeHints hints, ClassLoader classLoader) {
    hints.resources().registerPattern("db/*");
    hints.resources().registerPattern("messages/*");
    hints.resources().registerPattern("META-INF/resources/webjars/*");
  }

}
```

https://github.com/spring-projects/spring-boot/issues/32654

Logging

Logging

- Spring Boot provides default configuration files for four logging frameworks: Logback, Log4j, Log4j2 and java.util.Logging
- Starters use Logback with color output
- Default log level set to INFO
- Debug output can be easily enabled using `--debug` and `logging.level.com.acme=DEBUG`
- Log to console (`stdout` and `stderr`) by default
- For custom Logging in APP use SLF4j API
- `logging.file.name` and `logging.file.path` property to enable file

```
# application.properties
logging.level.com.example=DEBUG
```

More on Properties

YAML instead of Properties

- SnakeYAML dependency is added by Spring Boot Starter
- File extension `.yml`
- Replacement for `application.properties`
- `@PropertySource` does not support yaml
- Consider Tabs vs. Spaces

```yaml
# application.yml
spring:
 application:
   name: DemoApplication
server:
 port: 9000
```

Properties from CLI Arguments

- Java Style

```
java -Dproperty.name="value" -jar demo.jar
```

- Spring Boot Style

```
java -jar demo.jar --property.name="value"
```

Properties from Environment Variables

- Environments variables are available as properties
- Automatically binds properties in `camelCase`, `kebab-case`, `snake_case`, `SCREEMING_SNAKE_CASE`
- Relaxed conversion from uppercase to java style
    - E.g. `JAVA_HOME` -> `@Value("${java.home}")`

Remapping Property Values

- RandomValuePropertySource can randomize values

```
my.random.number=${random.int}
my.random.long=${random.long}
my.random.uuid=${random.uuid}
```

- Concatenating property values

```
host=localhost:8080
app.uri=/app
app.url=https://${host}${app.uri}
```

Property Sources

1. Devtools global settings properties
2. `@TestPropertySource`
3. `@SpringBootTest(properties = "…")`
4. Command line arguments
5. `SPRING_APPLICATION_JSON` (inline JSON)
6. `ServletConfig` init parameters
7. `ServletContext` init parameters
8. JNDI attributes from `java:comp/env`
9. Java System properties
10. OS environment variables
11. `RandomValuePropertySource`
12. Config data (`application.properties`, YAML, etc.)
13. `@PropertySource`
14. `SpringApplication.setDefaultProperties()`

See also: https://docs.spring.io/spring-boot/docs/current/reference/html/features.html#features.external-config

Profiles

@Profile

- Used to limit the availability of `@Bean`,`@Component` or `@Configuration`
- With one or more parameters
  - `@Profile("prod")` or `@Profile({"dev", "prod"})`
- Negation possible
  - `@Profile("!dev")`

```
@Bean
@Profile("dev")
public DevEmployeeService employeeService() { … }
```

```
@Service
@Profile("dev")
public class DevEmployeeService implements EmployeeService { … }
```

```
@Profile("!dev")
@Configuration
public class PrdConfiguration{ … }
```

Selecting Profiles

- Command Line

```
java -Dspring.profiles.active=local,jdbc –jar demo.jar
```

- Environment Variables

```
export SPRING_PROFILES_ACTIVE=local,jdbc
```

- Property File (`application.properties` or `application.yml`)

```
# application.properties
spring.profiles.active=local,jdbc
```

Selecting Profiles (2)

- Tests

```
@ActiveProfiles({"local","jdbc"})
```

- Programmatically setting profile

```
public static void main(String[]args) {
    SpringApplication app = new SpringApplication(DemoApplication.class);
    app.setAdditionalProfiles("local","jdbc");
}
```

Profile Specific Configurations

- `application-{profile}.properties`
- Loaded the from the same location as `application.properties`
- Will override default `application.properties`

```
# application-local.properties
db.url=jdbc:hsqldb:file:configurations
db.driver=org.hsqldb.jdbcDriver
db.username=sa
db.password=
```

```
# application-staging.properties
db.url=jdbc:oracle:thin:@//localhost:1521/configurations
db.driver=oracle.jdbc.driver.OracleDriver
db.username=<username>
db.password=<password>
```

YAML and Profiles

- A YAML file might contain more than one profile
- Default profile not named

```yaml
# default profile
server:
 port: 9000
---
spring:
  config:
    activate:
        on-profile: "prod"
server:
  port: 8000
---
spring:
  config:
    activate:
      on-profile: "staging"
server:
  port: 8080
```