Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

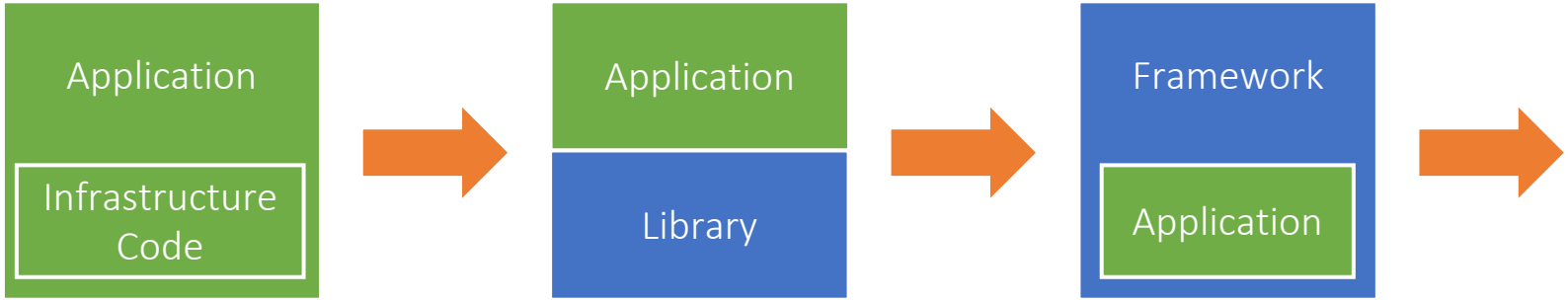
Spring and Spring Boot

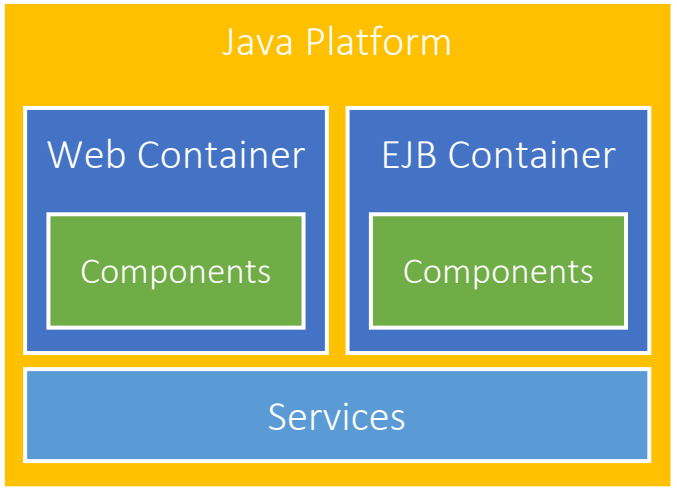
CAS Spring Application Development

Simon Martinelli, simon.martinelli@bfh.ch, v2025.10

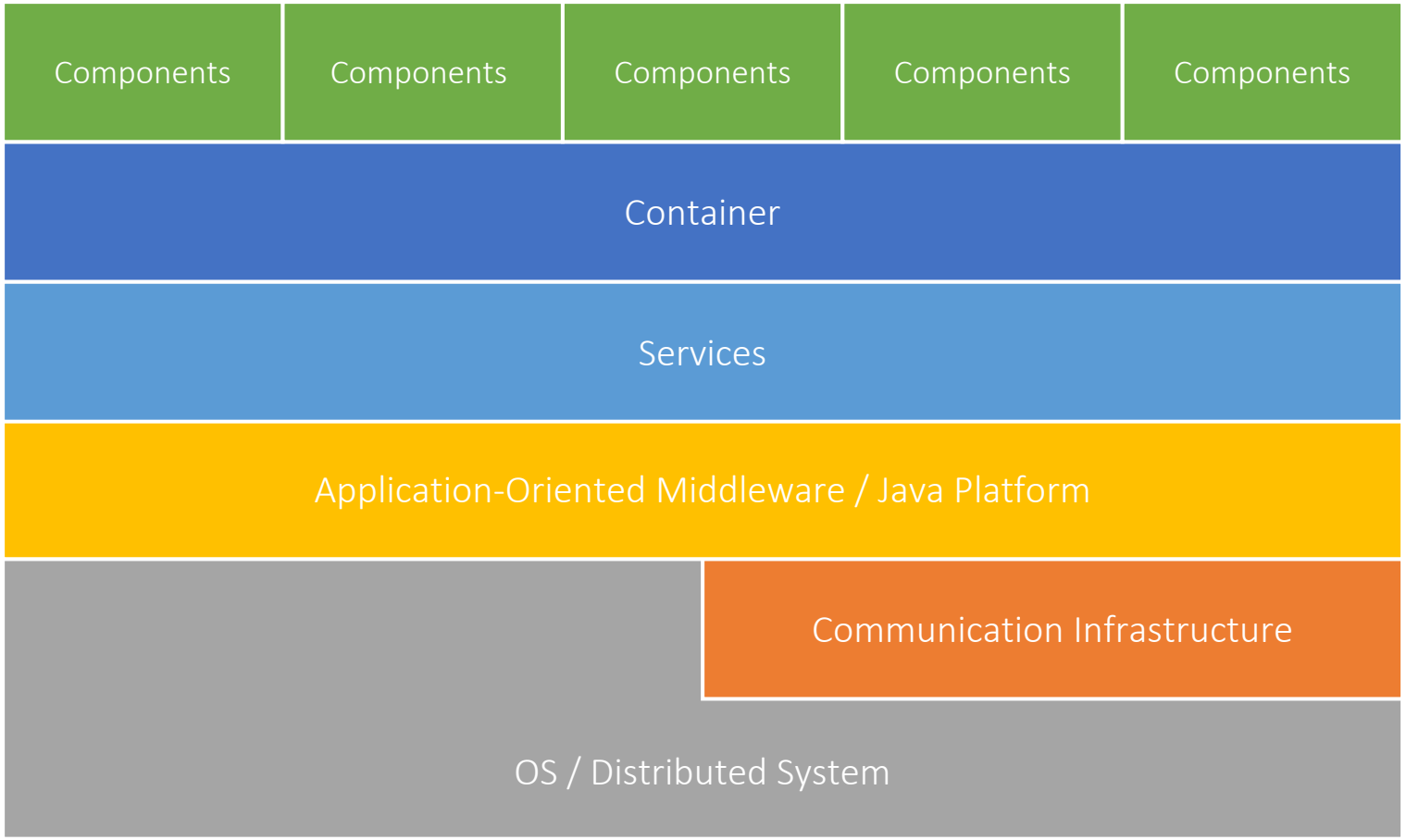
Overview

From Application to Platform

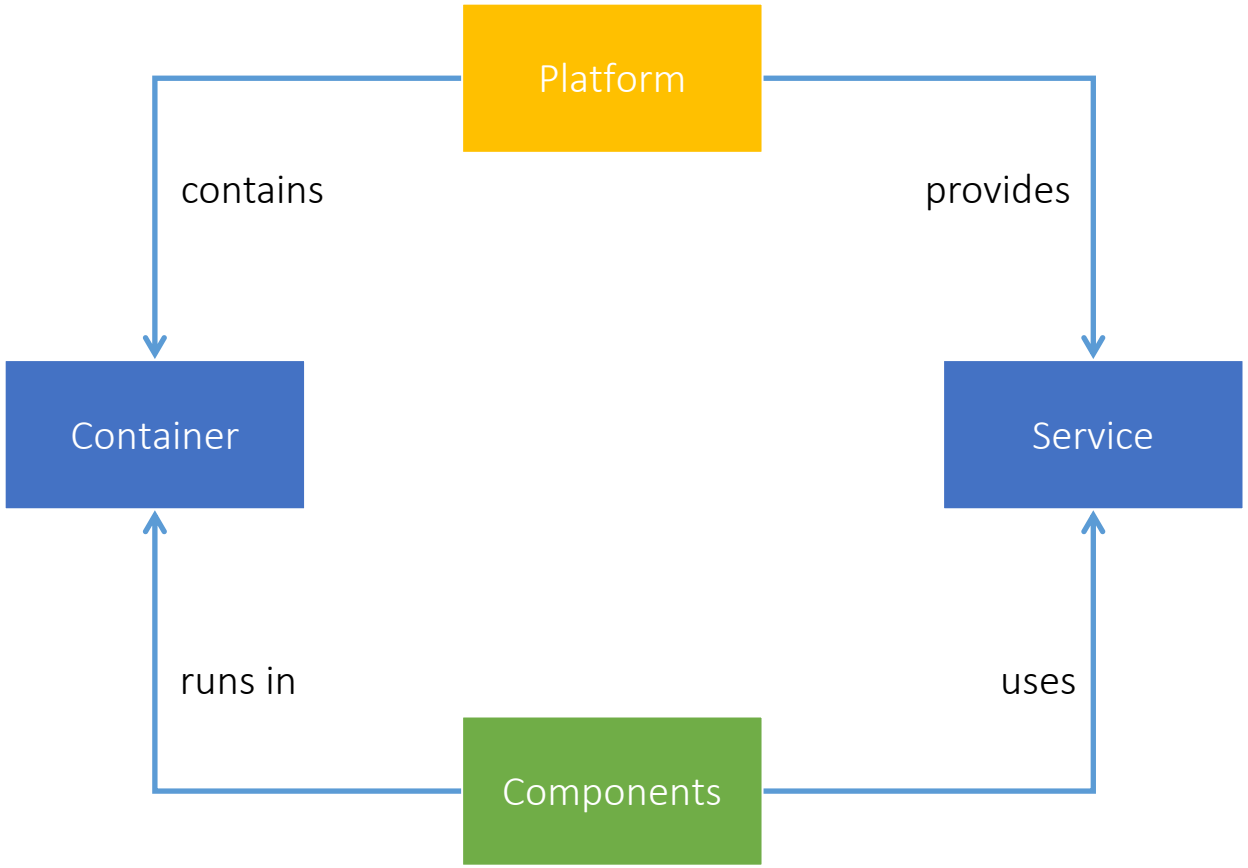




Platform


Components
Components
Components
Components
Components
Container
Services
Application-Oriented Middleware / Java Platform
Communication Infrastructure
OS / Distributed System

Container, Services and Components



Principles and Patterns

1. Injection and Factory
2. Context
3. Proxy
4. Interceptors

Don’t Call Us – We Call You!

Martin Fowler, 2004
http://martinfowler.com/articles/injection.html

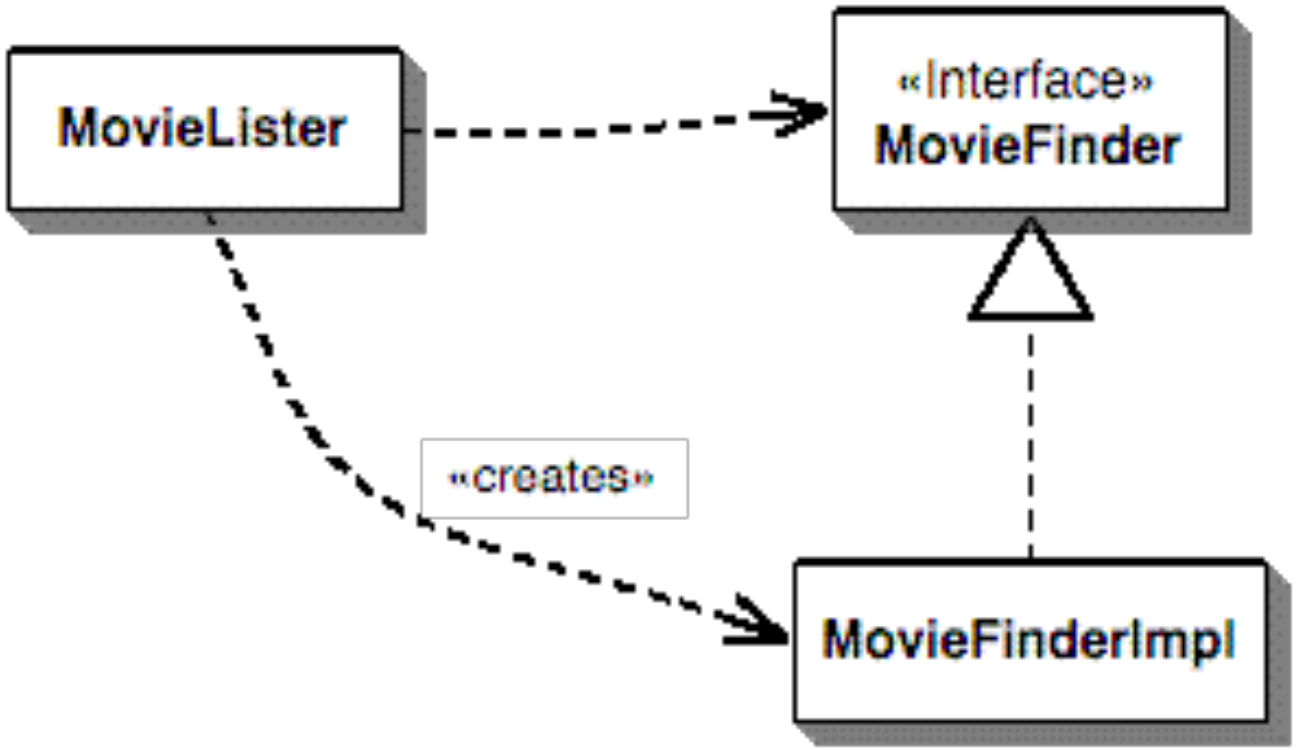
Background

The question is, what aspect of control are [they] inverting?

Martin Fowler asked this question about Inversion of Control (IoC) in 2004 on his website

Fowler proposed to rename the Principle “Dependency Injection”.
So that it is self-explanatory

A Naive Approach



Traditional Method

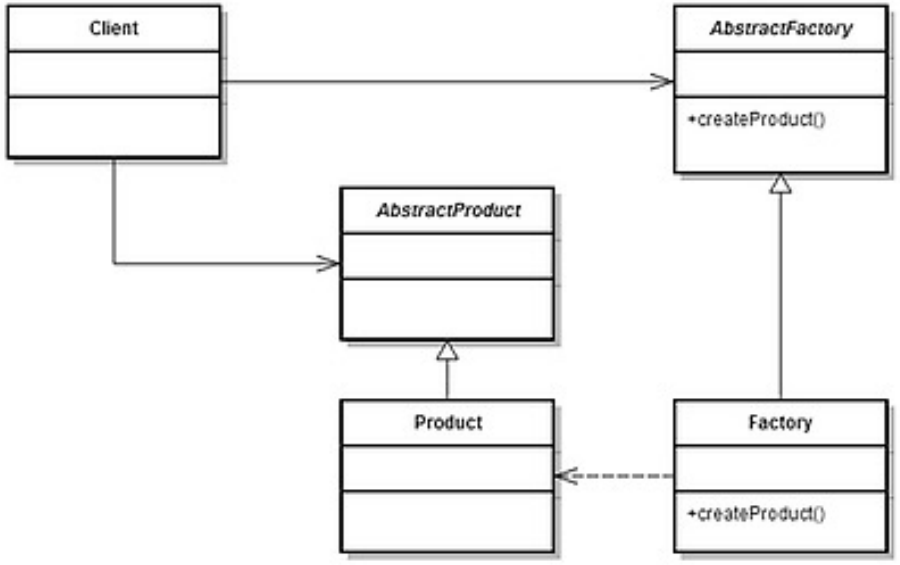
```
private MovieFinder finder;

public MovieLister() {
    finder = new ColonDelimitedMovieFinder("movies1.txt");
}
```

Problems

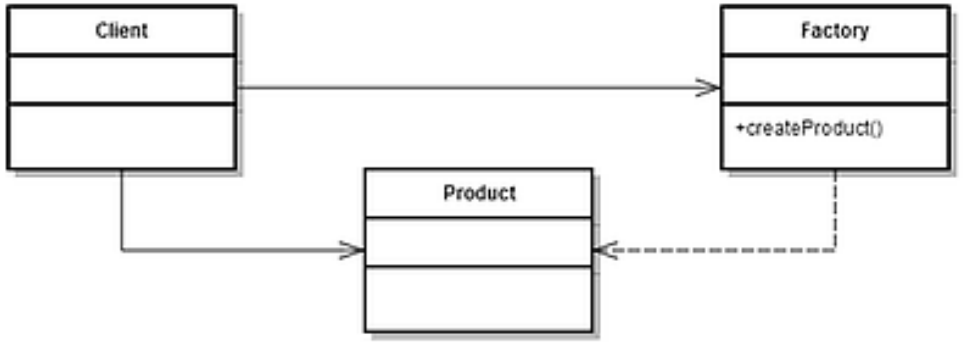
- Strong coupling
- Hard to extend

Factory Patterns

Abstract Factory



Factory Method



TODO HINZUFÜGEN

https://www.baeldung.com/java-factory-pattern

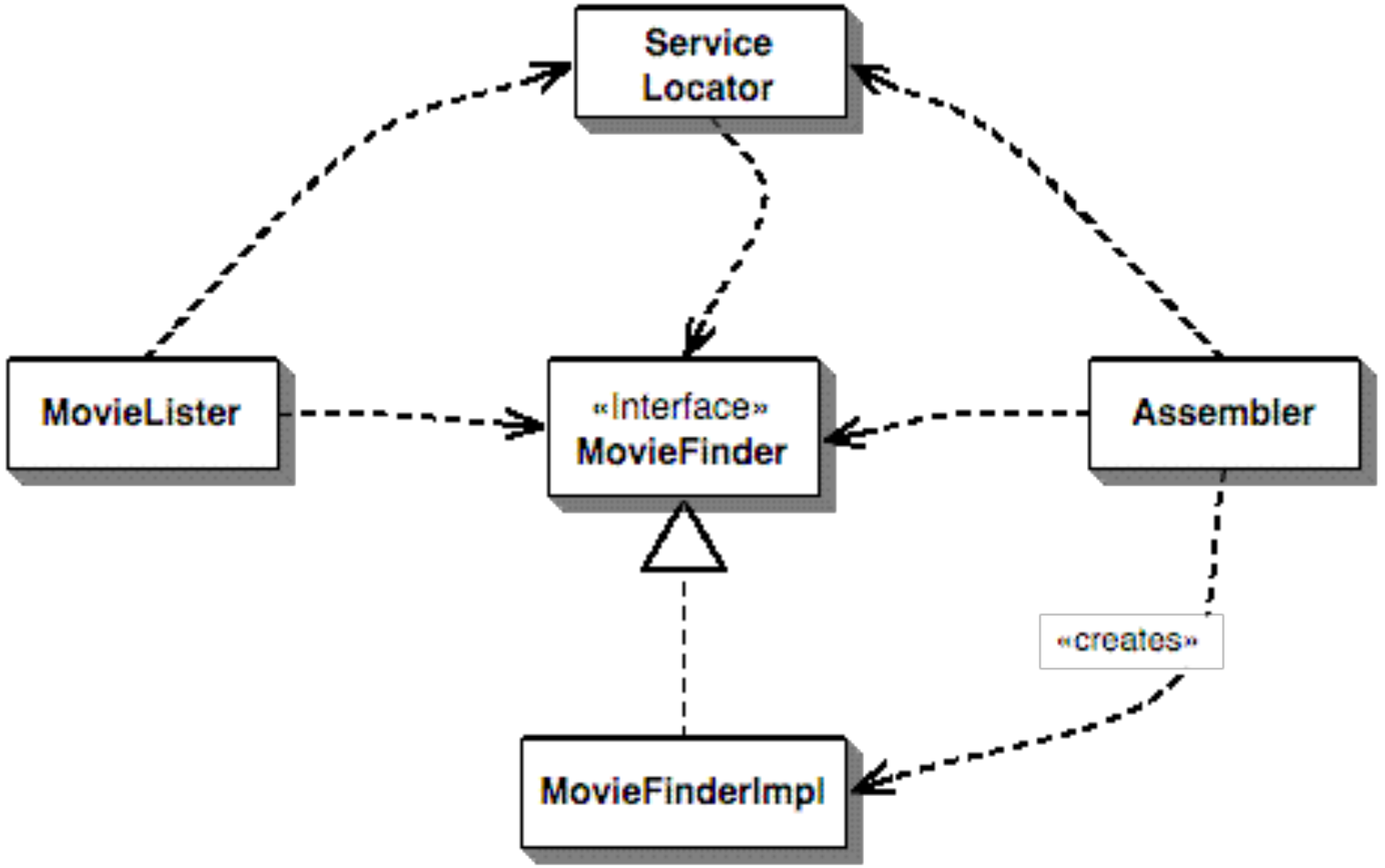
Solution: Factory Method

```
public class MoveFinderFactory() {

    public static MovieFinder createMovieFinder(String f) {
        return new ColonDelimitedMovieFinder(f);
    }
}
```

Problems

- Unnecessary code
- Problem moved to the factory
- How is the factory configured?

Solution: Service Locator



Solution: Dependency Injection

Don't call us - we call you! (Hollywood Principle) = Inversion of Control

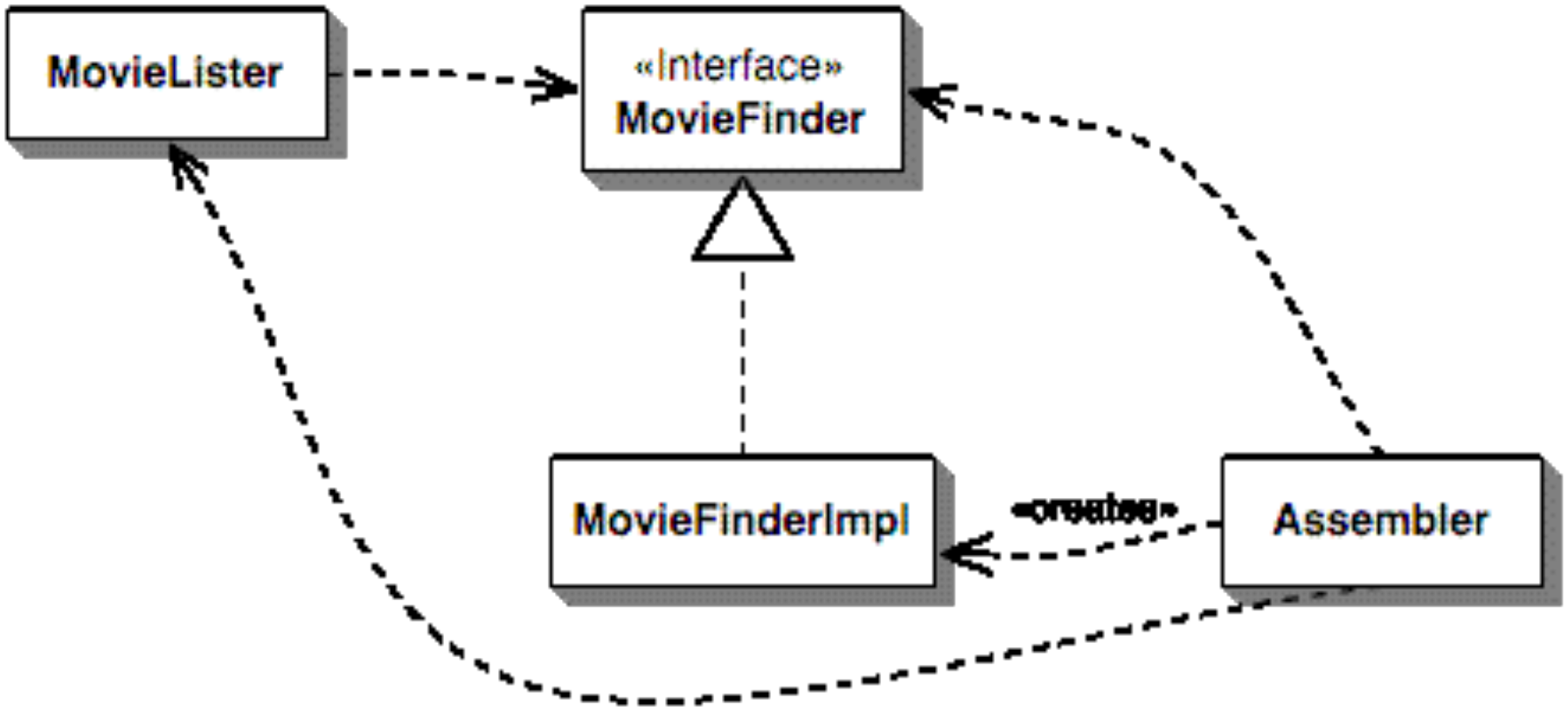
Relationships are set from the outside

- Constructor
- Field
- Setter

Frameworks

- Java EE
- Spring
- Google Guice

Assembler


MovieLister
«Interface»
MovieFinder
MovieFinderImpl
«creates»
Assembler

Metadata

Annotations

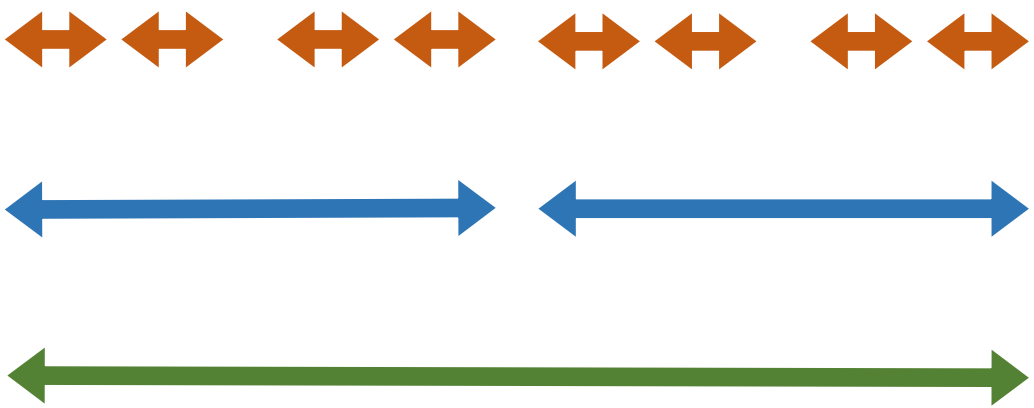
- Starting from Java 5
- Complier checked
- Business code “contamination"

XML

- No recompile for configuration changes
- Prone to errors (mostly IDE checked)

Context and Scopes

The life cycle and the interactions of stateful components are linked to well-defined but extensible contexts



Scopes

Request - A user interaction with a web application with an HTTP request or retrieval of a stateless session bean

Session - A user interaction with a web application through multiple HTTP requests or all requests for a stateful session bean

Application/Singleton - Shared state of all user interactions of a web application or all requests for a singleton session bean

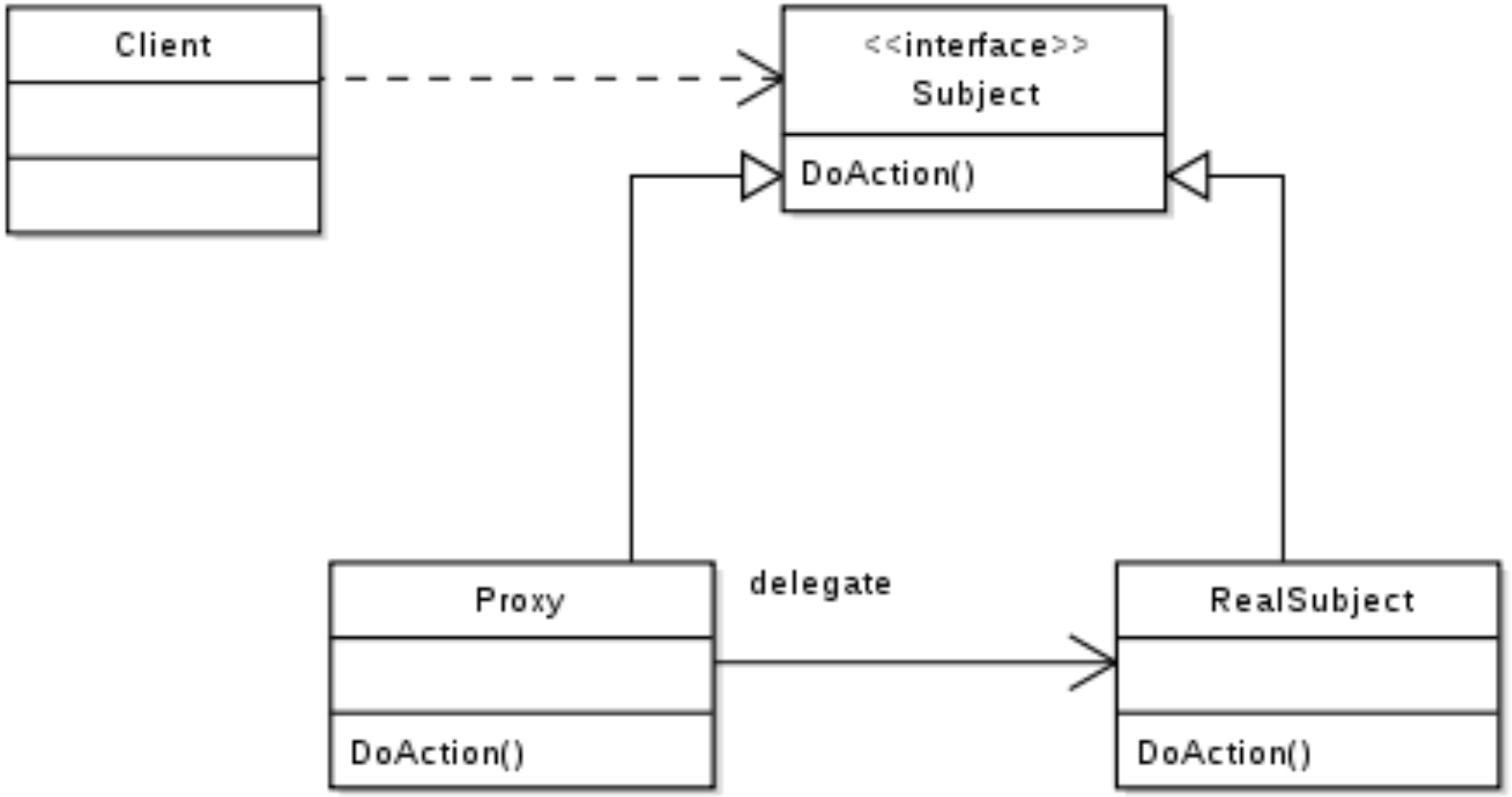
Scope Management

The scope gives an object a well-defined lifecycle context

An object is automatically created during first use

The object is automatically destroyed when the context ends

Proxy


Client
<<interface>>
Subject
DoAction()
Proxy
DoAction()
delegate
RealSubject
DoAction()

Proxy

A proxy in its most general form is a class that acts as an interface to another “subject"

This subject may, for example, be a network connection, a large object in the memory, a file or another resource

As the representative of this subject the proxy can control the generation of the subject as well as access to it

Proxies can also be used to prevent further operations on actual access to the object. The object thus remains independent of these operations

Proxy Scenarios

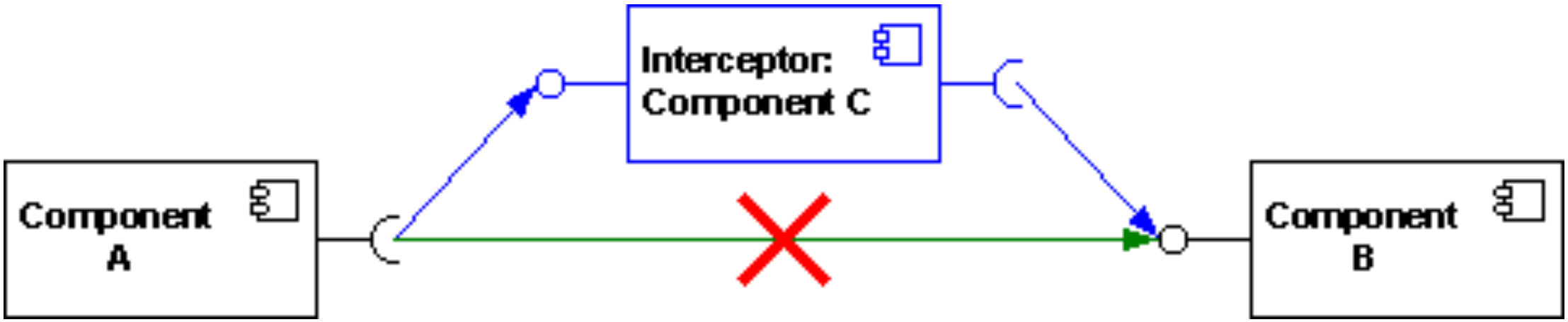
Virtual Proxy
Imagine a situation where there is a multiple database call to extract a very large image. Since this is an expensive operation, we could potentially use the proxy pattern which would create multiple proxies and highlight the large, memory-consuming object for further processing. The real object is only created when a client first requests/accesses the object and after that we can just refer to the proxy to reuse the object. This avoids duplication of the object, hence saving memory.

Remote Proxy
A remote proxy can be thought about as a stub in the RPC call. The remote proxy provides a local representation of the object which is present in the different address location. Another example can be providing an interface for remote resources such as web service or REST resources.

Protective Proxy
The protective proxy acts as an authorization layer to verify whether the actual user has access to appropriate content. An example can be thought about the proxy server which provides restrictive internet access in office. Only the websites and content which are valid will be allowed and the remaining ones will be blocked.

Smart Proxy
A smart proxy provides an additional layer of security by interposing specific actions when the object is accessed. An example could be to check if the real object is locked before it is accessed to ensure that no other object can change it.

Interceptor



Interceptor

The Interceptor pattern provides the possibility to transparently incorporate functionality into a framework

This is important especially if different applications need to access a framework and need special services, which are not provided directly through the framework

By using the interceptor pattern this functionality does not have to be integrated into the framework

This would help to keep its structure simple without having to omit the extended functionality

AOP - Aspect Oriented Programming

Aspect-Oriented Programming (AOP) is a programming paradigm using generic functionalities across multiple classes → Cross-Cutting Concerns

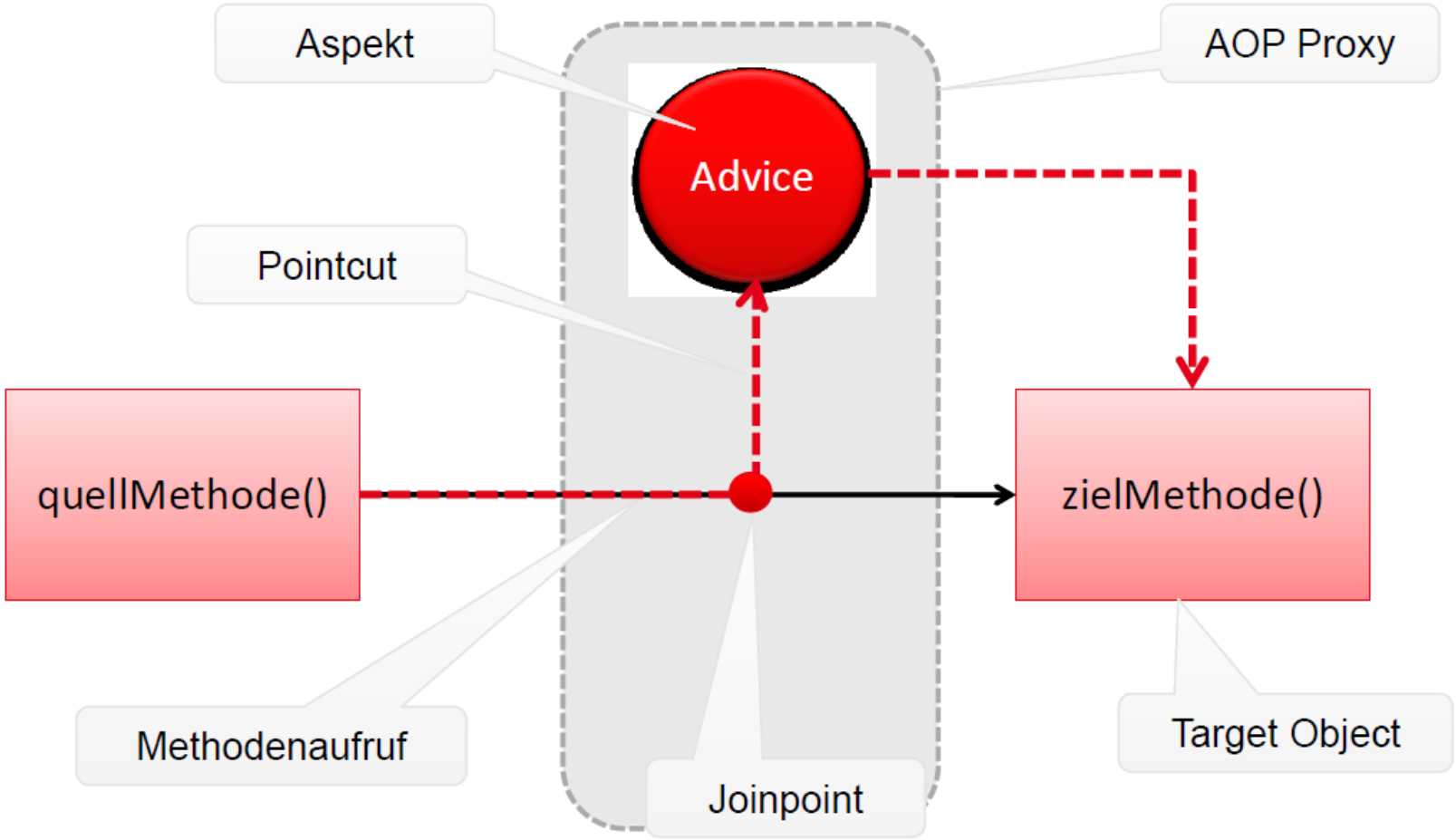
Logical aspects of an application program are thereby separated from actual business logic

Typical examples of applications include transaction management, auditability and logging

The concept of AOP was developed by Gregor Kiczales and his team at the company Xerox PARC

In 2001, the first AOP language AspectJ was also presented there

Advice

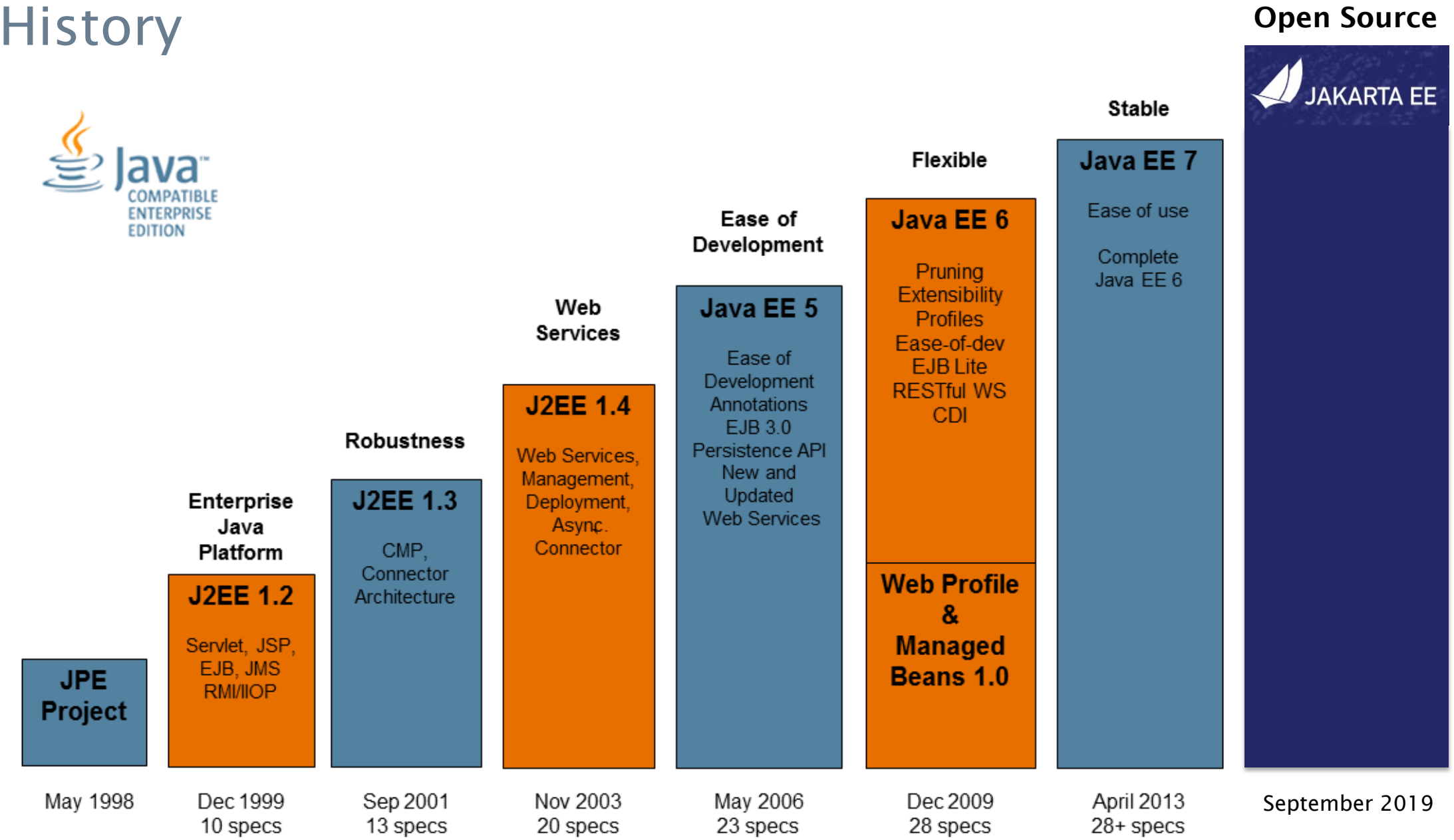

Aspekt
AOP Proxy
Advice
Pointcut
quellMethode()
zielMethode()
Methodenaufruf
Joinpoint
Target Object


YOU MUST CHOOSE A SIDE.....

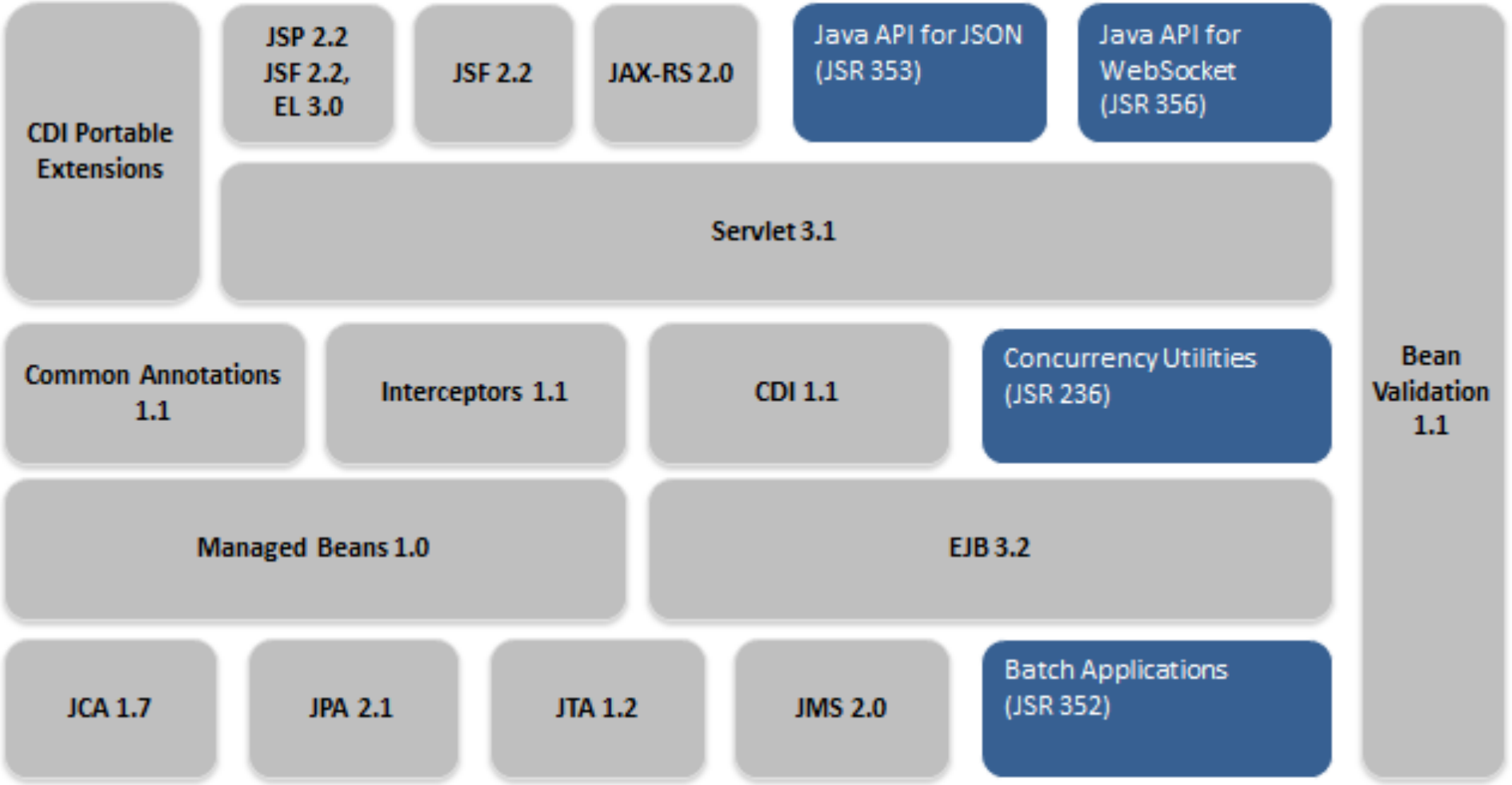
J2EE → Java EE → Jakarta EE

https://jakarta.ee/

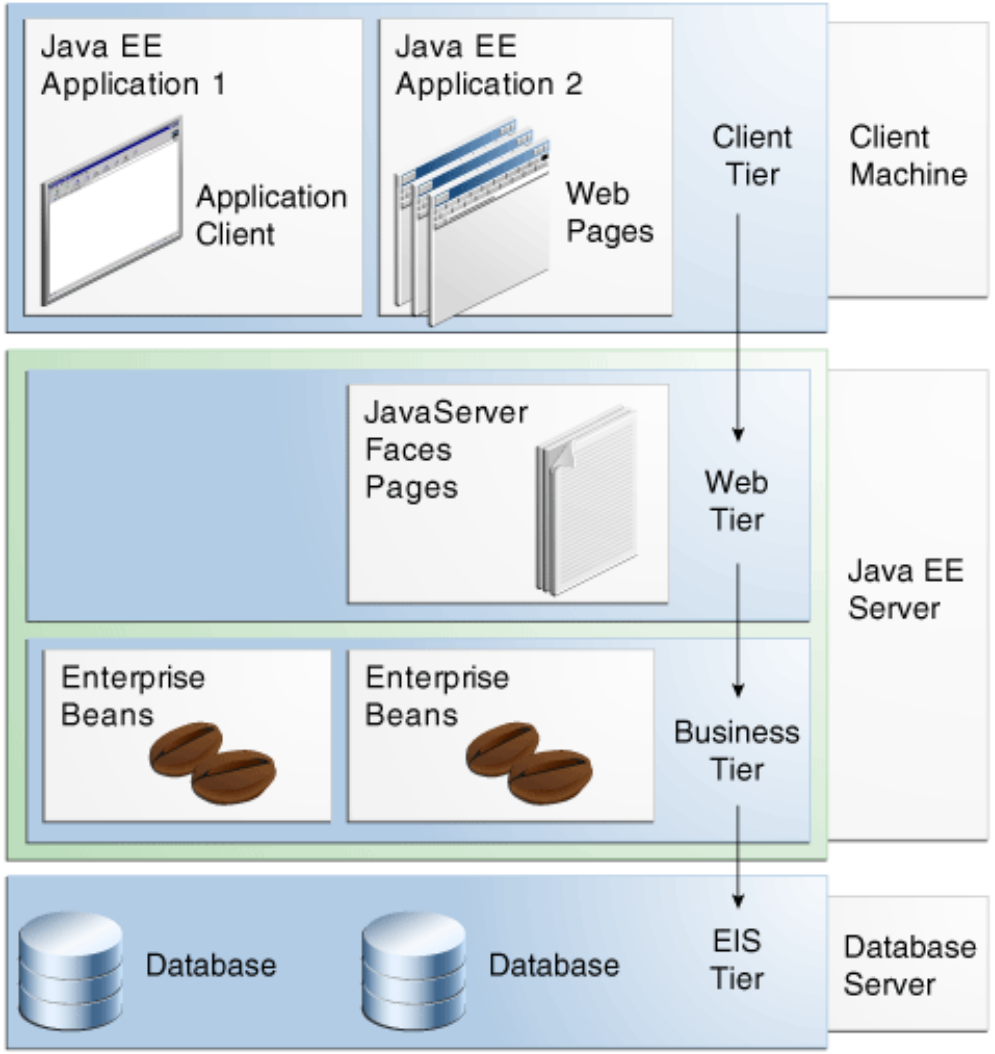
History


Java
COMPATIBLE
ENTERPRISE
EDITION
JPE
Project
May 1998
Enterprise
Java
Platform
J2EE 1.2
Servlet, JSP,
EJB, JMS
RMI/IIOP
Dec 1999
10 specs
Robustness
J2EE 1.3
CMP,
Connector
Architecture
Sep 2001
13 specs
Web
Services
J2EE 1.4
Web Services,
Management,
Deployment,
Async.
Connector
Nov 2003
20 specs
Ease of
Development
Java EE 5
Ease of
Development
Annotations
EJB 3.0
Persistence API
New and
Updated
Web Services
May 2006
23 specs
Flexible
Java EE 6
Pruning
Extensibility
Profiles
Ease-of-dev
EJB Lite
RESTful WS
CDI
Web Profile
&
Managed
Beans 1.0
Dec 2009
28 specs
Stable
Java EE 7
Ease of use
Complete
Java EE 6
April 2013
28+ specs
Open Source
JAKARTA EE
September 2019

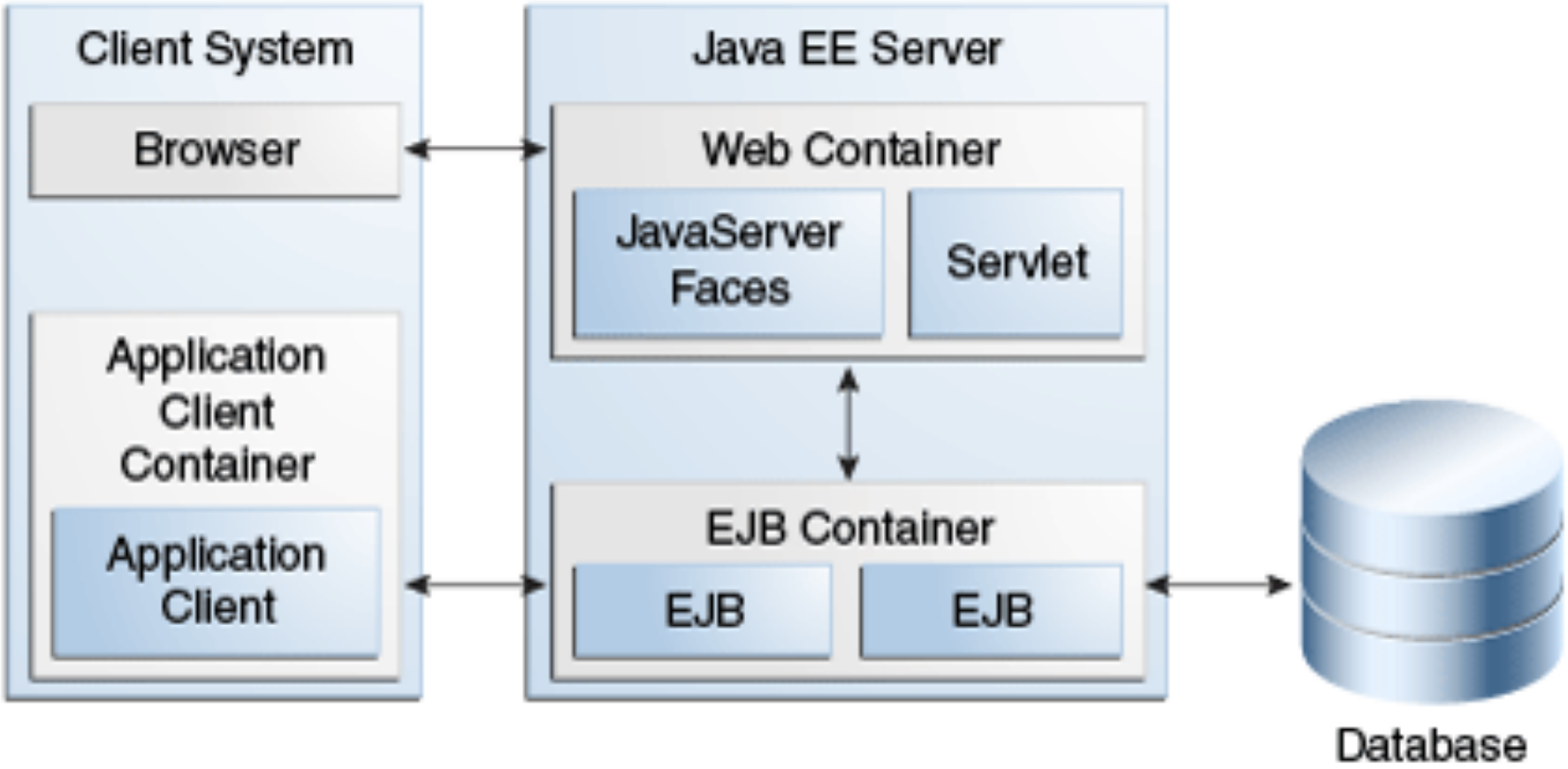
Java EE Specifications



Layers


Java EE
Application 1
Application
Client
Java EE
Application 2
Web
Pages
Client
Tier
Client
Machine
JavaServer
Faces
Pages
Web
Tier
Java EE
Server
Enterprise
Beans
Enterprise
Beans
Business
Tier
Database
Database
EIS
Tier
Database
Server

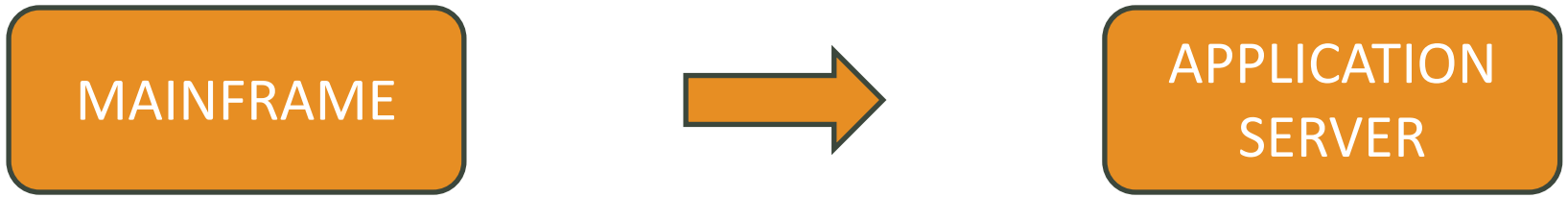
Container


Client System
Browser
Application Client Container
Application Client
Java EE Server
Web Container
JavaServer Faces
Servlet
EJB Container
EJB
EJB
Database

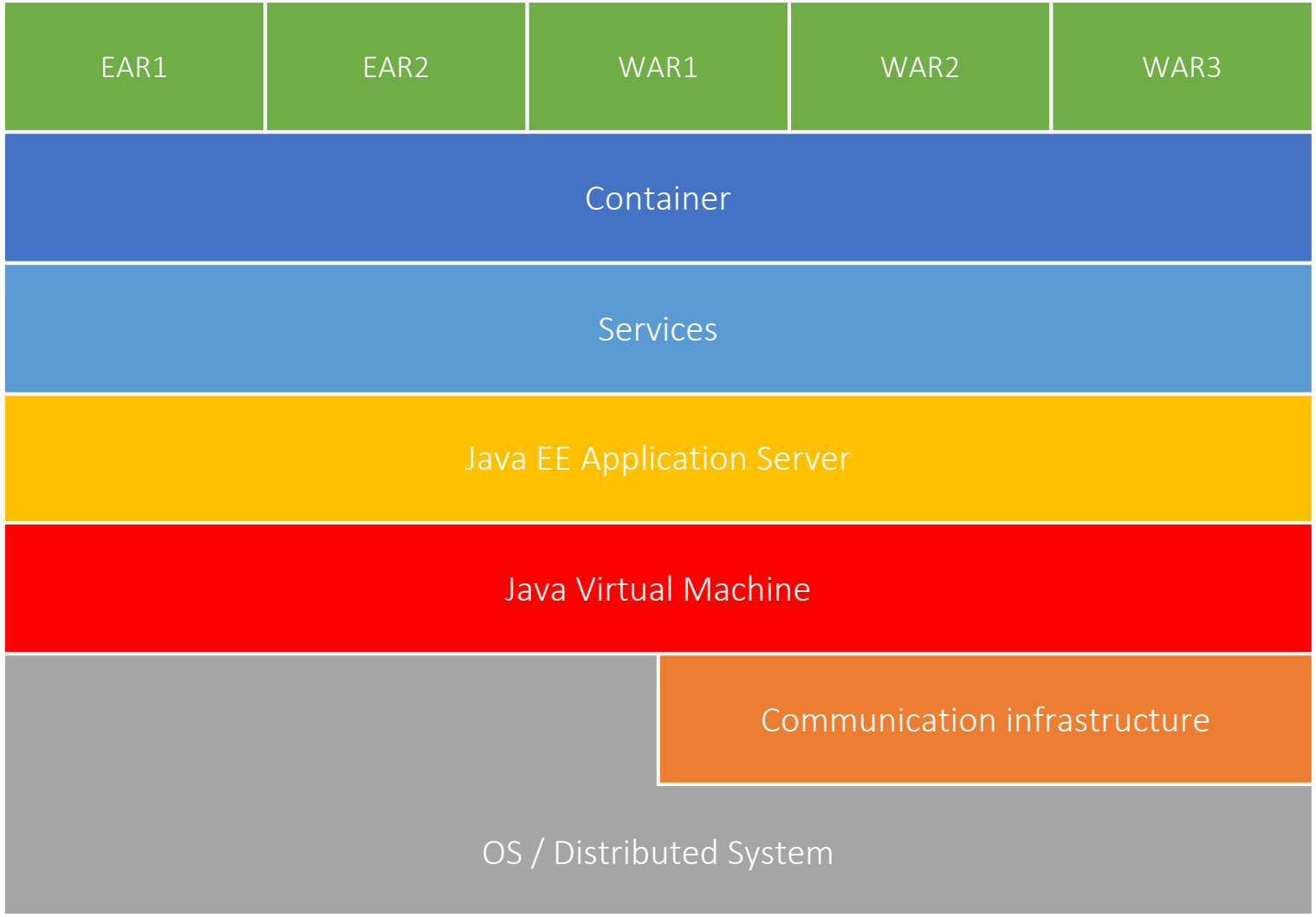
THE APPLICATION SERVER

< 1999

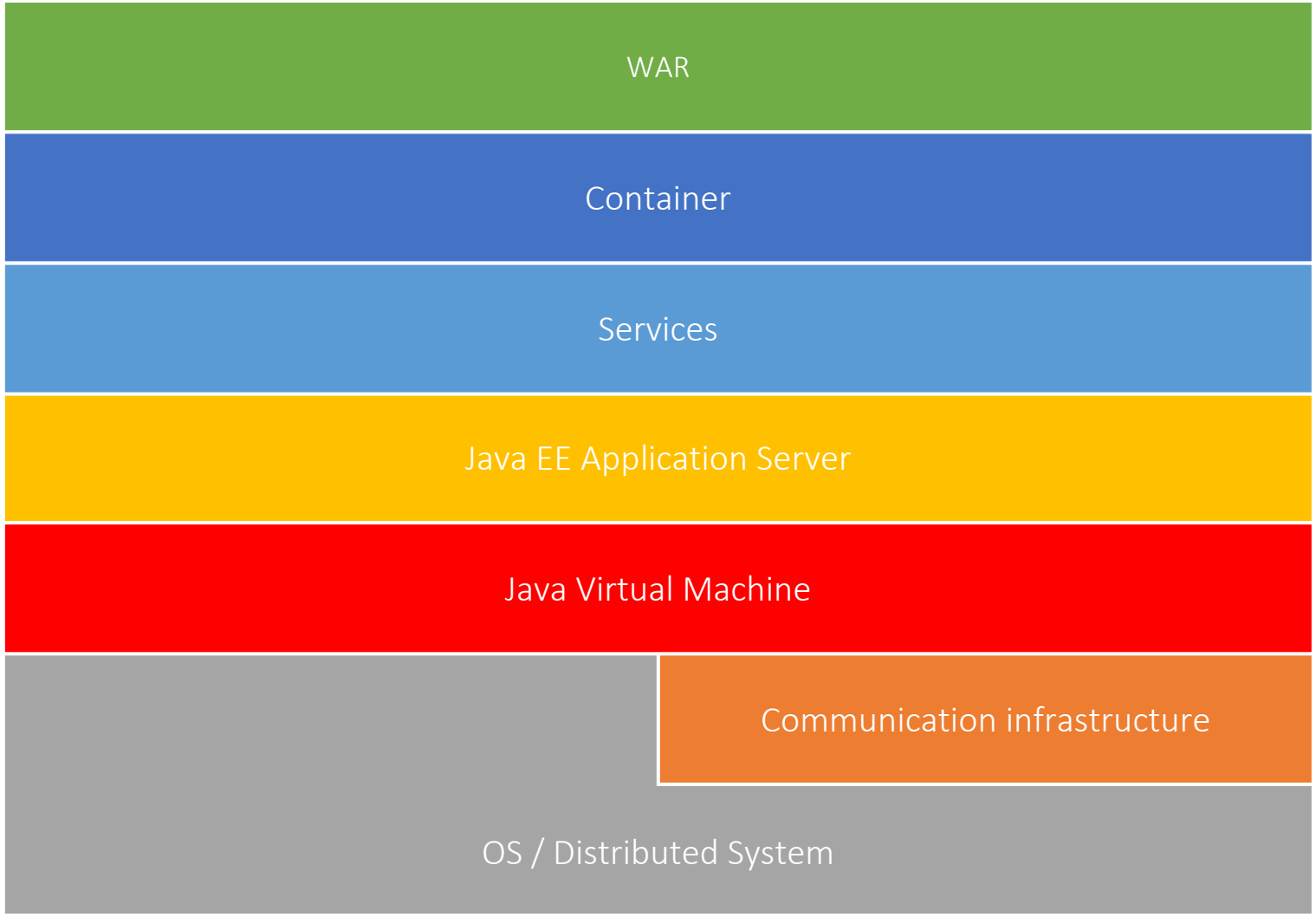
> 1999



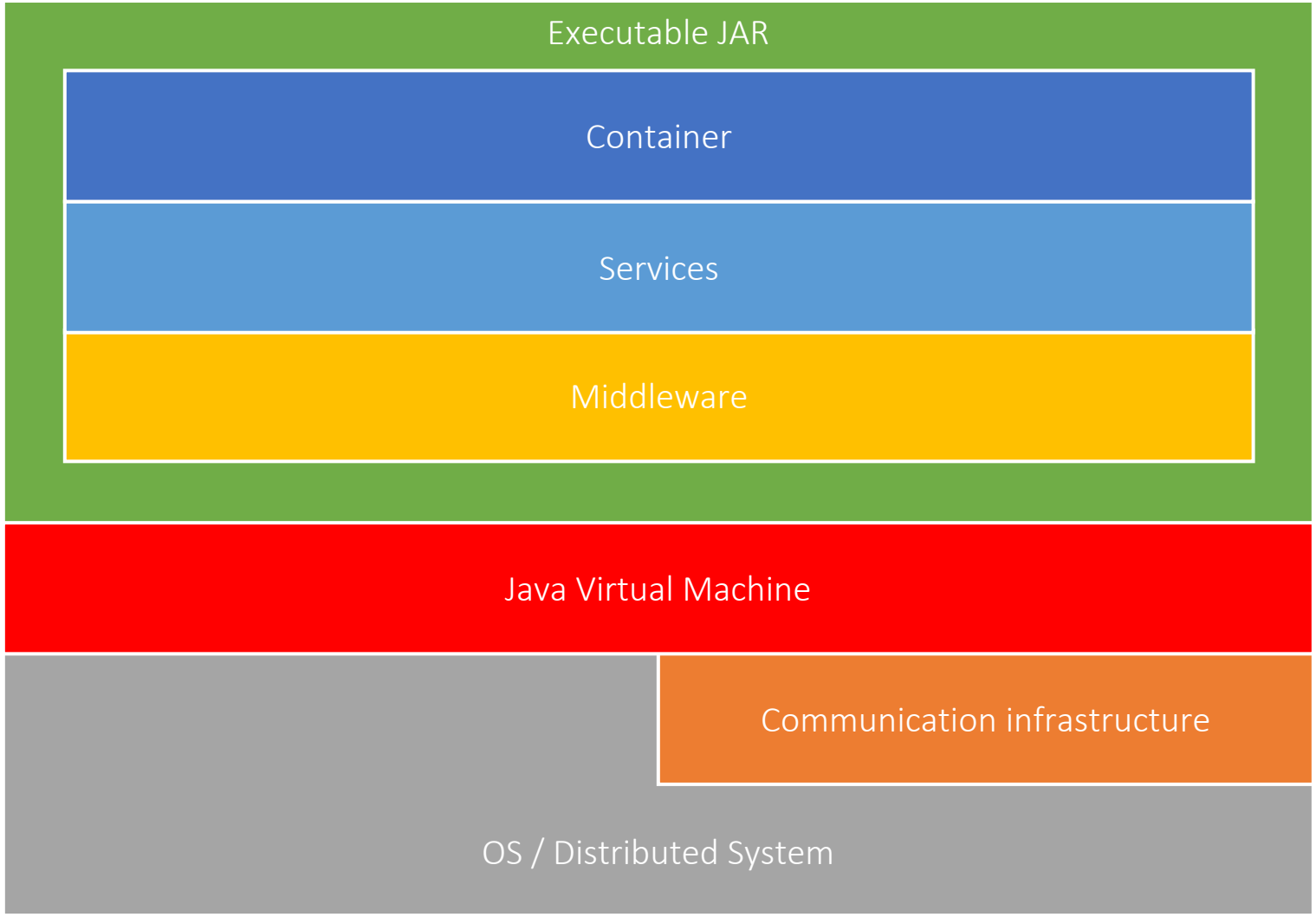
Shared Platform



Dedicated Platform



Self Hosted

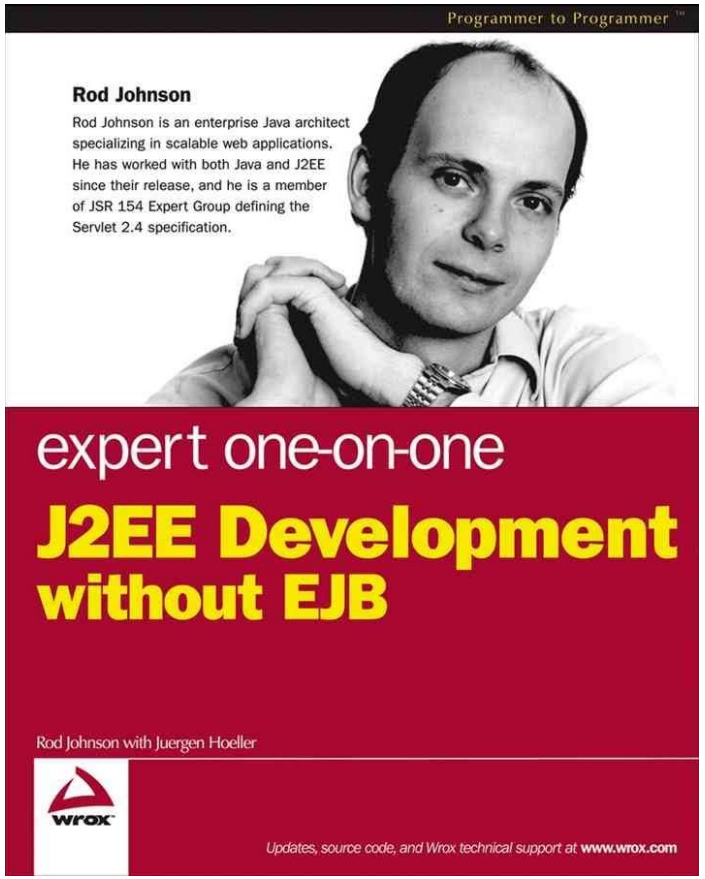

Executable JAR
Container
Services
Middleware
Java Virtual Machine
Communication infrastructure
OS / Distributed System

Spring Framework

Spring Framework History

2002

2004



The Project

https://spring.io

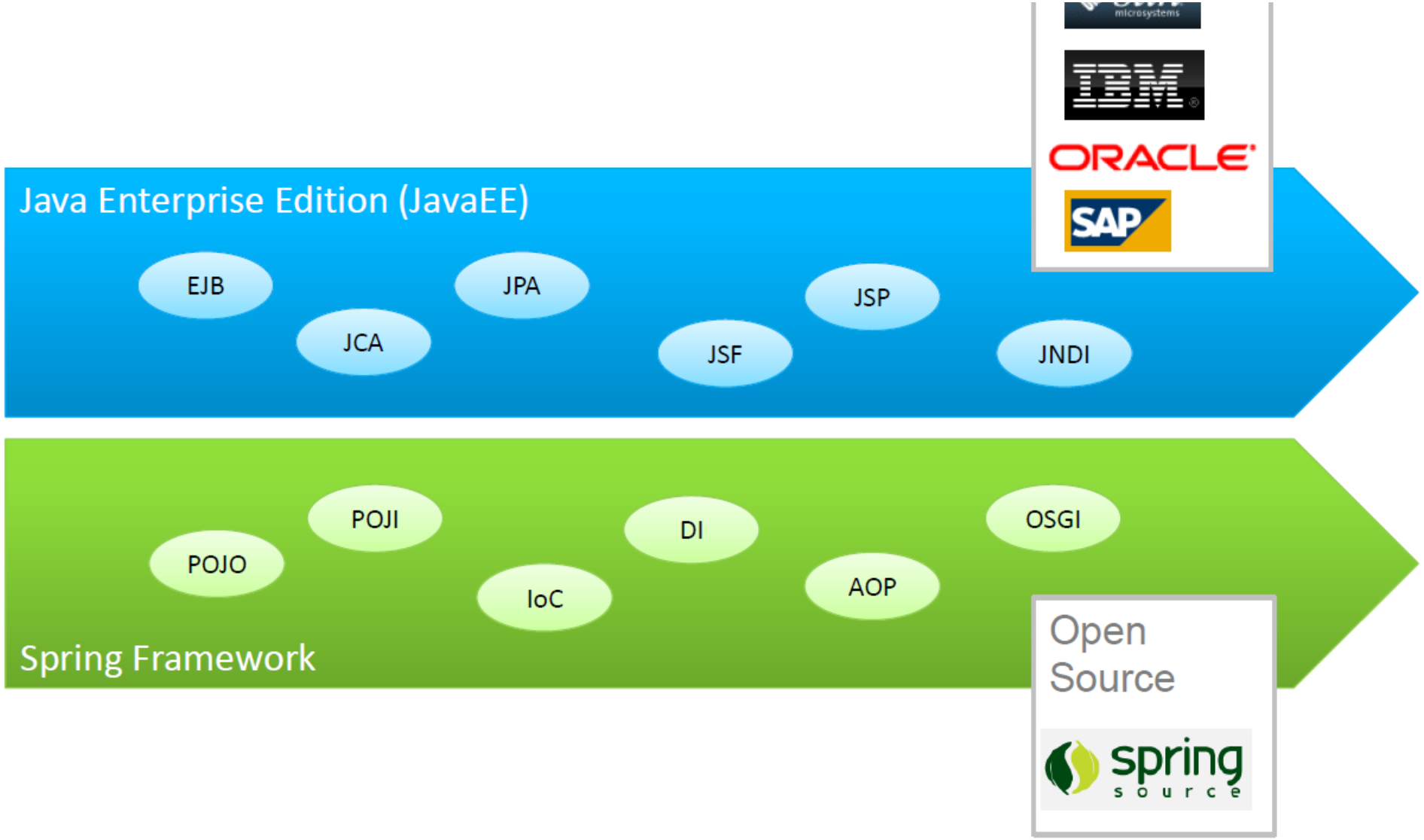
Spring created by Interface21

- Renamed Spring Source
- Currently developed by vmWare (Dell)

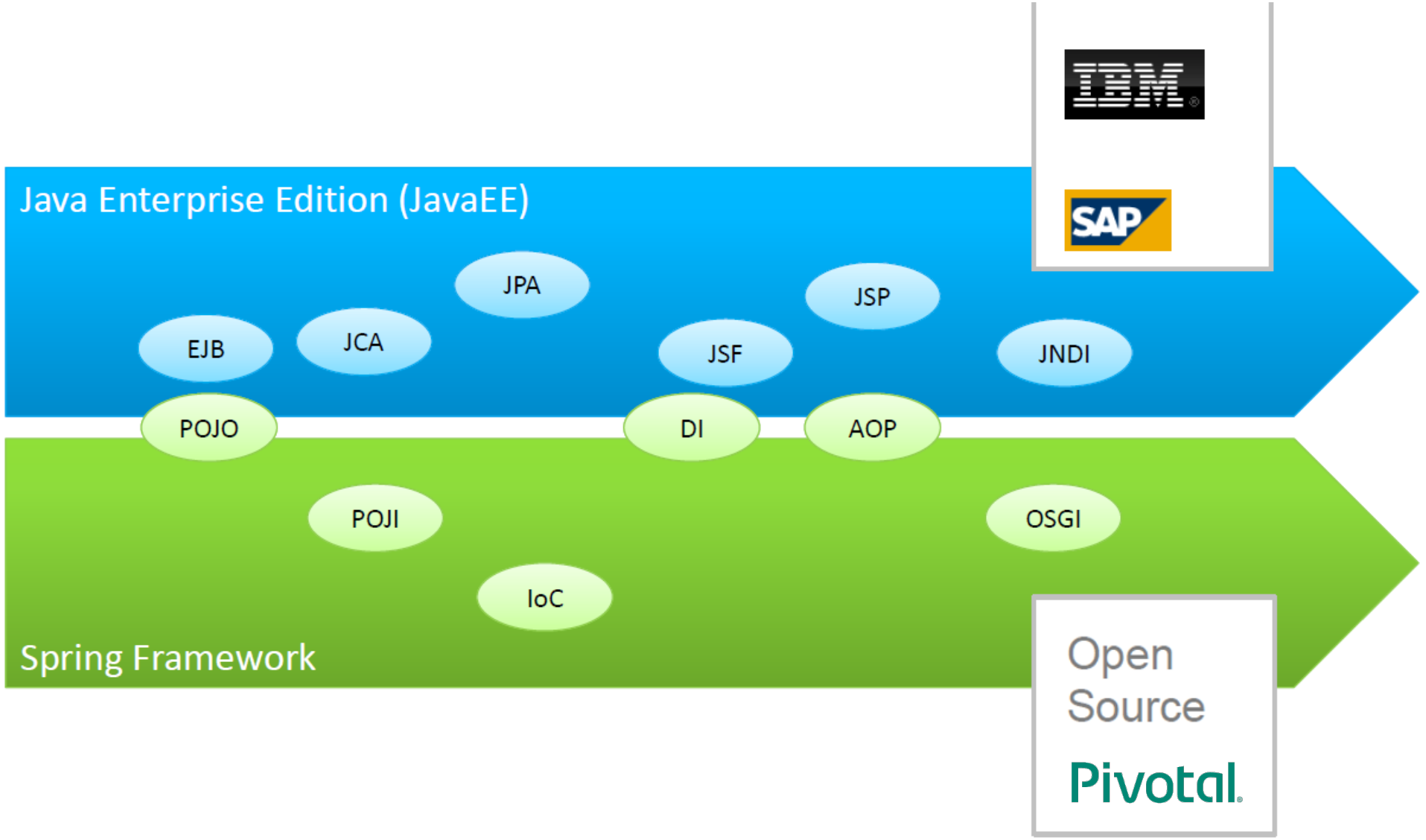
Open Source-Project started in February 2003

- Apache 2.0 License
- https://github.com/spring-projects/spring-framework

Java EE vs Spring



Not really…


IBM
SAP
Java Enterprise Edition (JavaEE)
JPA
JSP
EJB
JCA
JSF
JNDI
POJO
DI
AOP
POJI
OSGI
IoC
Spring Framework
Open Source
Pivotal

Goals

POJO-based development

- Generic component model for application development
- Flexible easy to use alternative to EJBs without being bound to J2EE

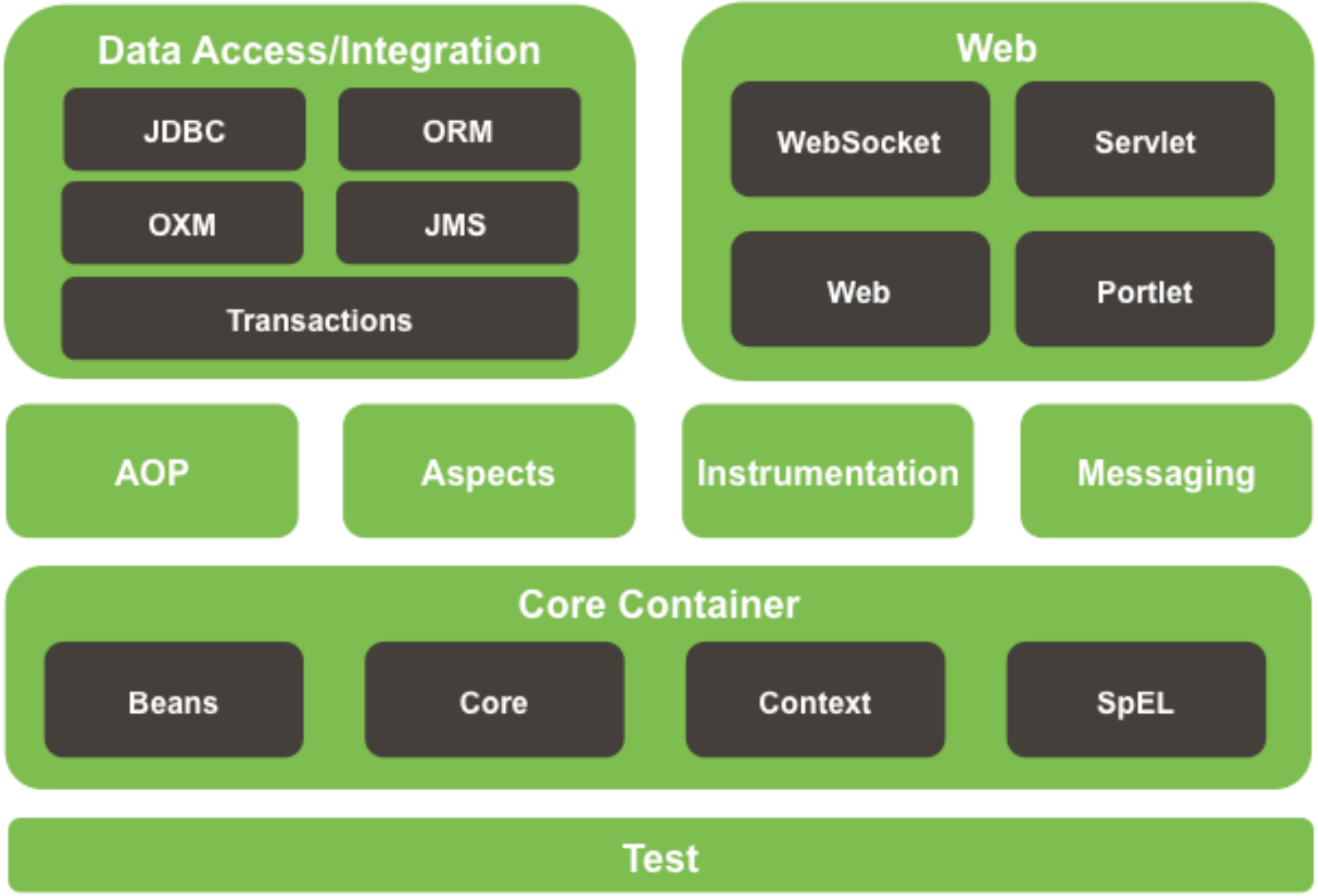
Application components are decoupled

- Instantiation, initialization, configuration and binding done by Inversion of Control / Dependency Injection framework
- Components without dependencies to Spring (non-invasive)
- Runs many runtime environments from standalone to full-scale Java EE application server

Generic Middleware-Services

- Provided through AOP
- Uses services from Java EE-Application-Servers if possible

Spring Framework Runtime



Spring Cloud

Distributed/versioned configuration
Service registration and discovery
Routing
Service-to-service calls
Load balancing
Circuit Breakers
Global locks
Leadership election and cluster state
Distributed messaging

Microservice Frameworks

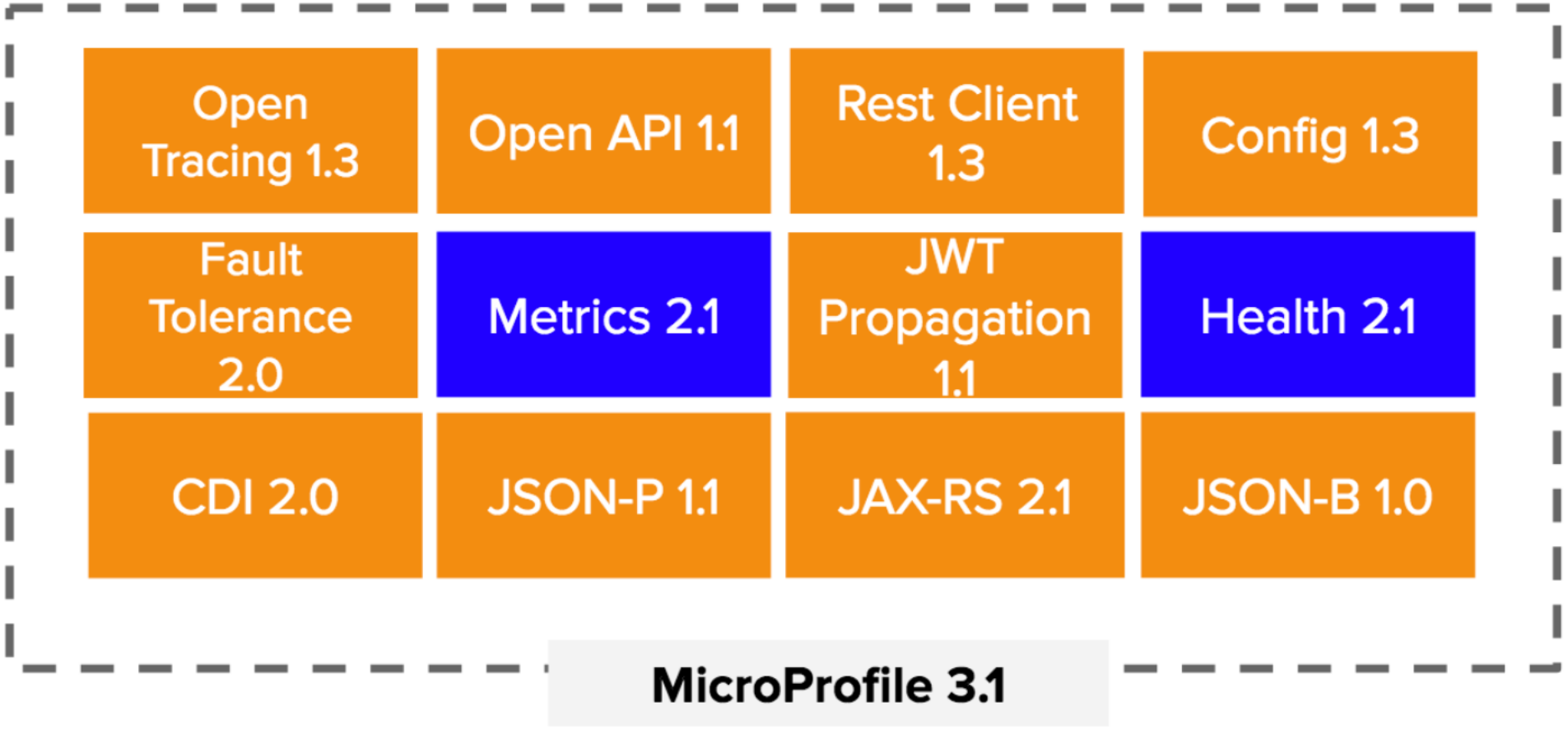
Frameworks, Frameworks, Frameworks

Source https://www.java-forum-stuttgart.de/_data/2019_G5_Schwoerer.pdf

Microprofile.io

https://microprofile.io/

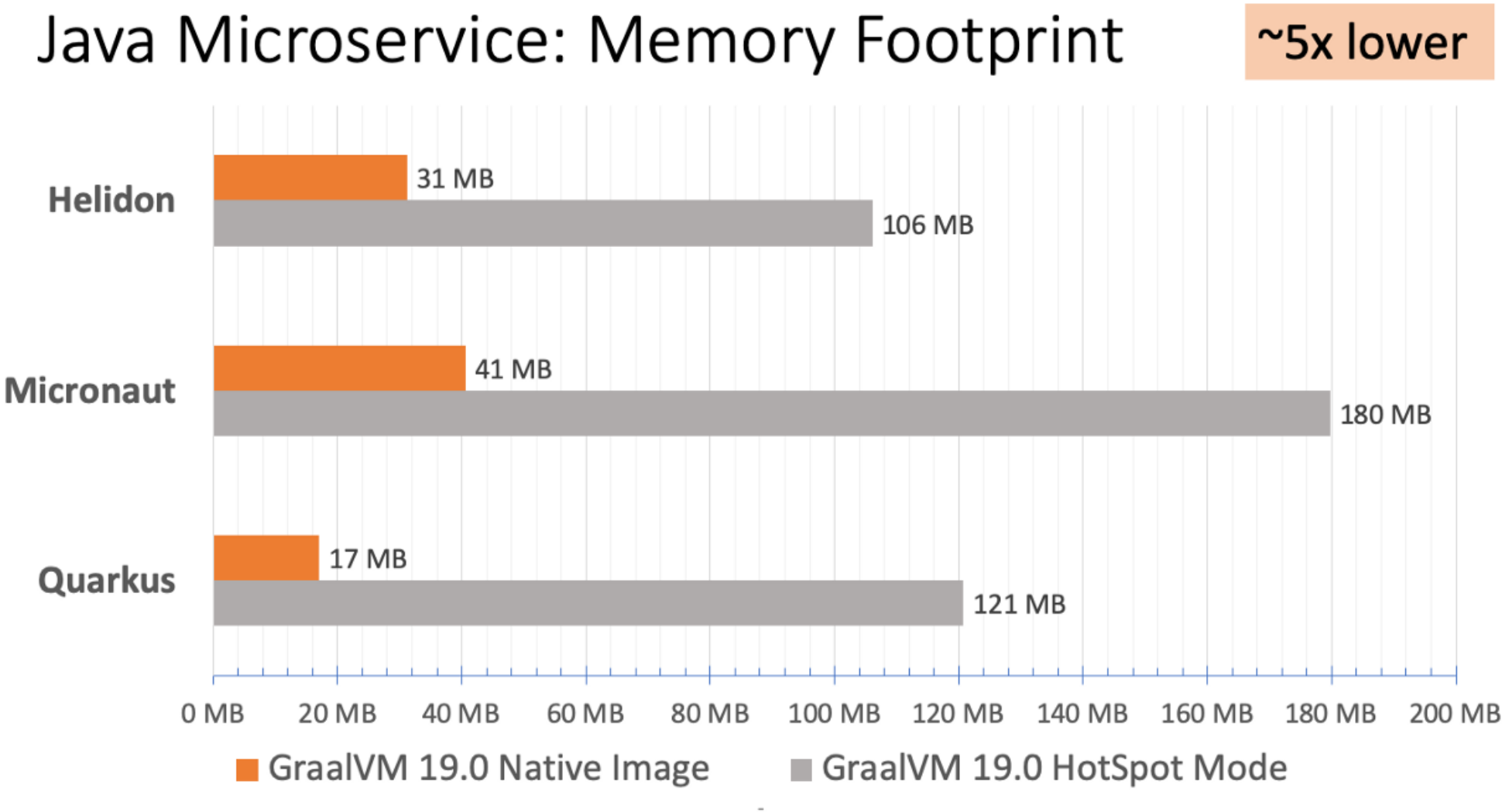
Enterprise Java for a Microservices Architecture



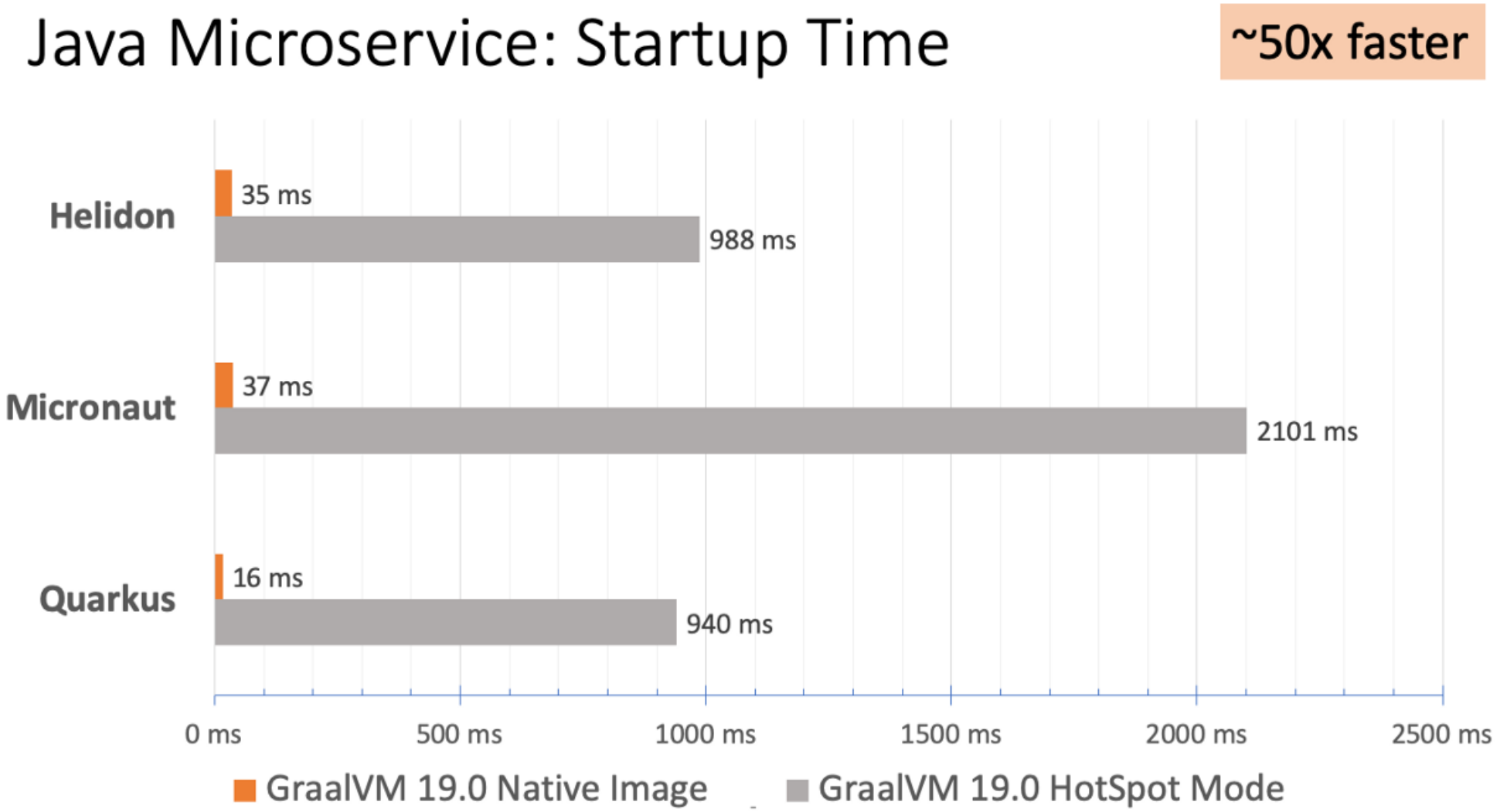
Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

GraalVM

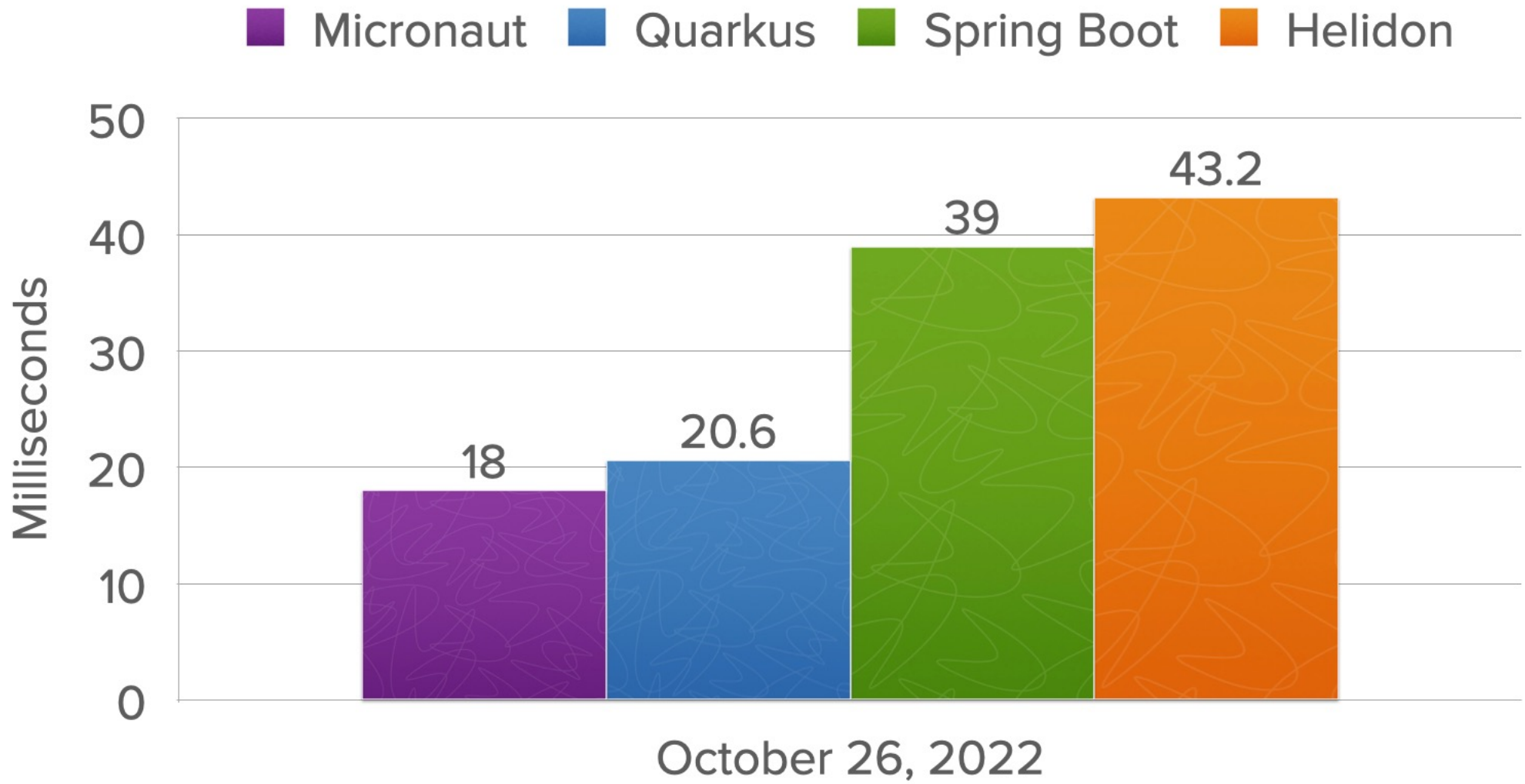
GraalVM


Java Microservice: Memory Footprint
~5x lower
Helidon
31 MB
106 MB
Micronaut
41 MB
180 MB
Quarkus
17 MB
121 MB
0 MB
20 MB
40 MB
60 MB
80 MB
100 MB
120 MB
140 MB
160 MB
180 MB
200 MB
GraalVM 19.0 Native Image
GraalVM 19.0 HotSpot Mode

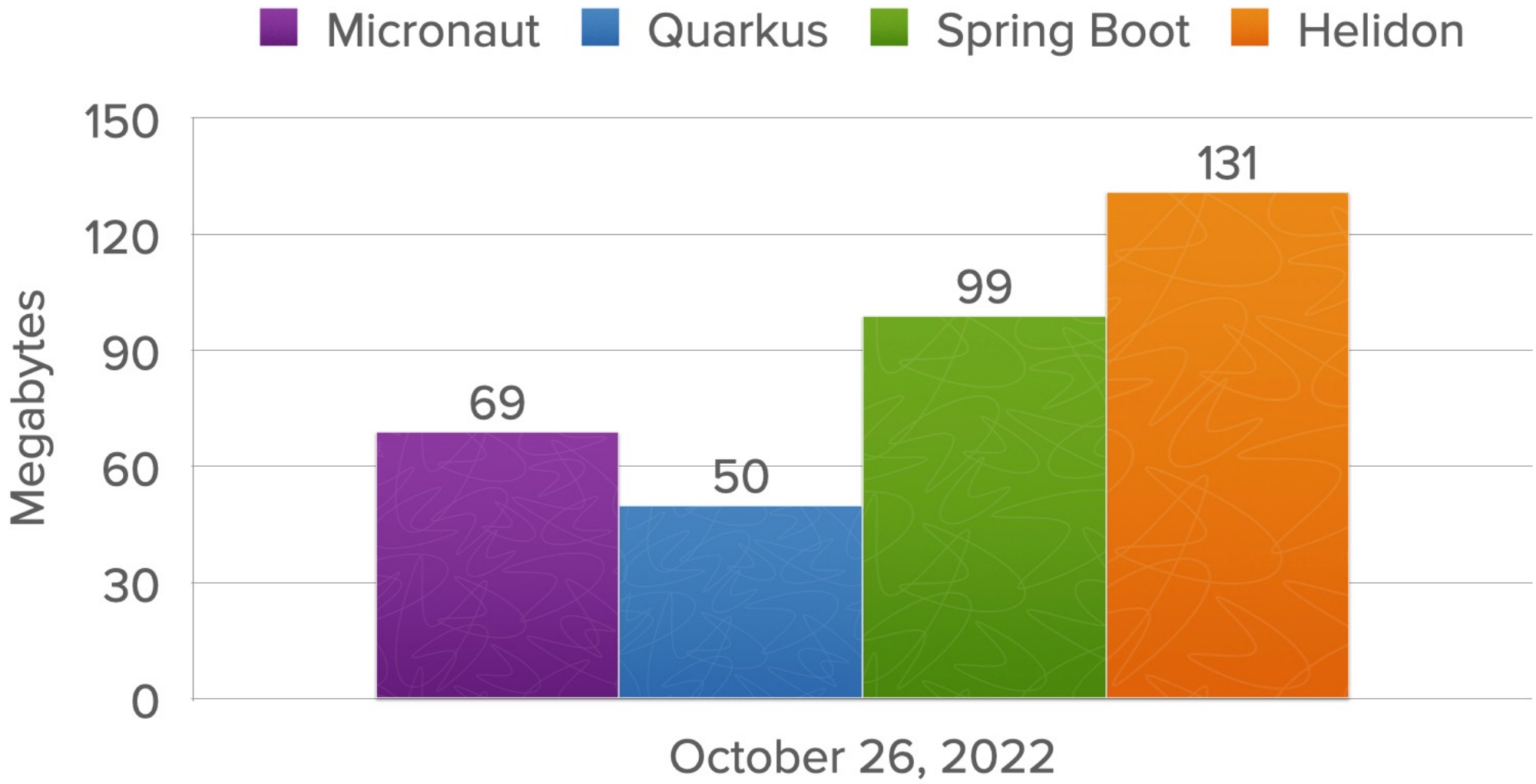
GraalVM


Java Microservice: Startup Time
~50x faster
Helidon
35 ms
988 ms
Micronaut
37 ms
2101 ms
Quarkus
16 ms
940 ms
0 ms
500 ms
1000 ms
1500 ms
2000 ms
2500 ms
GraalVM 19.0 Native Image
GraalVM 19.0 HotSpot Mode

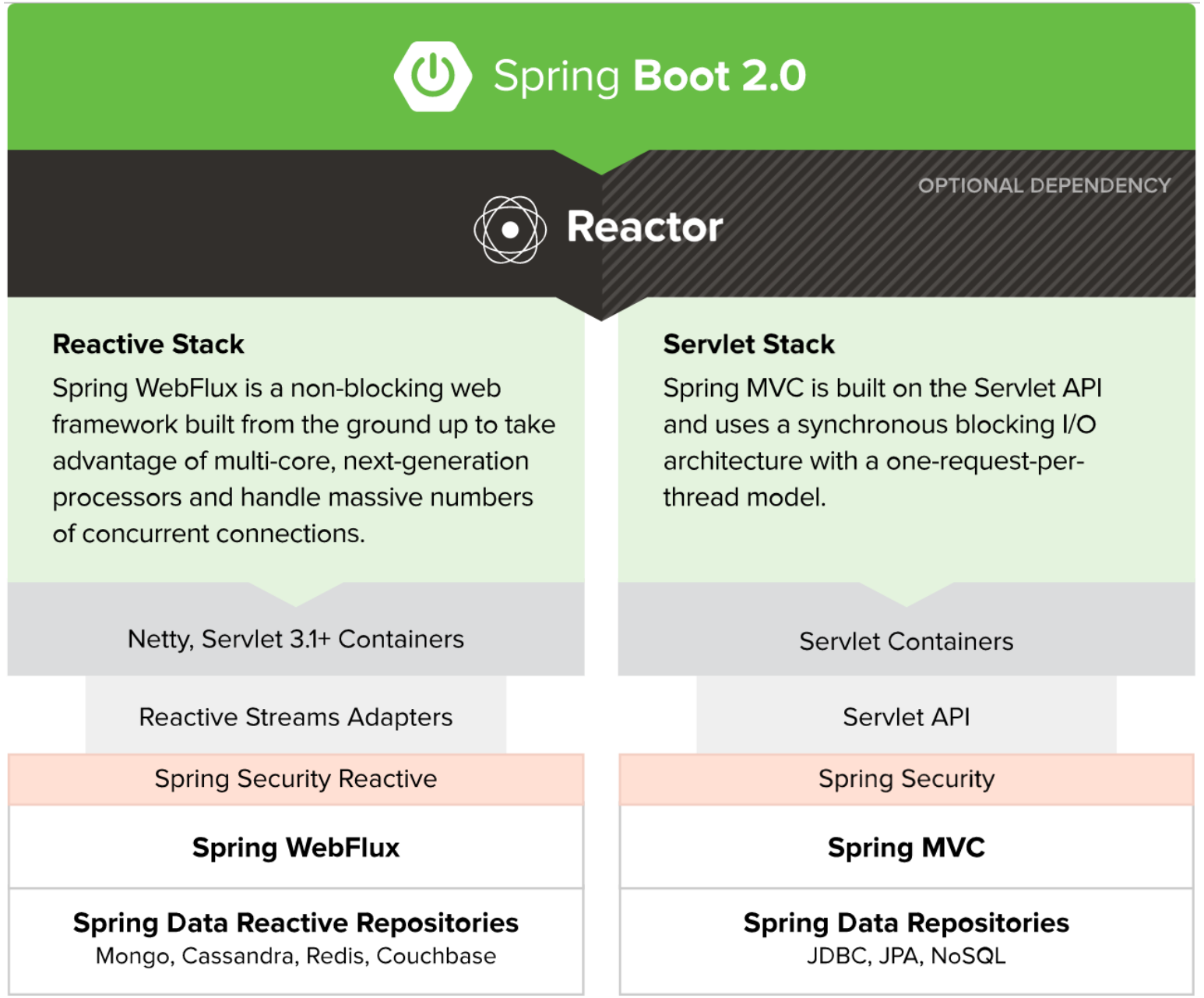
Native Startup Performance (GraalVM 22.3)



Native Memory Used (MB)



Spring Boot


Spring Boot 2.0
Reactor
OPTIONAL DEPENDENCY
Reactive Stack
Spring WebFlux is a non-blocking web framework built from the ground up to take advantage of multi-core, next-generation processors and handle massive numbers of concurrent connections.
Servlet Stack
Spring MVC is built on the Servlet API and uses a synchronous blocking I/O architecture with a one-request-per-thread model.
Netty, Servlet 3.1+ Containers
Servlet Containers
Reactive Streams Adapters
Servlet API
Spring Security Reactive
Spring Security
Spring WebFlux
Spring MVC
Spring Data Reactive Repositories
Mongo, Cassandra, Redis, Couchbase
Spring Data Repositories
JDBC, JPA, NoSQL

What's Spring Boot?

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

We take an **opinionated view** of the Spring platform and third-party libraries so you can get started with minimum fuss

Most Spring Boot applications **need very little** Spring **configuration**

Spring Boot Features

Create **stand-alone** Spring **applications**

Embed Tomcat, Jetty or **Undertow** directly (no need to deploy WAR files)

Provide opinionated **'starter' POMs** to simplify your Maven configuration

Automatically configure Spring whenever possible

Provide **production-ready features** such as metrics, health checks and externalized configuration

Absolutely **no code generation** and **no requirement for XML** configuration

Spring Boot Starters

Starters are a set of **convenient dependency descriptors** that you can include in your application.

You get a **one-stop shop** for all the Spring and related technologies that you need without having to hunt through sample code and copy-paste loads of dependency descriptors

For example, if you want to get started using Spring and JPA for database access, include the `spring-boot-starter-data-jpa` dependency in your project

The starters **contain a lot of the dependencies** that you need to get a project up and running quickly and with a consistent, supported set of managed transitive dependencies

What's new in Spring Boot 3?

Java 17

JakartA EE 9

- New namespace jakarta instead of javax

Native support

On your mark, ready, GO!

Spring Boot Website

https://spring.io/projects/spring-boot

Reference Documentation

https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/

Spring Initializer

https://start.spring.io/

Parent and Plugin

```
<!-- Inherit defaults from Spring Boot -->
<parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.0.6.RELEASE</version>
</parent>
```

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

Configuration

Feature	`@ConfigurationProperties`	`@Value`
Relaxed binding	Yes	No
Meta-data support	Yes	No
`SpEL` evaluation	No	Yes

Table 24.1. relaxed binding

Property	Note
`acme.my-project.person.first-name`	Kebab case, which is recommended for use in `.properties` and `.yml` files.
`acme.myProject.person.firstName`	Standard camel case syntax.
`acme.my_project.person.first_name`	Underscore notation, which is an alternative format for use in `.properties` and `.yml` files.
`ACME_MYPROJECT_PERSON_FIRSTNAME`	Upper case format, which is recommended when using system environment variables.

Main Class

```
@SpringBootApplication
public class Application {

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

}
```

Run

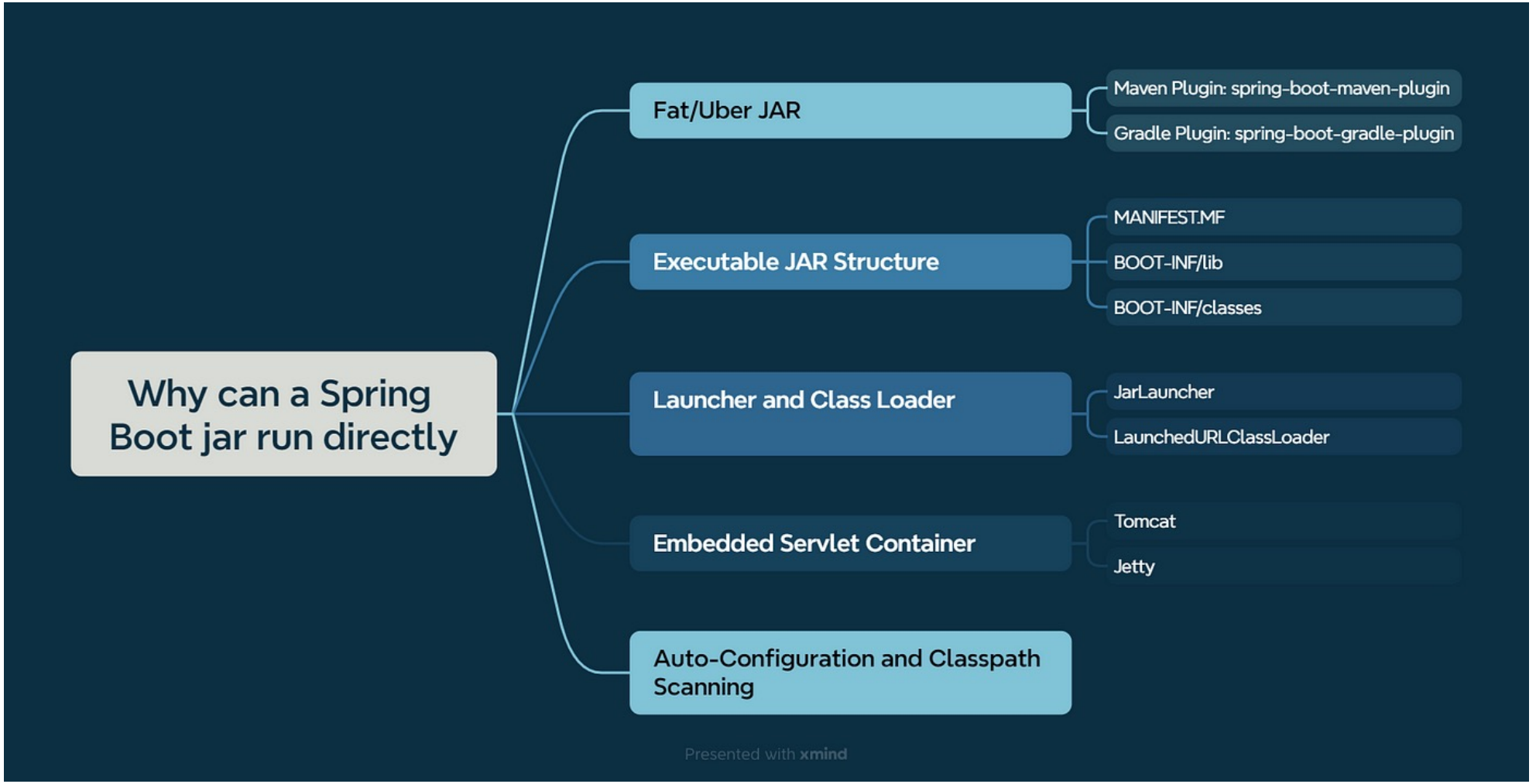
```
import org.springframework.boot.*;
import org.springframework.stereotype.*;

@Component
public class MyBean implements CommandLineRunner {

        public void run(String... args) {
                // Do something...
        }

}
```

Profiles

```
@Configuration
@Profile("production")
public class ProductionConfiguration {

        // ...

}
```

```
spring.profiles.active=dev,hsqldb
```

Why can Spring Boot jars run directly?



Source: https://blog.stackademic.com/why-can-spring-boot-jars-run-directly-ab51cd3a79a2