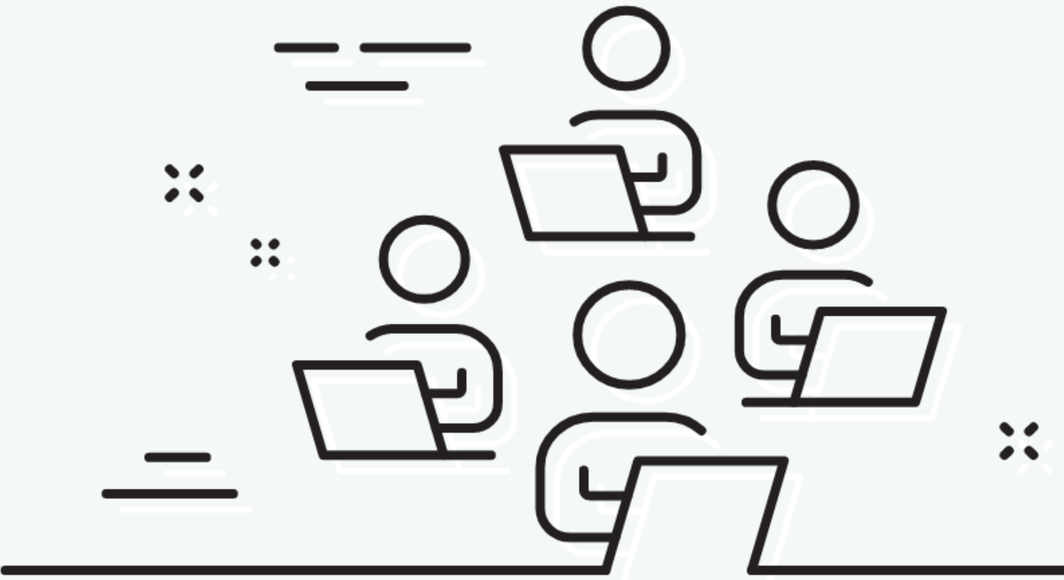
SPRING BOOT 3 & BEYOND

Dan Vega
Spring Developer Advocate at Broadcom

spring

OFFICE HOURS



https://www.springofficehours.io

AGENDA

SPRING BOOT 3.0

JDK 17+

Jakarta EE 9/10

AOT (Ahead-of-time)

Observability

SPRING BOOT 3.1

Docker Compose

Testcontainers

Connection Details

SPRING BOOT 3.2

Virtual Threads

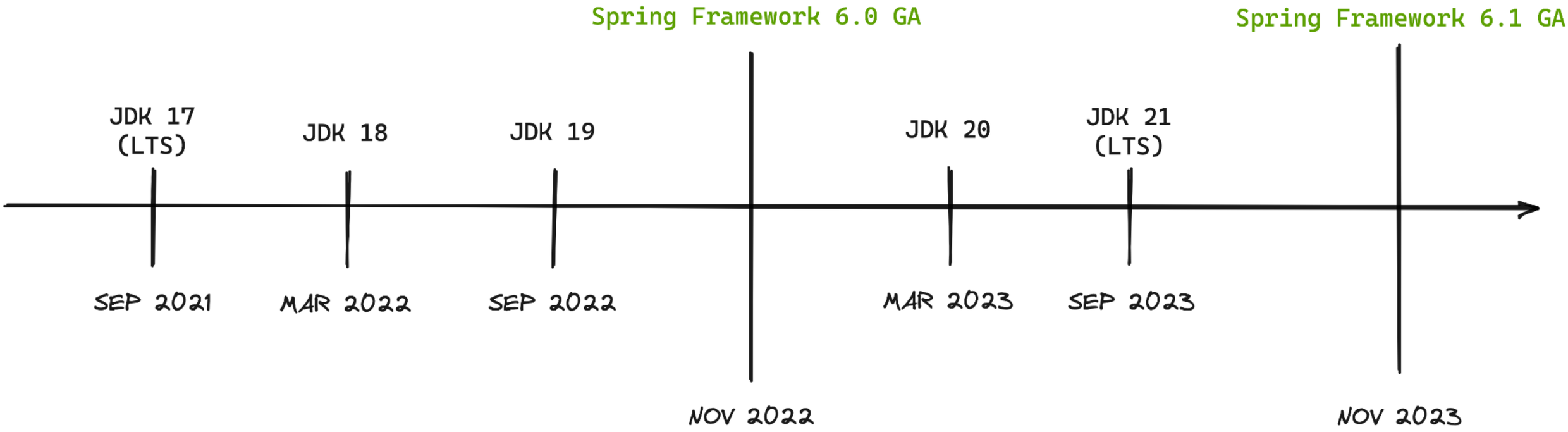
Project CRaC

Rest Client

JDBC Client

SPRING BOOT 3

JAVA AND JAKARTA


Spring Framework 6.0 GA
Spring Framework 6.1 GA
JDK 17
(LTS)
JDK 18
JDK 19
JDK 20
JDK 21
(LTS)
SEP 2021
MAR 2022
SEP 2022
NOV 2022
MAR 2023
SEP 2023
NOV 2023

BASELINE: JAKARTA EE 9

A new namespace: Java EE 8 → Jakarta EE 9

Servlet API 5.0 : javax.servlet → jakarta.servlet

JPA 3.0 : javax.persistence → jakarta.persistence

Bean Validation 3.0 : javax.validation → jakarta.validation

Dependency Injection 2.0 : javax.inject → jakarta.inject

All further API evolution to happen in jakarta namespace

Migrating with the help of IDEs & Tools

IntelliJ IDEA - Java EE → Jakarta EE

Open Rewrite

Spring Boot Migrator

AHEAD-OF-TIME (AOT)

AOT COMPILATION

What is it?

Reducing startup time and **memory footprint** in production

Runtime hints for reflection, resources, serialization, proxies

Optional for optimized JVM deployment

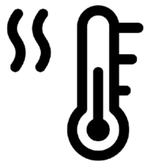
Precondition for GraalVM native executables

AOT is a tradeoff; extra build setup and less flexibility at runtime

WHY COMPILE TO NATIVE?

Instant Startup

Milliseconds for native instead of seconds for the JVM



No Warmup

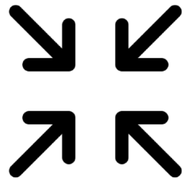
Performance benefits available immediately

Low Resource Usage

Lower Memory consumption no JIT compilation

Reduced Attack Surface

Closed world of dependencies with explicit reflection



Compact Packaging

Smaller containers and easier to deploy

Lower Compute Costs

Reduce your infrastructure costs

spring

USE CASES FOR NATIVE IMAGES

JVM Likely Better

- High Traffic Website
- Huge Memory and CPU
- Frequent deployments
- Big Monolithic Application

Assess

- Microservices
- Container Image Distribution
- Kubernetes
- Backoffices

Recommended

- Low Memory and CPU
- Function as a Service (FaaS)
- Scale to Zero
- CLIs

OBSERVABILITY

WHAT IS OBSERVABILITY?

Observability is the ability to observe the internal state of a running system from the outside. It consists of the three pillars **logging**, **metrics** and **traces**.

Direct Observability instrumentation with **Micrometer Observation** in several parts of the Spring Framework.

RestTemplate, **WebClient & RestClient** are instrumented to produce HTTP client request observations.

Spring Cloud Sleuth no longer needed for **Distributed Tracing**

```java
@SpringBootApplication
public class Application {

    private static final Logger log = LoggerFactory.getLogger(Application.class);

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

    @Bean
    CommandLineRunner commandLineRunner(ObservationRegistry registry) {
        return args -> {
            // Create an Observation and observe your code!
            Observation.createNotStarted( name: "user.name", registry)
                    .contextualName( s: "getting-user-name")
                    // let's assume that you can have 3 user types
                    .lowCardinalityKeyValue("userType", "userType1")
                    // let's assume that this is an arbitrary number
                    .highCardinalityKeyValue("userId", "1234")
                    // this is a shortcut for starting an observation, opening a scope,
                    // running user's code, closing the scope and stopping the observation
                    .observe(() -> log.info("Hello"));
        };
    }
}
```

```java
@SpringBootApplication
public class Application {

    private static final Logger log = LoggerFactory.getLogger(Application.class);

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

    @Bean
    @Observed(name = "command-line-runner")
    CommandLineRunner commandLineRunner(ObservationRegistry registry) {
        return args -> {
            log.info("Hello");
        };
    }
}
```

SPRING BOOT 3 - BUT WAIT THERE'S MORE!

HTTP Interface Client

RFC 7807 **Problem Details**

Spring **Data** 2022

Spring **Security** 6

Spring **Authorization** Server 1.0

Spring for **GraphQL** 1.1

More

SPRING BOOT 3.1

SOME REALLY COOL PROJECT

PostgreSQL 16.0 on port 5432

Kafka 3.4.1 on port 9092

Redis 6.0.20 on port 6379

Elasticsearch 8.8.2 in a 3 node cluster configuration on port 8201

MongoDB 6.0.8 in a 3 node cluster configuration on port 27017

Microservices

- The customer-service running on port 8081
- The product-service running on port 8082
- The order-service running on port 8083

git

RUN

DEVELOPER EXPERIENCE

Docker Compose

Testcontainers

Connection Details


docker

spring initializr

Project

- ○ Gradle - Groovy
- ○ Gradle - Kotlin
- ● Maven

Language

- ● Java
- ○ Kotlin
- ○ Groovy

Spring Boot

- ○ 3.2.0 (SNAPSHOT)
- ● 3.2.0 (RC1)
- ○ 3.1.6 (SNAPSHOT)
- ○ 3.1.5
- ○ 3.0.13 (SNAPSHOT)
- ○ 3.0.12
- ○ 2.7.18 (SNAPSHOT)
- ○ 2.7.17

Project Metadata

Group	dev.danvega
Artifact	cd
Name	cd
Description	Connection Details Demo
Package name	dev.danvega.cd
Packaging	● Jar ○ War
Java	● 21 ○ 17 ○ 11 ○ 8

Dependencies ADD DEPENDENCIES... ⌘ + B

Spring Web WEB

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Spring Data JDBC SQL

Persist data in SQL stores with plain JDBC using Spring Data.

PostgreSQL Driver SQL

A JDBC and R2DBC driver that allows Java programs to connect to a PostgreSQL database using standard, database independent Java code.

Docker Compose Support DEVELOPER TOOLS

Provides docker compose support for enhanced development experience.

demo.zip

- .gitignore
- .mvn
- HELP.md
- compose.yaml
- mvnw
- mvnw.cmd
- pom.xml
- src

DOWNLOAD COPY

```yaml
services:
  postgres:
    image: 'postgres:latest'
    environment:
      - 'POSTGRES_DB=mydatabase'
      - 'POSTGRES_PASSWORD=secret'
      - 'POSTGRES_USER=myuser'
    ports:
      - '5432'

```

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v3.2.0-RC1)

2023-11-01T11:19:57.979-04:00  INFO 76999 --- [           main] dev.danvega.cd.CdApplication             : Starting CdApplication using Java 21 with PID 76999 (/Users/vega/Downloads/cd/target/classes
2023-11-01T11:19:57.980-04:00  INFO 76999 --- [           main] dev.danvega.cd.CdApplication             : No active profile set, falling back to 1 default profile: "default"
2023-11-01T11:19:58.009-04:00  INFO 76999 --- [           main] .s.b.d.c.l.DockerComposeLifecycleManager : Using Docker Compose file '/Users/vega/Downloads/cd/compose.yaml'
2023-11-01T11:20:15.179-04:00  INFO 76999 --- [utReader-stderr] o.s.boot.docker.compose.core.DockerCli    :  Network cd_default  Creating
2023-11-01T11:20:15.204-04:00  INFO 76999 --- [utReader-stderr] o.s.boot.docker.compose.core.DockerCli    :  Network cd_default  Created
2023-11-01T11:20:15.204-04:00  INFO 76999 --- [utReader-stderr] o.s.boot.docker.compose.core.DockerCli    :  Container cd-postgres-1  Creating
2023-11-01T11:20:15.231-04:00  INFO 76999 --- [utReader-stderr] o.s.boot.docker.compose.core.DockerCli    :  Container cd-postgres-1  Created
2023-11-01T11:20:15.233-04:00  INFO 76999 --- [utReader-stderr] o.s.boot.docker.compose.core.DockerCli    :  Container cd-postgres-1  Starting
2023-11-01T11:20:15.409-04:00  INFO 76999 --- [utReader-stderr] o.s.boot.docker.compose.core.DockerCli    :  Container cd-postgres-1  Started
2023-11-01T11:20:15.409-04:00  INFO 76999 --- [utReader-stderr] o.s.boot.docker.compose.core.DockerCli    :  Container cd-postgres-1  Waiting
2023-11-01T11:20:15.916-04:00  INFO 76999 --- [utReader-stderr] o.s.boot.docker.compose.core.DockerCli    :  Container cd-postgres-1  Healthy
2023-11-01T11:20:17.521-04:00  INFO 76999 --- [           main] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data JDBC repositories in DEFAULT mode.
2023-11-01T11:20:17.528-04:00  INFO 76999 --- [           main] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 5 ms. Found 0 JDBC repository interfaces.
2023-11-01T11:20:17.708-04:00  INFO 76999 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port 8080 (http)
2023-11-01T11:20:17.712-04:00  INFO 76999 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2023-11-01T11:20:17.712-04:00  INFO 76999 --- [           main] o.apache.catalina.core.StandardEngine    : Starting Servlet engine: [Apache Tomcat/10.1.15]
2023-11-01T11:20:17.732-04:00  INFO 76999 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2023-11-01T11:20:17.733-04:00  INFO 76999 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 442 ms
2023-11-01T11:20:17.923-04:00  INFO 76999 --- [           main] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Starting...
2023-11-01T11:20:18.017-04:00  INFO 76999 --- [           main] com.zaxxer.hikari.pool.HikariPool        : HikariPool-1 - Added connection org.postgresql.jdbc.PgConnection@1213ffbc
2023-11-01T11:20:18.017-04:00  INFO 76999 --- [           main] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Start completed.
2023-11-01T11:20:18.093-04:00  INFO 76999 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port 8080 (http) with context path ''
2023-11-01T11:20:18.099-04:00  INFO 76999 --- [           main] dev.danvega.cd.CdApplication             : Started CdApplication in 20.271 seconds (process running for 20.598)
```

WHAT DOES THIS MEAN FOR ME?

You don't have to remember to run **docker compose up** before starting the application.

If the services are already running, it will detect that, too, and will use them.

The images started by Docker Compose are automatically detected and used to create **ConnectionDetails** beans pointing to the services.

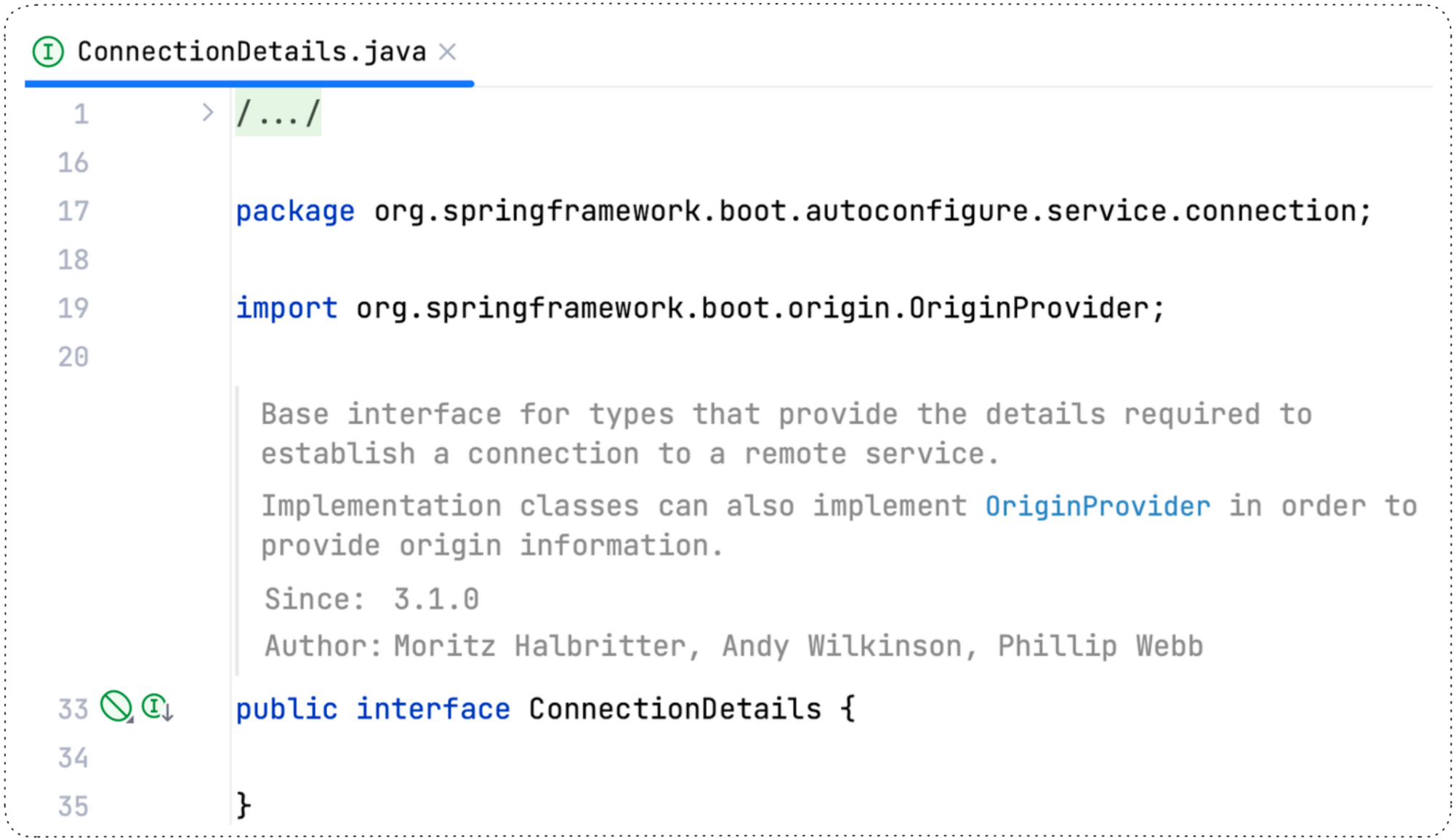
That means you don’t have to put properties in your configuration, you don’t have to remember how to construct PostgreSQL JDBC URLs, and so on.

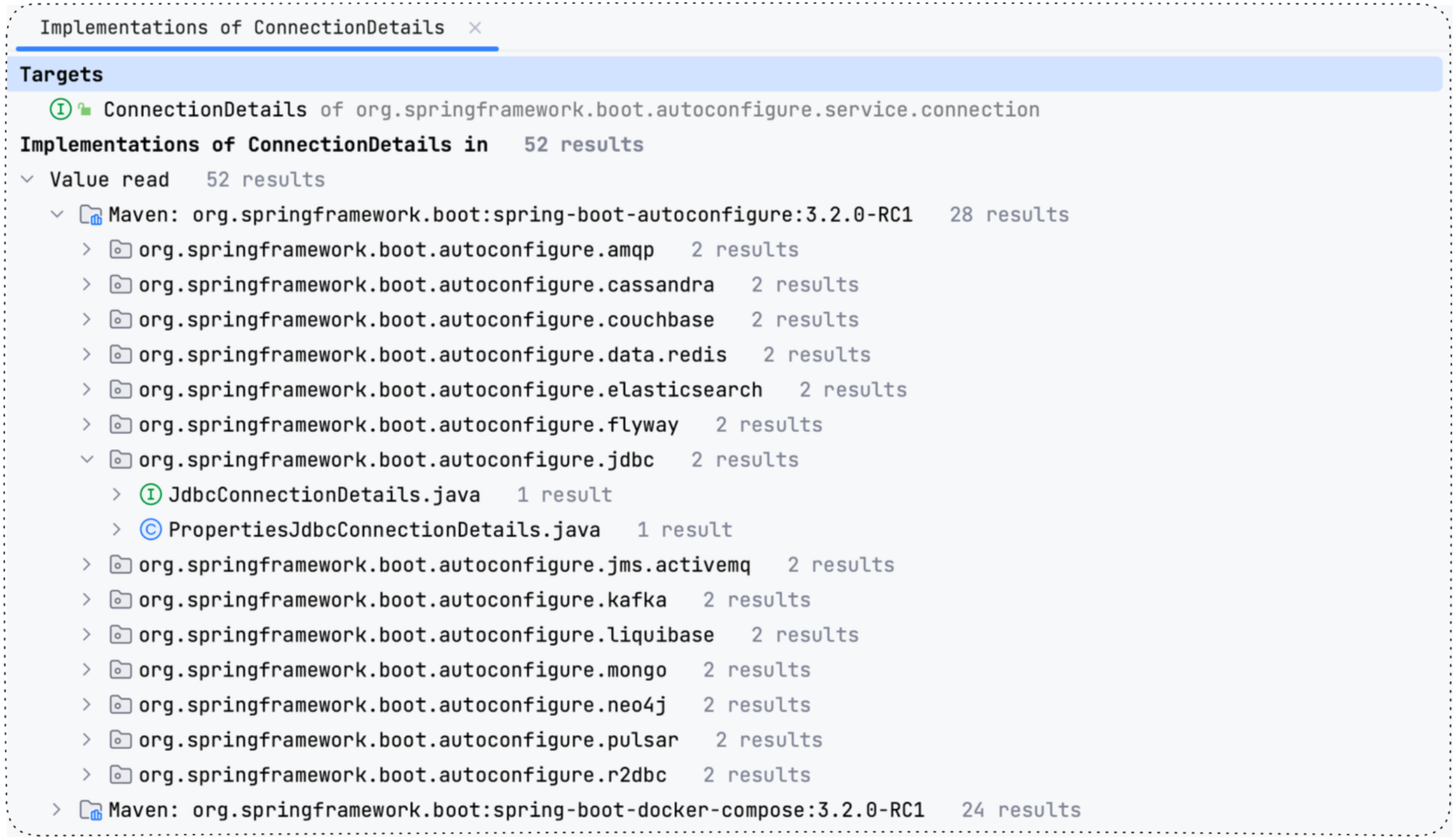
compose.yaml

```yaml
services:
  postgres:
    image: 'postgres:latest'
    environment:
      - 'POSTGRES_DB=mydatabase'
      - 'POSTGRES_PASSWORD=secret'
      - 'POSTGRES_USER=myuser'
    ports:
      - '5432'
```

application.properties

```properties
spring.datasource.url=jdbc:postgresql://localhost:32771/mydatabase
spring.datasource.username=myuser
spring.datasource.password=secret
```


ConnectionDetails.java
/.../
package org.springframework.boot.autoconfigure.service.connection;
import org.springframework.boot.origin.OriginProvider;
Base interface for types that provide the details required to establish a connection to a remote service.
Implementation classes can also implement OriginProvider in order to provide origin information.
Since: 3.1.0
Author: Moritz Halbritter, Andy Wilkinson, Phillip Webb
public interface ConnectionDetails {
}


Implementations of ConnectionDetails
Targets
ConnectionDetails of org.springframework.boot.autoconfigure.service.connection
Implementations of ConnectionDetails in 52 results
Value read 52 results
Maven: org.springframework.boot:spring-boot-autoconfigure:3.2.0-RC1 28 results
org.springframework.boot.autoconfigure.amqp 2 results
org.springframework.boot.autoconfigure.cassandra 2 results
org.springframework.boot.autoconfigure.couchbase 2 results
org.springframework.boot.autoconfigure.data.redis 2 results
org.springframework.boot.autoconfigure.elasticsearch 2 results
org.springframework.boot.autoconfigure.flyway 2 results
org.springframework.boot.autoconfigure.jdbc 2 results
JdbcConnectionDetails.java 1 result
PropertiesJdbcConnectionDetails.java 1 result
org.springframework.boot.autoconfigure.jms.activemq 2 results
org.springframework.boot.autoconfigure.kafka 2 results
org.springframework.boot.autoconfigure.liquibase 2 results
org.springframework.boot.autoconfigure.mongo 2 results
org.springframework.boot.autoconfigure.neo4j 2 results
org.springframework.boot.autoconfigure.pulsar 2 results
org.springframework.boot.autoconfigure.r2dbc 2 results
Maven: org.springframework.boot:spring-boot-docker-compose:3.2.0-RC1 24 results

SPRING BOOT 3.1

Using **Testcontainers** at Development Time

Docker Compose **spring-boot-docker-compose** module

Auto-Configuration for **Spring Authorization Server**

Building **Docker Images** with Maven or Gradle

Spring for GraphQL 1.2

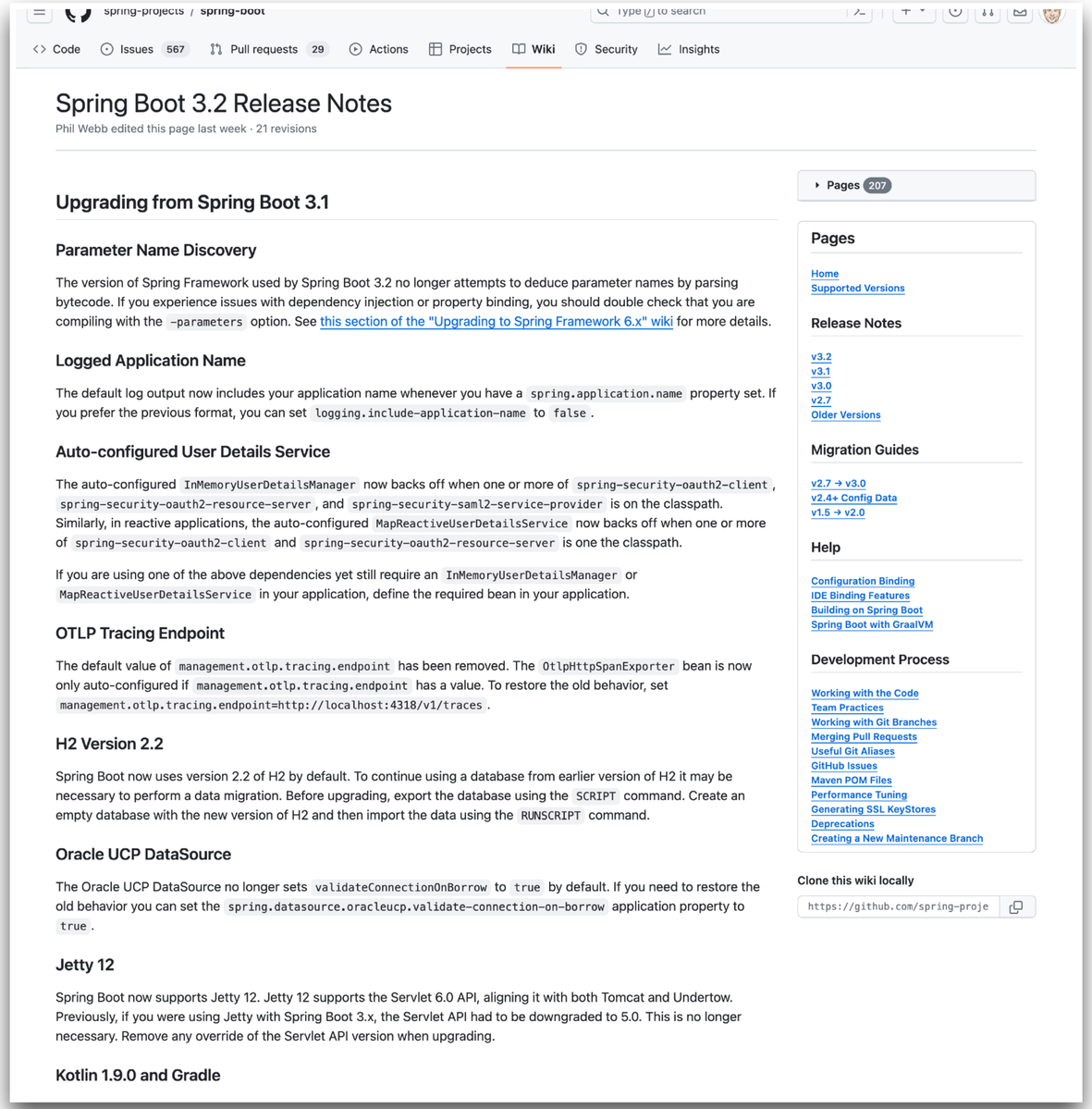
Dependency Upgrades

SPRING BOOT 3.2

SPRING BOOT 3.2

Release Notes

- Runtime Efficiency
  - Virtual Threads
  - Project CRaC (Coordinated Restore at Checkpoint)
- Client Abstractions
  - REST Client
  - JDBC Client
- Spring for Apache Pulsar Support
- SSL Bundle Reloading
- Observability Improvements
- More...



RUNTIME EFFICIENCY

PROJECT CRAC (COORDINATED RESTORE AT CHECKPOINT)

- Faster Startup Times (short time to first transaction)
  - Similar goals to AOT
- Snapshot of the Java process (checkpoint)
- Uses the snapshot to launch any number of JVMs
- CRaC API
- Spring Boot 3.2 adds initial support for CRaC
  - Automatic checkpoints

PROJECT LOOM / VIRTUAL THREADS

WHAT IS A THREAD IN JAVA

- Java is Multithreaded by default
- java.lang.Thread
  - Abstraction over operating system threads
  - Every thread is essentially a wrapper over an OS thread
- OS/JVM thread = platform thread
- Platform threads are expensive
- Take a lot of memory / take time to create & destroy
- Traditional threads are stored on the JVM stack

Project Loom is to intended to explore, incubate and deliver Java VM features and APIs built on top of them for the purpose of supporting **easy-to-use**, **high-throughput lightweight concurrency** and new programming models on the Java platform.

PROJECT LOOM

Project Loom includes the following JEPs.

- JEP 425: Virtual Threads went final in JDK 21
- JEP 428: Structured Concurrency (Preview) 2nd preview in JDK 21.
- JEP 429: Scoped Values (Preview) 2nd preview in JDK 21.

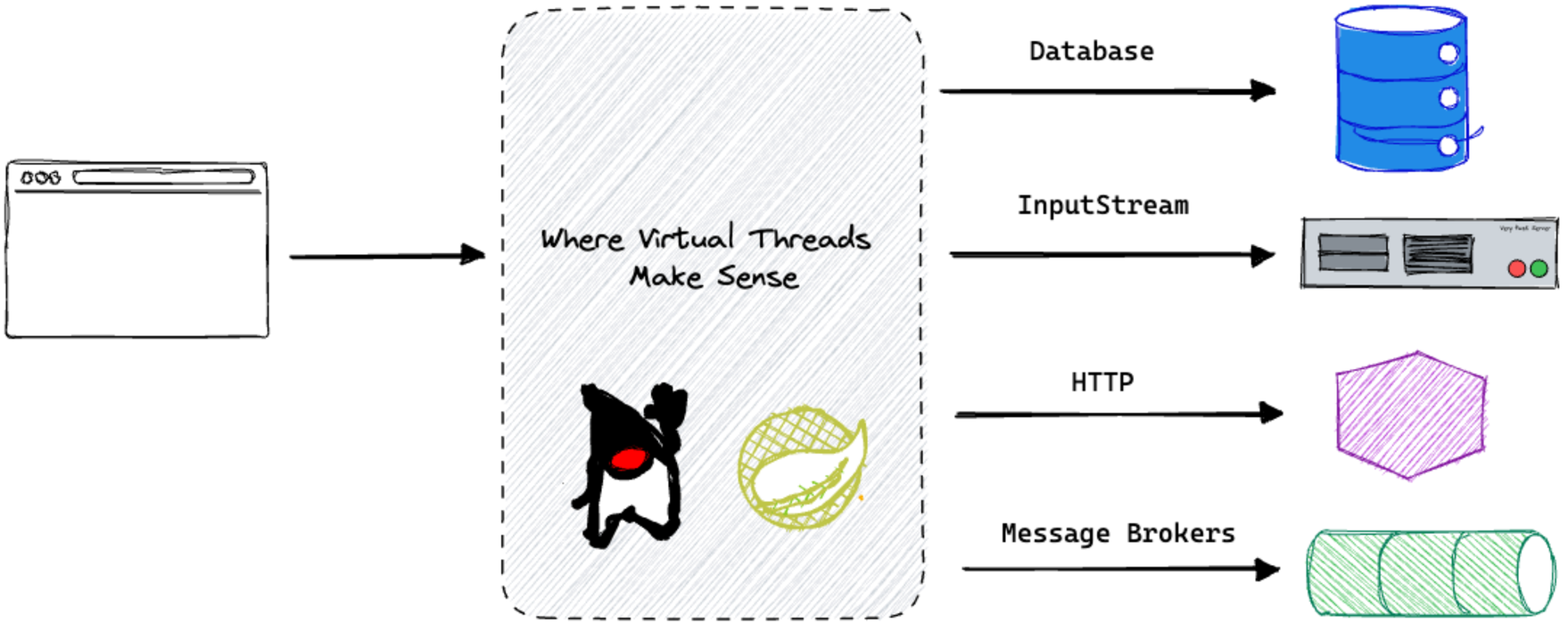
"PROJECT LOOM" VIRTUAL THREADS

- Final in JDK 21
- Not Linked to an OS thread
- Use the heap for the stack (potential implications for GC)
- Have their own scheduler
- Created for a task and then allowed to terminate
- Do not pool virtual threads

THREAD-PER-REQUEST

Embracing Virtual Threads

Project Loom aims to reduce the effort of writing, maintaining and observing high-throughput concurrent applications.



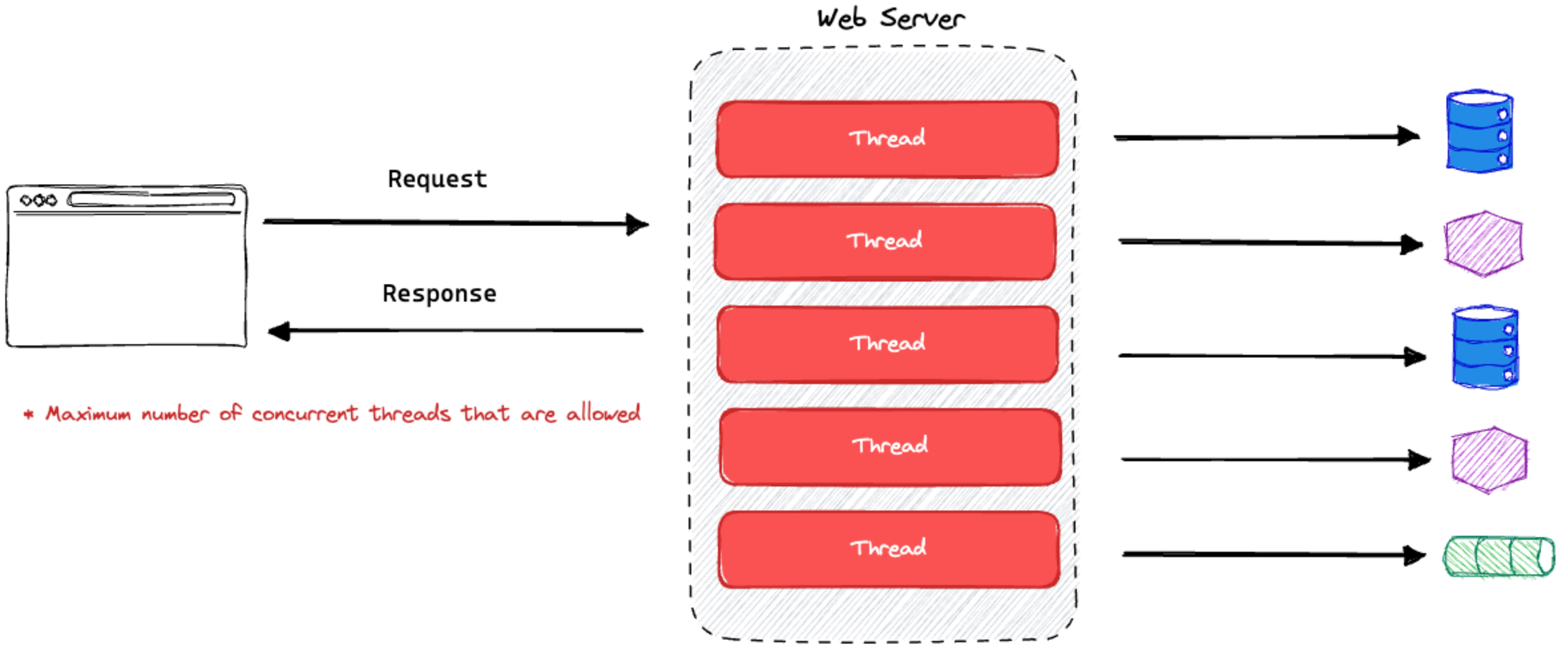
This application works great on your laptop and even in the initial stages of implementation. However, once news spreads about your exceptional new application, you start experiencing a high volume of traffic causing it to crash.

Why is this happening?

What can you do to prevent this?

spring

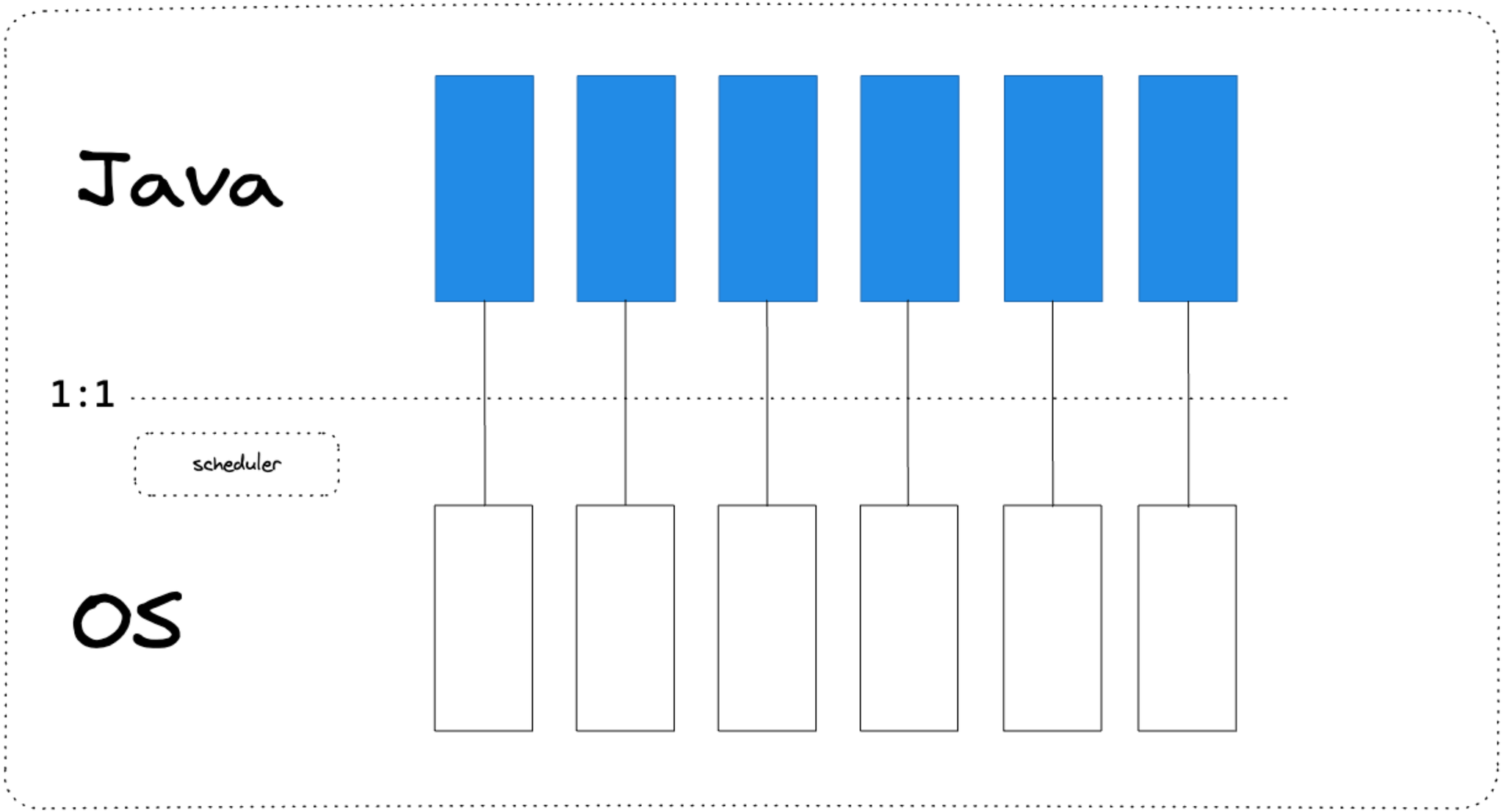
Thread Per Request



When the number of maximum threads has been reached each subsequent request will need to wait for a thread to be released to fulfill that request

spring

Java is Made of Threads



Scalability Solutions

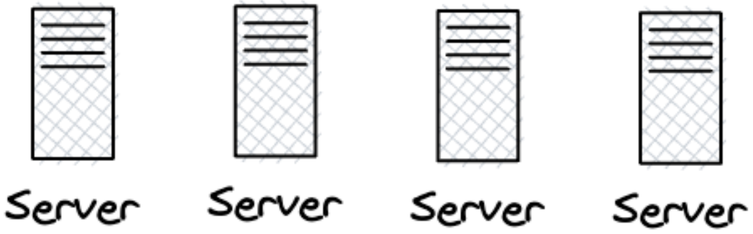
Hardware

You can address this issue by adding more hardware

Scaling Vertically

Memory CPU

Scaling Horiztonally



Asynchronous Programming

You can also solve this problem by writing non-blocking software and like any architecture choice there are pros/cons to each.

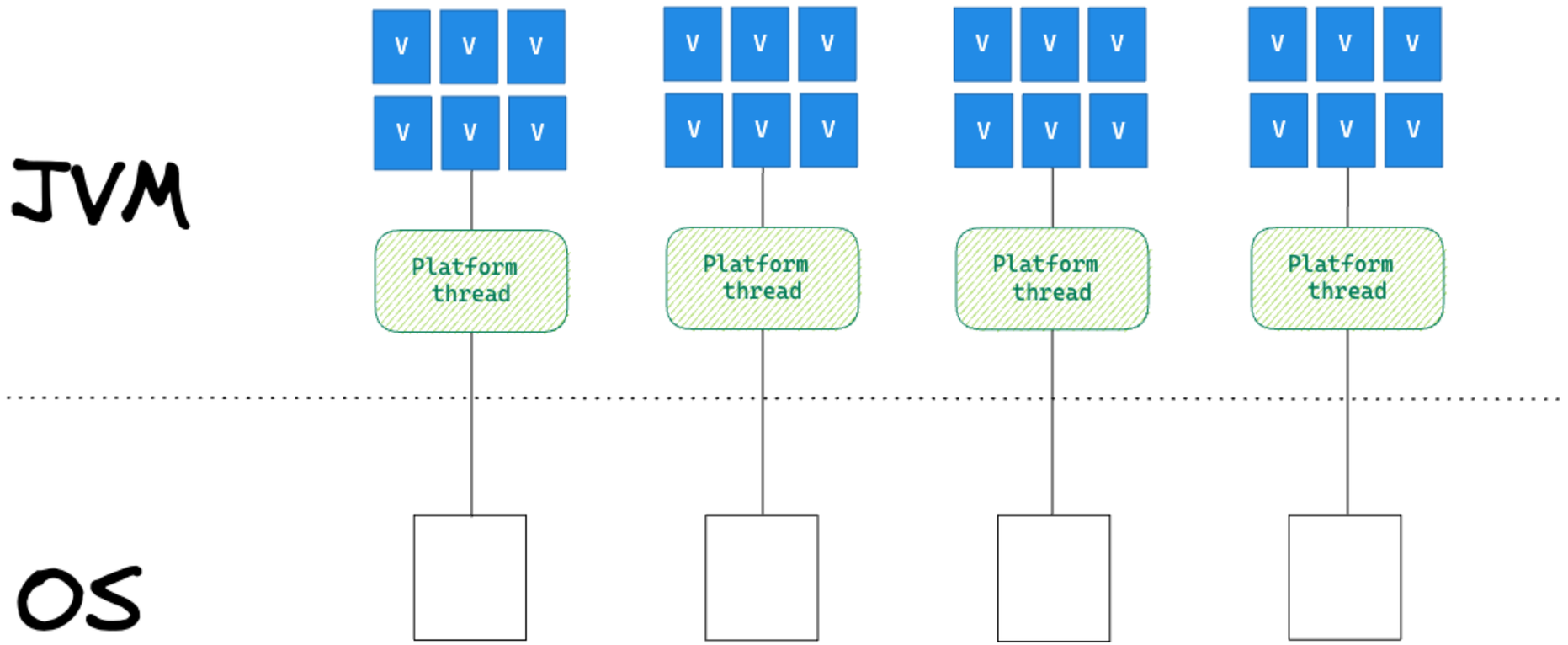
Pros

- Improved performance and responsiveness
- Improve performance when dealing with large datasets
- Scalability / Better resource utilization
- Composability

Cons

- All-or-nothing
- Separates the task from the thread
- Reactive Programming has a learning curve
- Can be harder to debug or profile

Virtual Threads



High-throughput servers with simple thread-per-request code using the same APIs

spring

VIRTUAL THREADS & SPRING APPLICATIONS

EMBRACING VIRTUAL THREADS IN SPRING APPLICATIONS

Web Applications.

- Servlet callback based on Input / Output Stream
- Tomcat / Jetty Executor setup for dispatching to request handlers
- “Loom-ready” database drivers, scalable connection pool setup
- Ideally no code changes in the main application codebase
- Virtual threads supported in GraalVM Native Image as well

- Significant scalability benefits for database driven web applications

EMBRACING VIRTUAL THREADS IN SPRING APPLICATIONS

Messaging / Scheduling

- Spring managed task executors with virtual thread options
  - JMS Message Listener containers
  - @Scheduled handler methods
- Many of those listeners and handlers methods trigger I/O operations

EMBRACING VIRTUAL THREADS IN SPRING APPLICATIONS

Where it doesn't make sense

- Purely CPU bound handler methods
- Reactive based web applications

VIRTUAL THREADS

Loom vs Reactive Programming

- Virtual Threads provide a different scalability mechanism
- Traditional imperative programming style
- Empowering Spring MVC to reach maximum scalability
- Spring Webflux as an architecture for stream based processing
- Reactive: Stream based processing not primarily for scalability (anymore)

NEW CLIENT ABSTRACTIONS

REST CLIENT

HISTORY OF CLIENT ABSTRACTIONS

- Rest Template
- Web Client
- Graphql Client
- Declarative Http Client
  - Spring Cloud Open Feign
  - Http Interfaces
- Rest Client

REST CLIENT

Perform HTTP Requests & Return Responses

Exposes a Fluent & Synchronous API

Builds on everything we learned from RestTemplate & WebClient

Http Interfaces using the Rest Client in an imperative application

Underlying HTTP Client Libraries

- JDK HttpClient (default)
- Apache HttpClient

JDBC CLIENT

HISTORY OF DATABASE ABSTRACTIONS

- **J**ava **D**ata**B**ase **C**onnection (JDBC)
- JDBC Template
- Spring Data
  - Spring Data JPA
  - Spring Data JDBC
- JDBC Client


DEMO

SPRING FRAMEWORK 6.1 - NOVEMBER 2023

- Foundation for Spring Boot 3.2 & 3.3
  - Spring Boot 3.2 - November 2023
  - Spring Boot 3.3 - May 2024
- Production-ready on JDK 17 & 21 LTS

THANK YOU

dan.vega@broadcom.com

@therealdanvega

https://www.danvega.dev