Fullstack in a native.image

Java Vienna - Dominik Dorn

About Me

— Dominik Dorn
— Software Engineer / Freelancer
— Java since 2006
— Twitter/X: x.com/domdorn
— LinkedIn: linkedin.com/in/dominik-dorn/

Agenda

1. What is GraalVM Native Image?
2. Demo Tech Stack
3. Demo #1: LiveChat - Vaadin + DB in a Native Image
4. Demo #2: Pplan - REST, HTMX, JobRunR
5. Obstacles & Lessons Learned
6. Performance

What is GraalVM Native Image?

— Ahead-of-Time (AOT) compiled Java application
— Produces a standalone native executable
— No JVM needed at runtime
— Sub-second startup times
— Lower memory footprint

Why Native Images?

- **Fast startup**: great for serverless, CLI tools, microservices
- **Low memory**: less overhead without the JVM
- **Instant peak performance**: no JIT warmup
- **Single binary**: easy to deploy, containerize

The Catch

— No dynamic class loading at runtime
— Reflection needs to be declared at build time
— Build takes longer (minutes vs seconds)
— Not all libraries support it (yet)

Demo Tech Stack

Component	Version
Java	25
Spring Boot	4.0.2
Vaadin	25.0.4
GraalVM	25
PostgreSQL	18
Flyway	via starter
JobRunR	8.4.2

Demo #1: LiveChat

A real-time chat app as a native binary

Project Setup

```xml
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>4.0.2</version>
</parent>

<properties>
    <java.version>25</java.version>
    <vaadin.version>25.0.4</vaadin.version>
</properties>
```

Dependencies: Vaadin BOM

Vaadin needs a BOM for version management:

```xml
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>com.vaadin</groupId>
            <artifactId>vaadin-bom</artifactId>
            <version>${vaadin.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Dependencies: Vaadin

```xml
<dependency>
    <groupId>com.vaadin</groupId>
    <artifactId>vaadin-spring-boot-starter</artifactId>
</dependency>

<!-- Required for dev mode in Vaadin 25! -->
<dependency>
    <groupId>com.vaadin</groupId>
    <artifactId>vaadin-dev</artifactId>
    <optional>true</optional>
</dependency>
```

Version comes from the BOM - no version tag needed here.

Dependencies: Database

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
</dependency>
<!-- Spring Boot 4: must use the STARTER! -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-flyway</artifactId>
</dependency>
```

The Application Class

```java
@SpringBootApplication
@Push
public class Application implements AppShellConfigurator {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

- `@Push` enables WebSocket server push for Vaadin
- `AppShellConfigurator` lets Vaadin configure the HTML shell

Domain Model with Records

```java
public record ChatMessage(
    Long id,
    String username,
    String content,
    Instant createdAt
) {
    public static ChatMessage create(String username, String content) {
        return new ChatMessage(null, username, content, Instant.now());
    }
}
```

Records work great with native images - simple, immutable, predictable.

Why JdbcClient? Why not JDBI?

We tried JDBI first. It failed spectacularly.

```
NoSuchMethodException: No constructor for class
  'org.jdbi.v3.core.config.internal.ConfigCaches'
NoSuchMethodException: No constructor for class
  'org.jdbi.v3.core.internal.OnDemandExtensions'
NoSuchMethodException: No constructor for class
  'org.jdbi.v3.sqlobject.statement.internal.
    SqlObjectStatementConfiguration'
```

JDBI uses heavy internal reflection. 50+ classes would need hints. Not practical.

Spring JdbcClient to the rescue

```java
@Repository
public class JdbcChatMessageRepository
    implements ChatMessageRepository {

    private final JdbcClient jdbcClient;

    public JdbcChatMessageRepository(JdbcClient jdbcClient) {
        this.jdbcClient = jdbcClient;
    }
```

Introduced in Spring Framework 6.1. First-class native image support via Spring AOT.

JdbcClient: Save

```java
@Override
public ChatMessage save(ChatMessage message) {
    KeyHolder keyHolder = new GeneratedKeyHolder();
    jdbcClient.sql("""
        INSERT INTO chat_message
          (username, content, created_at)
        VALUES (?, ?, ?)""")
        .param(message.username())
        .param(message.content())
        .param(Timestamp.from(message.createdAt()))
        .update(keyHolder, "id");

    Long id = keyHolder.getKeyAs(Long.class);
    return new ChatMessage(
        id, message.username(),
        message.content(), message.createdAt());
}
```

JdbcClient: Query

```java
@Override
public List<ChatMessage> findRecentMessages(int limit) {
    return jdbcClient.sql("""
        SELECT id, username, content, created_at
        FROM chat_message
        ORDER BY created_at DESC LIMIT ?""")
        .param(limit)
        .query((rs, rowNum) -> new ChatMessage(
            rs.getLong("id"),
            rs.getString("username"),
            rs.getString("content"),
            rs.getTimestamp("created_at").toInstant()
        ))
        .list()
        .reversed();
}
```

Vaadin UI: ChatView

```java
@Route("")
public class ChatView extends VerticalLayout {
    private final ChatMessageRepository repository;
    private final Div messageList;
    private Registration broadcasterRegistration;

    public ChatView(ChatMessageRepository repository) {
        this.repository = repository;
        // ... build UI components
        loadRecentMessages();
    }
```

Broadcaster

```java
public class Broadcaster {
    static final Executor executor =
        Executors.newSingleThreadExecutor();
    static final LinkedList<Consumer<ChatMessage>>
        listeners = new LinkedList<>();

    public static synchronized Registration
        register(Consumer<ChatMessage> listener) {
        listeners.add(listener);
        return () -> {
            synchronized (Broadcaster.class) {
                listeners.remove(listener);
            }
        };
    }

    public static synchronized void
        broadcast(ChatMessage message) {
        for (var listener : listeners)
            executor.execute(
                () -> listener.accept(message));
    }
}
```

Receiving Push Updates

```java
@Override
protected void onAttach(AttachEvent attachEvent) {
    UI ui = attachEvent.getUI();
    broadcasterRegistration = Broadcaster.register(
        message -> {
            ui.access(() -> {
                displayMessage(message);
                scrollToBottom();
            });
        });
}

@Override
protected void onDetach(DetachEvent detachEvent) {
    if (broadcasterRegistration != null) {
        broadcasterRegistration.remove();
    }
}
```

`ui.access()` is essential - it ensures thread-safe UI updates.

Flyway Migration

```sql
CREATE TABLE chat_message (
    id         BIGSERIAL PRIMARY KEY,
    username   VARCHAR(100) NOT NULL,
    content    TEXT NOT NULL,
    created_at TIMESTAMP WITH TIME ZONE
               DEFAULT CURRENT_TIMESTAMP
);

CREATE INDEX idx_chat_message_created_at
    ON chat_message(created_at DESC);
```

The Key: RuntimeHintsRegistrar

```java
@Configuration
@ImportRuntimeHints(NativeImageConfig.AppRuntimeHints.class)
public class NativeImageConfig {

    static class AppRuntimeHints
        implements RuntimeHintsRegistrar {
        @Override
        public void registerHints(
            RuntimeHints hints, ClassLoader classLoader) {

            // Domain classes for JDBC mapping
            hints.reflection().registerType(
                ChatMessage.class, MemberCategory.values());

            // Flyway migration files
            hints.resources().registerPattern("db/migration/*");
        }
    }
}
```

This is the Spring way to tell GraalVM what to include.

Maven Profiles

```xml
<profile>
    <id>production</id>
    <!-- Vaadin production frontend build -->
</profile>

<profile>
    <id>native</id>
    <build><plugins>
        <plugin>
            <groupId>org.graalvm.buildtools</groupId>
            <artifactId>native-maven-plugin</artifactId>
            <configuration>
                <imageName>livechat</imageName>
                <buildArgs>
                    <buildArg>--no-fallback</buildArg>
                    <buildArg>-H:+ReportExceptionStackTraces
                    </buildArg>
                </buildArgs>
            </configuration>
        </plugin>
    </plugins></build>
</profile>
```

Building & Running

```
# Development mode
./mvnw spring-boot:run

# Build native image
./mvnw -Pnative,production native:compile

# Run the native binary
./target/livechat
```

That's it. A single binary. No JVM required.

Performance: LiveChat

Metric	JVM Mode	Native Image
Startup Time	~2.0s	**0.13s**
Improvement	-	**15x faster**
Binary Size	N/A	~147 MB
Build Time	~2s	~75s

Demo #2: Pplan

REST + HTMX + Vaadin + JobRunR

Pplan: What is it?

- A doctor search application for Austria
- Vaadin 25 frontend for users (search, map, filters)
- REST API + HTMX admin dashboard
- Background jobs for web crawling (JobRunR)
- All running as a single native binary

Adding REST Endpoints

```java
@RestController
@RequestMapping("/api/admin/crawl")
public class AdminCrawlController {

    @PostMapping("/wien/search")
    public Map<String, String> triggerSearch() {
        crawlJobScheduler.triggerSearchCrawl(1);
        return Map.of("status", "Search crawl enqueued");
    }

    @PostMapping("/wien/fetch")
    public Map<String, String> triggerFetch() {
        crawlJobScheduler.triggerDetailFetch(1);
        return Map.of("status", "Detail fetch enqueued");
    }
}
```

REST and Vaadin co-exist in the same Spring Boot app.

HTMX Admin Dashboard

```
<script src="https://unpkg.com/htmx.org@2.0.4">
</script>

<button onclick="triggerJob('/api/admin/crawl/wien/search',
    this)">Search Crawl</button>


async function triggerJob(url, btn) {
    btn.disabled = true;
    try {
        const res = await fetch(url, {method: 'POST'});
        const data = await res.json();
        showToast(data.status || 'Job enqueued');
    } catch (e) {
        showToast('Error: ' + e.message);
    } finally {
        setTimeout(() => btn.disabled = false, 3000);
    }
}
```

Serving Static HTML in Native Image

```
@RestController
public class AdminPageController {

    @GetMapping({"/admin", "/admin/", "/admin/index.html"})
    public ResponseEntity<byte[]> adminPage() {
        // Serve static/admin/index.html
    }
}
```

Register the resource in RuntimeHints:

```
hints.resources().registerPattern("static/admin/*");
```

Multi-Port Architecture

Port	Purpose	Auth
50080	Vaadin UI (public)	None
50081	JobRunR Dashboard	-
50082	Admin REST + HTMX	HTTP Basic

```
@Component
public class AdminPortConfig {
    // Adds secondary Tomcat connector on admin port
}
```

Admin paths blocked on public port, non-admin paths blocked on admin port.

JobRunR: Setup

```xml
<dependency>
    <groupId>org.jobrunr</groupId>
    <artifactId>jobrunr-spring-boot-4-starter</artifactId>
    <version>8.4.2</version>
</dependency>
```

Note: There's a dedicated Spring Boot 4 starter!

JobRunR: Configuration

```
jobrunr.background-job-server.enabled=true
jobrunr.dashboard.enabled=true
jobrunr.dashboard.port=50081
jobrunr.database.skip-create=false
jobrunr.jobs.default-number-of-retries=3
jobrunr.background-job-server.poll-interval-in-seconds=15

# Virtual Threads!
jobrunr.background-job-server.thread-type=VirtualThreads
```

JobRunR: Why JobRequest pattern?

The typical JobRunR lambda approach:

```
// This does NOT work in native images!
BackgroundJob.enqueue(
    () -> myService.doWork());
```

Lambdas require serialization + reflection that breaks in native images.

JobRunR: The Native-Friendly Way

```java
public record PopulateReferenceDataRequest()
    implements JobRequest {
    @Override
    public Class<PopulateReferenceDataHandler>
        getJobRequestHandler() {
        return PopulateReferenceDataHandler.class;
    }
}

@Component
public static class PopulateReferenceDataHandler
    implements JobRequestHandler
        <PopulateReferenceDataRequest> {
    @Override
    public void run(PopulateReferenceDataRequest req)
        throws Exception {
        referenceDataService.populateReferenceData();
    }
}
```

Records as JobRequests + explicit handler classes = native image friendly.

JobRunR: Enqueuing Jobs

```java
@Component
public class CrawlJobScheduler {

    private final JobRequestScheduler jobRequestScheduler;

    @EventListener(ApplicationReadyEvent.class)
    public void onStartup() {
        jobRequestScheduler.enqueue(
            new PopulateReferenceDataRequest());
    }

    public void triggerSearchCrawl(int sourceId) {
        jobRequestScheduler.enqueue(
            new SearchCrawlRequest(sourceId));
    }
}
```

JobRunR: Native Image Hints

JobRunR state classes also need reflection hints:

```java
// JobRunR request & handler types
hints.reflection().registerType(
    CrawlJobScheduler.PopulateReferenceDataRequest.class,
    MemberCategory.values());
hints.reflection().registerType(
    CrawlJobScheduler.PopulateReferenceDataHandler.class,
    MemberCategory.values());
// ... for every request/handler pair
```

JobRunR: reachability-metadata.json

JobRunR state classes need separate metadata:

```
[
  { "type": "org.jobrunr.jobs.Job",
    "allDeclaredFields": true,
    "allDeclaredMethods": true,
    "allDeclaredConstructors": true },
  { "type": "org.jobrunr.jobs.states.EnqueuedState",
    "allDeclaredFields": true, ... },
  { "type": "org.jobrunr.jobs.states.ProcessingState", ... },
  { "type": "org.jobrunr.jobs.states.SucceededState", ... },
  { "type": "org.jobrunr.jobs.states.FailedState", ... },
  { "type": "org.jobrunr.jobs.states.DeletedState", ... },
  { "type": "org.jobrunr.jobs.states.ScheduledState", ... }
]
```

Place in `META-INF/native-image/.../reachability-metadata.json`

Virtual Threads + JobRunR

```
jobrunr.background-job-server.thread-type=VirtualThreads
```

— Uses Java 21+ virtual threads for background job execution

— Default thread pool: **16 x CPU core count**

— Many concurrent jobs with minimal OS thread overhead

— Especially useful for I/O-bound jobs (HTTP calls, DB queries)

RuntimeHints in the Real World

The LiveChat app needed hints for **2 classes**.

The Pplan app needs hints for **50+ classes**:

— Domain records (Doctor, Ordination, Specialty, ...)
— Repository row types
— Service classes
— JobRunR request + handler pairs

The NativeImageConfig Pattern

```
@Configuration
@ImportRuntimeHints(
    NativeImageConfig.AppRuntimeHints.class)
public class NativeImageConfig {

    static class AppRuntimeHints
            implements RuntimeHintsRegistrar {
        // next slide...
    }
}
```

One config class per application. Referenced via `@ImportRuntimeHints`.

RuntimeHintsRegistrar

```java
@Override
public void registerHints(
        RuntimeHints hints, ClassLoader classLoader) {

    // Reflection for domain classes
    hints.reflection().registerType(
        Doctor.class, MemberCategory.values());
    hints.reflection().registerType(
        Ordination.class, MemberCategory.values());

    // Resources (Flyway, static files)
    hints.resources().registerPattern("db/migration/*");
    hints.resources().registerPattern("static/admin/*");
}
```

Add every type that needs reflection. Register every resource pattern.

Obstacles We Hit

Obstacle #1: JDBI

— JDBI uses heavy dynamic reflection internally
— 50+ internal classes would need hints
— **Solution**: Use Spring JdbcClient instead

Obstacle #2: Spring Boot 4 Autoconfiguration

— Autoconfiguration was modularized in Spring Boot 4

— Just having `flyway-core` is not enough anymore

— **Solution**: Use `spring-boot-starter-flyway`

Obstacle #3: Vaadin 25 Dev Mode

- Vaadin 25 requires explicit `vaadin-dev` dependency
- Without it: `RuntimeException: 'vaadin-dev-server' not found`
- **Solution**: Add `vaadin-dev` with `<optional>true</optional>`

Obstacle #4: Flyway Resources

- Flyway SQL files must be included in the native image
- They're not automatically detected
- **Solution**: `hints.resources().registerPattern("db/migration/*")`

Obstacle #5: JobRunR Lambdas

— Lambda-based job scheduling doesn't work in native images

— Requires runtime serialization + reflection

— **Solution**: Use `JobRequest` / `JobRequestHandler` pattern with records

Obstacle #6: JobRunR State Classes

- JobRunR internally serializes/deserializes job states
- State classes need reflection metadata
- **Solution**: `reachability-metadata.json` for all state classes

Golden Rules for Native Images

1. **Prefer Spring-managed libraries** - they have AOT support
2. **Use starters** in Spring Boot 4 (modularized autoconfig)
3. **Register everything** that uses reflection in RuntimeHints
4. **Register resources** (SQL files, templates, static files)
5. **Avoid lambda serialization** - use explicit types

Summary

- Spring Boot 4 + Vaadin 25 + GraalVM = production ready
- REST APIs + HTMX + Vaadin coexist in one native binary
- JobRunR works with `JobRequest/JobRequestHandler` pattern
- Virtual Threads work in native images
- JdbcClient is the go-to for native-friendly database

Thank you!

Questions?

Twitter/X: x.com/domdorn
LinkedIn: linkedin.com/in/dominik-dorn/

Code: github.com/domdorn