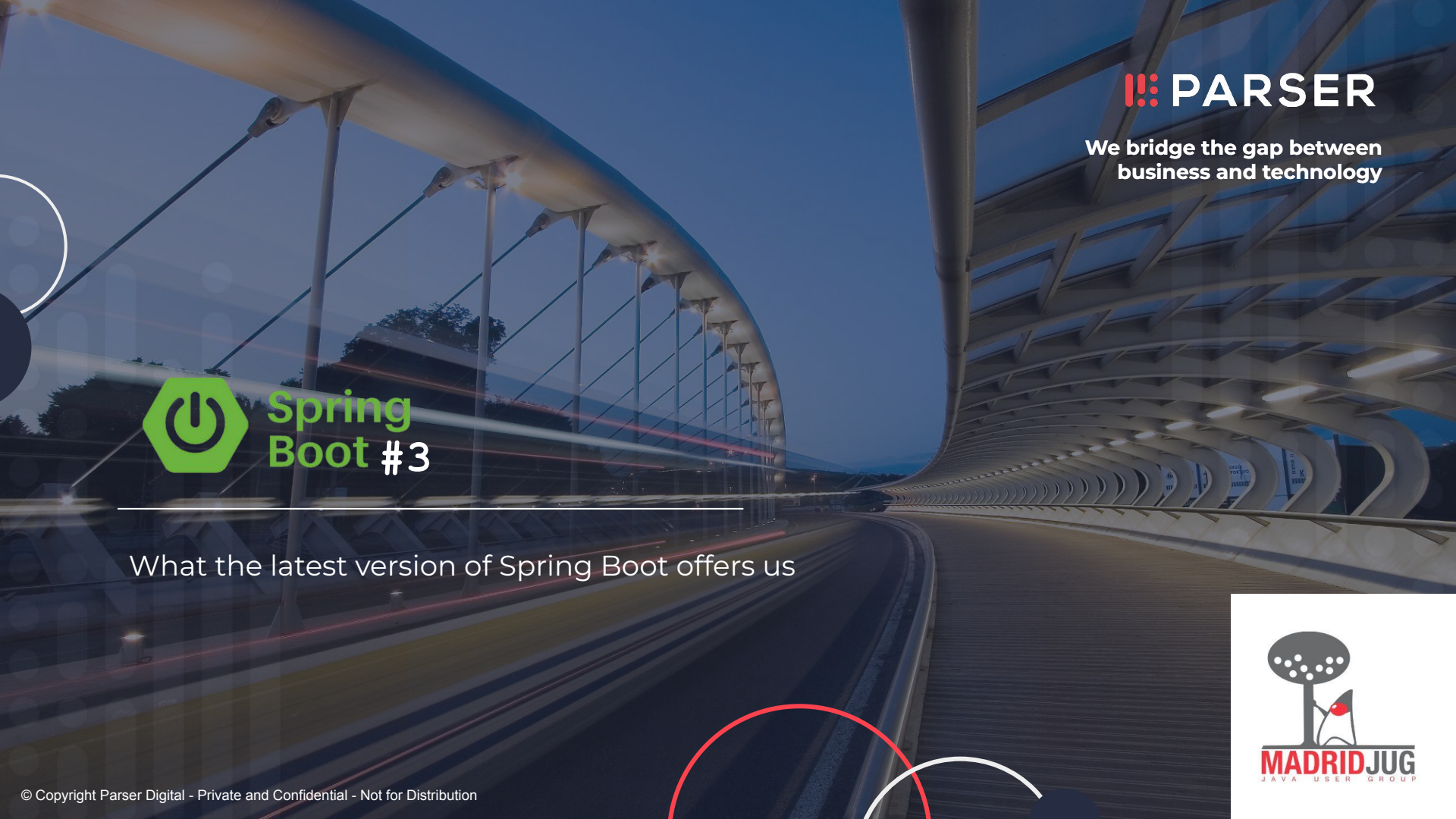


PARSER

We bridge the gap between business and technology

Spring Boot #3

What the latest version of Spring Boot offers us

MADRIDJUG
JAVA USER GROUP

Event Organisation

The team

After a couple of months of planning, the day has finally come!

Thanks to Flor, David and Aurea for making this possible.

Now... let's start! 🙌

PARSER

Florencia Paez

Organiser

Iván Pérez

Speaker

MADRIDJUG
JAVA USER GROUP

David Gómez

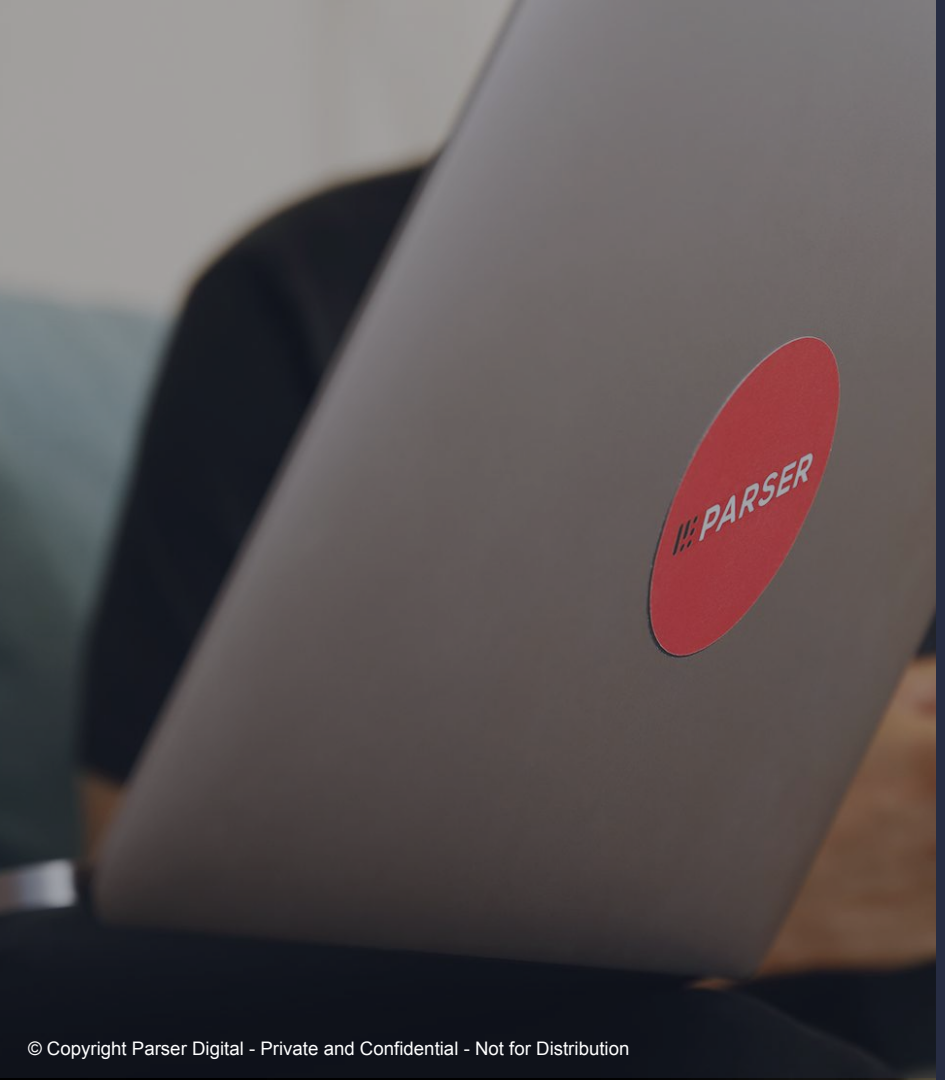
Event host

Aurea Muñoz

Event host

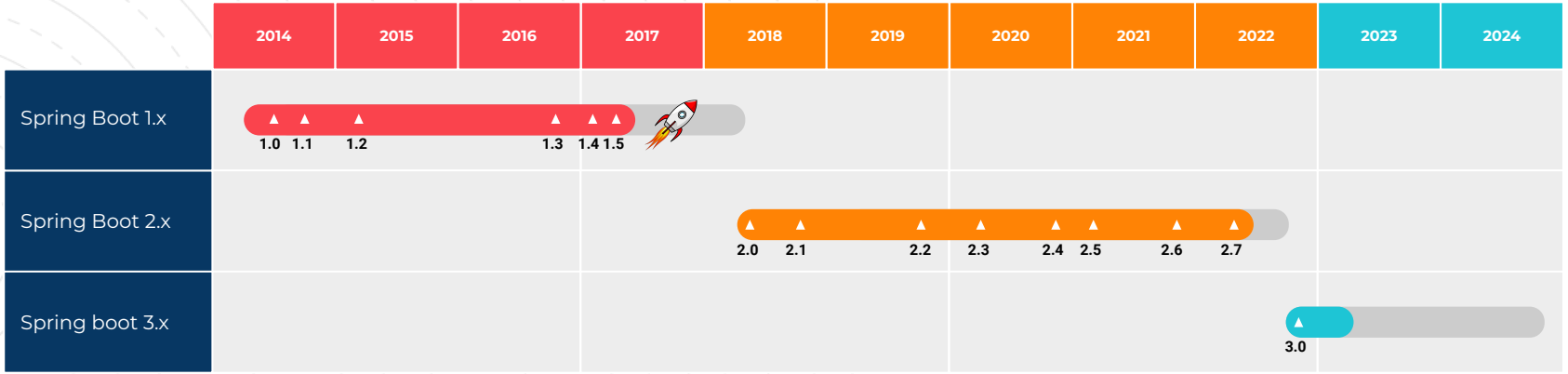
Agenda

- Release history
- Spring Boot 3
  - Highlights
  - Tips for migration to SB3
  - Working modes
    - JVM
    - Native
- Demo
- Questions



Release History

Spring Boot release history



- **2017 was the key year** in which Spring Boot took off and started to be adopted by quite a few companies for their services in a productive way.
- Spring Boot v2 **has been the most used so far**, being compatible from Java 8 onwards
- v3 aims to improve **compatibility with containers**, as well as latest versions of Java

Spring Boot 3

PARSER

Highlights

- Compatibility with **Java 17** onwards. Older versions of Java are no longer supported
- Migration from Java EE to **Jakarta EE** APIs. All packages related to `javax.*` are now `jakarta.*`
- **GraalVM** Native Image support
- Integration with **Micrometer** (tracing) to improve observability
- Built on top of **Spring Framework 6**, meaning we can use many other features, such as:
  - `ProblemDetail` for raising exceptions, aligned to RFC-7807
    https://docs.spring.io/spring-framework/docs/6.0.0-M3/javadoc-api/org/springframework/http/ProblemDetail.html
  - New `HttpInterface` to build http clients
    https://docs.spring.io/spring-framework/docs/6.0.0-RC1/reference/html/integration.html#rest-http-interface
  - Project Loom: virtual threads (still in preview)

Problem Detail

- It’s a representation of the RFC-7807 that includes all fields defined in the RFC
- We don’t need to create our custom exceptions object or make use of the ProblemJson developed by Zalando anymore.

```
{
  "type": "https://example.com/probs/out-of-credit",
  "title": "You do not have enough credit.",
  "status": 403,
  "detail": "Your current balance is 30, but that costs 50.",
  "instance": "/api/bank"
}
```

Http Interface

- New Java interface with annotated methods for HTTP exchanges
- Just declare an interface with your HTTP calls

```
@HttpExchange(url = "/repos/{owner}/{repo}", accept = "application/vnd.github.v3+json")
interface RepositoryService {

    @GetExchange
    Repository getRepository(@PathVariable String owner, @PathVariable String repo);

    @PatchExchange(contentType = MediaType.APPLICATION_FORM_URLENCODED_VALUE)
    void updateRepository(@PathVariable String owner, @PathVariable String repo,
            @RequestParam String name, @RequestParam String description, @RequestParam String homepage);

}
```

- And create a proxy that will perform the HTTP exchanges

```
WebClient client = WebClient.builder().baseUrl("https://api.github.com/").build();
HttpServiceProxyFactory factory = WebClientAdapter.createHttpServiceProxyFactory(client);
factory.afterPropertiesSet();

RepositoryService service = factory.createClient(RepositoryService.class);
```

Observability improvements

- What is observability?

> *"how well you can understand the internals of your system by examining its outputs"*
>
> https://spring.io/blog/2022/10/12/observability-with-spring-boot-3

- Spring Boot 3 has improved the integration with Micrometer
- Now it's **very easy to have distributed tracing, logs and metrics** → Autoconfiguration
- We can add "observations" to our application so that we can see how it is behaving
  - Help us to measure specific operations inside our app, create spans for distributed tracing, logs…

```
        Observation           Observation
        Context               Context
Created ----------> Started ----------> Stopped
```

Observability improvements (II)

AOP annotations

- Requires to register a new Bean `ObservedAspect`
- We can make observations by just adding the annotation `@Observed` to the method that we want to observe
- Logic in the method remains intact

Programmatically

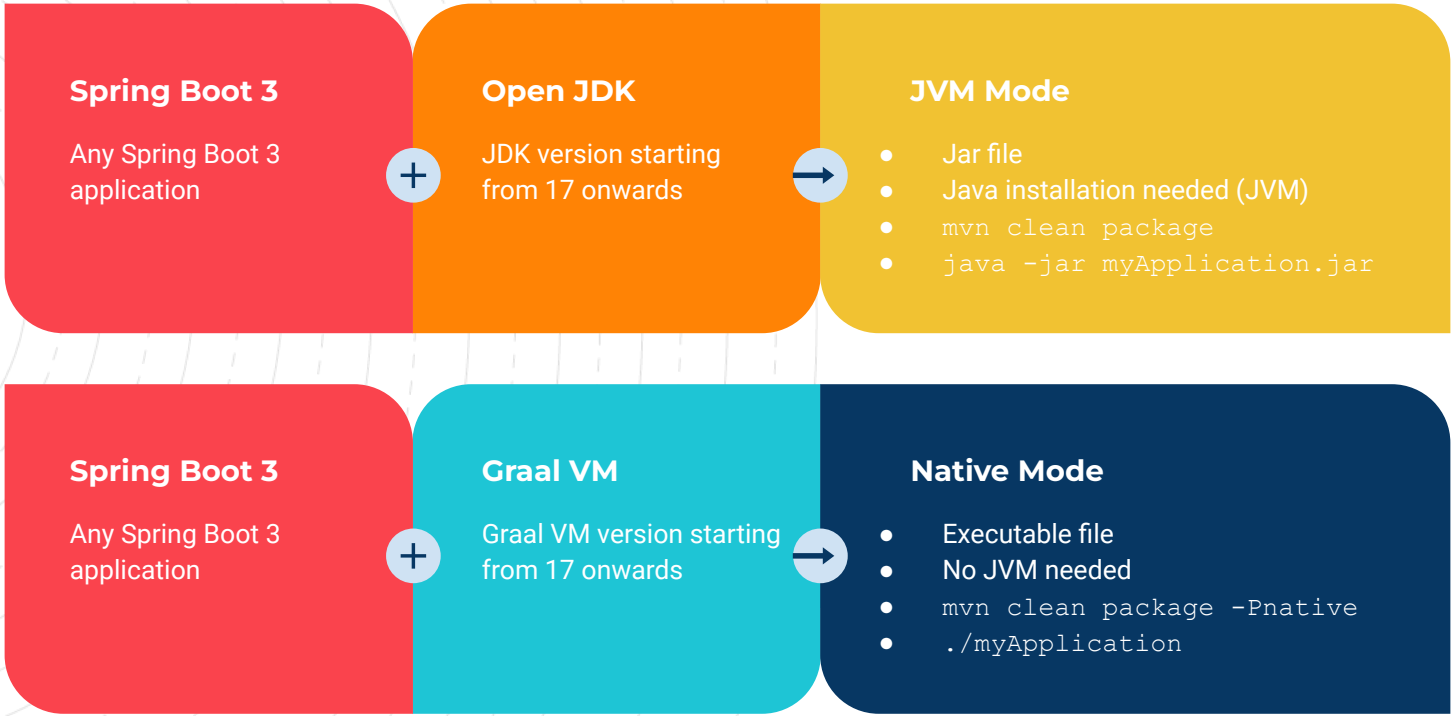
- No need to register any Bean
- We need to inject the `ObservationRegistry` in the class where we want to make observations
- Observation is created programmatically inside the method that we want to observe

Virtual Threads (preview)

- **Current challenges:**
  - Java relies on the OS kernel threads. Applications usually allow up to millions of transactions, users or sessions simultaneously, however there are not enough threads for it.
  - Common examples are *CompletableFuture* and *RxJava*
  - **Such APIs are harder to debug and integrate with legacy APIs**. Thus, there is a need for a lightweight concurrency construct which is independent of kernel threads.

- **Project Loom** allows developers to create **high concurrent applications** without the need of making big changes to the current code.
- Requires **JDK 19**
- Its goal is to dramatically reduce the effort of writing, maintaining, and observing high-throughput concurrent applications

Working modes



Migration steps

Pre-Migration steps

#1 - Install JDK 17 in your system

- **SDKman** is your tool → https://sdkman.io/
- Very easy to install/switch between different versions of JDK
- Will save us a lot of time building JVM/Native apps (JDK vS. GraalVM)
- Available for Windows/Linux/macOS


```
ivan_perez@PRM-FVFG92E6Q05P ~ % sdk list java
================================================================================
Available Java Versions for macOS ARM 64bit
================================================================================
 Vendor        | Use | Version      | Dist    | Status     | Identifier
--------------------------------------------------------------------------------
 Corretto      |     | 19.0.1       | amzn    |            | 19.0.1-amzn
               |     | 17.0.5       | amzn    |            | 17.0.5-amzn
               | >>> | 17.0.4       | amzn    | local only | 17.0.4-amzn
               |     | 11.0.17      | amzn    | installed  | 11.0.17-amzn
               |     | 8.0.352      | amzn    |            | 8.0.352-amzn
 Gluon         |     | 22.1.0.1.r17 | gln     |            | 22.1.0.1.r17-gln
               |     | 22.1.0.1.r11 | gln     |            | 22.1.0.1.r11-gln
 GraalVM       |     | 22.3.r19     | grl     |            | 22.3.r19-grl
               |     | 22.3.r17     | grl     | installed  | 22.3.r17-grl
               |     | 22.3.r11     | grl     |            | 22.3.r11-grl
               |     | 22.2.r17     | grl     |            | 22.2.r17-grl
               |     | 22.2.r11     | grl     |            | 22.2.r11-grl
               |     | 22.1.0.r17   | grl     |            | 22.1.0.r17-grl
               |     | 22.1.0.r11   | grl     | installed  | 22.1.0.r11-grl
 Java.net      |     | 21.ea.4      | open    |            | 21.ea.4-open
               |     | 21.ea.3      | open    |            | 21.ea.3-open
               |     | 21.ea.2      | open    |            | 21.ea.2-open
               |     | 21.ea.1      | open    |            | 21.ea.1-open
               |     | 20.ea.30     | open    |            | 20.ea.30-open
               |     | 20.ea.29     | open    |            | 20.ea.29-open
               |     | 20.ea.28     | open    |            | 20.ea.28-open
               |     | 20.ea.27     | open    |            | 20.ea.27-open
               |     | 20.ea.26     | open    |            | 20.ea.26-open
               |     | 19.0.1       | open    |            | 19.0.1-open
```

Pre-Migration steps

#2 - Update your project to the latest Spring Boot 2.7.x version

- Fix issues step by step will be less painful
- This way it will be easier to detect some incompatibilities before moving to 3.x
- Review your dependencies and upgrade those required to the latest version

Maven™

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot</artifactId>
    <version>2.7.7</version>
</dependency>
```

Gradle

```
implementation 'org.springframework.boot:spring-boot:2.7.7'
```

Pre-Migration steps

#3 - Pay special attention to Spring Security (if you use it)

- Spring Boot 3.0 works with Spring Security 6
- To make the transition to v3 simpler, the Spring team have released a Spring Security 5.8 compatible with Spring Boot 2.7.x, so it is recommended to upgrade
- There is an official guide about how to upgrade to Spring Security 5.8 and what you have to keep in mind

https://docs.spring.io/spring-security/reference/5.8/migration/index.html

https://docs.spring.io/spring-security/reference/migration/index.html

Pre-Migration steps

#4 - Review deprecations on latest version of Spring Boot 2.x

- There are some classes or methods that were deprecated on Spring Boot 2
- These won't be available on Spring Boot 3
- Remove them! (Or, replace them by their equivalent.)
- Some examples:
  - `WebSecurityConfigurerAdapter` has been removed
  - `@EnableGlobalWebSecurity` has been removed
  - `authorizeRequests()` renamed to `authorizeHttpRequests()`
  - `antMatchers()` renamed to `requestMatchers()`

Migration steps

Upgrade to Spring Boot 3

Maven™

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot</artifactId>
    <version>3.0.0</version>
</dependency>
```

Gradle

```
implementation 'org.springframework.boot:spring-boot:3.0.0'
```

Migration steps

#1 - Fix compilation errors

- Spring Boot 3 brings some **breaking changes**
- Reminder:
  - `javax.x` → `jakarta.x`
- If you did not remove the deprecations from Spring Boot 2.x, you must do it now
- Some **application properties** have been also removed or renamed
- To help developers on this matter, Spring Boot provides a `spring-boot-properties-migrator` library
- Once added to our project, it will not only analyse the application environment and print diagnostics at startup, but will also temporarily migrate properties at runtime for you

Migration steps

#2 - Verify core changes

- If you were using an image banner for your app, this is no longer supported. Text file does.
- Logging date format has changed for Logback and Log4j2 to be aligned with the ISO-8601 standard
  - `'yyyy-MM-dd'T'HH:mm:ss.SSSXXX'`
  - It uses a `T` separator between the date and time instead of a space
  - Also adds the timezone offset to the end

Summary

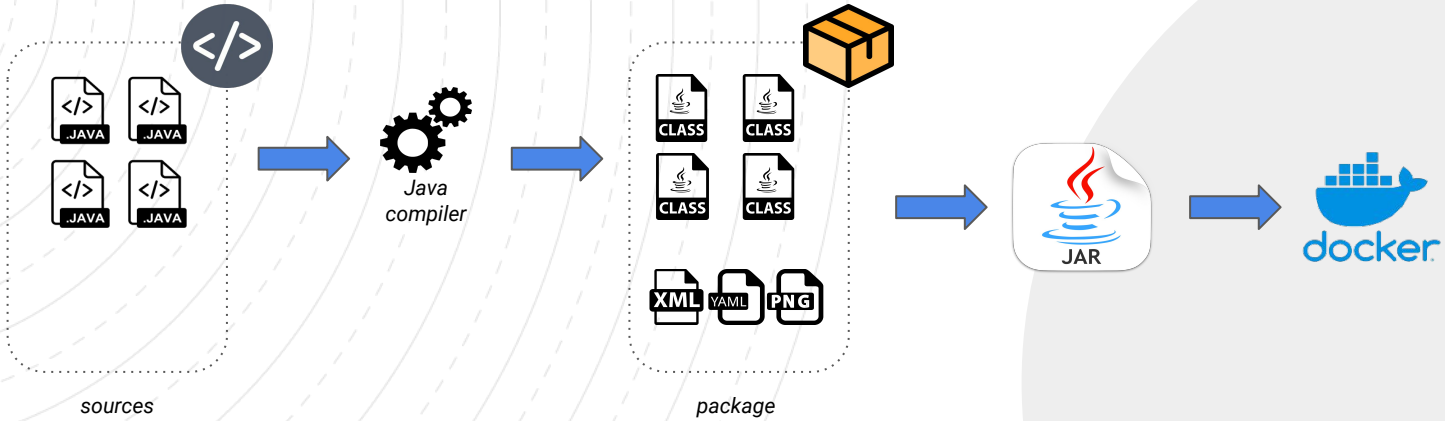
- Try to keep your dependencies up-to-date!
  - It will make the migration to Spring Boot 3 easier
  - You'll always have the latest features available, bug fixes
  - And safety! Most of the third party library updates are to patch vulnerabilities → OWASP Top Ten
  - There are tools like Renovate Bot or Dependa Bot that can make this very easy!
- Read the docs 🤓
  - There is a lot of official documentation on how to migrate applications to Spring Boot 3
- Don't give up, learn, and if possible, share your process. You might find issues that will help others:
  - https://stackoverflow.com/search?q=spring+boot
  - https://www.reddit.com/r/SpringBoot/

Working Modes

PARSER

JVM Mode

- **Standard mode** for all Spring Boot apps, also for those previous to v3
- Artifact is a **jar** file, which contains all the compiled code (`*.class`) and all data/resources needed for a Java application to work
- **A required Java installation** in the system to be able to run a jar file
- Easy to run inside a container (Docker)



Native Mode

- Spring Boot 3 brings support for **GraalVM** → https://www.graalvm.org
- Spring seeks to position itself against competitors in the world of native applications such as **Quarkus** or **Micronaut**
- Artifact is an **executable** file
- **No java installation** required to run the application
- Easy to run inside a container (*)
- There is a plugin available for both **Maven** and **Gradle** to create native images!
- Nice to have clarity on the next concepts before going native:
    - GraalVM, AOT processing, reachability metadata…

GraalVM

- Businesses are always looking for ways to increase application performance. To avoid adding costs on resources, they look for **efficiency** to get more out of their existing infrastructure.
  - Horizontally scalable **container-based** architectures and microservices

- **GraalVM** is a **high-performance runtime** built on Oracle Java SE. It adds new **compiler optimizations** that deliver the best performance for running Java applications.
  - Enhances the Java ecosystem by offering a compatible and better-performing Java Development Kit (JDK)
  - Includes a **Native Image** utility → AOT compilation → Native executable
    - These are **comparable to native C/C++ programs** in startup, memory usage, and performance, without leaving the powerful ecosystem of Java.

Ahead of Time (AOT) compilation

- Compilation means transforming source code (Java, Python…) into machine code, which is a set of low-level instructions to be executed in a microprocessor
- **JIT (*Just In Time*)** compilation is the default mode for Java apps → **JVM**
- **AOT** runs over the Graal VM compiler and compiles bytecode directly into machine code at build time

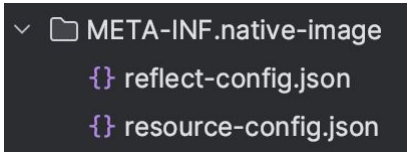
JIT

- Generates machine code **during** execution of the program
- Traditional computers **cannot execute** JVM bytecode directly.
- The **JVM interprets bytecode at runtime** and identifies the architecture where the program is running
- This makes **Java apps slower** in comparison with other programming languages such as Rust or C which produce native code directly

AOT

- Generates machine code **before** the execution of the program
- The resulting code is **tailored to a specific OS and hardware architecture** → very fast execution
- GraalVM compiler performs a very high AOT compilation of the JVM bytecode
- As output, a native executable is produced. It **doesn't require a JVM** installed in the machine
- Native apps consume much **less memory** and **start much faster** → eliminate overhead introduced by the JVM

Reachability metadata



- When we build a native image, **only elements that are reached from our application entry point are included**. These are libraries, JDK classes discovered through the static analysis, etc…
- **Some elements** such as classes, methods or even fields **may not be discovered** due to some aspects like reflection, dynamic proxies, serialisation…
- Even though the executable will be created successfully, if an element is not included, it could lead to **runtime failures**, therefore we need to manually tell the compiler that those elements must also be kept into account and included in the final artifact
- To help developers, there is a Github repository which includes a **base configuration for reachability metadata**. This repo is consulted during the creation of the native image and for example, if the compiler detects that our app uses PostgreSQL, the reachability metadata related to this will be downloaded from the repository
- **Sometimes this is not enough**, but we can always provide our own metadata configuration so that the compiler includes it

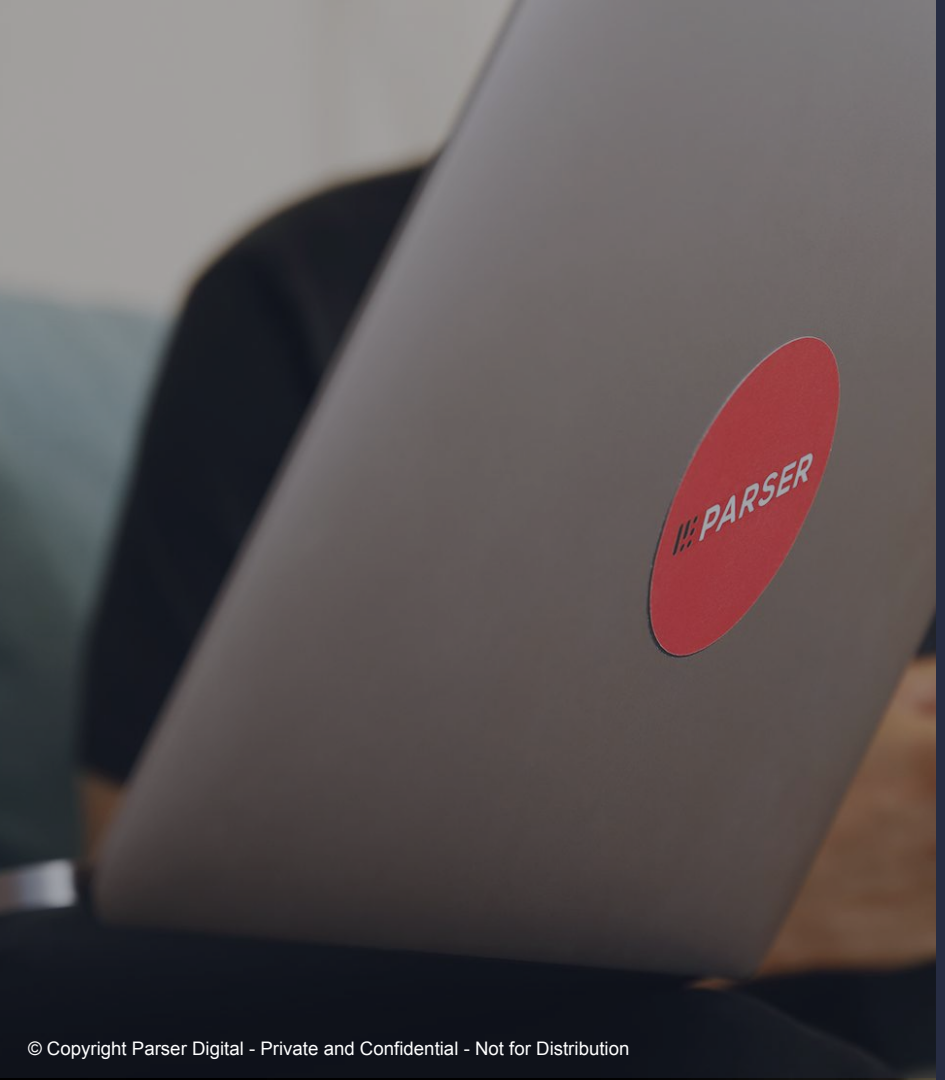
Native Mode

PROS

- Dead code elimination → Reduces attack surface
- Very fast start time → Easier horizontal scale
- Very low memory consumption
- Starting apps is as easy as any other executable file
- No need of Java installation on the machine
- Small size on disk → smaller containers → cheaper cloud infrastructure!

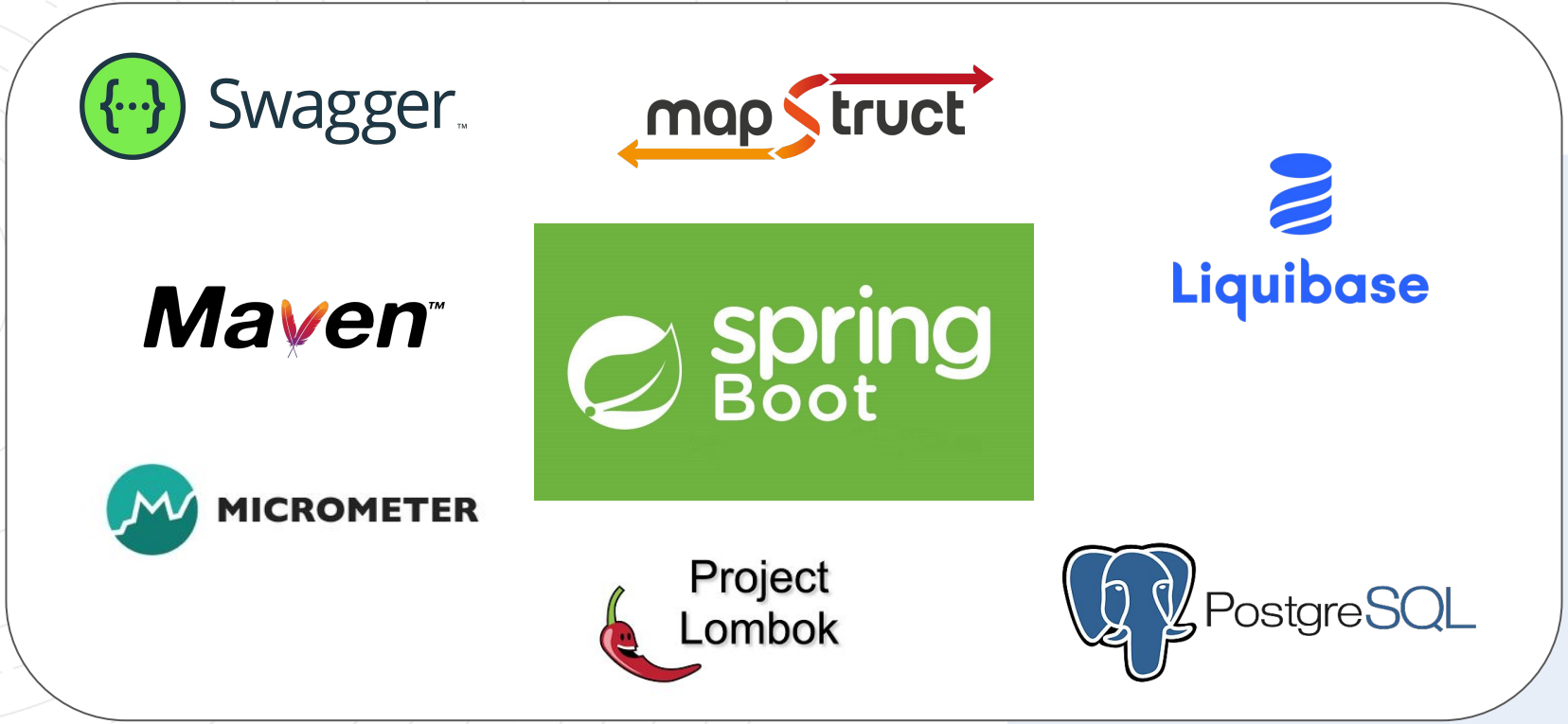
CONS

- More complicated initial configuration (specially on Windows), learning curve
- Much slower builds
- As native mode is based on reflection, even though we manage to create the executable successfully, it can fail during start up due to some missing configuration
- Executable tied to be run on a specific hardware architecture

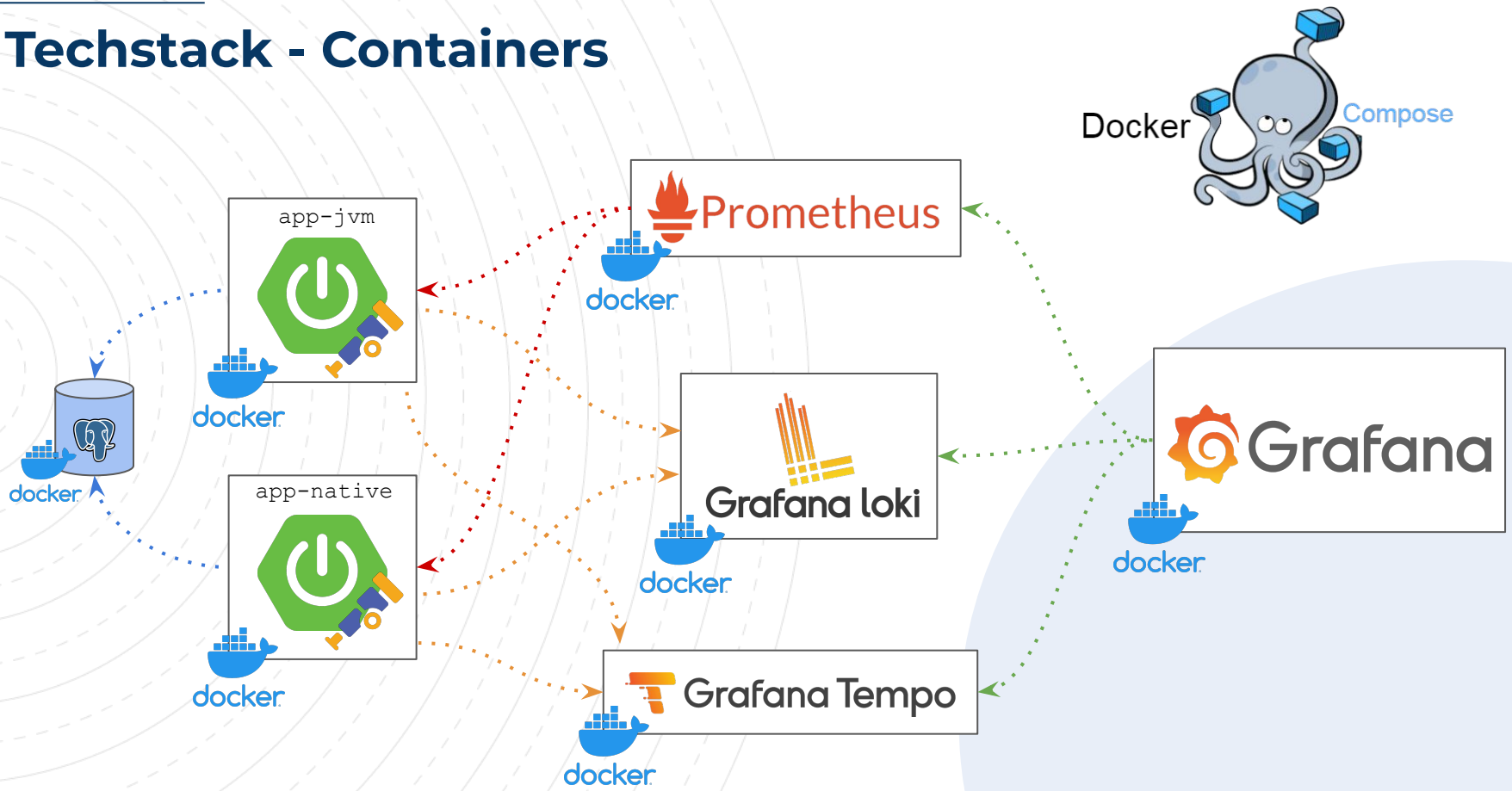


DEMO

Techstack - Application



Techstack - Containers



Questions?

Thank you!

Iván Pérez | *Technical Lead*
ivan.perez@parserdigital.com

https://github.com/parserdigital/springboot-3-overview.git