Schneller, smarter, besser!

Kotlin 2.0

Kotlin 2.0 bringt mit dem K2-Compiler und zahlreichen neuen Features spürbare Verbesserungen, die die Entwicklung noch effizienter machen. Gleichzeitig setzt sich Ktor als flexibles Framework für schlanke und anpassbare REST-Services immer mehr durch. Wir geben einen Überblick über die wichtigsten Neuerungen und zeigen, warum sie so spannend sind.

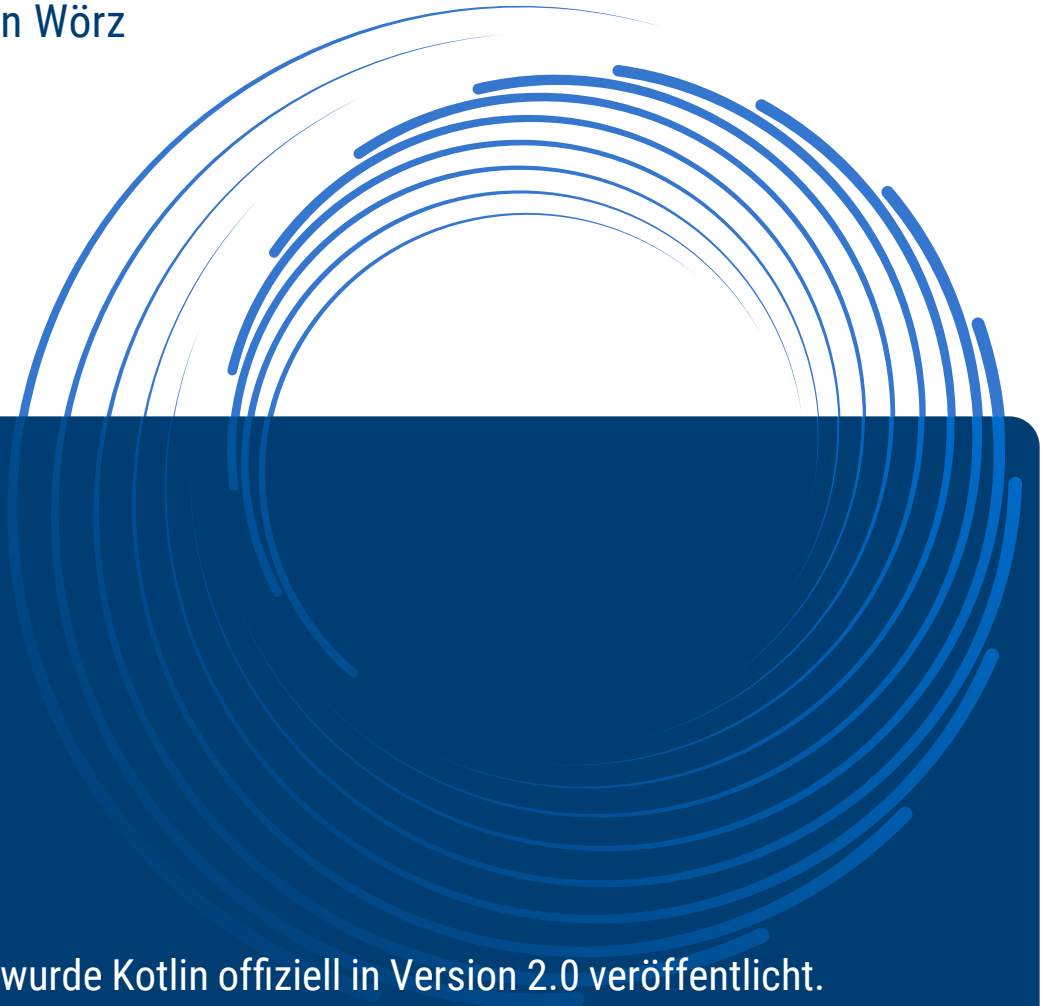
Kleine Schritte, große Wirkung
von Christian Wörz

Darf's noch etwas mehr sein?
von Werner Eberling

Kotlin 2.0 – neue Features und der vollständige K2-Compiler

Kleine Schritte, große Wirkung

von Christian Wörz



Letztes Jahr wurde Kotlin offiziell in Version 2.0 veröffentlicht. Das Release beinhaltet neben dem endlich vollständig verfügbaren K2-Compiler zahlreiche neue Features und Verbesserungen in allen Bereichen der Sprache.

Schon seit Langem hat Jetbrains am neuen Kotlin-Compiler, der seinen Vorgänger ablösen soll, gearbeitet. Über zehn Millionen Zeilen Code wurden verwendet, um seine Stabilität zu prüfen und die Qualität sicherzustellen. Mit Kotlin 2.0 ist dieser Stand nun endlich erreicht und alle Zielplattformen verwenden K2 nun standardmäßig: JVM, Native, WebAssembly und JavaScript.

Was sind nun die Vorteile dieser Migration? Zum einen ist der neue Compiler signifikant schneller als das alte, in die Jahre gekommene Modell. Bei größeren Projekten soll die Kompilierzeit gegenüber der bisherigen um die Hälfte kürzer sein. Auch IntelliJ wird mit dem sogenannten K2 Mode, der sich aktuell mit IntelliJ 2024.2 in der Betaphase befindet, spürbar schneller, wenn damit Kotlin-Code editiert wird. Syntax-Highlights sowie Autocompletion sind laut Jetbrains oftmals 1,8-mal beziehungsweise 1,5-mal schneller als bisher. Zudem können durch die Vereinheitlichung des Compilers für alle Plattformen neue Sprachfeatures viel einfacher entwickelt werden. Eine Menge Logik wird nun auf allen Plattformen wiederverwendet, und auch die Pipelines sind nun einheitlich.

Jedem Entwickler sollte klar sein, was das für die Geschwindigkeit der Umsetzung von neuen Features bedeutet: Der Entwicklungsprozess wird signifikant schlanker, wenn ein Feature nur einmal implementiert und für alle Plattformen wiederverwendet werden kann. Neben der allgemeinen Performanceverbesserung kommt K2 mit vielen neuen Features, die uns auch während der Entwicklungszeit direkt helfen werden. Die Wichtigsten werden wir nun näher beleuchten.

Smart-Cast-Verbesserungen

Kotlin konnte von Anfang an in der Vielzahl der Fälle herausfinden, welchen spezifischen Typ eine Variable beinhaltet, und sie implizit für uns in diesen spezifischeren Typ konvertieren. Mit K2 wurde diese Funktion nochmals stark erweitert. Hierzu einige Beispiele.

Bisher war es in einigen Konstellationen nicht möglich, auf Typinformationen von Variablen zuzugreifen, wenn diese außerhalb eines *if*-Blocks erstellt wurden. Somit konnte Kotlin diese Variable auch nicht mittels Smart Casting typisieren. Mit K2 ist das nun anders und Kotlin kann auf diese Information zugreifen. Schauen wir uns dazu das Codebeispiel in Listing 1 an.



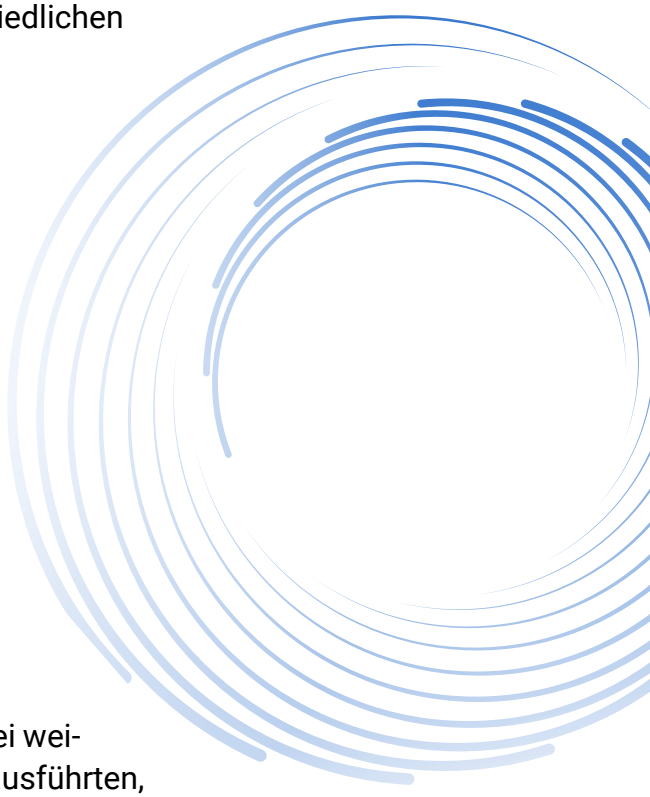
Listing 1

```
class Mensch {
   fun hallo() = println("Hallo")
}

fun menschSagHallo(lebensform: Any) {
   val isMensch = lebensform is Mensch
   if(isMensch) {
      lebensform.hallo() // <-- (1)
   }
}
```

Der Code in Zeile (1) hätte bislang einen Fehler verursacht, da *lebensform* nicht in *Mensch* gecastet worden wäre. Durch K2 ist das nun kein Problem mehr. Diese Verbesserung hilft uns genauso bei *when*- oder *while*-Konstrukten.

Schauen wir uns als Nächstes ein Codebeispiel für Type Checks mit *or* an und vergleichen dann, welche Typen die unterschiedlichen Compiler daraus ableiten würden (Listing 2).



Listing 2

```
interface Status {
   fun info() {}
}

interface Ok: Status
interface Fehler: Status
interface Unbekannt: Status

fun checkStatus(status: Any) {
   if(status is Ok || status is Fehler) {
      status.info() // <-- (1)
   }
}
```

Was passiert hier? Wir haben ein Interface, von dem drei weitere Interfaces ableiten. Wenn wir diesen Code bisher ausführten, hatte das einen Fehler zur Folge, da das Smart-Casting unsere *status*-Variable zu *Any* gecasted hätte. Doch K2 versteht, dass durch die Verwendung des ||-Operators *status* nur noch *Ok* oder *Fehler* als Typ haben kann, weshalb wir nun *info()* aufrufen können.

Zusätzlich zu den neuen Smart-Cast-Features wurde auch ein Fehler behoben, der auftrat, wenn eine Klasse eine Funktion als Property beinhaltete (Listing 3). Wenn wir diesen Code in Kotlin 1.9 oder früher ausführen, erhalten wir einen Fehler, da dem Compiler nicht klar ist, dass *func* zu diesem Zeitpunkt nicht *null* ist. Mit Kotlin 2.0 ist das Problem behoben und der Compiler verwendet einen Smart Cast.

Listing 3

```
class Wrapper(val func: (() -> Unit)?) {

  fun run() {
    if(func != null) {
      func() // <-- (1)
    }
  }
}
```

Eines meiner favorisierten Features im Bereich Smart Casting ist das verbesserte Fehlerhandling. Nun werden die Typinformationen aus dem *try*-Block ebenfalls in die *catch*- und *finally*-Blöcke weitergegeben, was es beispielsweise einfacher macht, mit nullfähigen Typen zu arbeiten (Listing 4).

Listing 4

```
fun test() {
  var einString: String?
  einString = "Ich bin nicht mehr null"

  try {
    println(einString.length)
    // 23

    einString = null

    if(true) throw Exception() // Fehler werfen
  } catch (e: Exception) {
    println(einString?.length) // <-- (1)
  }
}
```

Zeile (1) hätte in Kotlin 1.9 und früher fälschlicherweise eine Info angezeigt, dass es unnötig sei, einen Safe Call zu verwenden, da die Information, dass *einString* im *try*-block wieder *null* zugewiesen wird, nicht im *catch*-Block vorhanden war. K2 versteht dieses Konstrukt nun aber, wodurch ein Safe Call nun nicht nur ohne Warnung erlaubt, sondern auch benötigt wird.

Bis jetzt war Kotlin nicht bekannt, dass sich Typen verändern können, nachdem ein Inkrement- oder Dekrement-Operator (*++* oder --) verwendet wurde. Ein einfaches Beispiel dazu zeigt Listing 5.

Listing 5

```
interface Eins {
  operator fun inc(): Zwei = TODO()
}

interface Zwei : Eins {
  fun zwei() = Unit
}
interface Hundert {
  fun hundert() = Unit
}
```

```
fun main(zahl: Eins) {
  var unklar: Eins = zahl

  if (unklar is Hundert) { // (1)
    ++unklar // (2)

    unklar.zwei() // (3)
  }
}
```

Was war hier bislang das Problem? Kotlin hat in Zeile (1) angenommen, dass *unklar* vom Typ *Hundert* ist, und dadurch völlig die Information verloren, dass *unklar* auch vom Typ *Eins* sein kann, da es beide Interfaces (*Hundert* sowie *Eins*) implementieren kann. In der neuen Version castet der Compiler Zeile (2) zum Typ *Zwei*, da klar ist, dass *unklar* ebenfalls vom Typ *Eins* sein muss. Ein Nischenfehler, aber trotzdem ist es sehr beeindruckend, dass sich das Jetbrains-Kotlin-Team auch darum gekümmert hat.

Mit K2 wurden auch einige Verbesserungen im Multi-Platform-Bereich in Angriff genommen. Die zwei größten wollen wir uns nun genauer anschauen.

Trennung von Platform- und Common-Quellcode beim Kompilieren

Die Architektur des bisherigen Compilers verhinderte, dass die Platform- und Common-Source-Sets während des Kompilierschritts voneinander getrennt waren. Daher konnte Code aus dem Common-Teil auf plattformspezifischen Code zugreifen, was unschöne Nebeneffekte zur Folge hatte. Es führte zum Beispiel teils zu unterschiedlichen Verhaltensweisen für die Plattformen und sorgte zudem dafür, dass bestimmte Einstellungen des Common-Teils auf den Platform-Code Einfluss nehmen konnten.

Mit K2 ist das nun nicht mehr möglich, da die neue Architektur sicherstellt, dass die beiden Source-Sets strikt getrennt sind. Ein gutes Beispiel, um diese Anpassung in Aktion zu sehen, zeigt Listing 6.

Listing 6

```
// COMMON START
fun hallo(x: Any) = println("Von Common")

fun beispiel() = hallo(10)
// COMMON ENDE

// PLATFORM START
// JVM
fun hallo(x: Int) = println("Von Plattform")
// JavaScript hat keine 'hallo'-Funktion
```

Hier trat vor K2 ein seltsames Verhalten auf. Wenn *beispiel()* in Java aufgerufen wurde, wurde die *hallo*-Methode aus dem JVM-Code verwendet. Beim Aufruf aus der JavaScript-Plattform jedoch wurde die Funktion aus Common verwendet. Somit ergaben sich für unterschiedliche Plattformen unterschiedliche Verhaltensweisen. Mit K2 hat der Common-Code keinen Zugriff mehr auf den plattformspezifischen Code, wodurch auch bei der JVM-Plattform die Common-Methode *hallo* aufgerufen wird.

TRACK

W-JAX

Core Java & Languages:
Schöpfe das volle Potenzial der Java-Plattform aus!

Tauchen Sie tief ein in die neuesten Entwicklungen von Java und lernen Sie, wie Sie bessere, schnellere und effizientere Java-Anwendungen entwickeln können.

Lernen Sie von unseren Expert:innen:

- **Java 17 LTS, Java 21 LTS, Java 24:** Meistern Sie die wichtigsten Features und Funktionalitäten der aktuellsten Java-Versionen.
- **Cloud-native Java:** Erfahren Sie, wie Sie skalierbare Java-Anwendungen für moderne Cloud-Umgebungen erstellen und bereitstellen.
- **GraalVM:** Entdecken Sie die Leistungsfähigkeit der polyglotten Programmierung und integrieren Sie Java nahtlos mit anderen Sprachen.
- **JVM-Interna & Performance:** Entwickeln Sie tiefes Verständnis der Java Virtual Machine (JVM) und erkunden Sie Techniken zur Leistungsoptimierung.
- **High-Performance Java:** Schöpfen Sie das volle Potenzial Ihrer Java-Anwendungen aus, indem Sie Leistungsengpässe beseitigen.

Ein weiterer Unterschied konnte bisher auch bei bestimmten Konstellationen für die Entwicklungsumgebungen IntelliJ und Android Studio festgestellt werden. Beispielsweise wurden bestimmte Fehler nur in der Entwicklungsumgebung selbst angezeigt, der Compiler hingegen konnte damit umgehen. Auch das ist nun behoben und der Compiler kann die Fehlermeldungen nun ebenfalls anzeigen.

Kotlin auf der JVM-Plattform

Kotlin 2.0 unterstützt nun endlich die Generierung von Java-22-Bytecode. Andere JVM-spezifischen Änderungen sind in diesem Release eher ein Nebenschauplatz, aber trotzdem sollen sie nicht unerwähnt bleiben.

- *Leichtgewichtigere Lambdafunktionen:* Ab jetzt werden für Lambdafunktionen standardmäßig keine anonymen Klassen mehr verwendet, sondern es wird mittels *invokedynamic* generiert. Dadurch werden die erstellten Applikationen kleiner und leichtgewichtiger. Dieses Feature reiht sich in die kleinen, aber feinen Verbesserungen ein, die dieses Release so interessant machen.
- *API zum Lesen und Anpassen von Metadaten stable:* Lange war es zwar möglich, die Metadaten der vom Kotlin-JVM-Compiler erstellten Binärdateien mittels des *kotlinx-metadata-jvm* API zu lesen und anzupassen, aber wie es das „x“ in *kotlinx* vermuten ließ, befand sich das API noch in der Betaversion. Nun hat sich der Name verändert und die APIs sind mit *kotlin-metadata-jvm* stable.

Kotlin Native

Kotlin Native hat von allen Platform Targets die wenigsten Anpassungen erfahren. Wir wollen sie aber nicht außer Acht lassen und kurz betrachten:

- *Garbage-Collection-Performance-Messung:* In speziellen Fällen kann es notwendig sein, die Performance des Garbage Collectors zu messen, um Performanceprobleme zu analysieren. Diese Möglichkeit war in Xcode Instruments bisher für Kotlin Native stark eingeschränkt, da die einzige Möglichkeit darin bestand, die Logs direkt zu inspizieren. Mit Kotlin 2.0 wurde das verbessert und der Garbage Collector erstellt jetzt sogenannte Signposts, die von Xcode Instruments eingesehen und analysiert werden können. Damit haben wir nun eine einfachere Möglichkeit, Bottlenecks in unseren Apps zu analysieren, ohne uns durch die Logs zu quälen.

- *Objective-C-Methodenzugriff:* Wenn wir bisher Objective-C-Methoden in Kotlin aufrufen wollten, konnte das in einigen Fällen zu Problemen führen, etwa, wenn Methoden für Kotlin die gleiche Signatur aufwiesen. Ein Beispiel hierfür wäre *locationManager:didEnterRegion* und *locationManager:didExitRegion*. Wenn wir versuchen, diese Methoden in Kotlin zu verwenden, erhalten wir einen Fehler aufgrund eines Conflicting Overload, da Kotlin sie nicht unterscheiden kann. Um dieses Problem zu beheben, musste der Fehler bisher manuell unterdrückt werden. Das ist nun nicht mehr notwendig, denn jetzt existiert eine Annotation, die wir verwenden können, um den Compiler darauf hinzuweisen: *@ObjCSignatureOverride*. Sie sorgt dafür, dass der Compiler es ignoriert, wenn mehrere Methoden die gleiche Signatur aufweisen.

Kotlin für WebAssembly

Die meisten Anpassungen von Kotlin 2.0 im WebAssembly-(Wasm-)Bereich befassen sich mit der JavaScript-Kompatibilität und verbessern sie enorm. So werden zum Beispiel dank Binaryen schnellere Build-Zeiten erreicht. Binaryen ist ein Tool, das auch bisher schon in Kotlin/Wasm zum Einsatz kam. Vor Kotlin 2.0 musste es aber manuell aufgesetzt werden. Nun wird Binaryen automatisch bei jedem produktiven Build für alle Projekte ausgeführt. Laut dem Jetbrains-Team soll das die Größe des generierten Artefakts reduzieren und die Laufzeitperformance erhöhen. Genaue Messwerte sind bislang aber noch nicht veröffentlicht worden.

Eine weitere Neuerung bezieht sich auf die Named Imports. Wer schon einmal versucht hat, Funktionen von Kotlin/Wasm in eine JavaScript-Datei zu importieren, der musste feststellen, dass das nur über einen sogenannten Default Export möglich war:

```javascript
import Module from './util.mjs'

Module.halloWelt()
```

Stattdessen können wir nun einzelne Funktionen exportieren und somit auch importieren. Die Anwendung ist sehr einfach mit *@JsExport* möglich:

```kotlin
@JsExport
fun halloWelt() = "Hallo Welt"
```

Importieren können wir dann die Funktion mit dem bekannten Vorgehen:

```
```javascript
import { halloWelt } from './util.mjs'
```
```

Meines Erachtens eine großartige Anpassung, um die Les- und Wartbarkeit des Codes zu verbessern.

Weiter ist die Generierung von TypeScript-Typ-Files zu nennen. TypeScript ist aus der Webentwicklung nicht mehr wegzudenken. Das hat man auch bei Kotlin eingesehen und geht nun einen weiteren Schritt, um TypeScript optimal zu unterstützen. Kotlin 2.0 kann jetzt sogenannte *d.ts*-Dateien, also Typendefinitionen aus den exportierten *@JsExport*-Funktionen erstellen.

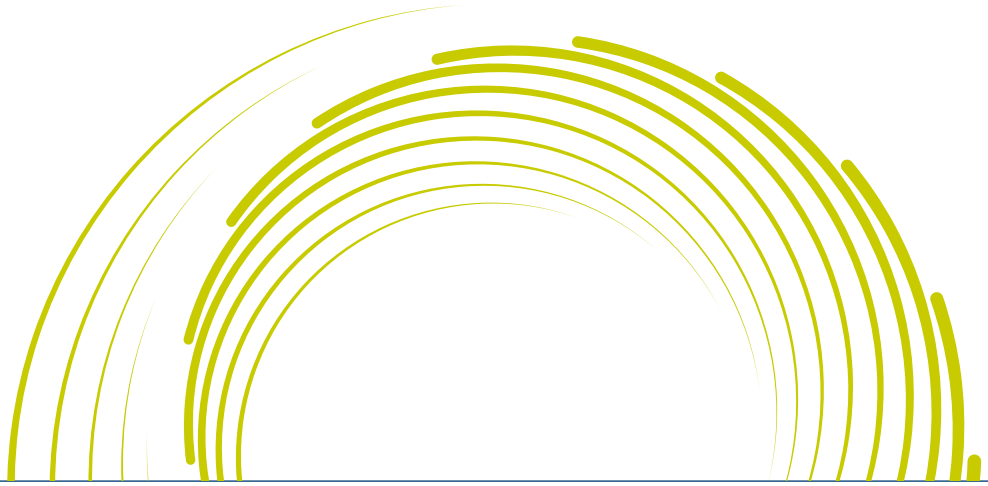
Dieses Feature hat zurzeit den Status „experimental“, kann sich also noch verändern. Wer aber trotzdem schon einen Blick riskieren will, kann das mit einer Erweiterung des Gradle Builds mit einfach einer hinzugefügten Zeile tun (Listing 7).



Listing 7

```
//build.gradle
kotlin {
  wasmJs {
    binaries.executable()
    browser {
    }
    generateTypeScriptDefinitions()
  }
}
```

Mit *generateTypeScriptDefinitions()* sucht der Compiler alle mit *@JSExport* annotierten Funktionen und stellt dafür Typen zu Verfügung. Als begeisterter TypeScript-Entwickler ist das sicher eins meiner Highlights in diesem Release.



Kotlin für die JavaScript-Plattform

Nachdem wir uns alle Verbesserungen in Kotlin/Wasm angeschaut haben und schon viele weitere im Wasm-JavaScript-Bereich bestaunen durften, lasst uns jetzt die JavaScript-Plattform selbst näher beleuchten.

So haben wir dank Kotlin 2.0 nun die Möglichkeit, durch das Setzen eines Compilation Targets alle ES2015-Features zu aktivieren: Wie bereits die Generierung der TypeScript-Files, kann auch diese Änderung einfach in der *build.gradle*-Datei aktiviert werden. Damit werden die seit Kotlin 1.9 experimentell unterstützten ES2015 Modules und ES2015 Classes aktiviert und generiert (Listing 8).

Listing 8

```
kotlin {
  js {
    compilerOptions {
      target.set("es2015")
    }
  }
}
```

Doch das ist nicht alles. Durch das ES2015 Target haben wir nun auch experimentellen Zugriff auf die ES2015 Generators. Dadurch können nun die *suspend*-Funktionen aus Kotlin mittels *generators* in JavaScript kompiliert werden.

Ein weiterer Punkt ist das Übergeben von Argumenten in die *Main*-Funktion. Mit *passAsArgumentToMainFunction* und *passProcessArgvToMainFunction* ist es nun viel einfacher, Argumente direkt der *Main*-Methode zu übergeben. Wir können einen beliebigen JavaScript-Ausdruck übergeben und dieser wird dann als der *args*-Parameter verwendet. Das macht es um einiges einfacher, unser Programm aus der Kommandozeile mit Parametern aufzurufen (Listing 9).

Listing 9

```
kotlin {
  js {
    binary.executable()
    passAsArgumentToMainFunction("process.argv")
  }
}
```

Im Fall von NodeJS müssen wir das aber nicht immer manuell tun, sondern können die *passProcessArgvToMainFunction* verwenden (Listing 10).

Listing 10

```
kotlin {
  js {
    binary.executable()
    nodejs {
      passProcessArgvToMainFunction()
    }
  }
}
```

Feingranulare JavaScript-Dateien liegen nun ebenso im Bereich des Möglichen. Bisher konnte Kotlin JavaScript-Dateien nur auf zwei Arten generieren. Entweder wurde eine einzige *.js*-Datei erstellt, die den ganzen Code beinhaltete, oder, und so ist es aktuell noch der Standard, es kann eine Datei pro Projektmodul erstellt werden.

Mit Kotlin 2.0 existiert nun aber eine dritte Möglichkeit. Wir können nun eine JavaScript-Datei pro Kotlin-Datei erstellen lassen. Somit sind die Dateien feingranular aufgeteilt. Um dieses Feature zu aktivieren, können wir entweder das Target auf *es2015* setzen oder das spezifische ES2015-Module-Feature in *build.gradle* aktivieren und *gradle.properties* anpassen (Listing 11).

Listing 11

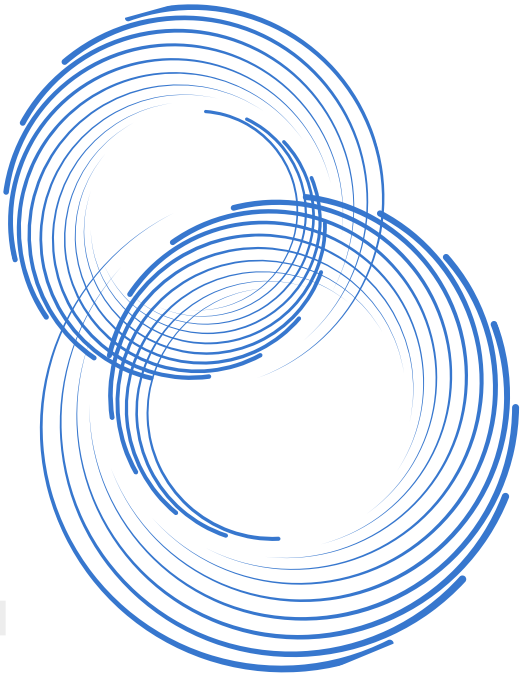
```
// build.gradle.kts
kotlin {
  js(IR) {
    useEsModules() // ENTWEDER

    browser()
    compilerOptions {
      target.set("es2015") // ODER
    }
  }
}

# gradle.properties
kotlin.js.ir.output.granularity=per-file
```

Obschon immer noch an der Interoperability zwischen Kotlin und Java Collections gearbeitet wird, erhalten wir in dieser Version bereits einen Vorgeschmack, in welche Richtung es sich in Zukunft weiterentwickeln wird.

Es wird nämlich möglich, Lists, Maps und Sets, seien es die Mutable- oder Immutable-Varianten, in JavaScript zu verwenden. Wir können aktuell zwar noch keine neuen Collections erstellen, aber bestehende verwenden. Die Anwendung ist dabei sehr intuitiv. Sagen wir, wir haben eine bestehende Liste in Kotlin (Listing 12). Dann kann diese ohne Probleme in JavaScript verwendet werden (Listing 13).



Listing 12

```
// Kotlin
@JsExport
data class Leser(
   val name: String,
   val geleseneArtikel: List<Artikel> = emptyList()
)

@JsExport
data class Artikel(val artikelName: String)

@JsExport
val leser = Leser(
   name = "Christian",
   geleseneArtikel = listOf(Artikel(artikelName = "Kotlin 2.0"))
)
```

Listing 13

```
// JavaScript
import { Leser, Artikel, leser, KtList } from "my-module"

const alleArtikelNamen = leser.geleseneArtikel
   .asJsReadonlyArrayView()
   .map(x => x.artikelName) // ["Kotlin 2.0"]
```

Wie eindrucksvoll zu sehen ist, können wir mittels *KtList* und *asJsReadonlyArrayView* auf die Liste zugreifen. Mit zukünftigen Versionen sollten wir auch die Möglichkeit erhalten, in JavaScript neue Collections zu erstellen. Wir dürfen gespannt sein.

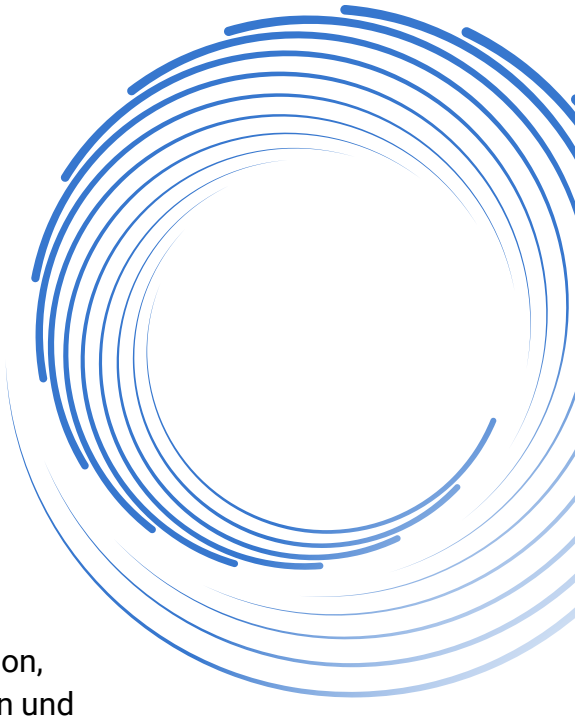
Neu ist auch der npm-Support. Bisher konnte innerhalb des Kotlin-Multiplatform-Gradle-Plug-ins nur Yarn als Package Manager verwendet werden. Ab jetzt wird auch npm, der bekannteste Package Manager, unterstützt. Doch er muss manuell in *gradle.properties* aktiviert werden. Ansonsten ist standardmäßig weiterhin Yarn aktiv:

```
kotlin.js.yarn=false
```

Erweiterte Standardbibliothek

Neben all den Änderungen, die spezifisch für ein Platform Target ausgerollt wurden, gibt es auch spannende Anpassungen in der Standard-Lib von Kotlin. Zum einen sind mehrere Funktionen jetzt stable und können ohne Bedenken bezüglich möglicher Anpassungen in zukünftigen Releases verwendet werden, zum anderen wurden einige völlig neue Features veröffentlicht.

So wurde die *enumValues*-Funktion de facto durch *enumEntries* ersetzt. Die Funktion *enumEntries<T>* war zwar schon in älteren Versionen verfügbar, aber noch nicht stable. Sie „ersetzt" die bestehende *enumValues<T>*-Funktion. Diese ist zwar immer noch verfügbar, aber das Entwicklungsteam empfiehlt klar, *enumEntries<T>* zu verwenden. Sie ist um einiges performanter, da bei ihrer Verwendung jedes Mal ein neues Array erstellt wird. Die neue Funktion gibt euch eine Liste aller Enum-Einträge des generischen Typs *T* zurück (Listing 14).



Listing 14

```
enum class COMPILERS { K2, LEGACY }

inline fun <reified T : Enum<T>> printAllValues() {
  print(enumEntries<T>().joinToString { it.name })
}

printAllValues<COMPILERS>()
// K2, LEGACY
```

Wie schon *enumEntries* ist nun auch *AutoClosable* in Kotlin 2.0 stabil. Java-Entwickler werden dieses Interface und dessen Vorteile bereits gut kennen. Es ermöglicht uns, nicht mehr benötigte Ressourcen einfach schließen zu können. In Kotlin erhalten wir dazu auch die nützliche *use*-Funktion, die wir verwenden können, um einen Block Code auszuführen und die Ressource danach direkt zu schließen (Listing 15).

Listing 15

```
val autoCloseable = AutoCloseable { writer.flushAndClose() } // (1)

autoCloseable.use {
  writer.doStuff()
}
```

Hier ist schön zu sehen, wie simpel wir ein solches *AutoClosable*-Konstrukt erstellen können. Zuerst wird in Zeile 1 eine *AutoCloseable*-Instanz erstellt, und wir geben die Information mit, was geschehen soll, wenn alle Aufgaben in *use* durchlaufen sind. Hier soll der Writer geflushed und geschlossen werden. Wir können nun sicher sein, dass diese Funktion am Ende des *Use*-Blocks – oder im Fall eines Fehlers – aufgerufen wird.

Auch die Common-Bibliothek hat Erweiterungen erfahren. In Kotlin 2.0 wurden mehrere Funktionen hinzugefügt, die vorher nur für spezifische Plattformen zu Verfügung standen. Für die *AbstractMutableList*-Klasse sind das die Methoden *modCount* und *removeRange*. Sie waren zuvor bereits für jede Plattform vorhanden, aber noch nicht im Common Target. Das hatte zur Folge, dass eigene Implementationen dieser Funktionen in jeder Plattform einzeln implementiert werden mussten, was nun nicht mehr der Fall ist.

Ebenfalls neu ist die Stringmethode *toCharArray(destination)* in der Common-Bibliothek. Bisher war sie nur in der JVM-Plattform vorhanden. Es gab bereits die *toCharArray*-Funktion, die aber jedes Mal ein eigenes neues *CharArray* erstellt hat. Die neue Funktion tut das nicht, sondern benutzt das Array *destination*, das der Methode übergeben wurde (Listing 16).

TRACK

W-JAX

Serverside Java: Entwickle dich weiter

Dieser Track vermittelt wertvolles Wissen, um robuste, skalierbare und Cloud-native Java-Anwendungen zu entwickeln. Erfahre mehr über die neuesten Frameworks, Bibliotheken und Best Practices, um in der dynamischen Java-Welt immer auf dem neuesten Stand zu bleiben.

Lernen Sie von unseren Expert:innen:

- **Spring Boot & Spring Cloud:** Entwickle skalierbare, Cloud-native Anwendungen mit Spring Boot und Spring Cloud für die reibungslose Integration von Cloud-Diensten.
- **Quarkus & GraalVM:** Entdecke die Serverless-Plattform Quarkus und ihre nahtlose Integration mit GraalVM für polyglotte Programmierung und ultimative Performance.
- **Jakarta EE 9 & MicroProfile:** Moderne Java-EE-Entwicklung mit Jakarta EE 9 und MicroProfile für Microservices-Architekturen.
- **Reactive Java mit RxJava & Project Reactor:** Implementiere asynchrone und reaktive Programmierparadigmen für hochperformante und skalierbare Anwendungen mit RxJava und Project Reactor.
- **Performance mit Java Flight Recorder & Prometheus:** Erhalte tiefgreifende Einblicke in die Performance deiner Anwendungen und optimiere sie mit Java Flight Recorder und Prometheus für maximale Effizienz.

Listing 16

```
fun main() {
   val myString = "Hallo Java Magazin!"
   val destinationArray = CharArray(myString.length)

   myString.toCharArray(destinationArray)

   for (char in destinationArray) {
     print("$char ")
     // H a l l o  J a v a  M a g a z i n !
   }
}
```

Gradle-Verbesserungen

Die Integration des Build-Tools für Kotlin, Gradle, hat in Kotlin 2.0 ebenfalls einige Verbesserungen erhalten. Wir werden diese Anpassungen hier nicht im Detail beleuchten, da sie oftmals im Bereich der Vereinfachung der Konfiguration der Kotlin-Projekte liegen.

Ein Punkt aber ist wichtig für die Entwicklung von Kotlin-2.0-Projekten, nämlich dass Gradle 6.8.3 bis einschließlich Version 8.5 unterstützt wird. Das heißt, wir haben weiterhin eine große Bandbreite an verfügbaren Versionen, die wir verwenden können.

Fazit: Wow!

Kotlin 2.0 scheint in Bezug auf Features auf den ersten Blick keine riesigen Schritte gemacht zu haben. Und obwohl sich die neuen Sprachfeatures in Grenzen halten, ist dieses Release unglaublich wichtig und interessant für die Zukunft von Kotlin. Kennen wir es nicht alle? Wir entwickeln eine Applikation und bauen Feature auf Feature. Da kann es schnell passieren, dass die Entwicklung mit der Zeit langsamer wird, da man sich nicht die Zeit nimmt, die Basis zu betrachten und Refactorings vorzunehmen, um zukünftige Entwicklungen einfacher zu gestalten.

Und genau dem hat Kotlin hier nun vorgebeugt und mit den internen Anpassungen des K2-Compilers den Grundstein neu gelegt, um die Entwicklung zukünftiger Features viel einfacher zu gestalten. Wir dürfen also gespannt sein, was die Zukunft bringen wird.



Christian Wörz arbeitet seit über zehn Jahren als Softwareentwickler. Nebst seiner Arbeit als Contractor für Kotlin, Java, TypeScript und JavaScript liebt er es, sein Wissen in Trainings und Talks weiterzugeben und selbst Neues zu lernen.

Ktor mit eigenen Plug-ins erweitern

Darf’s noch etwas mehr sein?

von Werner Eberling



Wenn es darum geht, REST/HTTP Services in Kotlin zu implementieren, dann ist Ktor immer häufiger das Mittel der Wahl. Das auf Kotlin basierende Framework zeichnet sich etwa dadurch aus, dass – bei kleinem Serverkern – einzelne Funktionalitäten per Plug-in modular hinzukonfiguriert werden können. So ist nur der Code in der Anwendung vorhanden, der wirklich gebraucht wird. Und sollte eine Funktionalität fehlen, so lassen sich unkompliziert eigene Plug-ins schreiben.

Webservices in purem Kotlin? Mit Ktor ist nichts leichter als das. Das Framework für HTTP-basierte Services hat seinen Bekanntheitsgrad in den vergangenen Jahren deutlich gesteigert. Und wie es bei solchen Web-Frameworks häufig der Fall ist, geht auch Ktor inzwischen mit jeder Menge funktionaler Erweiterungen an den Start. Da diese aber in Form von Plug-ins hinzukonfiguriert werden, blähen sie die eigene Anwendung nicht unnötig auf. Die Funktionalitäten sind nur dann mit paketiert, wenn sie wirklich benötigt werden.

Zudem können Entwickler das Framework nach eigenem Gusto erweitern und so eventuell fehlende Funktionalität einfach dazubauen. Schon praktisch, so ein Plug-in-basierter Ansatz!

Webservices mit Ktor – ein kurzes Recap

Bevor wir uns aber in die Welt der Ktor-Plug-ins und deren Entwicklung stürzen, soll ein kurzes „Hallo“ an die Welt dort draußen zeigen – oder, je nach Vorkenntnissen, in Erinnerung rufen –, wie Ktor grundsätzlich funktioniert. Listing 1 zeigt, wie ein einfacher *HelloWorld*-Service mit Ktor implementiert werden kann. Die dafür bereitgestellten Funktionen und Parameter fügen sich dabei zu einer schön lesbaren Kotlin-basierten „Webservices-DSL“ zusammen, sodass die Funktionsweise der Services direkt aus dem Code heraus ersichtlich wird – ohne zusätzlichen technischen Boilerplate-Code.

Listing 1: HelloWorld.kt

```
fun main() {
  embeddedServer(Netty, port = 8080, host = "0.0.0.0") {
    routing {
      route("/hello") {
        get { call.respondText("Hello Ktor!") }
      }
    }
  }.start(wait = true)
}
```

In diesem Fall soll der Service über einen integrierten Netty Server an Port 8080 an allen verfügbaren IP-Netzwerkadressen des Rechners bereitgestellt werden. Über die Funktion *routing* wird die einzige vom Service unterstütze Webroute */hello* definiert und im Fall eines GET Request im inneren Funktionsbody auch direkt die Antwort auf diese Route implementiert. Hier ist es die Rückgabe des statischen Textes *„Hello Ktor!“*. Wenn man nun nicht vergisst, den so vorkonfigurierten Server mittels seiner Methode *start* hochzufahren und ihm dabei aufzutragen, dass er bitte auf Anfragen warten soll *(wait = true)*, dann steht einem Test des Service über den Browser nichts mehr im Weg.

Wurde die Anwendung gestartet, kann mit dem Browser der Wahl über den URL *http://localhost:8080/hello* auf den Service zugegriffen werden. Die Antwort erscheint im Browserfenster (**Abb. 1**).



ABB. 1 Hello World im Browser

Soviel zum (Wieder-)Einstieg in die Ktor-Welt. Wer noch mehr über die grundlegenden Funktionsweisen des Frameworks erfahren möchte, dem sei der Artikel unter [1] oder ein Blick in die Ktor-Dokumentation [2] ans Herz gelegt.

Das gerade betrachtete Beispiel war allerdings mehr als nur ein einfacher Ktor-Service, der dem Leser wieder die Funktionsweise von Ktor ins Gedächtnis rufen sollte. Vielmehr kam hier schon fast unbemerkt das erste Plug-in zum Einsatz. Die im Beispiel verwendete Routendefinition ist so nur möglich, wenn das entsprechende Routing-Plug-in verfügbar und installiert ist. Womit wir schon mitten im Thema wären – aber eins nach dem anderen.

Standards und mehr – Funktionalität einer Ktor-Anwendung zusammenstellen

Der modulare Plug-in-basierte Aufbau einer Ktor-Anwendung wird schon bei der Zusammenstellung der benötigten Bibliotheken auf der Ktor-Starter-Website [3] ersichtlich. Die Nutzung dieser Website bietet sich an, um das benötigte Projektgerüst inklusive Bibliotheksreferenzen aufzubauen (Besitzer einer IntelliJ-IDEA-Ultimate-Lizenz können ein Ktor-Projekt auch direkt über den entsprechenden Wizard ihrer IDE generieren).

Hierbei werden erst einmal grundlegende Einstellungen vorgenommen, wie die Auswahl der zu verwendenden Ktor-Version (zum Entstehungszeitpunkt dieses Artikels war Version 3.0.1 latest and greatest), des gewünschten Build-Systems (Gradle, Maven oder das JetBrain-Eigengewächs Amper) und der zu verwendeten Laufzeitumgebung (Netty, Jetty, CIO oder Tomcat).

Der Hauptteil der Website beschäftigt sich aber mit der Auswahl der für die gewünschte Funktionalität benötigten Plug-ins. Hier werden „Standard-Plug-ins“ wie Routing, Content Negotiation und JSON-Serialisierung fast immer auf der Liste der benötigten Bibliotheken landen. Aber auch Themen, wie z. B. Authentifizierung auf den unterschiedlichsten Wegen (Basic, Digest, JWT, OAuth etc.), stehen zur Verfügung und werden vermutlich in der ein oder anderen Variante in die Auswahl wandern (**Abb. 2**). Wurden alle benötigten Plug-ins ausgewählt, kann über GENERATE PROJECT eine ZIP-Datei mit dem benötigten Projektskelett und auf Wunsch auch einer ersten Beispielanwendung heruntergeladen werden.

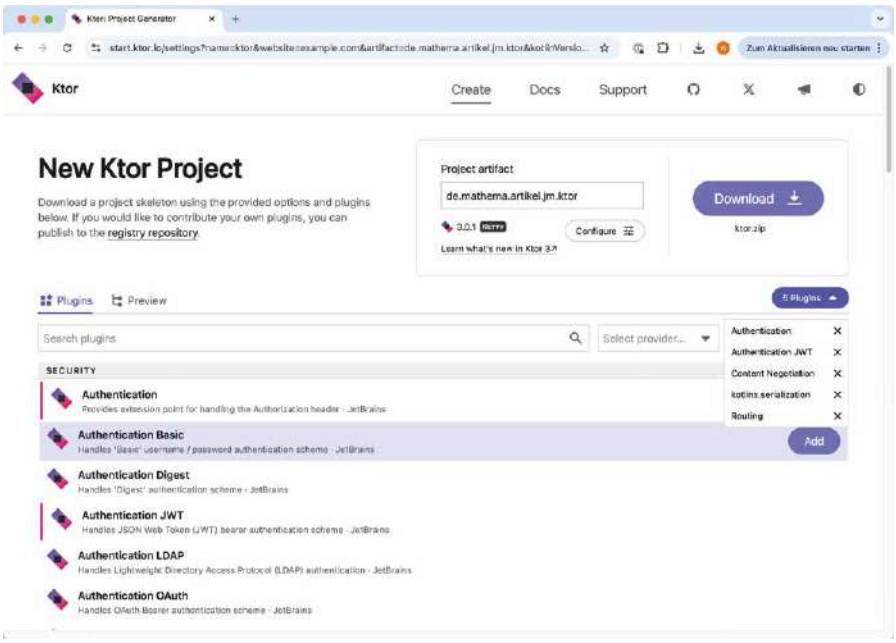


ABB. 2 Ktor-Starter

Wurde bei der Zusammenstellung des Projekts die Option zur Erzeugung von Beispielcode aktiviert, lässt sich das Vorgehen zur Installation und Nutzung der ausgewählten Plug-ins direkt im generierten Code nachvollziehen. Im Ordner PLUGINS wurde für jedes ausgewählte Plug-in eine Kotlin-Datei angelegt. Jede dieser Dateien enthält den für die Installation des jeweiligen Plug-ins notwendigen Code (**Abb. 3**).

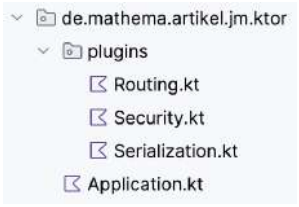


ABB. 3 Plug-in-Ordner

Dazu wird die Plug-in-Referenz an die *install*-Funktion übergeben und so der Anwendung bekannt gemacht. Zusätzlich kann innerhalb einer Lambda Konfigurationscode für das Plug-in ausgeführt werden. Im Fall des Serialisierungs-Plug-in wird hier beispielsweise JSON als mögliches Format registriert, indem die entsprechende Funktion angegeben wird:

```
fun Application.configureSerialization() {
  install(ContentNegotiation) {
    json()
  }
}
```

Der Registrierungscode wird als Extension Function der eigentlichen Ktor *Application* erzeugt, sodass er alle notwendigen Funktionen im Zugriff hat und bei der Konfiguration der Anwendung einfach aufgerufen werden kann:

```
fun Application.module() {
  configureSerialization()
  configureSecurity()
  configureRouting()
}
```

Bei genauerer Betrachtung des generierten Codes fällt auf, dass sich die beiden anderen Plug-ins nicht an das Muster *install(PLUG-IN)* halten. Vielmehr scheinen hier mit *routing {…}* und *authentication {…}* spezielle Schlüsselwörter – oder besser Funktionen – zu existieren, um das jeweilige Plug-in zu installieren. Hierbei handelt es sich tatsächlich um Komfortfunktionen, die für häufig genutzte Plug-ins bereitgestellt werden. Diese ersetzen bzw. kapseln den standardmäßigen Aufruf von *install(PLUG-IN*)und sollen so die Nutzung erleichtern (Listing 2).

Listing 2: Routing.kt und Security.kt (Ausschnitte)

```
fun Application.configureSecurity() {
  authentication {…}
}

fun Application.configureRouting() {
  routing {…}
}
```

Dem geneigten Leser, der das z. B. für das Routing-Plug-in direkt ausprobieren möchte, sei noch der Hinweis mitgegeben, dass das für diesen Fall benötigte Plug-in RoutingRoot heißt. Der äquivalente *install*-Aufruf würde damit *install(RoutingRoot)* und nicht *install(Routing)* lauten. Der Versuch, Letzteres auszuführen, wird mit dem Hinweis auf ein fehlendes *Companion-Object* und einer somit fehlenden Initialisierung scheitern.

Eigene Plug-ins implementieren und installieren

Was mit Standard-Plug-ins funktioniert, ist natürlich auch für selbst geschriebene möglich [4]. Ist das Plug-in erst einmal erzeugt und die Referenz darauf verfügbar, so kann es auch per *install*-Funktion registriert werden und seine Arbeit verrichten.

Hierzu heißt es nun aber erst einmal, ein entsprechendes Plug-in zu erzeugen. Das ist in Ktor nicht komplizierter, als eine vom Framework bereitgestellte Funktion zu rufen. In diesem Fall die Funktion *createApplicationPlug-in*. Ihr werden der Name des Plug-ins, Informationen zu dessen Konfiguration (dazu später mehr) und der eigentliche Code des Plug-ins als Lambda übergeben. Der Code, der innerhalb der Lambda-Funktion definiert wird, wird zum Installationszeitpunkt des Plug-ins ausgeführt. Dabei können neben direkten Aktionen auch weitere Lambdas für spezielle Callback-Funktionen registriert werden. Callbacks sind für vordefinierte Zustände in der Request-Verarbeitung oder auch des Anwendungslebenszyklus vorhanden. Die dort registrierten Funktionen werden von Ktor in den entsprechenden Situationen automatisch gerufen, sodass bei Bedarf in die Verarbeitungskette eingegriffen oder zusätzliche Logik implementiert werden kann.

```
val MyFirstPlug-in = createApplicationPlug-in(name = "MyFirstPlug-in") {
   println(“Mein erstes Plug-in wurde registriert!!!")
   onCall {call -> println("Request für ${call.request.uri} entgegengenommen.")}
}
```

Wird das in obigem Code definierte Plug-in beim Start der Anwendung registriert (siehe auch Listing 3)

```
val fun Application.configureCustomPlug-in(){
    install(MyFirstPlug-in)
}
```

Listing 3: Application.kt

```
val fun Application.module() {
   configureSerialization()
   configureSecurity()
   configureRouting()
   configureCustomPlug-in()
}
```

und nach dem Start der Anwendung im Browser der URL *http://localhost:8080* gerufen, finden sich die Ausgaben aus Listing 4 im Logfile.

Listing 4

```
[main] INFO  Application - Autoreload is disabled because the development
                                                                mode is off.
Mein erstes Plug-in wurde registriert!!!
[main] INFO  Application - Application started in 0.209 seconds.
[DefaultDispatcher-worker-1] INFO  Application - Responding at
                                                        http://0.0.0.0:8080
Request für / entgegengenommen.
…
```

Wie erwähnt, kann im Plug-in-Code u. a. im Rahmen der Request-Verarbeitung auf eine Reihe von Ereignissen reagiert werden. Tabelle 1 zeigt die Callback-Funktionen, die hierfür genutzt werden können.

Funktion	Aufrufzeitpunkt/Möglichkeiten
onCall()	Wird beim Eintreffen eines Client-Calls gerufen und ermöglicht den Zugriff auf die Metadaten dieses Calls sowohl bzgl. des Requests (z. B. URI, Queryparameter, Header etc.) als auch bzgl. der Response (z. B. Header). Auch deren Manipulation ist möglich (z. B. Setzen von Custom Headern in der Antwort).
onCallReceive()	Wird beim Empfangen des Client-Calls, speziell der übertragenen Daten, gerufen. Anders als bei *onCall()* hat die Funktion hier Zugriff auf die im Rahmen des Calls übertragenen Daten (Request Body) und kann diese bei Bedarf manipulieren.
onCallRespond()	Wird vor dem Ausliefern der Antwort an den Client gerufen. Auch hier hat die Funktion die Möglichkeit, den Inhalt der Antwort (Response Body) zu ändern, bevor diese übertragen wird.
on(…)	Möglichkeit, auf weitere vordefinierte Ereignisse zu reagieren. Die verfügbaren Ereignisse im Rahmen der Request-Verarbeitung werden in den folgenden Zeilen beschrieben.
on(CallSetup)	Wird vor der eigentlichen Call-Verarbeitung gerufen und kann für frühzeitige, grundlegende Call-Validierungen genutzt werden.
on(ReceiveRequestBytes)	Wird vor der Aufbereitung des Client Request Bodies gerufen und ermöglicht den Zugriff auf die Rohdaten des Requests.
on(ResponseBodyReadyForSend)	Wird gerufen, bevor die komplett aufbereitete Antwort (alle Datentransformationen sind bereits durchgeführt) an den Client gesendet wird.
on(ResponseSent)	Wird aufgerufen, nachdem die Antwort erfolgreich an den Client gesendet wurde. Bietet die Möglichkeit, Metrikmessungen zu beenden oder gehaltene Ressourcen freizugeben.
On(CallFailed)	Wird aufgerufen, falls innerhalb der Verarbeitungskette ein Fehler auftritt. Kann zur zentralen Fehlerbehandlung (Vermeidung von HTTP-500) genutzt werden.

Tabelle 1: Call-back-Funktionen für das Reagieren auf bestimmte Ereignisse

Es existieren noch weitere Ereignisse, auf die theoretisch reagiert werden könnte, wie z. B. Metrics. Diese sind aber als intern gekennzeichnet und sollten nicht benutzt werden, da sie mit jedem Versionsupdate geändert oder entfernt werden können.

Auch auf Ereignisse aus dem Lebenszyklus der Anwendung selbst kann in einem Plug-in reagiert werden. Hierzu muss lediglich mit der bereits bekannten *on(…)*-Funktion eine Callback-Funktion für das passende *MonitoringEvent* registriert werden (z. B. *on(MonitoringEvent (ApplicationStarted))*) – mehr hierzu unter [5].

Wie lange brauchst du? – Zustand im Call speichern

Aufbauend auf den bisher vorgestellten Mechanismen ist es jetzt möglich, ein Plug-in zu schreiben, das die Verarbeitungszeit eines Request misst und sie sowohl im Log der Anwendung ausgibt als auch – in Form eines Custom Headers – dem Client zurückliefert. Hierzu kann im Callback *onCall()* die aktuelle Systemzeit ausgelesen und gespeichert werden. Bevor die Antwort an den Client ausgeliefert wird, wird in *onCallRespond()* erneut die Systemzeit ausgelesen (soll die vollständige Aufbereitung der Antwort mit in die Berechnung aufgenommen werden, so kann auch der Callback *on(ResponseBodyReadyForSend)* zur Bestimmung des Endzeitpunkte genutzt werden). Die Differenz beider Werte ergibt die Zeit der Request-Verarbeitung, die nun im Logfile ausgegeben werden kann. Zusätzlich lässt sich mit *call.response.header(…)* ein entsprechender Custom Header in die Response setzen und so die Information in den Metadaten der Antwort an den Client senden.

Soweit der grundsätzliche Plan, doch wie soll die Anfangszeit aus *onCall()* zur Berechnung in die Callback-Funktion von *onResponse()* gelangen? Ein Ansatz könnte die Speicherung innerhalb von Variablen des Plug-ins sein. Hierbei ist allerdings zu beachten, dass alle Requests durch dasselbe Plug-in laufen, die Verarbeitung also hoch nebenläufig ist. Hier müsste also auf jeden Fall eine Datenstruktur gewählt werden, die einen parallelen Zugriff unterstützt (z. B. *kotlin.concurrent.AtomicLong*).

Obwohl das für globale Zustände eine Lösung darstellen kann, hilft der Ansatz im vorliegenden Beispiel nicht weiter. Hier soll ja die Zeit pro Call gemessen werden, d. h., es wird eine Datenspeicherung im Scope des aktuellen Calls benötig. Glücklicherweise sieht Ktor das Sharing von Daten zwischen verschiedenen Callbacks innerhalb einer Request-Verarbeitung bereits Framework-seitig vor. Jedem Call können dazu mittels *call.attributes.put*(…) Attribute mitgegeben werden, die in späteren Callbacks wieder ausgelesen werden können.

Somit haben wir alles zusammen, um das beschriebene Plug-in zu implementieren. Listing 5 zeigt das Ergebnis, das über den bereits bekannten Weg (Anwendung starten und *http://localhost:8080* aufrufen) getestet werden kann.

Listing 5: MyFirstPlug-in.kt

```
val MyFirstPlug-in = createApplicationPlug-in(name = "MyFirstPlug-in") {
  println("Mein erstes Plug-in wurde registriert!!!")
  val startTimeKey = AttributeKey<Long>("requestStartTime")

  onCall {
    call -> call.apply {
      println("Request für ${call.request.uri} entgegengenommen.")
      attributes.put(startTimeKey, System.currentTimeMillis())
    }
  }

  onCallRespond {
    call -> call.apply {
      val now = System.currentTimeMillis()
      println("Total request computation time: ${now -
                                              attributes[startTimeKey]}ms")
      response.header("C-Cust-Rct", "${now - attributes[startTimeKey]}ms")
    }
  }
}
```

Ähnlich, aber immer anders – Plug-ins konfigurieren

Um die Funktionalität eines Plug-ins an die jeweiligen Bedürfnisse anpassen zu können, können Plug-ins bei der Installation konfiguriert werden. Schon das im Beispiel enthaltene Routing-Plug-in enthält eine solche Konfiguration mit dem Typ *Routing*. Sie ermöglicht die Definition der eigentlichen Anwendungsrouten (Listing 6).

Listing 6: Routing.kt

```
val fun Application.configureRouting() {
  // entspricht: install(RoutingRoot)
  routing {
    get("/") { call.respondText("Hello World!") }
  }
}
```

Auch selbst geschriebene Plug-ins können auf diesem Weg mit Konfigurationswerten versorgt werden. Alles, was es dazu braucht, ist eine Klasse, die die Konfigurationswerte kapselt. Im aktuellen Beispiel könnten dies der Name des Custom Headers in der Antwort sowie die Aktivierung der Ausgabe der gemessenen Zeit im Logfile und als Header sein. Zusätzlich werden für alle drei Werte innerhalb der Konfigurationsklasse Defaultwerte hinterlegt:

```
class MyFirstPlug-inConfiguration() {
   var logTime: Boolean = true
   var addTimeCustomHeader: Boolean = false
   var customHeaderName = "X-Cust-Rct"
}
```

Nun muss die neu geschaffene Konfigurationsklasse noch als Konfiguration des Plug-ins registriert werden (Listing 7). Das geschieht bei Erzeugung des Plug-ins. Neben Namen und Plug-in-Code gibt es hier die Möglichkeit, die Referenz auf eine Konfigurationsklasse anzugeben. Wird hier die neu geschaffene Klasse angegeben, so ist sie innerhalb der Plug-in-Codes verfügbar und die gesetzten Konfigurationen können ausgelesen werden. Da diese grundsätzlich änderbar sind, empfiehlt es sich, sie zu Beginn der Plug-in-Initialisierung auszulesen und unveränderbar zu speichern.

TRACK

W-JAX

Web Development & JavaScript:
Moderne performante Frontends für Java-Systeme

Lernen Sie von renommierten Expert:innen, wie Sie mit Angular oder React hochperformante und skalierbare Frontends bauen, moderne Architekturen für optimale User Experience implementieren, die Performance optimieren und umfassenden Schutz vor Sicherheitsrisiken gewährleisten.

Lernen Sie von unseren Expert:innen:

- **JavaScript-Frameworks (React, Angular, SvelteKit):** Erfahre, wie du diese leistungsstarken Frameworks nutzen kannst, um dynamische und ansprechende Benutzeroberflächen zu erstellen.
- **Microfrontends:** Entdecke diesen revolutionären Ansatz zum Erstellen von groß angelegten Anwendungen mit verbesserter Skalierbarkeit und Wartbarkeit.
- **Performance-Optimierung:** Meistere wichtige Techniken, um sicherzustellen, dass deine Webanwendungen eine reibungslose und reaktionsschnelle Benutzererfahrung bieten.
- **Testing (Cypress, Playwright):** Schreibe robuste und wartbare Testsuites, um Fehler frühzeitig im Entwicklungsprozess zu erkennen.
- **Zukunft der Webentwicklung:** Erhalte Einblicke in die neuesten Trends und Innovationen, die die Zukunft des Webs prägen.

Listing 7: MyFirstPlug-in.kt (mit Konfigurationsklasse)

```
val MyFirstPlug-in = createApplicationPlug-in(
  name = "MyFirstPlug-in",
  createConfiguration = ::MyFirstPlug-inConfiguration
  ) {
    val logTime = Plug-inConfig.logTime
    val addHeader = Plug-inConfig.addTimeCustomHeader
    val headerName = Plug-inConfig.customHeaderName
    // ...
onCallRespond {
      call -> call.apply {
        val now = System.currentTimeMillis()
        if (logTime) println("Total request computation time: ${now -
                                                attributes[startTimeKey]} ms")
        if (addHeader) response.header(customHeaderName, "${now -
                                                attributes[startTimeKey]} ms")
      }
    }
  }
```

Nachdem Konfigurationsklasse und Plug-in nun miteinander verbunden sind, kann das Plug-in ab sofort bei der Installation konfiguriert werden. Werte, die dabei nicht explizit angegeben werden, fallen auf ihre Defaultwerte zurück (Listing 8).

Listing 8: CustomPlug-in.kt

```
val fun Application.configureCustomPlug-in() {
  install(MyFirstPlug-in) {
logTime = true
    addTimeCustomHeader = true
  }
}
```

Es muss nicht immer Code sein – Konfiguration via Konfigurationsfile

Die nun verfügbare Plug-in-Konfiguration kann bei Bedarf noch um die Möglichkeit zur Verwendung der *application.conf* bzw. *application.yaml* als Quelle der Konfiguration erweitert werden. Dafür muss zuerst die Pfadstruktur der Konfiguration definiert werden. Im aktuellen Beispiel wird ein neuer Konfigurationsblock mit der Bezeichnung *mfp (my first Plug-in)* verwendet, der die drei Konfigurationsparameter enthält (Hier wird die HOCON-Syntax [6] für die Darstellung der Anwendungskonfiguration verwendet, das Vorgehen funktioniert aber natürlich genauso mit einer YAML-basierten Konfiguration [7]):

```
mfp {
  logTime = "true"
  addTimeCustomHeader = "true"
  customHeaderName = "X-Cust-Rct"
}
```

Um nun auf die Anwendungskonfiguration zuzugreifen, muss diese bei der Erzeugung der Konfigurationsklasse übergeben werden. Das heißt, der Konstruktor der Klasse muss einen entsprechenden Parameter erhalten. Dadurch ist die Konfiguration im Zugriff und kann innerhalb der Klasse ausgelesen und zur Initialisierung der Felder genutzt werden (Listing 9).

Listing 9: MyFirstPlug-inConfiguration.kt

```
class MyFirstPlug-inConfiguration(appConf : ApplicationConfig) {
  var logTime: Boolean = appConf.tryGetString("logTime")?.toBoolean() ?: true
  var addTimeCustomHeader: Boolean =
    appConf.tryGetString("addTimeCustomHeader")?.toBoolean() ?: false
  var customHeaderName = appConf.tryGetString("customHeaderName")
                                                          ?: "X-Cust-Rct"
}
```

Dem aufmerksamen Leser mag auffallen, dass der Zugriff auf die einzelnen Werte innerhalb der Anwendungskonfiguration ohne Pfadangabe erfolgt. Damit das auch funktioniert, muss bei der Erzeugung des Plug-ins der Pfad zu den für das Plug-in gültigen Werten als *configurationPath* mit angegeben werden. Womit der letzte noch nicht betrachtete Parameter der Funktion *createApplicationPlug-in(…)* mit Leben gefüllt wäre. Der endgültige Funktionsaufruf sieht damit wie folgt aus:

```
val MyFirstPlug-in = createApplicationPlug-in(
  name = "MyFirstPlug-in",
  configurationPath = "mfp",
  createConfiguration = ::MyFirstPlug-inConfiguration
  ) { … }
```

Ein letzter Test – oder vielleicht doch nur der erste?

Wird das komplette Plug-in nun zusammen mit der Anwendung gestartet und mittels *http://localhost:8080* auf den Root Context von *localhost* zugegriffen, so lässt sich sowohl im Logfile als auch im Header der Antwort die für die Request-Verarbeitung benötigte Zeit ablesen (Listing 10, **Abb. 4**).

Listing 10

```
[main] INFO  Application - Autoreload is disabled because the development
                                                              mode is off.
Mein erstes Plug-in wurde registriert!!!
[main] INFO  Application - Application started in 0.209 seconds.
[DefaultDispatcher-worker-1] INFO  Application -
                                          Responding at http://0.0.0.0:8080
Request für / entgegengenommen.
…
Total request computation time: 9 ms
```

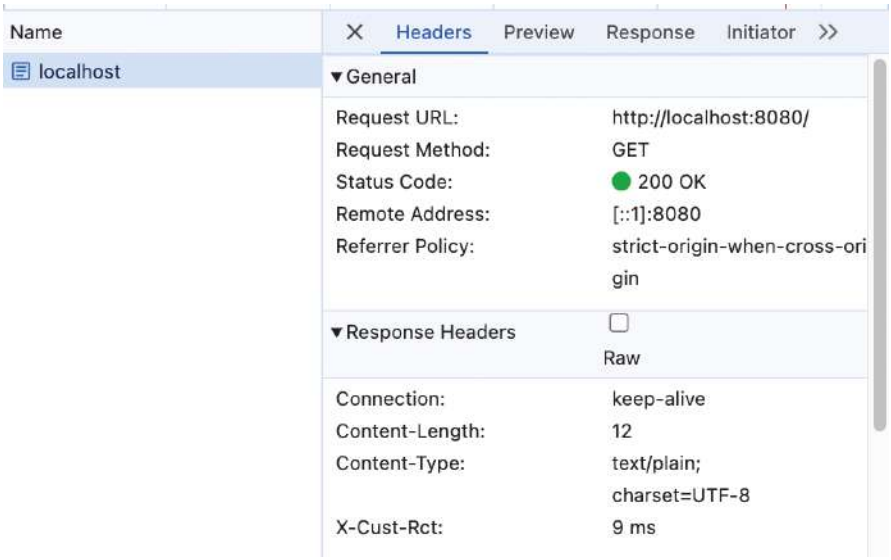


ABB. 4 Custom Header mit berechneter Verarbeitungszeit

Zugegeben, Logfiles und Header im Browser zu prüfen, funktioniert. Allerdings fühlt sich das Vorgehen alles in allem irgendwie falsch an. Daher sei zum Abschluss dieses Artikels noch erwähnt, dass sich die Funktion des eigenen Plug-ins als Bestandteil der Anwendung natürlich auch im Unit-Test prüfen lässt. Da zuletzt die Konfiguration per Anwendungskonfiguration betrachtet wurde, soll dieses Vorgehen nun auch im Unit-Test zum Einsatz kommen. Einmal mit aktiver Headerkonfiguration:

```
mfp {
  logTime = "true"
  addTimeCustomHeader = "true"
  customHeaderName = "X-Cust-Rct"
}
```

Und einmal mit inaktiver Headerkonfiguration:

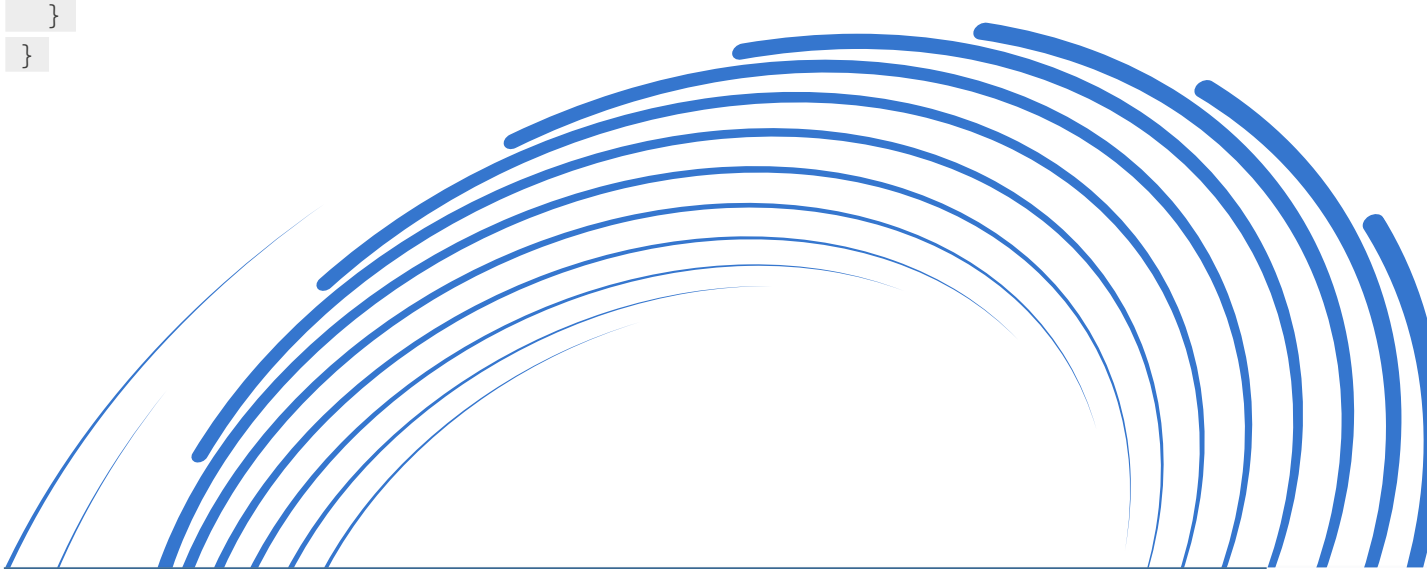
```
mfp {
  logTime = “true”
  addTimeCustomHeader = “false”
  customHeaderName = “X-Cust-Rct”
}
```

Im Unit-Test kann nun einmal auf die eine und einmal auf die andere Konfiguration zurückgegriffen werden, um auf das Vorhandensein oder eben Nichtvorhandensein des Headers zu prüfen (Listing 11).

Listing 11: ApplicationTest.kt

```
class ApplicationTest {
  @Test
    fun testHeaderPresentWhenConfigured() = testApplication {
      environment {
        config = ApplicationConfig("application-test-header-on.conf")
      }
      client.get("/").apply {
      assertTrue(call.response.headers.contains("X-Cust-Rct"))
    }
  }

  @Test
  fun testHeaderNotPresentWhenNotConfigured() = testApplication {
    environment {
      config = ApplicationConfig("application-test-header-off.conf")
    }
    client.get("/").apply {
      assertFalse(call.response.headers.contains("X-Cust-Rct"))
    }
  }
}
```



Fazit

Der modulare Aufbau von Ktor macht es einfach, das Framework um eigene Funktionalitäten zu erweitern. Nicht nur, dass Ktor ein einfaches und schnell erlernbares Programmiermodell bereitstellt, in dem ein Plug-in mit einem einzigen Funktionsaufruf erzeugt werden kann. Über wohldefinierte Callback-Funktionen wird der Eingriff in die Verarbeitung klar strukturiert und ist mit minimalem Aufwand möglich. Besonders die Tatsache, dass die meisten Funktionen in Ktor selbst in Plug-ins organisiert sind (frei nach dem Motto: „Eat your own dogfood!"), stellt sicher, dass der Erweiterungsmechanismus nicht nur stiefmütterlich „mitgeschleift" wird, sondern als „first class citizen" gepflegt und weiterentwickelt wird. In diesem Sinne: Happy Plug-in Coding!

(Der Quellcode zu den hier im Artikel gezeigten Beispielen ist auf GitHub verfügbar [8].)



Werner Eberling ist Principal Consultant und Trainer bei MATHEMA GmbH. Seit 1999 befasst er sich mit verteilten Systemen – angefangen bei C++/CORBA über die Java EE bis hin zu Microservices- und Cloud-Architekturen. Er ist Software-Architekt in größeren Scrum-Projekten und bietet Technologietrainings im deutschsprachigen Raum an. Sein Wissen gibt er auch als Autor von Fachbüchern und Fachartikeln sowie als Speaker auf IT-Konferenzen weiter.

@Wer_Eb

Links & Literatur

[1] Eberling, Werner: „Mit Ktor REST APIs in ‚purem' Kotlin entwickeln", Java Magazin 12.2022: https://entwickler.de/java/kotlin-rest-api

[2] Ktor Documentation – Welcome: https://ktor.io/docs/welcome.html

[3] Ktor Project Generator: https://start.ktor.io

[4] Extending Ktor – Custom Plug-ins: https://ktor.io/docs/custom-Plug-ins.html

[5] Extending Ktor – Handle events Plug-in: https://ktor.io/docs/server-events.html#handle-events-Plug-in

[6] Human-Optimized Config Object Notation: https://github.com/lightbend/config/blob/main/HOCON.md

[7] HOCON- und YAML-Konfiguration: https://ktor.io/docs/server-configuration-file.html

[8] Codebeispiele aus diesem Artikel auf GitHub: https://github.com/wern/ktor-custom-Plug-in-sample

W-JAX

DIE KONFERENZ FÜR JAVA, ARCHITEKTUR UND SOFTWARE-INNOVATION

