

GOTO **AMSTERDAM 2023**

Spring Framework 6

#GOTOams

Sam Brannen
@sam_brannen

Disclaimer

- This presentation may contain product features or functionality that are currently under development.
- This overview of new technology represents no commitment from VMware to deliver these features in any generally available product.
- Features are subject to change, and must not be included in contracts, purchase orders, or sales agreements of any kind.
- Technical feasibility and market demand will affect final delivery.
- Pricing and packaging for any new features/functionality/technology discussed or presented, have not been determined.
- The information in this presentation is for informational purposes only and may not be incorporated into any contract. There is no commitment or obligation to deliver any items presented herein.

Sam Brannen

- Staff Software Engineer



- Java Developer for 25 years



- Spring Framework Core Committer since 2007



- JUnit 5 Core Committer since 2015



Topics

- Spring Framework 6.0
 - Baseline Upgrades
 - Declarative HTTP Clients
 - Spring AOT and GraalVM
- Spring Framework 6.1 and Beyond
 - Project Leyden
 - Virtual Threads (Project Loom)
 - JVM Checkpoint Restore (Project CRaC)

Java and Jakarta

Baseline: JDK 17 LTS

- **Language:** text blocks, switch expressions, instanceof pattern matching
- **Core libraries:** collection factory methods, etc.
- **Type system:** records, sealed classes
- *A great baseline for modern-day Java!*

Coming up: JDK 21 LTS

- **Language:** pattern matching for switch, record patterns
- **Core libraries:** sequenced collections, etc.
- **Runtime:** virtual threads, generational ZGC
- *Fully supported in Spring Framework 6.1 already*

Baseline: 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
- *Same APIs as with Java EE 8, just in a different namespace*

Current: Jakarta EE 10

- **Servlet API 6.0:** e.g. Tomcat 10.1, Jetty 12
 - Servlet 6.0 in the build, Servlet 5.0 compatibility at runtime
 - Servlet 6.0 for mocks in spring-test
- **JPA 3.1:** e.g. Hibernate ORM 6.2
- **Bean Validation 3.0:** e.g. Hibernate Validator 8.0
- *Default EE API level in Spring Boot 3*

Coming up: Jakarta EE 11

- **JDK 21** as the official minimum requirement ⚠
- **Servlet API 6.1** embracing virtual threads
- **Tomcat 11** expected to require JDK 21 as well
- *Optional Tomcat 11 upgrade in Spring Boot 3.3 ?*

Odds and Ends

Module-path and Class-path Scanning Enhancements

- Module path scanning support for "classpath*:" resource prefix
 - For example, with a patched module using Maven Surefire
- Class path scanning support in custom filesystems
 - For example, the GraalVM native image filesystem

Declarative HTTP Clients

Interface-based HTTP Clients

- Annotate an interface – Spring translates it into actual HTTP client requests
- Analogous to OpenFeign/Feign but without the related issues
 - lack of non-blocking support, dependency on third parties, etc.
- Infrastructure and annotations live in Core Spring (spring-web)

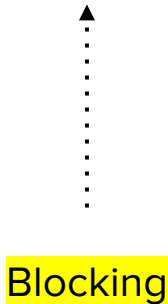
HTTP Client Annotations and Proxies

- `@HttpExchange`, `@PostExchange`, `@GetExchange`, etc.
 - instead of `@RequestMapping`, `@PostMapping`, etc.
- Reuses several parameter-level annotations from Spring MVC
 - `@RequestHeader`, `@PathVariable`, `@RequestBody`, etc.
- `HttpServiceProxyFactory`: creates proxies for annotated client interfaces

Example: @PostExchange - Blocking

```
public interface VerificationService {  
  
    @PostExchange("/verify")  
    CustomerVerificationResult verify(@RequestBody CustomerApplication app);  
  
}
```

Blocking



Example: @PostExchange - Reactive

```
public interface VerificationService {  
  
    @PostExchange("/verify")  
    Mono<CustomerVerificationResult> verify(@RequestBody CustomerApplication app);  
  
}
```

Reactive



Example: @PostExchange - ResponseEntity

```
public interface VerificationService {  
  
    @PostExchange("/verify")  
    ResponseEntity<CustomerVerificationResult> verify(@RequestBody CustomerApplication app);  
}
```



status, headers, body

Creating an HTTP Service Proxy

```
VerificationService client =  
    httpServiceProxyFactory.createClient(VerificationService.class);
```



```
@Bean  
public HttpServiceProxyFactory httpServiceProxyFactory() {  
    WebClient client = WebClient.create(this.baseUrl);  
    return new HttpServiceProxyFactory(new WebClientAdapter(client));  
}
```

Ahead-Of-Time

Spring AOT

- Reduces startup time and memory footprint in production
- **Runtime hints** for reflection, resources, serialization, proxies
- Optional for optimized JVM deployments
- Precondition for **GraalVM native executables**
- Core infrastructure in Spring Framework 6
- **Build tools** in Spring Boot 3
- **Test** within a native image with JUnit 5 and GraalVM Native Build Tools

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

GraalVM Native Image

- GraalVM is the de-facto standard for native executables
- Strong **closed-world assumption**, no runtime adaptations
- AOT-processed application as input → native executable
- Very **long build time** for actual native code generation

A different mode of deployment with strong benefits and limitations

Project Leyden

- <https://openjdk.org/projects/leyden/>
- OpenJDK aims to introduce **well-defined static images**
 - For example, custom HotSpot-based runtime images for specific applications
- **Incremental approach:** weaker constraints → more runtime flexibility
 - Strict closed-world constraints as the final goal

Spring's AOT strategy aligns with Leyden's JVM strategy

Virtual Threads

Project Loom: Virtual Threads preview in JDK 19

- Lightweight threading model within the JVM
- Designed as virtual variant of **java.lang.Thread**
- Better scalability for **imperative programming**
- *Not blocking an operating system thread on I/O operations*

JDK 21 takes Virtual Threads out of preview

- A regular JVM feature now, even part of an LTS release
- New builder API on **java.lang.Thread**
- New **java.util.concurrent.ExecutorService** variants
- *Seamless interoperability with existing code*
- *Avoid synchronization around I/O operations!*

Virtual Threads in Spring Framework 6.1

- New **virtualThreads** flag in **SimpleAsyncTaskExecutor**
- Dedicated **VirtualThreadTaskExecutor** variant
- *A simple replacement for existing TaskExecutor setups*
- *Individually configurable for messaging, scheduling, etc.*

Virtual Threads for Spring MVC applications

- **Tomcat/Jetty executor setup** for virtual threads
- Latest **database drivers** behind JDBC and JPA
- First-class setup option expected for **Spring Boot 3.2**
- *Ideally no changes necessary in the application codebase*

Spring WebFlux and Virtual Threads

- WebFlux: scalability through a **reactive programming model**
- Stream-based access with **backpressure-enabled** drivers
- Efficient CPU usage through non-blocking handling already
- Can **run potentially blocking user tasks on virtual threads**
- *A reactive-centric web stack with blocking escape options*

Spring MVC and Reactive Programming

- Spring MVC **understands reactive return types** as well (!)
- E.g. reactive datastore access **for specific web endpoints**
- Presence of Reactor necessary for reactive web endpoints
- That aside, **Spring MVC is a lean stack on virtual threads**
- *A virtual-thread-powered web stack with reactive options*

Benefits of Virtual Threads

- **Higher scalability** for existing applications
- Or same scalability with a **smaller footprint**
- Strong benefits for **JDBC/JPA interactions**
- A good fit for HTTP interactions via RestTemplate
- *Take your Spring web applications for a test run!*

JVM Checkpoint Restore

Project CRaC: Coordinated Restore at Checkpoint

- <https://github.com/CRaC/docs>
- Bootstrapping from a **warmed-up HotSpot JVM image**
- Originally developed by Azul for **OpenJDK on Linux**
- Adopted by Amazon for AWS Lambda SnapStart
- *A simple approach towards immediate JVM startup*

Project CRaC: Requirements

- **At checkpoint time**, pause the application
- No open network connections
- No open file handles
- **At restore time**, re-establish connections / listeners

Project CRaC support in Spring Framework 6.1

- <https://docs.spring.io/spring-framework/reference/6.1/integration/checkpoint-restore.html>
- Custom checkpoints after application startup (+ warmup)
- ApplicationContext gets **checkpoint/restore notifications**
- Propagates stop/restart signals to participating beans
- **org.springframework.context.Lifecycle** interface

Recommendations for Lifecycle implementations

- For example, Spring's own JMS message listener containers
- Stop all asynchronous processing on **Lifecycle.stop**
- Keep state intact; **stop is not equal to destroy call**
- Remain able to restart async work on **Lifecycle.start**
- When **destroy follows stop**, shut down completely

Common Spring apps working out-of-the-box

- **Standard lifecycle coordination** through the framework
- **Embedded Tomcat/Jetty** to participate in stop/restart
- **Spring Boot 3.2** can support common checkpoint choices
- Alternatively, custom checkpoints will be possible as well
- *Ideally no changes necessary in the application codebase*

Roadmap

Spring Framework 6.1 + Spring Boot 3.2

- Spring Framework 6.1 M2 in mid July
- Spring Boot 3.2 M1 in mid July
- **Release candidates in October 2023**
- **General availability in November 2023**
- *Give our milestones a try on JDK 21 EA builds!*

Thank You!

Sam Brannen

@sam_brannen