Spring ohne Magie

die Welt von Ktor

Christian Babsek · Syna GmbH

Java Forum Stuttgart 2026

Agenda

1. **Spring** — Vorteile & Nachteile
2. **Ktor** als Framework
3. Wie sieht **Ktor** aus?
4. Erweiterte Features mit **Ktor**
5. Vergleiche
6. **Live-Coding**
7. **Fazit**

Spring — Vorteile & Nachteile

Spring ist großartig — aber der Komfort hat einen Preis.

Vorteile

- Riesiges, ausgereiftes **Ökosystem**
- **Auto-Configuration** & Starter — schnell produktiv
- Riesige **Community** & Dokumentation
- Enterprise-Integrationen out of the box

Nachteile

- Viel **Magie**: vieles passiert implizit & zur Laufzeit
- **Reflection / Proxies** — schwer nachzuvollziehen
- Hohe Startzeit und RAM-Auslastung
- Fehlersuche führt durch Framework-Code statt durch eigenen

Ktor als Framework

Ein **Kotlin-natives** Web-Framework von **JetBrains**.

- **Code statt Annotationen** — der Server wird programmiert, nicht „gescannt“
- **Coroutines** im Kern — asynchron & non-blocking
- **Modular** — alles ist ein Plugin, das du explizit installierst
- **Leichtgewichtig** — kleiner Footprint, schneller Start
- **Bequem** — schlanke, gut lesbare APIs
- **Multiplatform** — Server auf JVM & Nativ

Das Plugin-Ökosystem

Der Generator `start.ktor.io` bietet Dutzende Plugins — jedes ein bewusstes `install( ... )`:

DI & FRAMEWORKS


Koin


Ktor DI


kotlinx-rpc

DATENBANKEN


Exposed


PostgreSQL


MongoDB

MESSAGING & ECHTZEIT


Kafka


RabbitMQ


WebSockets

SECURITY


JWT


OAuth


Firebase Auth

OBSERVABILITY


Micrometer


OpenTelemetry


CallLogging

WEB & APIS


Swagger


OpenAPI


HTMX

und viele mehr

Ein simples Ktor-Projekt

Der kleinste lauffähige Server — reiner Kotlin-Code:

```
fun main() {
    embeddedServer(Netty, port = 8080) {
        routing {
            get("/") { call.respondText("Hello, Ktor!") }
        }
    }.start(wait = true)
}
```

- `embeddedServer(Engine, port) { … }` — **erzeugt** den Server, `.start(wait = true)` **startet** ihn
- Alles in einer Datei, explizit, nachvollziehbar und ohne Magie
- Eine `main`-Funktion — kein Framework, das vor dir startet

Ein simples Ktor-Projekt

Der kleinste lauffähige Server — reiner Kotlin-Code:

```
fun main() {
    embeddedServer(Netty, port = 8080) {
        routing {
            get("/") { call.respondText("Hello, Ktor!") }
        }
    }.start(wait = true)
}
```

- `embeddedServer(Engine, port) { … }` — **erzeugt** den Server, `.start(wait = true)` **startet** ihn
- Alles in einer Datei, explizit, nachvollziehbar und ohne Magie
- Eine `main`-Funktion — kein Framework, das vor dir startet

Ein simples Ktor-Projekt

Der kleinste lauffähige Server — reiner Kotlin-Code:

```
fun main() {
    embeddedServer(Netty, port = 8080) {
        routing {
            get("/") { call.respondText("Hello, Ktor!") }
        }
    }.start(wait = true)
}
```

- `embeddedServer(Engine, port) { … }` — **erzeugt** den Server, `.start(wait = true)` **startet** ihn
- Alles in einer Datei, explizit, nachvollziehbar und ohne Magie
- Eine `main`-Funktion — kein Framework, das vor dir startet

Ein simples Ktor-Projekt

Der kleinste lauffähige Server — reiner Kotlin-Code:

```
fun main() {
    embeddedServer(Netty, port = 8080) {
        routing {
            get("/") { call.respondText("Hello, Ktor!") }
        }
    }.start(wait = true)
}
```

- `embeddedServer(Engine, port) { … }` — **erzeugt** den Server, `.start(wait = true)` **startet** ihn
- Alles in einer Datei, explizit, nachvollziehbar und ohne Magie
- Eine `main`-Funktion — kein Framework, das vor dir startet

Ein simples Ktor-Projekt

Der kleinste lauffähige Server — reiner Kotlin-Code:

```
fun main() {
    embeddedServer(Netty, port = 8080) {
        routing {
            get("/") { call.respondText("Hello, Ktor!") }
        }
    }.start(wait = true)
}
```

- `embeddedServer(Engine, port) { … }` — **erzeugt** den Server, `.start(wait = true)` **startet** ihn
- Alles in einer Datei, explizit, nachvollziehbar und ohne Magie
- Eine `main`-Funktion — kein Framework, das vor dir startet

Das Build-Setup

Ein schlankes Gradle-Setup genügt:

```
plugins {
    kotlin("jvm")
    kotlin("plugin.serialization")     // JSON zur Compile-Zeit
}

dependencies {
    // jede Abhängigkeit ist eine bewusste Entscheidung
    implementation("io.ktor:ktor-server-core")
    implementation("io.ktor:ktor-server-netty")          // die Engine
    implementation("io.ktor:ktor-server-content-negotiation")
    implementation("io.ktor:ktor-serialization-kotlinx-json")
}
```

- Die **Engine** (`netty`) ist eine normale Dependency — austauschbar gegen CIO, Jetty, …
- Im Repo gepinnt über einen **Version-Catalog** (`libs.versions.toml`)

Das Build-Setup

Ein schlankes Gradle-Setup genügt:

```
plugins {
    kotlin("jvm")
    kotlin("plugin.serialization")     // JSON zur Compile-Zeit
}

dependencies {
    // jede Abhängigkeit ist eine bewusste Entscheidung
    implementation("io.ktor:ktor-server-core")
    implementation("io.ktor:ktor-server-netty")          // die Engine
    implementation("io.ktor:ktor-server-content-negotiation")
    implementation("io.ktor:ktor-serialization-kotlinx-json")
}
```

- Die **Engine** (`netty`) ist eine normale Dependency — austauschbar gegen CIO, Jetty, …
- Im Repo gepinnt über einen **Version-Catalog** (`libs.versions.toml`)

Das Build-Setup

Ein schlankes Gradle-Setup genügt:

```
plugins {
    kotlin("jvm")
    kotlin("plugin.serialization")     // JSON zur Compile-Zeit
}

dependencies {
    // jede Abhängigkeit ist eine bewusste Entscheidung
    implementation("io.ktor:ktor-server-core")
    implementation("io.ktor:ktor-server-netty")          // die Engine
    implementation("io.ktor:ktor-server-content-negotiation")
    implementation("io.ktor:ktor-serialization-kotlinx-json")
}
```

- Die **Engine** (`netty`) ist eine normale Dependency — austauschbar gegen CIO, Jetty, …
- Im Repo gepinnt über einen **Version-Catalog** (`libs.versions.toml`)

Das Build-Setup

Ein schlankes Gradle-Setup genügt:

```
plugins {
    kotlin("jvm")
    kotlin("plugin.serialization")     // JSON zur Compile-Zeit
}

dependencies {
    // jede Abhängigkeit ist eine bewusste Entscheidung
    implementation("io.ktor:ktor-server-core")
    implementation("io.ktor:ktor-server-netty")          // die Engine
    implementation("io.ktor:ktor-server-content-negotiation")
    implementation("io.ktor:ktor-serialization-kotlinx-json")
}
```

- Die **Engine** (`netty`) ist eine normale Dependency — austauschbar gegen CIO, Jetty, …
- Im Repo gepinnt über einen **Version-Catalog** (`libs.versions.toml`)

Das Build-Setup

Ein schlankes Gradle-Setup genügt:

```
plugins {
    kotlin("jvm")
    kotlin("plugin.serialization")     // JSON zur Compile-Zeit
}

dependencies {
    // jede Abhängigkeit ist eine bewusste Entscheidung
    implementation("io.ktor:ktor-server-core")
    implementation("io.ktor:ktor-server-netty")          // die Engine
    implementation("io.ktor:ktor-server-content-negotiation")
    implementation("io.ktor:ktor-serialization-kotlinx-json")
}
```

- Die **Engine** (`netty`) ist eine normale Dependency — austauschbar gegen CIO, Jetty, …
- Im Repo gepinnt über einen **Version-Catalog** (`libs.versions.toml`)

Der Application-Block

`main()` startet die Engine, ein **Application-Block** konfiguriert sie:

```
fun main() {
    embeddedServer(Netty, port = 8080, host = "0.0.0.0") {
        module()                        // dein Application-Block
    }.start(wait = true)
}

fun Application.module() {              // Extension auf Application
    install(ContentNegotiation) { json() }
    install(CORS) { anyHost() }
    install(CallLogging)                // Request-Logging
    routing {
        get("/health") { call.respondText("OK") }
        post("/echo") { call.respond(call.receive<Item>()) }
    }
}
```

Der Application-Block

`main()` startet die Engine, ein **Application-Block** konfiguriert sie:

```
fun main() {
    embeddedServer(Netty, port = 8080, host = "0.0.0.0") {
        module()                        // dein Application-Block
    }.start(wait = true)
}

fun Application.module() {              // Extension auf Application
    install(ContentNegotiation) { json() }
    install(CORS) { anyHost() }
    install(CallLogging)                // Request-Logging
    routing {
        get("/health") { call.respondText("OK") }
        post("/echo") { call.respond(call.receive<Item>()) }
    }
}
```

Der Application-Block

`main()` startet die Engine, ein **Application-Block** konfiguriert sie:

```
fun main() {
    embeddedServer(Netty, port = 8080, host = "0.0.0.0") {
        module()                        // dein Application-Block
    }.start(wait = true)
}

fun Application.module() {              // Extension auf Application
    install(ContentNegotiation) { json() }
    install(CORS) { anyHost() }
    install(CallLogging)                // Request-Logging
    routing {
        get("/health") { call.respondText("OK") }
        post("/echo") { call.respond(call.receive<Item>()) }
    }
}
```

Der Application-Block

`main()` startet die Engine, ein **Application-Block** konfiguriert sie:

```
fun main() {
    embeddedServer(Netty, port = 8080, host = "0.0.0.0") {
        module()                        // dein Application-Block
    }.start(wait = true)
}

fun Application.module() {              // Extension auf Application
    install(ContentNegotiation) { json() }
    install(CORS) { anyHost() }
    install(CallLogging)                // Request-Logging
    routing {
        get("/health") { call.respondText("OK") }
        post("/echo") { call.respond(call.receive<Item>()) }
    }
}
```

Der Application-Block

`main()` startet die Engine, ein **Application-Block** konfiguriert sie:

```
fun main() {
    embeddedServer(Netty, port = 8080, host = "0.0.0.0") {
        module()                        // dein Application-Block
    }.start(wait = true)
}

fun Application.module() {              // Extension auf Application
    install(ContentNegotiation) { json() }
    install(CORS) { anyHost() }
    install(CallLogging)                // Request-Logging
    routing {
        get("/health") { call.respondText("OK") }
        post("/echo") { call.respond(call.receive<Item>()) }
    }
}
```

Der Application-Block

`main()` startet die Engine, ein **Application-Block** konfiguriert sie:

```
fun main() {
    embeddedServer(Netty, port = 8080, host = "0.0.0.0") {
        module()                        // dein Application-Block
    }.start(wait = true)
}

fun Application.module() {              // Extension auf Application
    install(ContentNegotiation) { json() }
    install(CORS) { anyHost() }
    install(CallLogging)                // Request-Logging
    routing {
        get("/health") { call.respondText("OK") }
        post("/echo") { call.respond(call.receive<Item>()) }
    }
}
```

Routing mit GET & POST

Routen sind Code, kein Annotations-Geflecht:

```
routing {
    get("/items") {
        call.respond(items)                    // → 200 + JSON-Liste
    }

    post("/items") {
        val draft = call.receive<ItemDraft>()      // JSON → Objekt
        val created = repository.add(draft)
        call.respond(HttpStatusCode.Created, created)
    }
}
```

- Routing ist eine DSL — verschachtelte Funktionen statt `@GetMapping`
- `receive<T>()` liest den Body, `respond()` schreibt die Antwort

Routing mit GET & POST

Routen sind Code, kein Annotations-Geflecht:

```
routing {
    get("/items") {
        call.respond(items)                    // → 200 + JSON-Liste
    }

    post("/items") {
        val draft = call.receive<ItemDraft>()      // JSON → Objekt
        val created = repository.add(draft)
        call.respond(HttpStatusCode.Created, created)
    }
}
```

- Routing ist eine **DSL** — verschachtelte Funktionen statt `@GetMapping`
- `receive<T>()` liest den Body, `respond()` schreibt die Antwort

Routing mit GET & POST

Routen sind Code, kein Annotations-Geflecht:

```
routing {
    get("/items") {
        call.respond(items)                    // → 200 + JSON-Liste
    }

    post("/items") {
        val draft = call.receive<ItemDraft>()      // JSON → Objekt
        val created = repository.add(draft)
        call.respond(HttpStatusCode.Created, created)
    }
}
```

- Routing ist eine DSL — verschachtelte Funktionen statt `@GetMapping`
- `receive<T>()` liest den Body, `respond()` schreibt die Antwort

Routing mit GET & POST

Routen sind Code, kein Annotations-Geflecht:

```
routing {
    get("/items") {
        call.respond(items)                    // → 200 + JSON-Liste
    }

    post("/items") {
        val draft = call.receive<ItemDraft>()      // JSON → Objekt
        val created = repository.add(draft)
        call.respond(HttpStatusCode.Created, created)
    }
}
```

- Routing ist eine DSL — verschachtelte Funktionen statt `@GetMapping`
- `receive<T>()` liest den Body, `respond()` schreibt die Antwort

JSON-Serialisierung konfigurieren

JSON ist kein Default — du aktivierst es bewusst:

```
install(ContentNegotiation) {
    json(Json {
        prettyPrint = true
        ignoreUnknownKeys = true
        encodeDefaults = true
    })
}

@Serializable
data class Item(val id: Int, val name: String, val done: Boolean = false)
```

- `ContentNegotiation` handelt das Format zwischen Client & Server aus (`Accept` / `Content-Type`)
- `kotlinx.serialization` erzeugt den Serializer **zur Compile-Zeit** ohne Reflection
- weitere Varianten (XML, etc.) sind ebenfalls verfügbar

JSON-Serialisierung konfigurieren

JSON ist kein Default — du aktivierst es bewusst:

```
install(ContentNegotiation) {
    json(Json {
        prettyPrint = true
        ignoreUnknownKeys = true
        encodeDefaults = true
    })
}

@Serializable
data class Item(val id: Int, val name: String, val done: Boolean = false)
```

- `ContentNegotiation` handelt das Format zwischen Client & Server aus (`Accept` / `Content-Type`)
- `kotlinx.serialization` erzeugt den Serializer **zur Compile-Zeit** ohne Reflection
- weitere Varianten (XML, etc.) sind ebenfalls verfügbar

JSON-Serialisierung konfigurieren

JSON ist kein Default — du aktivierst es bewusst:

```
install(ContentNegotiation) {
    json(Json {
        prettyPrint = true
        ignoreUnknownKeys = true
        encodeDefaults = true
    })
}

@Serializable
data class Item(val id: Int, val name: String, val done: Boolean = false)
```

- `ContentNegotiation` handelt das Format zwischen Client & Server aus (`Accept` / `Content-Type`)
- `kotlinx.serialization` erzeugt den Serializer **zur Compile-Zeit** ohne Reflection
- weitere Varianten (XML, etc.) sind ebenfalls verfügbar

JSON-Serialisierung konfigurieren

JSON ist kein Default — du aktivierst es bewusst:

```
install(ContentNegotiation) {
    json(Json {
        prettyPrint = true
        ignoreUnknownKeys = true
        encodeDefaults = true
    })
}

@Serializable
data class Item(val id: Int, val name: String, val done: Boolean = false)
```

- `ContentNegotiation` handelt das Format zwischen Client & Server aus (`Accept` / `Content-Type`)
- `kotlinx.serialization` erzeugt den Serializer **zur Compile-Zeit** ohne Reflection
- weitere Varianten (XML, etc.) sind ebenfalls verfügbar

Module: CORS & Co.

Jedes Feature ist ein **Plugin** — sichtbar per `install( ... )`:

```
install(CORS) {
    allowMethod(HttpMethod.Post)
    allowHeader(HttpHeaders.ContentType)
    anyHost()                       // TODO: in Produktion einschränken
}

install(StatusPages) {
    exception<ItemNotFoundException> { call, e → call.respond(NotFound, e.message ?: "Item not found") }
    exception<ValidationException> { call, e → call.respond(BadRequest, e.message ?: "Validation failed") }
}
install(CallLogging) { level = Level.INFO }
```

- **CORS** — Cross-Origin-Zugriffe steuern
- **StatusPages** — Domain-Exceptions zentral auf HTTP-Status mappen (404, 400, …)
- **CallLogging** — jeder Request landet im Log (`level` -gesteuert)

Module: CORS & Co.

Jedes Feature ist ein **Plugin** — sichtbar per `install( ... )`:

```
install(CORS) {
    allowMethod(HttpMethod.Post)
    allowHeader(HttpHeaders.ContentType)
    anyHost()                       // TODO: in Produktion einschränken
}

install(StatusPages) {
    exception<ItemNotFoundException> { call, e → call.respond(NotFound, e.message ?: "Item not found") }
    exception<ValidationException> { call, e → call.respond(BadRequest, e.message ?: "Validation failed") }
}
install(CallLogging) { level = Level.INFO }
```

- **CORS** — Cross-Origin-Zugriffe steuern
- **StatusPages** — Domain-Exceptions zentral auf HTTP-Status mappen (404, 400, …)
- **CallLogging** — jeder Request landet im Log (`level` -gesteuert)

Module: CORS & Co.

Jedes Feature ist ein **Plugin** — sichtbar per `install( ... )`:

```
install(CORS) {
    allowMethod(HttpMethod.Post)
    allowHeader(HttpHeaders.ContentType)
    anyHost()                       // TODO: in Produktion einschränken
}

install(StatusPages) {
    exception<ItemNotFoundException> { call, e → call.respond(NotFound, e.message ?: "Item not found") }
    exception<ValidationException> { call, e → call.respond(BadRequest, e.message ?: "Validation failed") }
}
install(CallLogging) { level = Level.INFO }
```

- **CORS** — Cross-Origin-Zugriffe steuern
- **StatusPages** — Domain-Exceptions zentral auf HTTP-Status mappen (404, 400, …)
- **CallLogging** — jeder Request landet im Log (`level` -gesteuert)

Module: CORS & Co.

Jedes Feature ist ein **Plugin** — sichtbar per `install( ... )`:

```
install(CORS) {
    allowMethod(HttpMethod.Post)
    allowHeader(HttpHeaders.ContentType)
    anyHost()                       // TODO: in Produktion einschränken
}

install(StatusPages) {
    exception<ItemNotFoundException> { call, e → call.respond(NotFound, e.message ?: "Item not found") }
    exception<ValidationException> { call, e → call.respond(BadRequest, e.message ?: "Validation failed") }
}
install(CallLogging) { level = Level.INFO }
```

- **CORS** — Cross-Origin-Zugriffe steuern
- **StatusPages** — Domain-Exceptions zentral auf HTTP-Status mappen (404, 400, …)
- **CallLogging** — jeder Request landet im Log (`level` -gesteuert)

Module: CORS & Co.

Jedes Feature ist ein **Plugin** — sichtbar per `install( ... )`:

```
install(CORS) {
    allowMethod(HttpMethod.Post)
    allowHeader(HttpHeaders.ContentType)
    anyHost()                       // TODO: in Produktion einschränken
}

install(StatusPages) {
    exception<ItemNotFoundException> { call, e → call.respond(NotFound, e.message ?: "Item not found") }
    exception<ValidationException> { call, e → call.respond(BadRequest, e.message ?: "Validation failed") }
}
install(CallLogging) { level = Level.INFO }
```

- **CORS** — Cross-Origin-Zugriffe steuern
- **StatusPages** — Domain-Exceptions zentral auf HTTP-Status mappen (404, 400, …)
- **CallLogging** — jeder Request landet im Log (`level` -gesteuert)

Dependency Injection mit Koin

Drei Schritte: deklarieren → als Plugin installieren → auflösen.

```
val appKoinModule = module {                        // 1) Bindungen deklarieren
    single<ItemRepository> { ExposedItemRepository() }
    single<ItemService> { ItemService(itemRepository = get()) }
}

fun Application.configureKoin() =                   // 2) Koin als Plugin installieren
    install(Koin) { modules(appKoinModule) }

val itemRepository by inject<ItemRepository>()      // 3) dort auflösen, wo gebraucht
```

- `single<T> { … }` bindet ein Interface an **eine** Implementierung — sichtbar im Modul
- `get()` liefert die Implementierung des verlangten Typs
- `inject()` löst sie lazy auf; der ganze Graph steht als gewöhnlicher Kotlin-Code da

Dependency Injection mit Koin

Drei Schritte: deklarieren → als Plugin installieren → auflösen.

```
val appKoinModule = module {                        // 1) Bindungen deklarieren
    single<ItemRepository> { ExposedItemRepository() }
    single<ItemService> { ItemService(itemRepository = get()) }
}

fun Application.configureKoin() =                   // 2) Koin als Plugin installieren
    install(Koin) { modules(appKoinModule) }

val itemRepository by inject<ItemRepository>()      // 3) dort auflösen, wo gebraucht
```

- `single<T> { … }` bindet ein Interface an **eine** Implementierung — sichtbar im Modul
- `get()` liefert die Implementierung des verlangten Typs
- `inject()` löst sie lazy auf; der ganze Graph steht als gewöhnlicher Kotlin-Code da

Dependency Injection mit Koin

Drei Schritte: deklarieren → als Plugin installieren → auflösen.

```
val appKoinModule = module {                        // 1) Bindungen deklarieren
    single<ItemRepository> { ExposedItemRepository() }
    single<ItemService> { ItemService(itemRepository = get()) }
}

fun Application.configureKoin() =                   // 2) Koin als Plugin installieren
    install(Koin) { modules(appKoinModule) }

val itemRepository by inject<ItemRepository>()      // 3) dort auflösen, wo gebraucht
```

- `single<T> { … }` bindet ein Interface an **eine** Implementierung — sichtbar im Modul
- `get()` liefert die Implementierung des verlangten Typs
- `inject()` löst sie lazy auf; der ganze Graph steht als gewöhnlicher Kotlin-Code da

Dependency Injection mit Koin

Drei Schritte: deklarieren → als Plugin installieren → auflösen.

```
val appKoinModule = module {                        // 1) Bindungen deklarieren
    single<ItemRepository> { ExposedItemRepository() }
    single<ItemService> { ItemService(itemRepository = get()) }
}

fun Application.configureKoin() =                   // 2) Koin als Plugin installieren
    install(Koin) { modules(appKoinModule) }

val itemRepository by inject<ItemRepository>()      // 3) dort auflösen, wo gebraucht
```

- `single<T> { … }` bindet ein Interface an **eine** Implementierung — sichtbar im Modul
- `get()` liefert die Implementierung des verlangten Typs
- `inject()` löst sie lazy auf; der ganze Graph steht als gewöhnlicher Kotlin-Code da

Dependency Injection mit Koin

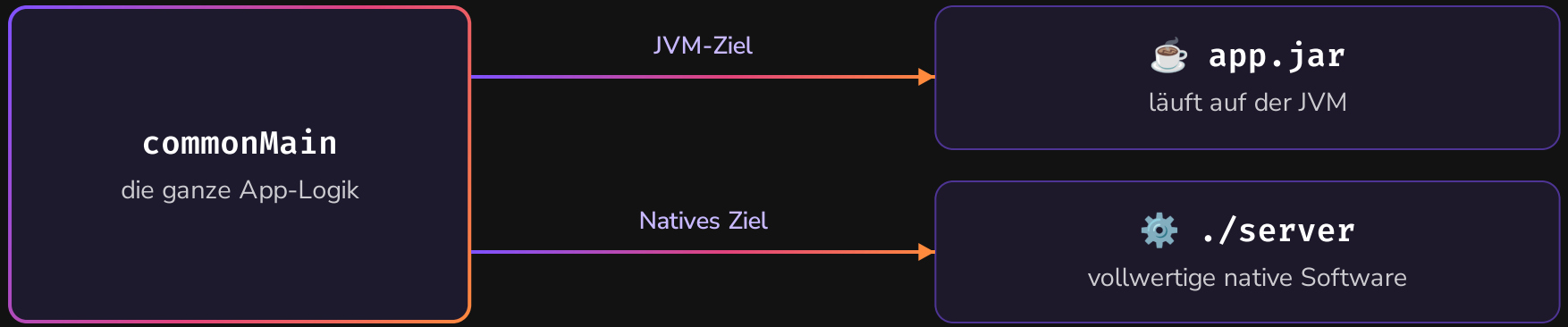
Drei Schritte: deklarieren → als Plugin installieren → auflösen.

```
val appKoinModule = module {                        // 1) Bindungen deklarieren
    single<ItemRepository> { ExposedItemRepository() }
    single<ItemService> { ItemService(itemRepository = get()) }
}

fun Application.configureKoin() =                   // 2) Koin als Plugin installieren
    install(Koin) { modules(appKoinModule) }

val itemRepository by inject<ItemRepository>()      // 3) dort auflösen, wo gebraucht
```

- `single<T> { … }` bindet ein Interface an **eine** Implementierung — sichtbar im Modul
- `get()` liefert die Implementierung des verlangten Typs
- `inject()` löst sie lazy auf; der ganze Graph steht als gewöhnlicher Kotlin-Code da

Dieselbe Codebasis — auch nativ kompiliert

Derselbe Kotlin-Code, zwei Compile-Targets:



- Genau **eine** Codebasis — kompiliert wahlweise zur JVM **oder** als native Binary
- Nativ: kein JRE, kein Cold-Start — eine Datei, die sofort startet (Linux & macOS)

Im Vergleich: Ein Endpunkt

Annotierte Controller-Klasse vs. Funktionen in einer Routing-DSL.

Spring

```
@RestController
@RequestMapping("/items")
class ItemController(
    val itemsRepo: ItemRepository,
) {
    @GetMapping
    fun all() = itemsRepo.findAll()

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    fun add(@RequestBody item: ItemDraft) =
        itemsRepo.save(item)
    @DeleteMapping("/{itemId}")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    fun remove(@PathVariable itemId: Long) =
        itemsRepo.delete(itemId)
}
```

Ktor

```
fun Route.itemRoutes(
    itemsRepo: ItemRepository,
) {
    get("/items") {
        call.respond(itemsRepo.findAll())
    }
    post("/items") {
        val item = call.receive<ItemDraft>()
        call.respond(Created, itemsRepo.save(item))
    }
    delete("/items/{itemId}") {
        val itemId = call.parameters["itemId"]?.toLongOrNull()
            ?: throw BadRequestException("itemId")
        itemsRepo.delete(itemId)
        call.respond(NoContent)
    }
}
```

Im Vergleich: JSON-Serialisierung

Implizit über Reflection vs. bewusst installiert, Codegen zur Compile-Zeit.

Spring

```
// Jackson: aktiv, sobald im Klassenpfad
// feinjustiert via application.yml / @Bean

data class Item(
    val id: Long,
    val name: String,
)

// (De-)Serializer per Reflection
// zur Laufzeit
```

Ktor

```
install(ContentNegotiation) {
    json(Json { prettyPrint = true })
}

@Serializable
data class Item(
    val id: Long,
    val name: String,
)
// Serializer zur Compile-Zeit
```

Im Vergleich: Dependency Injection

Classpath-Scan & Auto-Wiring vs. ein sichtbarer Modul-Graph.

Spring

```
@Service
class ExposedItemRepository :
    ItemRepository

@RestController
class ItemController(
    // @Autowired (implizit)
    val repo: ItemRepository,
)
// Bean-Auswahl per Classpath-Scan
```

Ktor + Koin

```
val appModule = module {
    single<ItemRepository> {
        ExposedItemRepository()
    }
}

val repo by inject<ItemRepository>()

// Graph steht sichtbar im Modul
```

Im Vergleich: App-Start

Auto-Configuration beim Boot vs. ein expliziter Startpfad.

Spring

```
@SpringBootApplication
class App

fun main(args: Array<String>) {
    runApplication<App>(*args)
}

// Auto-Configuration scannt den
// Klassenpfad und entscheidet für dich
```

Ktor

```
fun main() {
    embeddedServer(Netty, port = 8080) {
        module()   // du rufst alles auf
    }.start(wait = true)
}

// nichts passiert „automatisch“ –
// der Startpfad ist dein Code
```

LIVE

Genug Theorie

Live-Coding 🚀

Fazit

Kein „besser oder schlechter“ — ein bewusster Trade-off:

Spring

deklarativ & battle-tested

- **Auto-Configuration** & Starter — schnell produktiv
- riesiges, ausgereiftes **Ökosystem** & Community
- Enterprise-Integrationen **out of the box** (Data, Security, Batch, …)

→ Enterprise & Battle-Tested

Ktor

explizit & leichtgewichtig

- sichtbarer Code — keine Laufzeit-Magie
- **Compile-Zeit** statt Reflection — schneller Start, kleiner Footprint
- **Coroutines** im Kern + **Multiplatform** (JVM & Native)
- sehr schlank, auch für Microservices

→ schlanke Services, native Binaries, volle Kontrolle

Vielen herzlichen Dank!

Jede Frage ist erlaubt

Slides & Code github.com/chrisbabsek/jfs2026-christianbabsek-ktor