



# Kotlinx.serialization: Сериализуют все!

Кулешов Андрей

**Positive Technologies**

Руководитель разработки

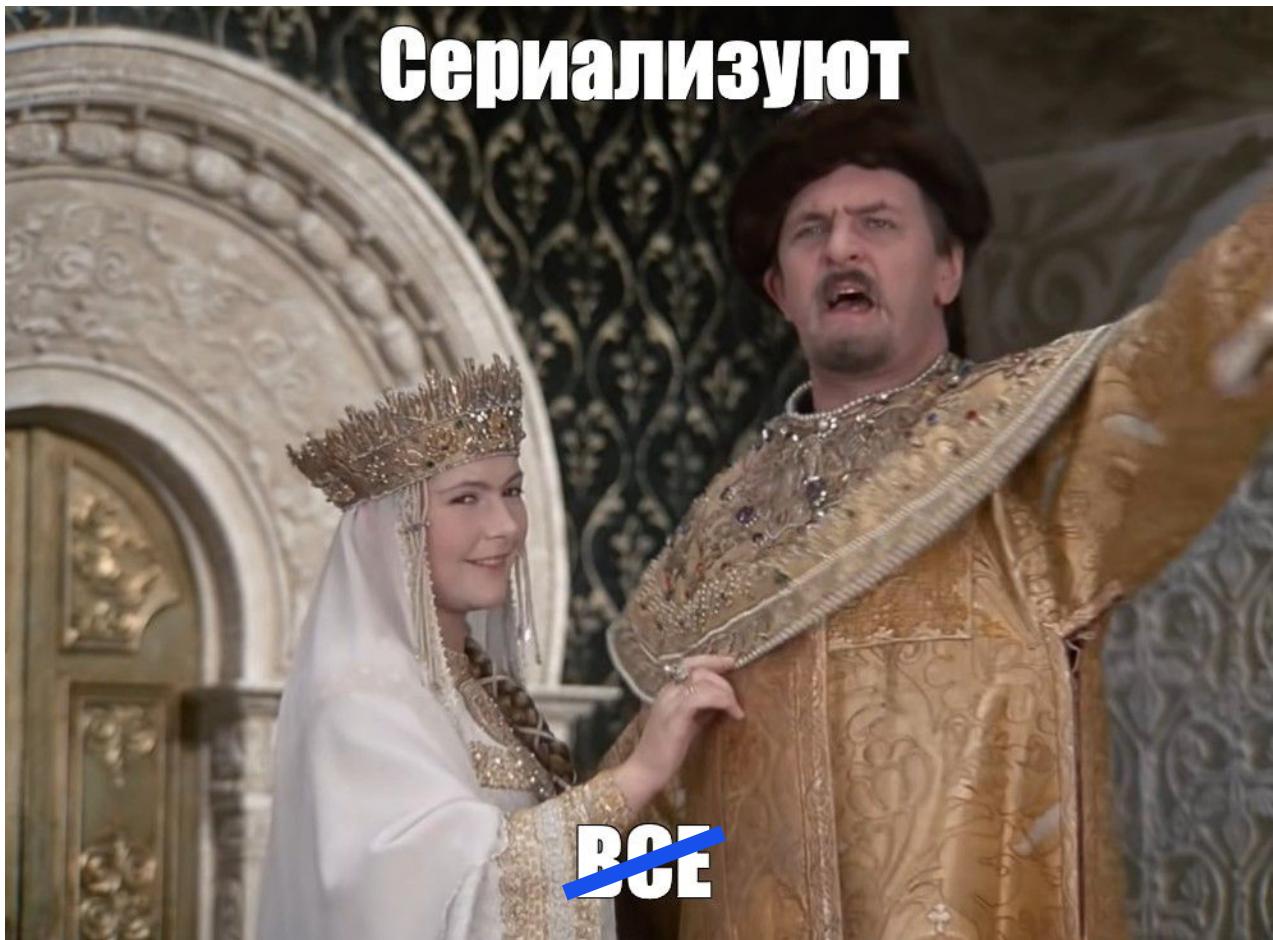
**Сериализуют**

**ВСЕ**



Сериализуют

~~Все~~



**Сериализуют**

**ВСЁ**



# \$ whoami?



## Андрей Кулешов

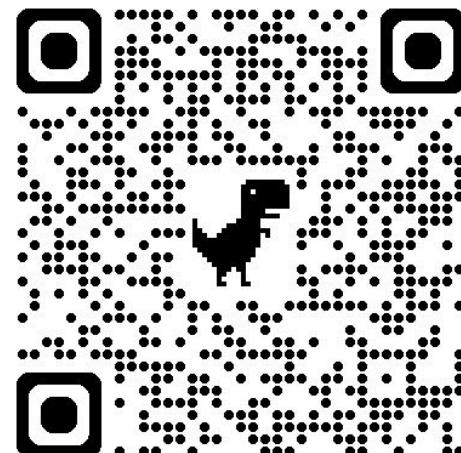


Стачка

phd 2



JPoint  
Joker<?>



# О чём поговорим?

`kotlinx.serialization`

# О чём поговорим?

`kotlinx.serialization`

что это за зверь

# О чём поговорим?

`kotlinx.serialization`

что это за зверь

что там под капотом

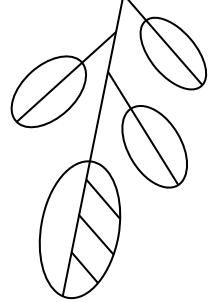
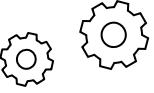
# О чём поговорим?

`kotlinx.serialization`

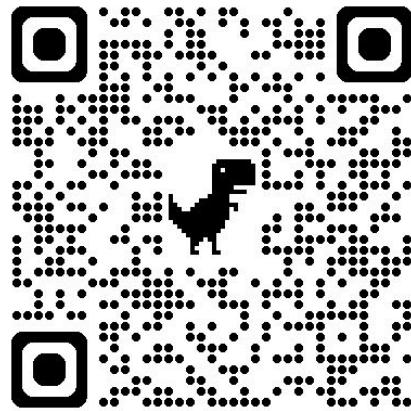
что это за зверь

что там под капотом

как это кастомизировать

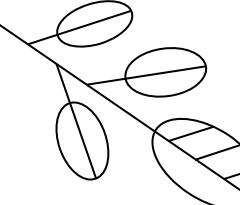
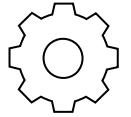


# Почему я?



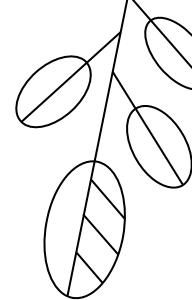
[KT]oml

powered by kotlinx.serialization



# Предыстория

# Kotlinx в Spring-backend мире



The screenshot shows the top navigation bar of the Joker conference website. It includes links for 'Расписание' (Schedule), 'Спикеры' (Speakers), 'Партнеры' (Partners), 'О нас' (About us), 'Архив' (Archive), 'Эксперты' (Experts), 'Code of Conduct', and a dropdown menu 'Еще'. There are also buttons for 'Стать спикером' (Become a speaker), 'EN', and 'Войти' (Login).

ДОКЛАД Libraries and Frameworks 15.10 / 14:30 – 15:15 (UTC+3)

## Spring и Kotlin: работай с любым форматом!

RU



Презентация pdf



Поговорим о том, как можно организовать десериализацию данных в Spring-приложении, когда вам приходится работать с различными форматами данных.

Обычно при разработке очередного технического сервиса для получения данных мы используем JSON как основной формат передачи данных. Это просто и работает из коробки: берем Java, с помощью Spring достаточно создать контроллеры, а Jackson в нем под капотом будет автоматически десериализовать входящие данные.

Но что делать, если ваша система должна работать не только с JSON, но и с другими форматами данных? Например, с YAML, XML, TOML, Protobuf или CBOR. В этом случае возникает множество вопросов. Нужно ли писать отдельную логику для каждого формата? Как справляться с различными интерфейсами и API сериализационных библиотек? Что делать с разнообразными аннотациями и как в одном классе описывать схему данных? А может, необходимо определять отдельные классы для разных сериализаторов, дублируя логику приложения?

Представьте, что вы работаете с платформой, которая должна принимать данные в различных форматах. Если для каждого формата писать отдельную логику десериализации, это приведет к сложному и запутанному коду. Кроме того, вам придется добавлять множество аннотаций от различных библиотек на целевые классы, что также усложнит поддержку кода. Здесь на помощь приходит Kotlin, а точнее – библиотека `kotlinx.serialization`. Она представляет собой компиляторский плагин и набор интерфейсов, имплементируя которые вы можете создать свою собственную библиотеку сериализации. Дополнительное весомое преимущество: она работает без рефлекшена и строит ветви процессинга данных уже на этапе компиляции.

### Спикеры

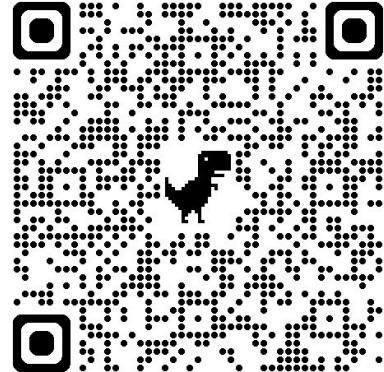


Андрей Кулешов  
Positive Technologies

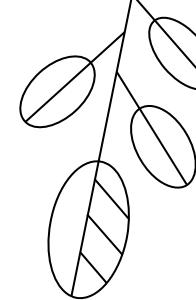
### Приглашенные эксперты



Александр Нозик  
МФТИ



# Базовые вещи с со стороны разработчиков сериализаторов



JPoint      Расписание    Спикеры    Партнеры    О нас    Архив    Эксперты    Ведущие    [Новый JPoint](#)

ДОКЛАД    [Kotlin](#)    14.06 / 18:30 – 19:30 (UTC+3)

## Kotlinx.serialization: готовим свою собственную библиотеку для сериализации

RU   

Презентация [pdf](#)

### Спикеры



Андрей Кулешов

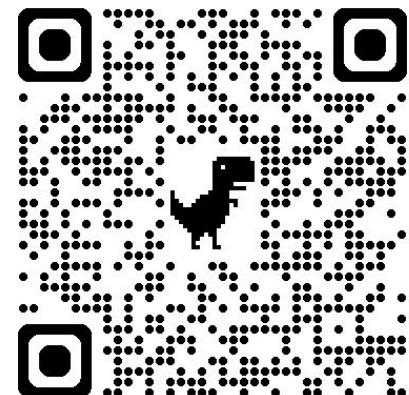
Huawei

Автор расскажет о kotlinx.serialization: как работать с этой библиотекой непосредственному пользователю и создателям сериализаторов, которые будут основываться на этом фреймворке.

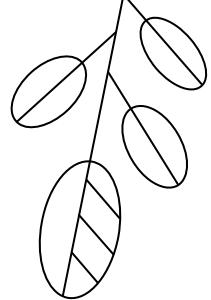
Основываясь на своем опыте разработки мультиплатформенной библиотеки KToml для сериализации формата TOML, автор расскажет о подводных камнях написания собственной open-source-библиотеки для сериализации и десериализации на Kotlin. Речь пойдет об особенностях использования библиотек из kotlinx.serialization в коде, о том, как устроена эта библиотека и как лучше организовать архитектуру своего сериализатора.

Доклад будет интересен Kotlin-программистам разных уровней, как пользователям сериализаторов, так и тем, кто может заинтересоваться созданием своего собственного сериализатора.

#compilerplugin #library



# Сегодня снова не будет 😭



g

@guai9632 1 year ago

могу поспорить, что ты еще забыл про версионирование, циклические графы и секурность

2 Reply



@telephon3208 1 year ago

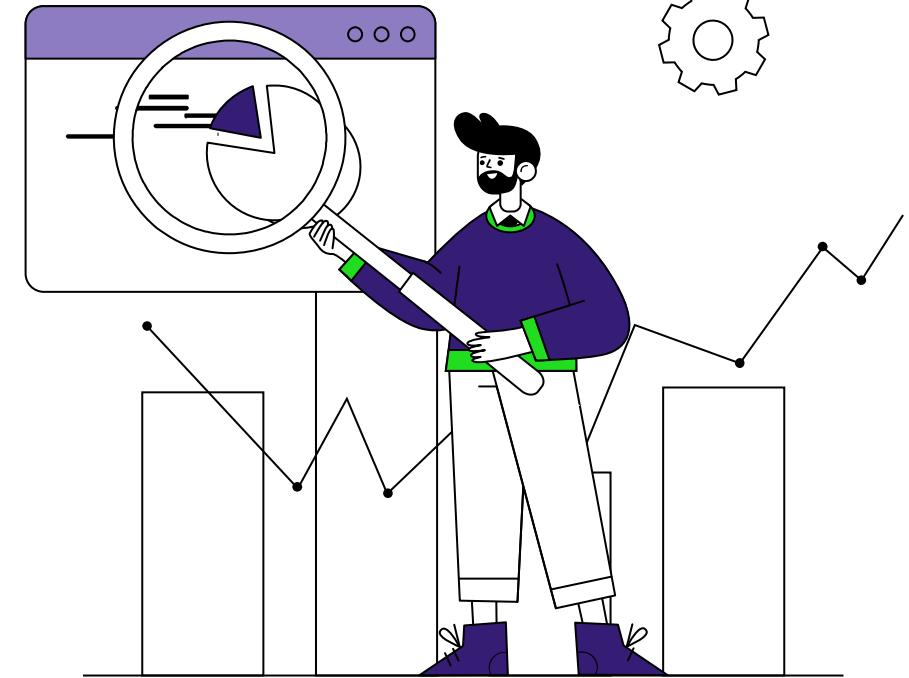
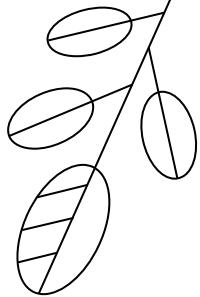
бесценный доклад

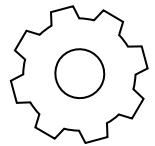
1 Reply



01

# Проблема





# У вас есть данные

Текстовый формат

Бинарный формат





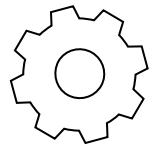
# У данных есть схема

Имена

Типы

Формат



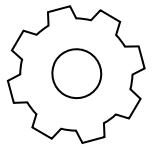


Надо данные разобрать  
и работать с  
**ними** внутри программы

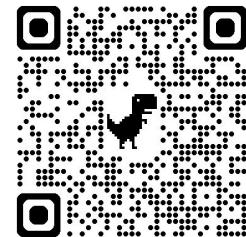




# Схема внутри кода



```
@Serializable  
data class MyClass(  
    val someBooleanProperty: Boolean,  
    val table1: Table1,  
    val table2: Table2,  
    @SerializedName("gradle-libs-like-property")  
    val kotlinJvm: GradlePlugin,  
    val myMap: Map<String, String>  
)
```





# Схема внутри кода

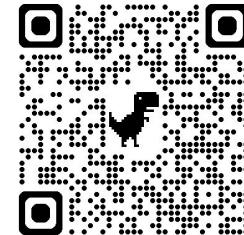


```
@Serializable  
data class MyClass(  
    val someBooleanProperty: Boolean,  
    val table1: Table1,  
    val table2: Table2,  
    @SerializedName("gradle-libs-like-property")  
    val kotlinJvm: GradlePlugin,  
    val myMap: Map<String, String>  
)
```

имена

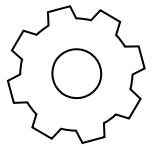
Boolean,

типы





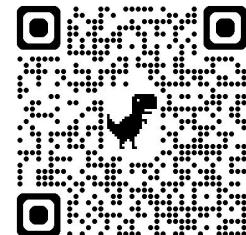
# Схема внутри кода



```
@Serializable  
data class MyClass(  
    val someBooleanProperty: Boolean,  
    val table1: Table1,  
    val table2: Table2,  
    @SerializedName("gradle-libs-like-property")  
    val kotlinJvm: GradlePlugin,  
    val myMap: Map<String, String>  
)
```

имена

типы

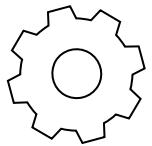


Код как схема данных





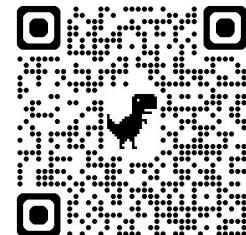
# Схема внутри кода

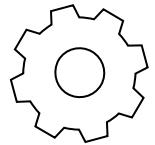


```
@Serializable  
data class MyClass(  
    val someBooleanProperty: Boolean,  
    val table1: Table1,  
    val table2: Table2,  
    @SerialName("gradle-libs-like-property")  
    val kotlinJvm: GradlePlugin,  
    val myMap: Map<String, String>  
)
```

...

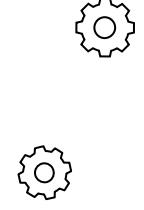
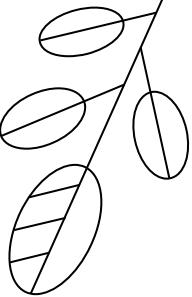
Как это работает?





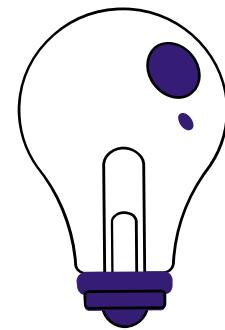
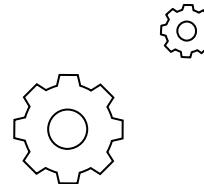
Легко, если есть возможность  
встраиваться в язык

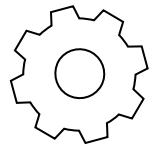




02

# kotlinx.serialization





# Но сначала терминология



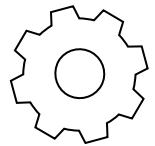


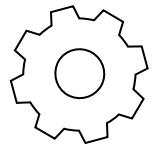
объекты

примитивы

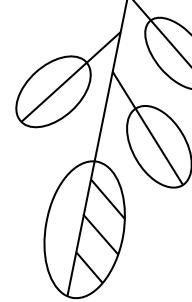
входные/выходные  
форматы







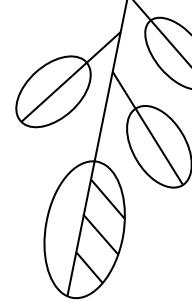
# Вкратце



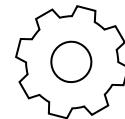
- Мультиплатформенная (**KMP**) библиотека



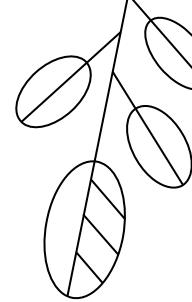
# Вкратце



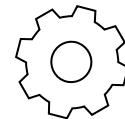
- Мультиплатформенная (КМР) библиотека
- Является **компиляторным плагином**



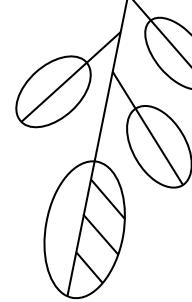
# Вкратце



- Мультиплатформенная (КМР) библиотека
- Является **компиляторным плагином**
- **НЕ** использует reflection



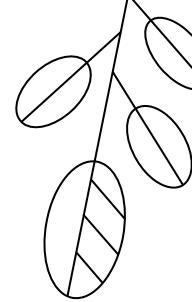
# Вкратце



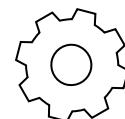
- Мультиплатформенная (КМР) библиотека
- Является **компиляторным плагином**
- **НЕ** использует reflection
- Встраивается в проект с помощью Maven/Gradle плагина



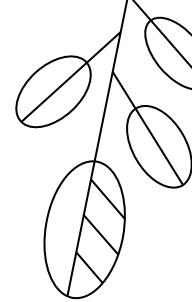
# Вкратце



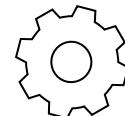
- Мультиплатформенная (КМР) библиотека
- Является **компиляторным плагином**
- **НЕ** использует reflection
- Встраивается в проект с помощью Maven/Gradle плагина
- 5 официальных форматов Json, Protobuf, CBOR, HOCON, Properties



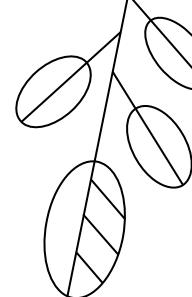
# Вкратце



- Мультиплатформенная (КМР) библиотека
- Является **компиляторным плагином**
- **НЕ** использует reflection
- Встраивается в проект с помощью Maven/Gradle плагина
- 5 официальных форматов Json, Protobuf, CBOR, HOCON, Properties
- 19+ коммьюнити библиотек для других форматов



# Вкратце



- Мультиплатформенная (**KMP**) библиотека
- Является **компиляторным плагином**
- **НЕ** использует reflection
- Встраивается в проект с помощью Maven/Gradle плагина
- 5 официальных форматов Json, Protobuf, CBOR, HOCON, Properties
- 19+ коммьюнити библиотек для других форматов



**sandwwraith** commented on 17 Jan

Member

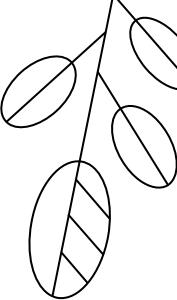


...

We don't have plans to merge more formats in this repository; I think the way it works now — with the list of community-supported formats — is the most optimal.



# Еще более кратко



README Code of conduct Apache-2.0 license

## Kotlin multiplatform / multi-format reflectionless serialization

stable JetBrains official license Apache License 2.0 build passing kotlin 2.0.20 maven-central v1.7.3 API reference KDoc  
chat slack

Kotlin serialization consists of a compiler plugin, that generates visitor code for serializable classes, runtime library with core serialization API and support libraries with various serialization formats.

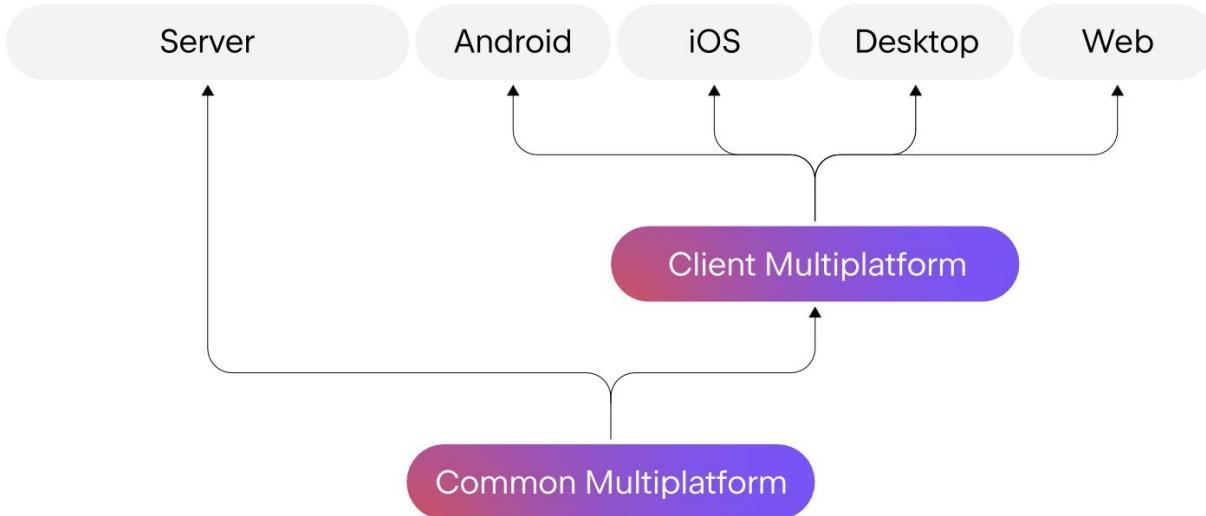
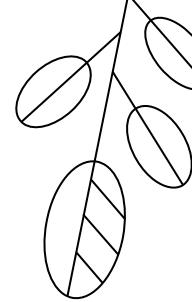
- Supports Kotlin classes marked as `@Serializable` and standard collections.
- Provides [JSON](#), [Protobuf](#), [CBOR](#), [Hocon](#) and [Properties](#) formats.
- Complete multiplatform support: JVM, JS and Native.

### Table of contents

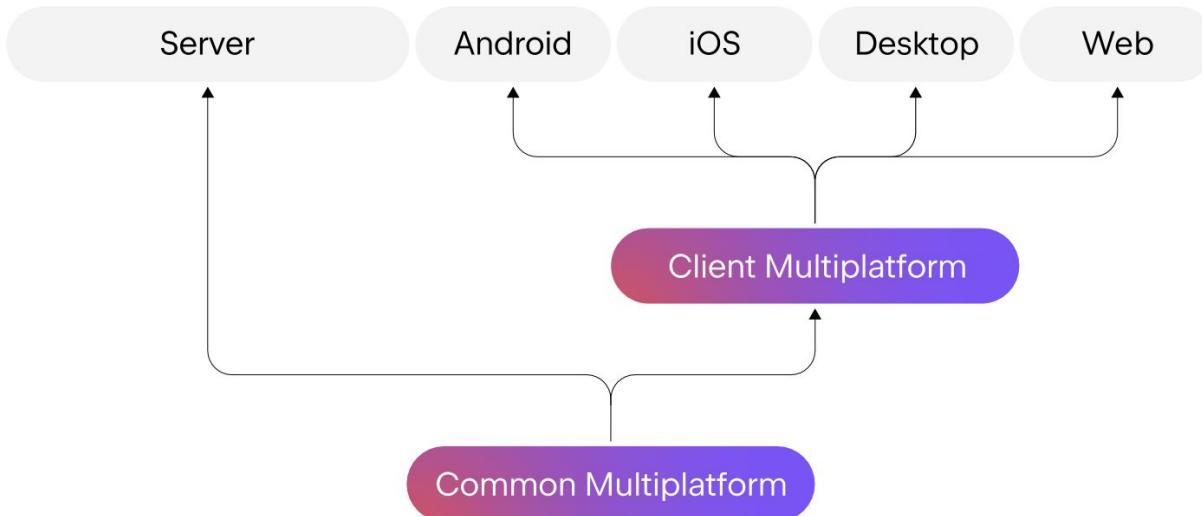
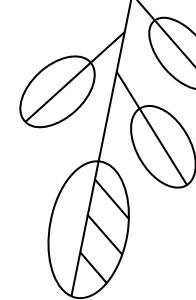
- [Introduction and references](#)
- [Setup](#)
  - [Gradle](#)
    - [1\) Setting up the serialization plugin](#)
    - [2\) Dependency on the JSON library](#)
  - [Android](#)
  - [Multiplatform \(Common, JS, Native\)](#)
  - [Maven](#)
  - [Bazel](#)
- [Additional links](#)
  - [Kotlin Serialization Guide](#)
  - [Full API reference](#)
  - [Submitting issues and PRs](#)



# На всякий случай



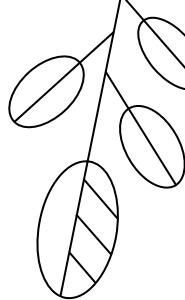
# На всякий случай



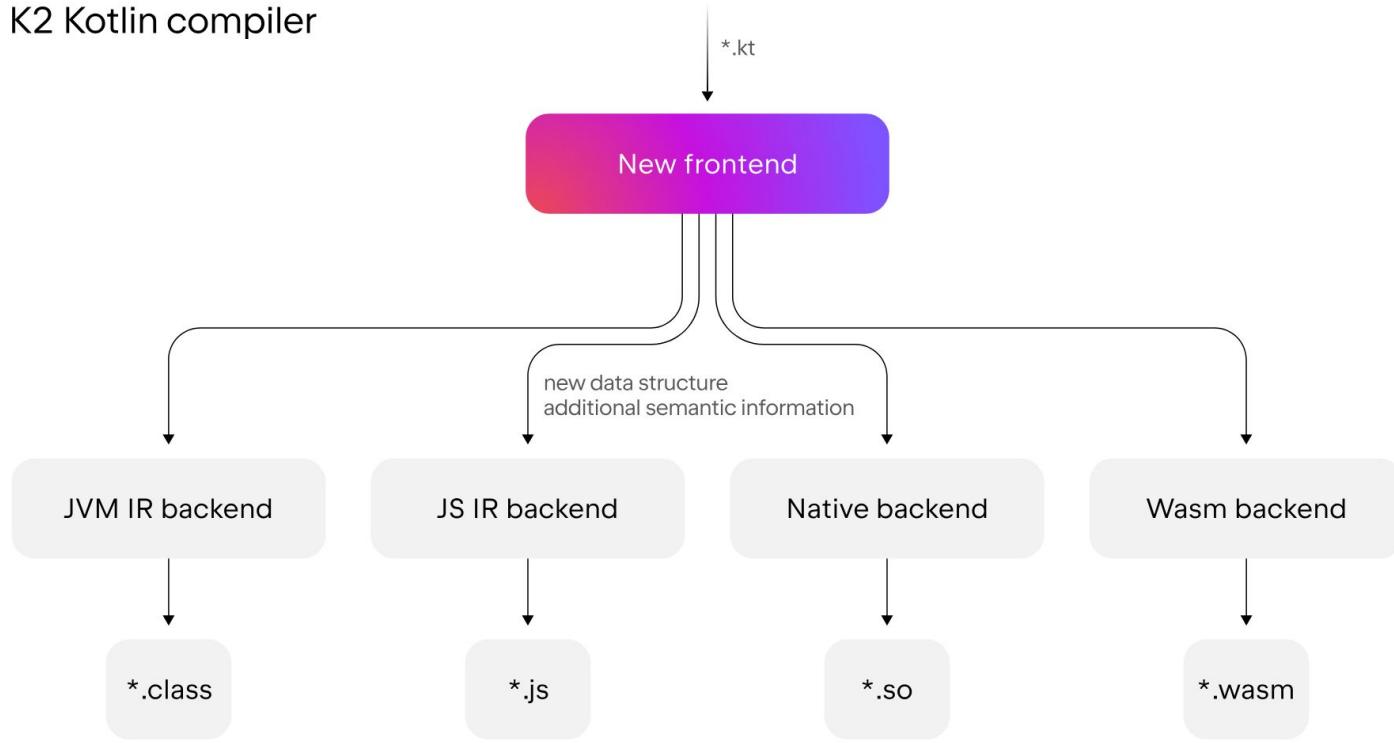
```
ktoml-core
├── build
└── src
    ├── commonMain
    ├── commonTest
    ├── iosMain
    ├── jsMain
    ├── jvmMain
    ├── linuxMain
    ├── macosArm64Main
    ├── macosX64Main
    └── mingwX64Main
build.gradle.kts
```



# Компиляторные плагины

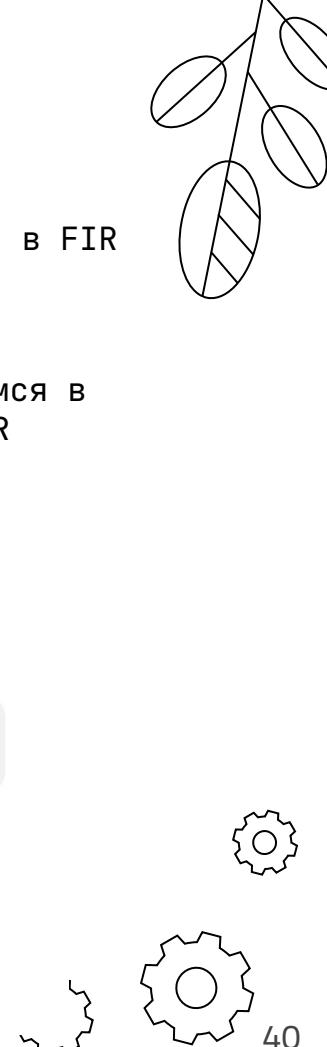
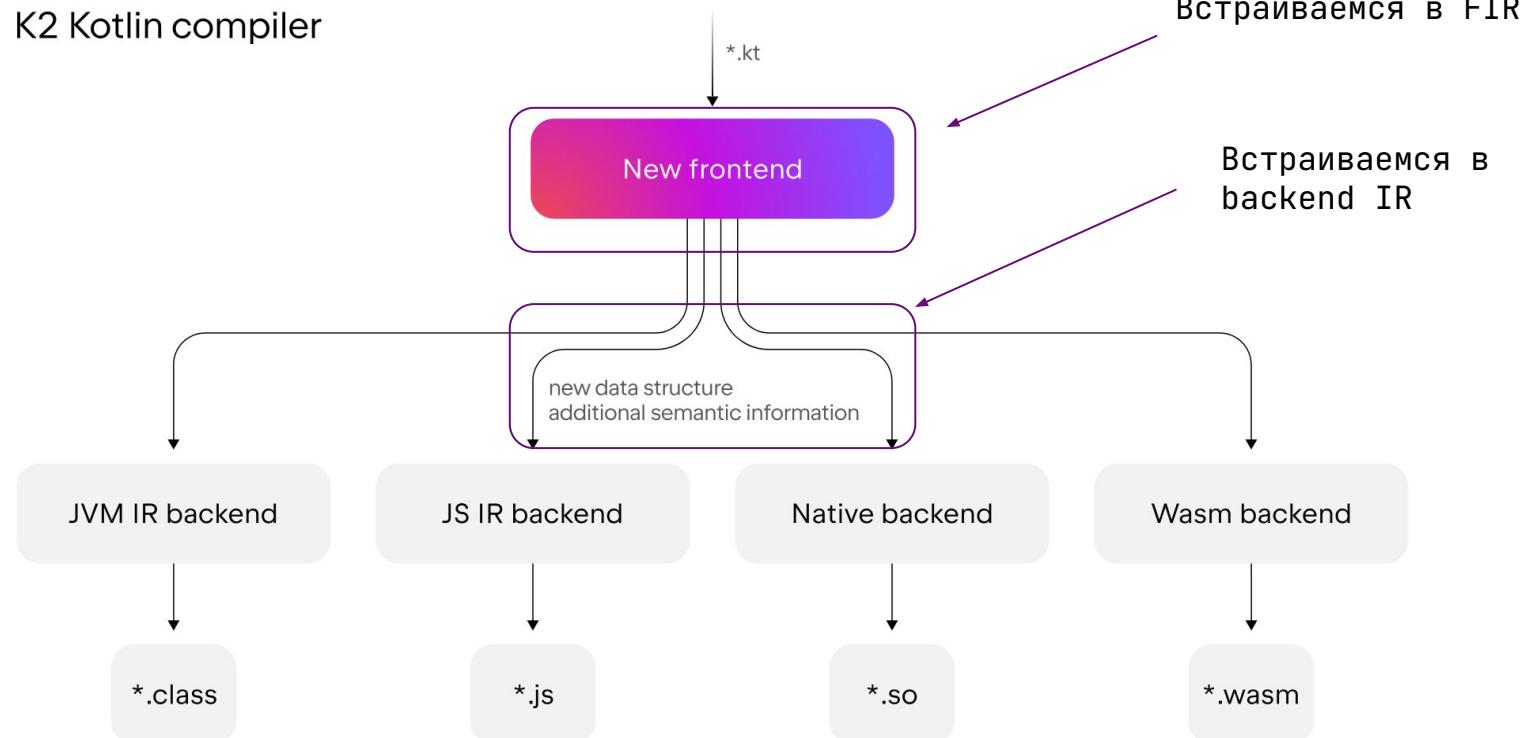


K2 Kotlin compiler

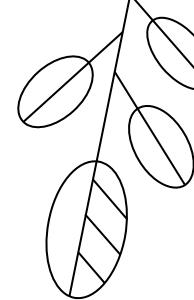


# Компиляторные плагины

K2 Kotlin compiler

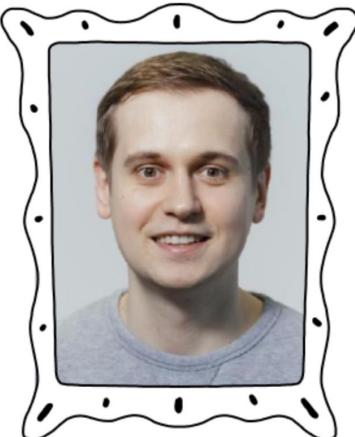


# Компиляторные плагины

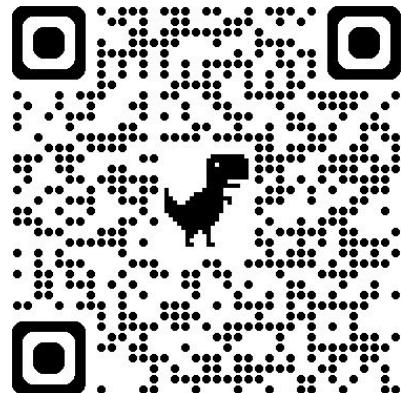
[Купить билет](#)[Спикеры](#)[Как это было в прошлом году](#)

## Компиляторные плагины в Котлин: почувствуй себя системным программистом

Компиляторные плагины – потрясающая фича, которая позволяет нам, обычным разработчикам, почувствовать себя в шкуре системных программистов без зазубривания теории компиляции, педантичной поддержки краевых случаев разных платформ и написания собственных лексеров с парсерами. Вместо этого мы можем сразу работать с абстрактными синтаксическими деревьями (AST) или любым другим промежуточным представлением (IR) и изменять код программ или процесс компиляции под себя.

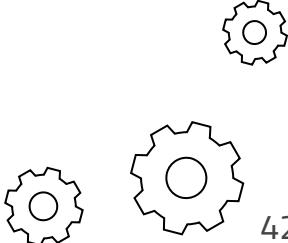
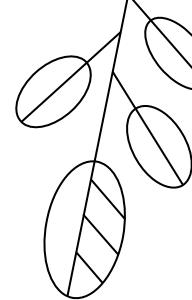


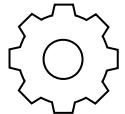
АНДРЕЙ КУЛЕШОВ,  
HUAWEI



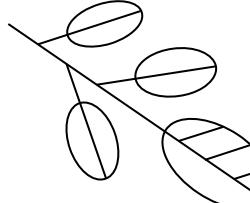
# Компиляторные плагины

а зачем мне эта  
информация





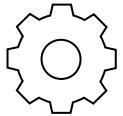
# Пример использования



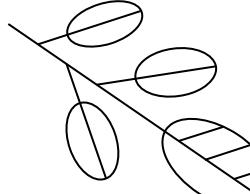
```
package com.akuleshov7
```

```
data class A(  
    val b: Int  
)
```

```
fun main() {  
    val a = A(0)  
}
```



# Пример использования



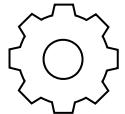
```
package com.akuleshov7
```

```
data class A(  
    val b: Int  
)
```

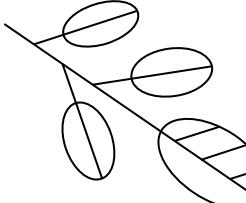
```
fun main() {  
    val a = A(0)  
    serializeToJsonSomehow(a)  
}
```

Дайте мне метод, чтобы хоть  
как-то это сериализовать!





# Пример использования

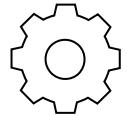


```
package com.akuleshov7
```

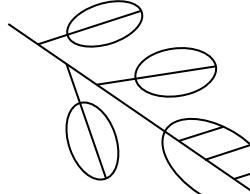
```
data class A(  
    val b: Int  
)
```

```
fun main() {  
    val a = A(0)  
}
```

```
dependencies {  
    implementation("org.jetbrains.kotlinx:kotlinx-serialization-json:1.7.3")  
}
```



# Пример использования



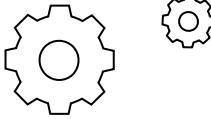
```
package com.akuleshov7

import kotlinx.serialization.encodeToString
import kotlinx.serialization.json.Json

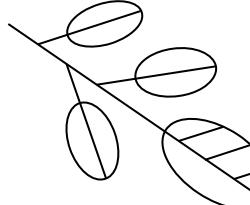
data class A(
    val b: Int
)

fun main() {
    val a = A(0)
    Json.encodeToString(a)
```

Build нормальный



# Ловим ошибку



```
package com.akuleshov7

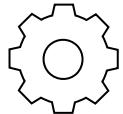
import kotlinx.serialization.encodeToString
import kotlinx.serialization.json.Json

data class A(
    val b: Int
)

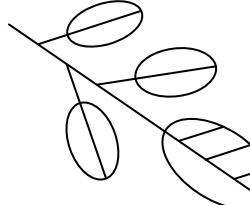
fun main() {
    val a = A(0)
    Json.encodeToString(a)
}
```

Build нормальный, а это на runtime

Exception in thread "main" kotlinx.serialization.SerializationException: **Serializer** for class 'A' is **not** found.



# Добавляем Serializable

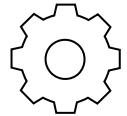


```
package com.akuleshov7

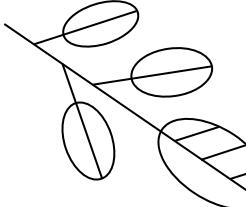
import kotlinx.serialization.Serializable
import kotlinx.serialization.encodeToString
import kotlinx.serialization.json.Json

@Serializable
data class A(
    val b: Int
)

fun main() {
    val a = A(0)
    Json.encodeToString(a)
}
```



# ЛОВИМ ОШИБКУ x2



```
package com.akuleshov7

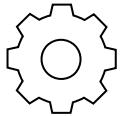
import kotlinx.serialization.Serializable
import kotlinx.serialization.encodeToString
import kotlinx.serialization.json.Json

@Serializable
data class A(
    val b: Int
)

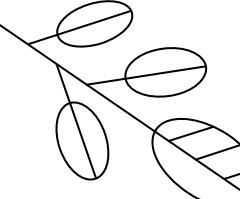
fun main() {
    val a = A(0)
    Json.encodeToString(a)
}
```



Exception in thread "main" kotlinx.serialization.SerializationException: **Serializer** for class 'A' is **not** found.



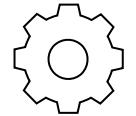
# Приисматриваемся



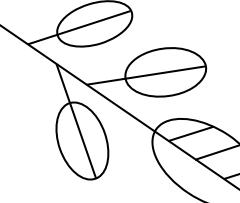
```
1 package com.akuleshov7
2
3 import kotlinx.serialization.Serializable
4 import kotlinx.serialization.encodeToString
5 import kotlinx.serialization.json.Json
6
7 @Serializable
8 data class Test {
9     val test = "test"
10 }
11
12 fun main() {
13     val json = Json.encodeToString(Test())
14 }
15
16 }
```

kotlinx.serialization compiler plugin is not applied to the module, so this annotation would not be processed. Make sure that you've setup your buildscript correctly and re-import project.

The main entry point to the serialization process. Applying `Serializable` to the Kotlin class instructs the serialization plugin to automatically generate implementation of `KSerializer` for the current class, that can be used to serialize and deserialize the class. The generated serializer can be accessed with `T.serializer()` extension function on the class companion, both are generated by the plugin as well.



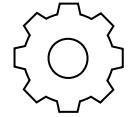
# Приисматриваемся



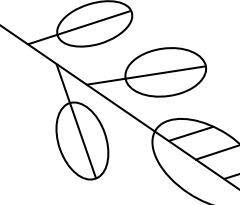
```
1 package com.akuleshov7
2
3 import kotlinx.serialization.Serializable
4 import kotlinx.serialization.encodeToString
5 import kotlinx.serialization.json.Json
6
7 @Serializable
8 data class Test {
9     val test = "test"
10 }
11
12 fun main() {
13     val json = Json.encodeToString(Test())
14 }
15
16 }
```

kotlinx.serialization compiler plugin is not applied to the module, so this annotation would not be processed. Make sure that you've setup your buildscript correctly and re-import project.

The main entry point to the serialization process. Applying `Serializable` to the Kotlin class instructs the serialization plugin to automatically generate implementation of `KSerializer` for the current class, that can be used to serialize and deserialize the class. The generated serializer can be accessed with `T.serializer()` extension function on the class companion, both are generated by the plugin as well.



# Присматриваемся



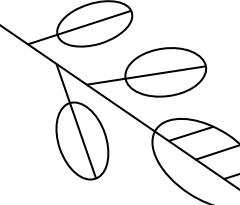
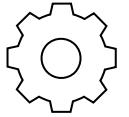
```
1 package com.akuleshov7
2
3 import kotlinx.serialization.Serializable
4 import kotlinx.serialization.encodeToString
5 import kotlinx.serialization.json.Json
6
7 @Serializable
8 data class Test {
9     val test = "test"
10 }
11
12 fun main() {
13     val json = Json.encodeToString(Test())
14     println(json)
15 }
16
```

kotlinx.serialization compiler plugin is not applied to the module, so this annotation would not be processed. Make sure that you've setup your buildscript correctly and re-import project.

public constructor Serializable(  
 val with: KClass<out KSerializer<\*>> = KSerializer::class)

А тут можно подложить кастомизацию, кстати

The main entry point to the serialization process. Applying `Serializable` to the Kotlin class instructs the serialization plugin to automatically generate implementation of `KSerializer` for the current class, that can be used to serialize and deserialize the class. The generated serializer can be accessed with `T.serializer()` extension function on the class companion, both are generated by the plugin as well.

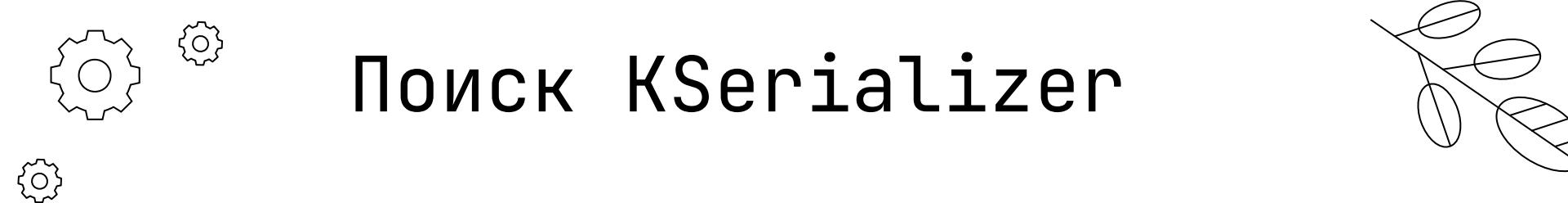


# KSerializer

Как обнаруживается?

Зачем он вообще нужен?

Как генерируется?



# Поиск KSerializer

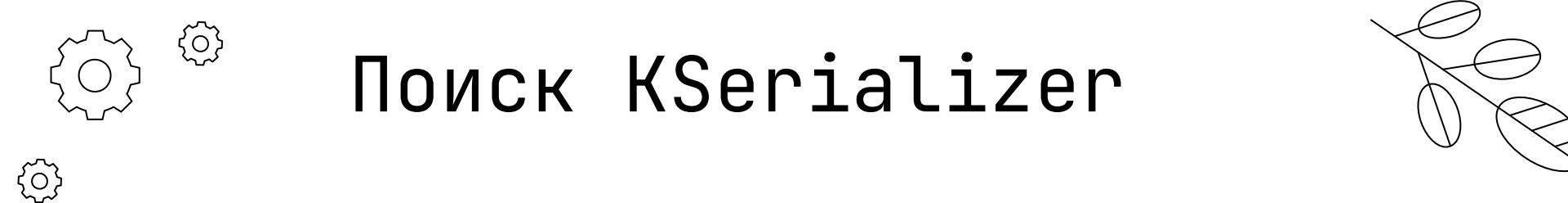
```
public fun SerializersModule.serializer(type: KType): KSerializer<Any?> =  
    serializerByKTypeImpl(type, failOnMissingTypeArgSerializer = true) ?: type.kclass()  
        .platformSpecificSerializerNotRegistered()
```



# Поиск KSerializer

```
public fun SerializersModule.serializer(type: KType): KSerializer<Any?> =  
    serializerByKTypeImpl(type, failOnMissingTypeArgSerializer = true) ?: type.kclass()  
        .platformSpecificSerializerNotRegistered()
```

< ... в конечном итоге приходим к ...>

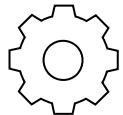


# Поиск KSerializer

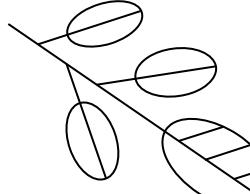
```
public fun SerializersModule.serializer(type: KType): KSerializer<Any?> =  
    serializerByKTypeImpl(type, failOnMissingTypeArgSerializer = true) ?: type.kclass()  
        .platformSpecificSerializerNotRegistered()
```

< ... в конечном итоге приходим к ... >

```
@InternalSerializationApi  
public fun <T : Any> KClass<T>.serializerOrNull(): KSerializer<T>? =  
    compiledSerializerImpl() ?: builtinSerializerOrNull()
```

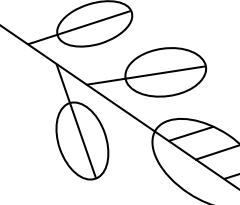
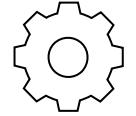


# Встроенные KSerializer



```
@OptIn(ExperimentalSerializationApi::class)
internal actual fun initBuiltins(): Map<KClass<*>, KSerializer<*>> =
    buildMap {
        // Standard classes are always present
        put(String::class, String.serializer())
        put(Char::class, Char.serializer())
        putCharArray::class, CharArraySerializer())
        put(Double::class, Double.serializer())
        put(DoubleArray::class, DoubleArraySerializer())
        put(Float::class, Float.serializer())
        put(FloatArray::class, FloatArraySerializer())
    }
```

<...>



На пальцах пример  
для *String*



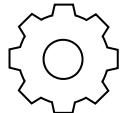
# Пример для String

```
@PublishedApi
internal object StringSerializer : KSerializer<String> {
    override val descriptor: SerialDescriptor =
        PrimitiveSerialDescriptor("kotlin.String", PrimitiveKind.STRING)

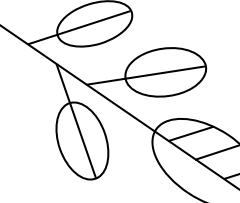
    override fun serialize(encoder: Encoder, value: String): Unit =
        encoder.encodeString(value)

    override fun deserialize(decoder: Decoder): String =
        decoder.decodeString()
}
```

[core/commonMain/src/kotlinx/serialization/Serializers.kt](#)



# Пример для String

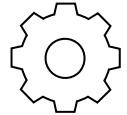


Метаинформация (схема класса итд.)

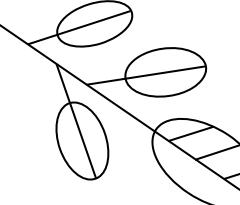
```
@PublishedApi
internal object StringSerializer : KSerializer<String> {
    override val descriptor: SerialDescriptor =
        PrimitiveSerialDescriptor("kotlin.String", PrimitiveKind.STRING)

    override fun serialize(encoder: Encoder, value: String): Unit =
        encoder.encodeString(value)

    override fun deserialize(decoder: Decoder): String =
        decoder.decodeString()
}
```



# Пример для String



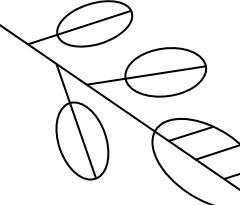
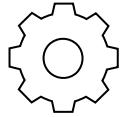
Логика сериализации для конкретного формата/библиотеки

```
@PublishedApi
internal object StringSerializer : KSerializer<String> {
    override val descriptor: SerialDescriptor =
        PrimitiveSerialDescriptor("kotlin.String", PrimitiveKind.STRING)

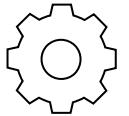
    override fun serialize(encoder: Encoder, value: String): Unit =
        encoder.encodeString(value)

    override fun deserialize(decoder: Decoder): String =
        decoder.decodeString()
}
```

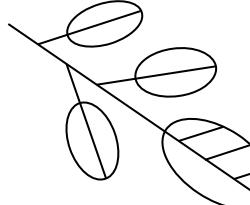
[core/commonMain/src/kotlinx/serialization/Serializers.kt](#)



# Пора добавить плагин



# Включаем плагин



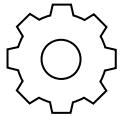
```
package com.akuleshov7

import
kotlinx.serialization.Serializable
import
kotlinx.serialization.encodeToString
import kotlinx.serialization.json.Json

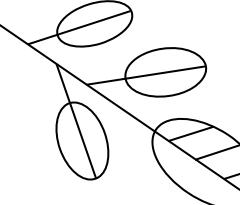
@Serializable
data class A(
    val b: Int
)

fun main() {
    val a = A(0)
    println(A.serializer())
    println(Json.encodeToString(a))
}
```

```
plugins {
    kotlin("plugin.serialization") version "2.0.20"
}
```



# Все работает



```
package com.akuleshov7

import
kotlinx.serialization.Serializable
import
kotlinx.serialization.encodeToString
import kotlinx.serialization.json.Json

@Serializable
data class A(
    val b: Int
)

fun main() {
    val a = A(0)
    println(A.serializer())           com.akuleshov7.A$$serializer@32d992b2
    println(Json.encodeToString(a))   {"b":0}
}
```

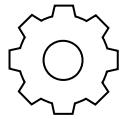
ПОЭТому НУЖНО ПОНИМАТЬ  
ЧТО ПРОИСХОДИТ ПОД  
КАПОТОМ



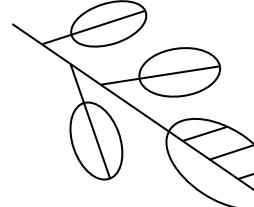


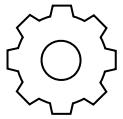
Посмотрим на  
сгенерированный код



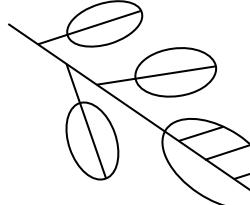


# Как посмотреть самому?





# Как посмотреть самому?



The screenshot shows the IntelliJ IDEA interface with the following details:

- Project View:** Shows the project structure for "TestKotlinSerialization". A purple rounded rectangle highlights the "build/classes/kotlin/main/com/akuleshov7" directory, which contains files "A" and "MainKt.class".
- Code Editor:** The file "A.class" is open, showing its decompiled Java code. The code is as follows:

```
// IntelliJ API Decompiler stub source generated from a class file
// Implementation of methods is not available

package com.akuleshov7

@kotlinx.serialization.Serializable public final data class A public constructor(b: kotlin.Int) {
    public companion object {
        public final fun serializer(): kotlinx.serialization.KSerializer<com.akuleshov7.A> { /* compiled code */ }
    }

    internal constructor(seen0: kotlin.Int, b: kotlin.Int, serializationConstructorMarker: kotlinx.serialization.internal.SerializationConstructorMarker) : this(b)

    public final val b: kotlin.Int /* compiled code */

    public final operator fun component1(): kotlin.Int { /* compiled code */ }

    public open operator fun equals(other: kotlin.Any?): kotlin.Boolean { /* compiled code */ }

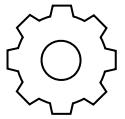
    public open fun hashCode(): kotlin.Int { /* compiled code */ }

    public open fun toString(): kotlin.String { /* compiled code */ }

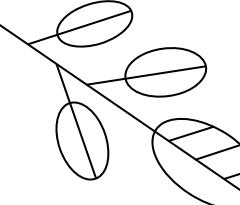
    @kotlin.jvm.JvmStatic internal final fun `write$Self`(self: com.akuleshov7.A, output: kotlinx.serialization.encoding.CompositeEncoder) { /* compiled code */ }

    @kotlin.Deprecated public object `$serializer` : kotlinx.serialization.internal.GeneratedSerializer<com.akuleshov7.A> {
        public final val descriptor: kotlinx.serialization.descriptors.SerialDescriptor /* compiled code */
    }
}
```

At the bottom of the interface, it says "BUILD SUCCESSFUL in 1s".



# Как посмотреть самому?

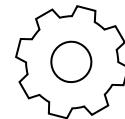
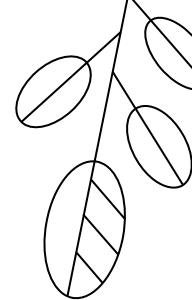


The screenshot shows the IntelliJ IDEA interface with the following details:

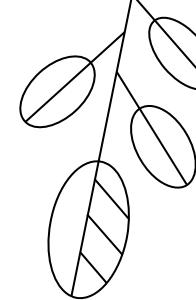
- Project View:** The left sidebar displays the project structure under "TestKotlinSerialization". It includes a "build" folder containing "classes" (with "kotlin" and "main" subfolders) and "META-INF". Other folders like ".idea", ".kotlin", "libs", "tmp", "gradle", "src", "test", ".gitignore", and "build.gradle.kts" are also visible.
- Tools Bar:** The top navigation bar has "Tools" selected, which is currently open as a dropdown menu.
- Code Editor:** The main right pane shows Java code for a class named "A". The code is annotated with comments indicating it is a compiler stub source generated from a class file. A tooltip for the "decompile" action is visible over the code.
- Toolbars and Status:** The top right features standard OS X-style controls (minimize, maximize, close) and a status bar indicating the build was successful ("BUILD SUCCESSFUL in 1s").

# Что будет сгенерировано?

```
public class $serializer implements GeneratedSerializer
```



# Что будет сгенерировано?



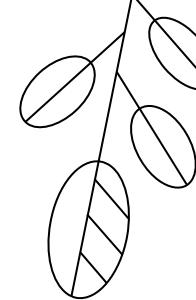
```
public class $serializer implements GeneratedSerializer
```



```
/**  
 * An interface for a [KSerializer] instance generated by the compiler plugin.  
 *  
 * Should not be implemented manually or used directly.  
 */  
@InternalSerializationApi  
public interface GeneratedSerializer<T> : KSerializer<T> {  
    public fun childSerializers(): Array<KSerializer<*>>  
    public fun typeParametersSerializers(): Array<KSerializer<*>>  
}
```



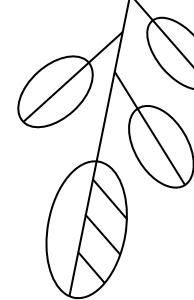
# Сериализатор



```
public class $serializer implements GeneratedSerializer {  
  
    public final void serialize(@NotNull Encoder encoder, @NotNull A value) {  
        Intrinsics.checkNotNullParameter(encoder, "encoder");  
        Intrinsics.checkNotNullParameter(value, "value");  
        SerialDescriptor var3 = descriptor;  
        CompositeEncoder var4 = encoder.beginStructure(var3);  
        A.write$Self$TestKotlinSerialization(value, var4, var3);  
        var4.endStructure(var3);  
    }  
}
```



# Сериализатор

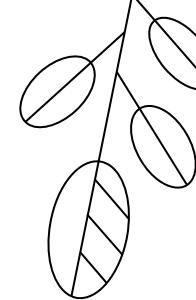


```
public class $serializer implements GeneratedSerializer {  
  
    public final void serialize(@NotNull Encoder encoder, @NotNull A value) {  
        Intrinsics.checkNotNullParameter(encoder, "encoder");  
        Intrinsics.checkNotNullParameter(value, "value");  
        SerialDescriptor var3 = descriptor;  
        CompositeEncoder var4 = encoder.beginStructure(var3);  
        A.write$Self$TestKotlinSerialization(value, var4, var3);  
        var4.endStructure(var3);  
    }  
}
```

“Движок”  
конкретного формата



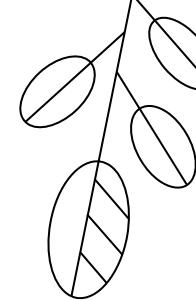
# Сериализатор



```
public class $serializer implements GeneratedSerializer {  
  
    public final void serialize(@NotNull Encoder encoder, @NotNull A value) {  
        Intrinsics.checkNotNullParameter(encoder, "encoder");  
        Intrinsics.checkNotNullParameter(value, "value");  
        SerialDescriptor var3 = descriptor;  
        CompositeEncoder var4 = encoder.beginStructure(var3);  
        A.write$Self$TestKotlinSerialization(value, var4, var3);  
        var4.endStructure(var3);  
    }  
}
```



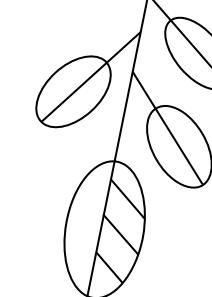
# Сериализатор



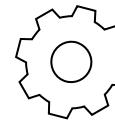
```
public class $serializer implements GeneratedSerializer {  
  
    public final void serialize(@NotNull Encoder encoder, @NotNull A value) {  
        Intrinsics.checkNotNullParameter(encoder, "encoder");  
        Intrinsics.checkNotNullParameter(value, "value");  
        SerialDescriptor var3 = descriptor;  
        CompositeEncoder var4 = encoder.beginStructure(var3);  
        A.write$Self$TestKotlinSerialization(value, var4, var3);  
        var4.endStructure(var3);  
    }  
}
```



# Сериализатор

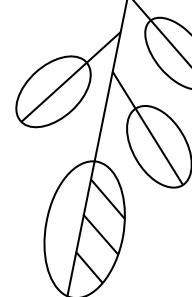


```
public class $serializer implements GeneratedSerializer {  
  
    public final void serialize(@NotNull Encoder encoder, @NotNull A value) {  
        Intrinsics.checkNotNullParameter(encoder, "encoder");  
        Intrinsics.checkNotNullParameter(value, "value");  
        SerialDescriptor var3 = descriptor;  
        CompositeEncoder var4 = encoder.beginStructure(var3);  
        A.write$Self$TestKotlinSerialization(value, var4, var3);  
        var4.endStructure(var3);  
    }  
  
    @JvmStatic  
    public static final void write$Self$TestKotlinSerialization(  
        A self, CompositeEncoder output, SerialDescriptor serialDesc  
    ) {  
        output.encodeIntElement(serialDesc, 0, self.b);  
    }  
}
```



# Десериализатор

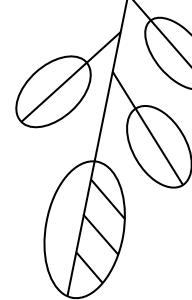
```
public class $serializer implements GeneratedSerializer {  
  
    @NotNull  
    public final A deserialize(@NotNull Decoder decoder) {  
        Intrinsics.checkNotNullParameter(decoder, "decoder");  
        SerialDescriptor var2 = descriptor;  
        boolean var3 = true;  
        int var5 = 0;  
        int var6 = 0;  
        CompositeDecoder var7 = decoder.beginStructure(var2);  
        ...  
    }  
}
```



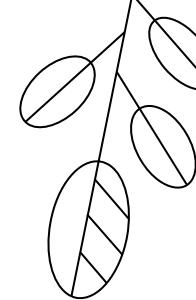
# Десериализатор

```
public class $serializer implements GeneratedSerializer {  
  
    @NotNull  
    public final A deserialize(@NotNull Decoder decoder) {  
        Intrinsics.checkNotNullParameter(decoder, "decoder");  
        SerialDescriptor var2 = descriptor;  
        boolean var3 = true;  
        int var5 = 0;  
        int var6 = 0;  
        CompositeDecoder var7 = decoder.beginStructure(var2);  
        ...  
    }  
}
```

“Движок”  
конкретного формата



# Десериализатор



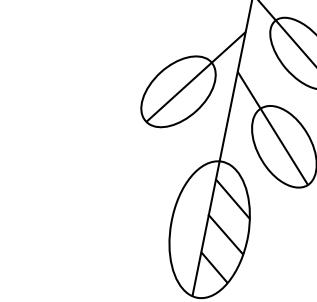
```
public class $serializer implements GeneratedSerializer {  
  
    @NotNull  
    public final A deserialize(@NotNull Decoder decoder) {  
        Intrinsics.checkNotNullParameter(decoder, "decoder");  
        SerialDescriptor var2 = descriptor;  
        boolean var3 = true;  
        int var5 = 0;  
        int var6 = 0;  
        CompositeDecoder var7 = decoder.beginStructure(var2);  
        ...  
    }  
}
```

Начинаем обрабатывать структуру



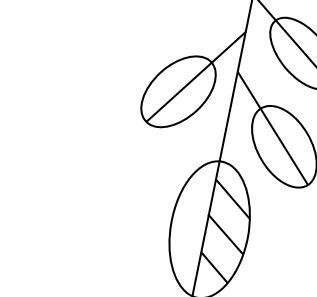
# Десериализатор

```
public class $serializer implements GeneratedSerializer {  
  
    @NotNull  
    public final A deserialize(@NotNull Decoder decoder) {  
        ...  
        while(var3) {  
            int var4 = var7.decodeElementIndex(var2);  
            switch (var4) {  
                case -1:  
                    var3 = false;  
                    break;  
                case 0:  
                    var6 = var7.decodeIntElement(var2, 0);  
                    var5 |= 1;  
                    break;  
                default:  
                    throw new UnknownFieldException(var4);  
            }  
        }  
        ...  
    }  
}
```



# Десериализатор

```
public class $serializer implements GeneratedSerializer {  
  
    @NotNull  
    public final A deserialize(@NotNull Decoder decoder) {  
        ...  
        while(var3) {  
            int var4 = var7.decodeElementIndex(var2);  
            switch (var4) {  
                case -1:  
                    var3 = false;  
                    break;  
                case 0:  
                    var6 = var7.decodeIntElement(var2, 0);  
                    var5 += 1;  
                    break;  
                default:  
                    throw new UnknownFieldException(var4);  
            }  
        }  
        ...  
    }  
}
```

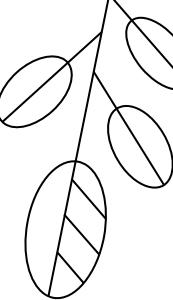


# Десериализатор

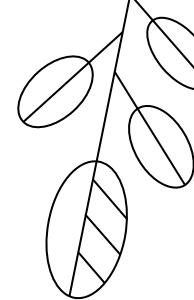
```
public class $serializer implements GeneratedSerializer {
```

```
@NotNull
public final A deserialize(@NotNull Decoder decoder) {
    ...
    while(var3) {
        int var4 = var7.decodeElementIndex(var2);
        switch (var4) {
            case -1:
                var3 = false;
                break;
            case 0:
                var6 = var7.decodeIntElement(var2, 0);
                var5 |= 1;
                break;
            default:
                throw new UnknownFieldException(var4);
        }
    }
    ...
}
```

NAME	INDEX
b	0



# Десериализатор

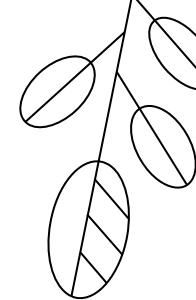


```
public class $serializer implements GeneratedSerializer {  
  
    @NotNull  
    public final A deserialize(@NotNull Decoder decoder) {  
        ...  
        while(var3) {  
            int var4 = var7.decodeElementIndex(var2); ← вытаскивается из дескриптора, который содержит эту метадату  
            switch (var4) {  
                case -1:  
                    var3 = false;  
                    break;  
                case 0:  
                    var6 = var7.decodeIntElement(var2, 0);  
                    var5 += 1;  
                    break;  
                default:  
                    throw new UnknownFieldException(var4);  
            }  
        }  
        ...  
    }  
}
```

вытаскивается из дескриптора, который содержит эту метадату

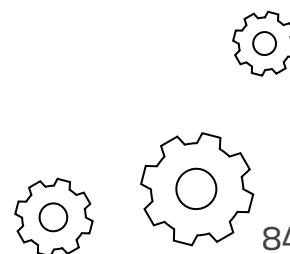


# Десериализатор

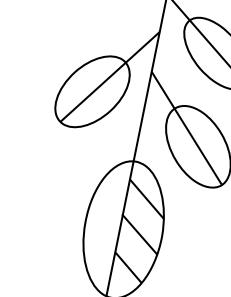


```
public class $serializer implements GeneratedSerializer {  
  
    @NotNull  
    public final A deserialize(@NotNull Decoder decoder) {  
        ...  
        while(var3) {  
            int var4 = var7.decodeElementIndex(var2);  
            switch (var4) {  
                case -1:  
                    var3 = false;  
                    break;  
                case 0:  
                    var6 = var7.decodeIntElement(var2, 0);  
                    var5 |= 1;  
                    break;  
                default:  
                    throw new UnknownFieldException(var4);  
            }  
        }  
        ...  
    }  
}
```

Разбираем наше поле **b** как **Int**



# Десериализатор

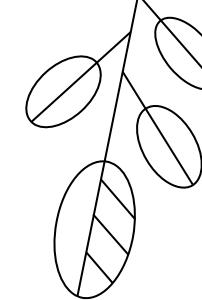


```
public class $serializer implements GeneratedSerializer {  
  
    @NotNull  
    public final A deserialize(@NotNull Decoder decoder) {  
        ...  
        while(var3) {  
            int var4 = var7.decodeElementIndex(var2);  
            switch (var4) {  
                case -1:  
                    var3 = false;  
                    break;  
                case 0:  
                    var6 = var7.decodeIntElement(var2, 0);  
                    var5 |= 1;  
                    break;  
                default:  
                    throw new UnknownFieldException(var4);  
            }  
        }  
        ...  
    }  
}
```

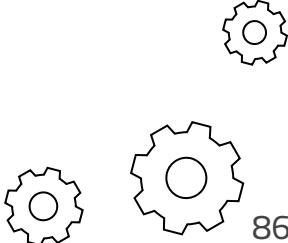
DECODE\_DONE → break@loop



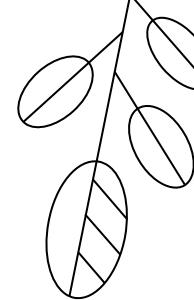
# Десериализатор



```
public class $serializer implements GeneratedSerializer {  
  
    @NotNull  
    public final A deserialize(@NotNull Decoder decoder) {  
        ...  
        while(var3) {  
            int var4 = var7.decodeElementIndex(var2);  
            switch (var4) {  
                case -1:  
                    var3 = false;  
                    break;  
                case 0:  
                    var6 = var7.decodeIntElement(var2, 0);      Не знаем индекс, мусор  
                    var5 |= 1;  
                    break;  
                default:  
                    throw new UnknownFieldException(var4);  
            }  
        }  
        ...  
    }  
}
```



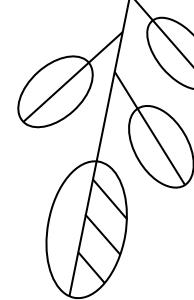
# Десериализатор



```
public class $serializer implements GeneratedSerializer {  
  
    @NotNull  
    public final A deserialize(@NotNull Decoder decoder) {  
  
        ...  
  
        var7.endStructure(var2); ← Обработали структуру, все хорошо  
  
        return new A(var5, var6, (SerializationConstructorMarker)null);  
    }  
}
```



# Десериализатор

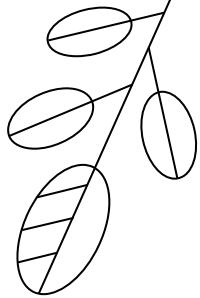


```
public class $serializer implements GeneratedSerializer {  
  
    @NotNull  
    public final A deserialize(@NotNull Decoder decoder) {  
  
        ...  
  
        var7.endStructure(var2);  
  
        return new A(var5, var6, (SerializationConstructorMarker)null);  
    }  
}
```



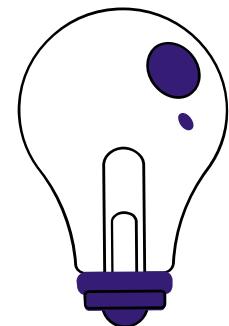
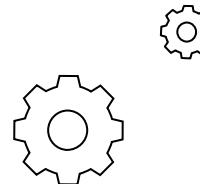
Хитрый синтетический конструктор для нашего класса, чтобы проверить то, что поля не забыты/потеряны. И присвоить значения



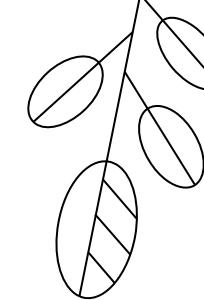


03

## Склейваем вместе



# Цепочки вызовов

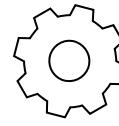


```
package com.akuleshov7

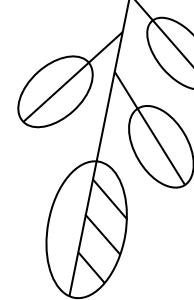
import kotlinx.serialization.decodeFromString
import kotlinx.serialization.json.Json

@Serializable
data class A(
    val b: Int
)

fun main() {
    val a = A(0)
    Json.decodeFromString(a)
}
```



# Цепочки вызовов



```
package com.akuleshov7

import kotlinx.serialization.decodeFromString
import kotlinx.serialization.json.Json

@Serializable
data class A(
    val b: Int
)

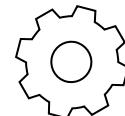
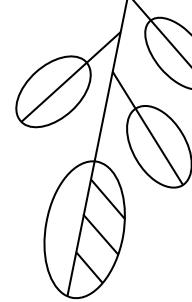
fun main() {
    val a = A(0)
    Json.decodeFromString(a)
}
```

Ответственные мы,  
обычные девелоперы

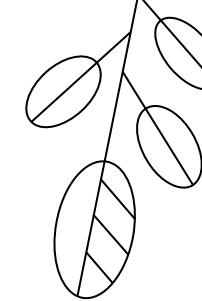


# Цепочки вызовов

`Json.decodeFromString(a)`



# Цепочки вызовов

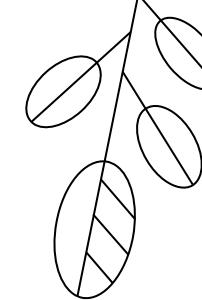


`Json.decodeFromString(a)`

```
decodeFromString(  
    serializersModule.serializer(),  
    string  
)
```



# Цепочки вызовов

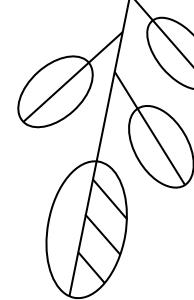


`Json.decodeFromString(a)`

```
decodeFromString(  
    serializersModule.serializer(),  
    string  
)
```



# Цепочки вызовов



`Json.decodeFromString(a)`

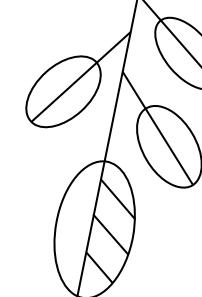
```
decodeFromString(  
    serializersModule.serializer(),  
    string  
)
```



Помните, как мы  
искали  
сгенерированный по  
`@Serializable`  
аннотации код?



# Цепочки вызовов



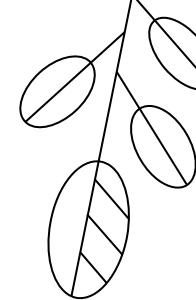
`Json.decodeFromString(a)`

```
decodeFromString(  
    serializersModule.serializer(),  
    string  
)
```

Ответственный компилятор и  
авторы компиляторных плагинов



# Цепочки вызовов



`Json.decodeFromString(a)`

```
decodeFromString(  
    serializersModule.serializer(),  
    string  
)
```

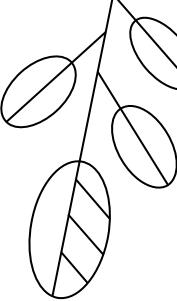
`KSerializer`: сгенерированный алгоритм того, как обходить схему и мапить примитивы на метод декодинга

Ответственный компилятор и авторы компиляторных плагинов



# Цепочки вызовов

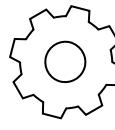
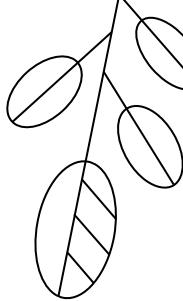
```
class Format : StringFormat {  
    public final override fun <T> decodeFromString(  
        deserializer: DeserializationStrategy<T>, // KSerializer  
        string: String  
    ): T {  
    }  
}
```



# Цепочки вызовов

```
class Format : StringFormat {  
    public final override fun <T> decodeFromString(  
        deserializer: DeserializationStrategy<T>, // KSerializer  
        string: String  
    ): T {  
    }  
}
```

Верхнеуровневый API, чтобы  
соответствовать kotlinx интерфейсам



# Цепочки вызовов

```
class Format : StringFormat {  
    public final override fun <T> decodeFromString(  
        deserializer: DeserializationStrategy<T>, // KSerializer  
        string: String  
    ): T {  
    }  
}
```

Верхнеуровневый API, чтобы  
соответствовать kotlinx интерфейсам

Json.decodeFromString()  
Toml.decodeFromString()  
Yaml.decodeFromString()



# Цепочки вызовов

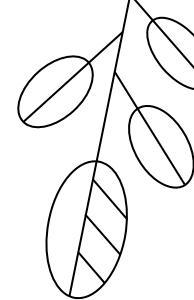
```
class Format : StringFormat {  
    public final override fun <T> decodeFromString(  
        deserializer: DeserializationStrategy<T>, // KSerializer  
        string: String  
    ): T {  
    }  
}
```

Верхнеуровневый API, чтобы  
соответствовать kotlinx интерфейсам

Ответственный автор библиотеки  
для конкретного формата



# Цепочки вызовов



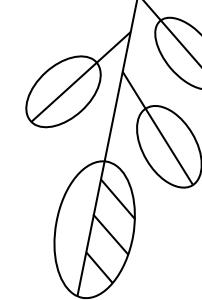
```
class Format : StringFormat {  
    public final override fun <T> decodeFromString(  
        deserializer: DeserializationStrategy<T>, // KSerializer  
        string: String  
    ): T {  
        val myDecoder = // decoder для нашего конкретного формата  
    }  
}
```



Движок, в котором имплементирована  
работа с конкретным форматом и  
конкретными примитивами



# Цепочки вызовов



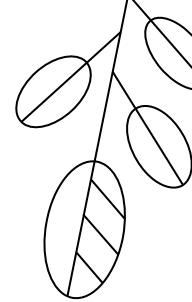
```
class Format : StringFormat {  
    public final override fun <T> decodeFromString(  
        deserializer: DeserializationStrategy<T>, // KSerializer  
        string: String  
    ): T {  
        val myDecoder = // decoder для нашего конкретного формата  
        val result = myDecoder.decodeSerializableValue(deserializer)  
        return result  
    }  
}
```



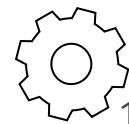
Внутри вызов того самого  
deserializer.deserialize(),



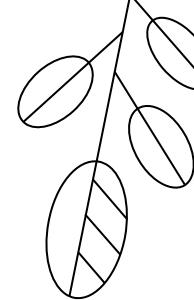
# Цепочки вызовов



```
class MyDecoder: Decoder {  
}
```



# Цепочки вызовов

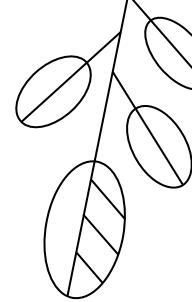


```
class MyDecoder: Decoder {  
    public fun decodeInt(): Int  
    public fun decodeString(): String  
  
    ...  
    public fun beginStructure(descriptor: SerialDescriptor): CompositeDecoder  
  
    ...  
    public fun decodeEnum(enumDescriptor: SerialDescriptor): Int  
}
```

Ответственный автор библиотеки  
для конкретного формата



# Цепочки вызовов



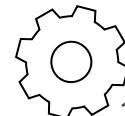
Мы

Не забываем  
@Serializable

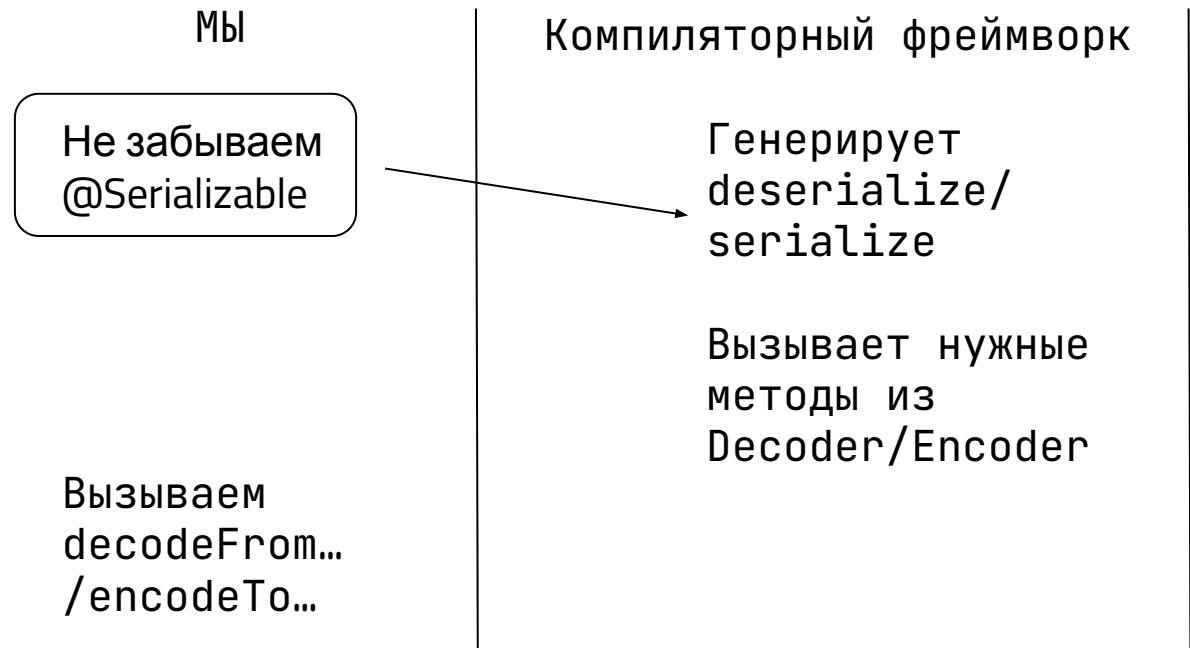
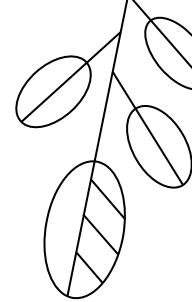
Вызываем  
decodeFrom...  
/encodeTo...

Компиляторный фреймворк

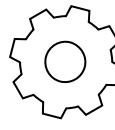
Либы



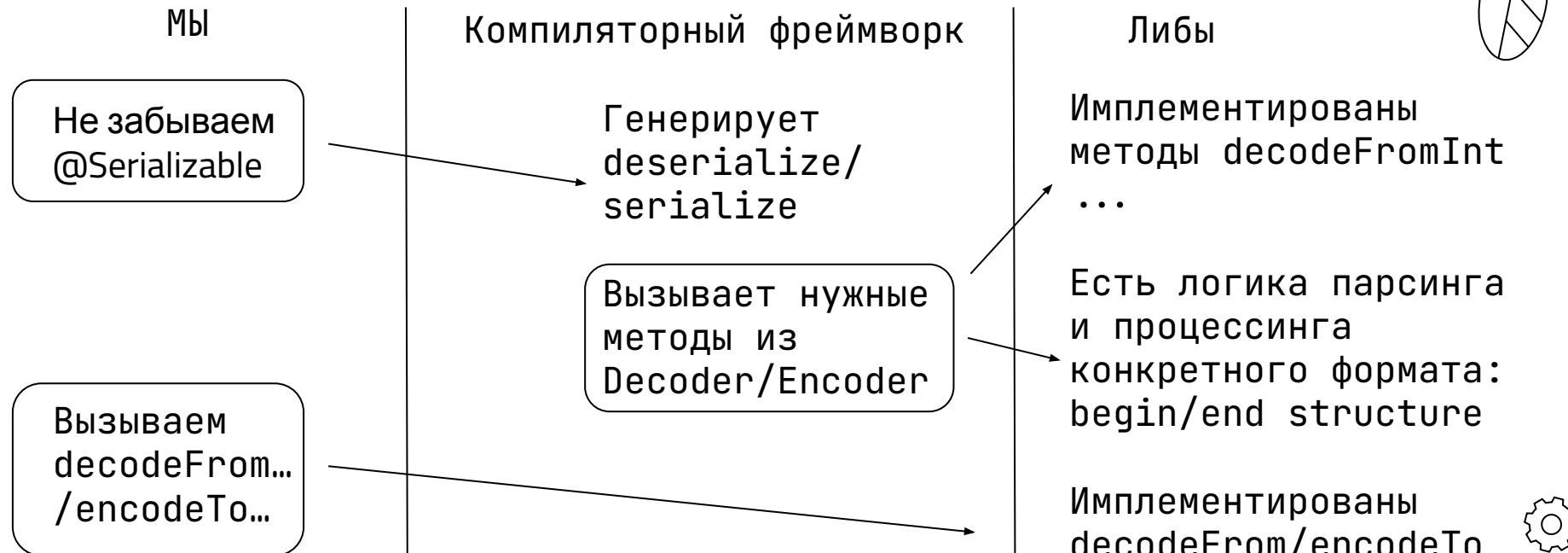
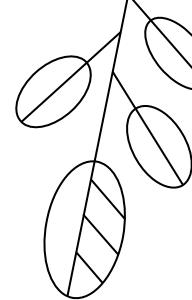
# Цепочки вызовов



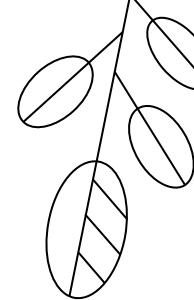
Либы



# Цепочки вызовов



# Цепочки вызовов



Мы

Не забываем  
@Serializable

Вызываем  
decodeFrom...  
/encodeTo...

Компиляторный фреймворк

Генерирует  
deserialize/  
serialize

Вызывает нужные  
методы из  
Decoder/Encoder

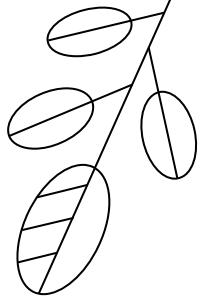
Либы

Имплементированы  
методы decodeFromInt  
...

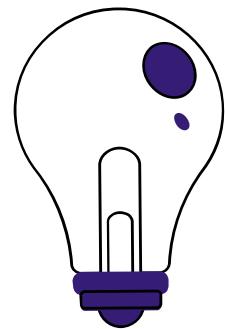
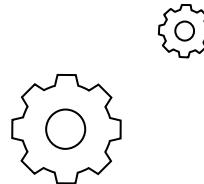
Есть логика парсинга  
и процессинга  
конкретного формата:  
begin/end structure

Имплементированы  
decodeFrom/encodeTo

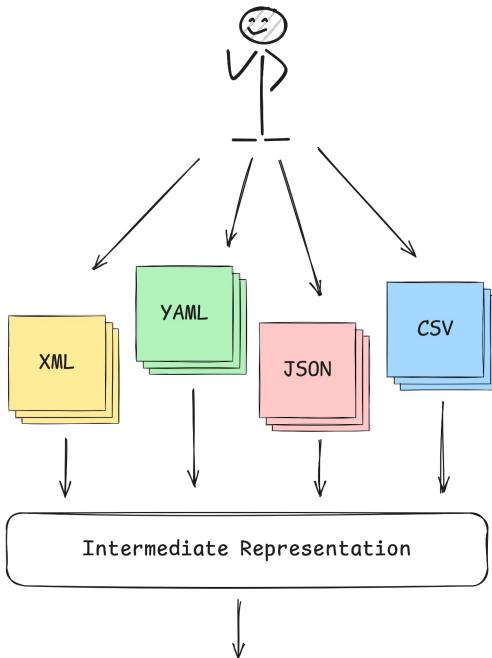


**04**

## Хитрый сценарий для лучшего понимания

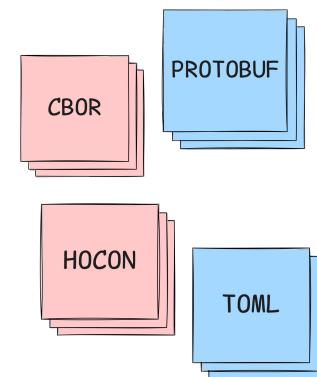


# Наша проблема



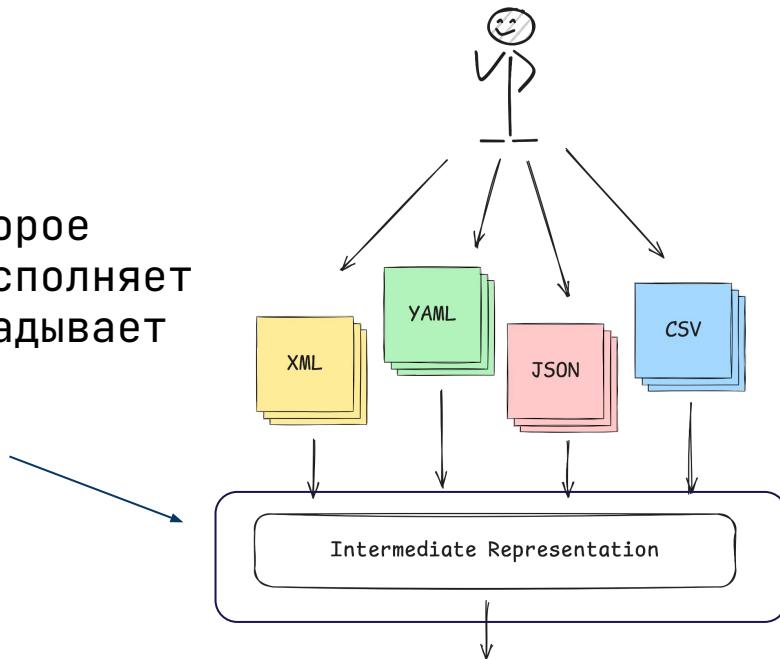
вероятно бинарные  
и нестандартные  
форматы

+



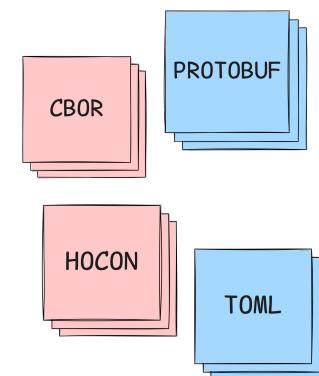
# Наша проблема

приложение, которое  
десериализует, исполняет  
логику и перекладывает  
в другой формат

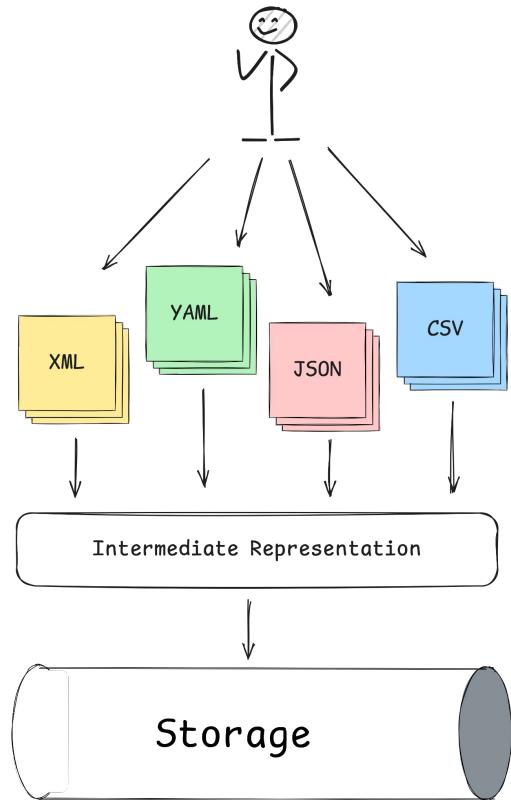


вероятно бинарные  
и нестандартные  
форматы

+

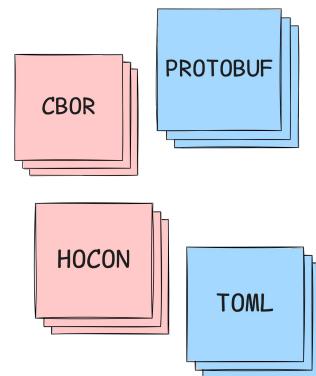


# Наша проблема



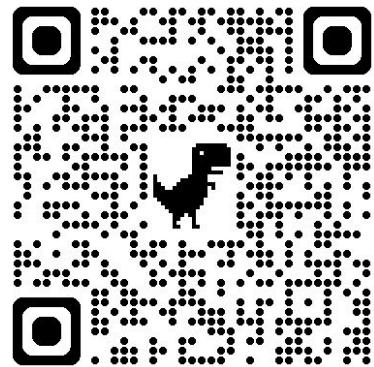
вероятно бинарные  
и нестандартные  
форматы

+





А вот в качестве IR у нас не просто  
схема из кода, а **AVRO**





# Наш IR: AVRO schema

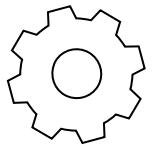


```
{  
    "type" : "record",  
    "name" : "Message",  
    "namespace" : "schema.namespace",  
    "fields" : [  
        {  
            "name" : "title",  
            "type" : [ "null", "string" ],  
            "default": null  
        },  
        {  
            "name" : "message",  
            "type" : "string"  
        }  
    ]  
}
```





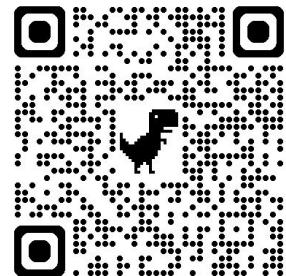
# Наш IR: AVRO schema



```
{  
    "type" : "record",  
    "name" : "Message",  
    "namespace" : "schema.namespace",  
    "fields" : [  
        {  
            "name" : "title",  
            "type" : [ "null", "string" ],  
            "default": null  
        },  
        {  
            "name" : "message",  
            "type" : "string"  
        }  
    ]  
}
```

имена

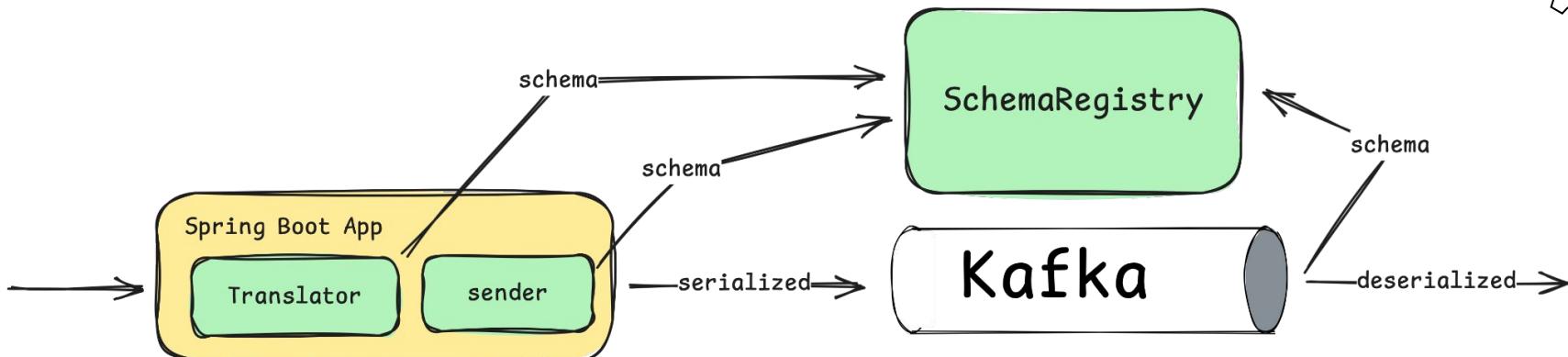
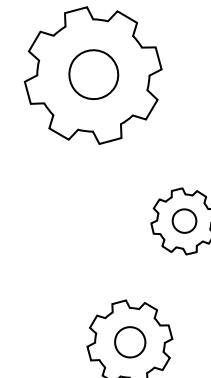
типы





# Наш IR: AVRO

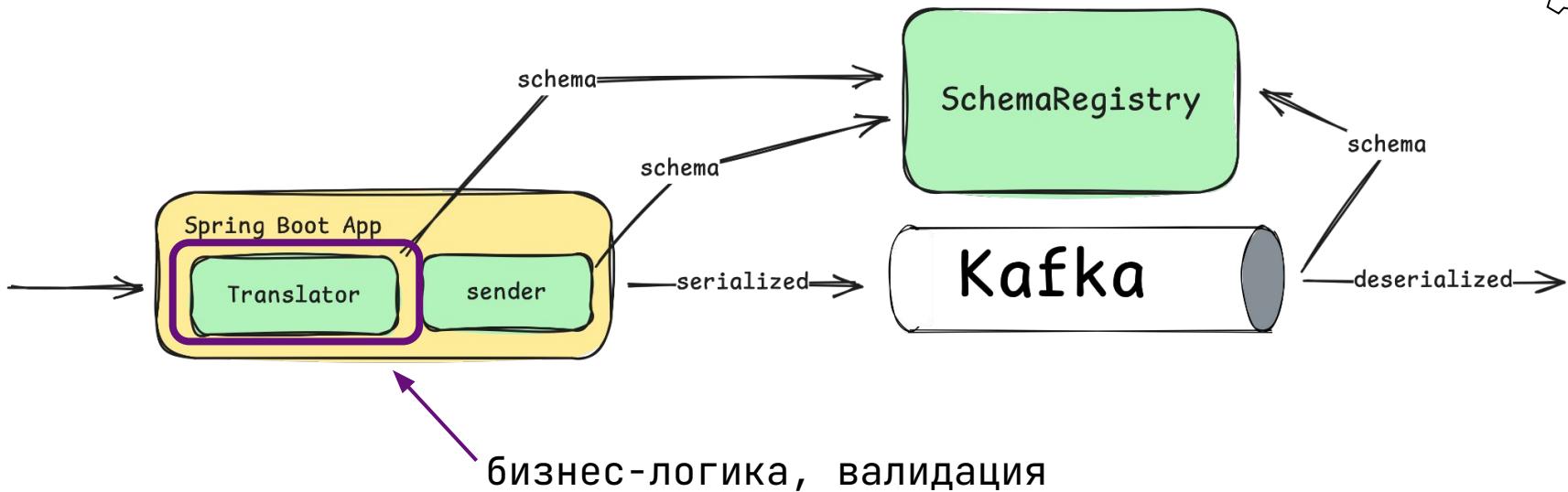
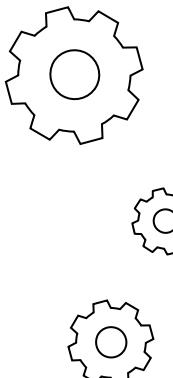
## почему?





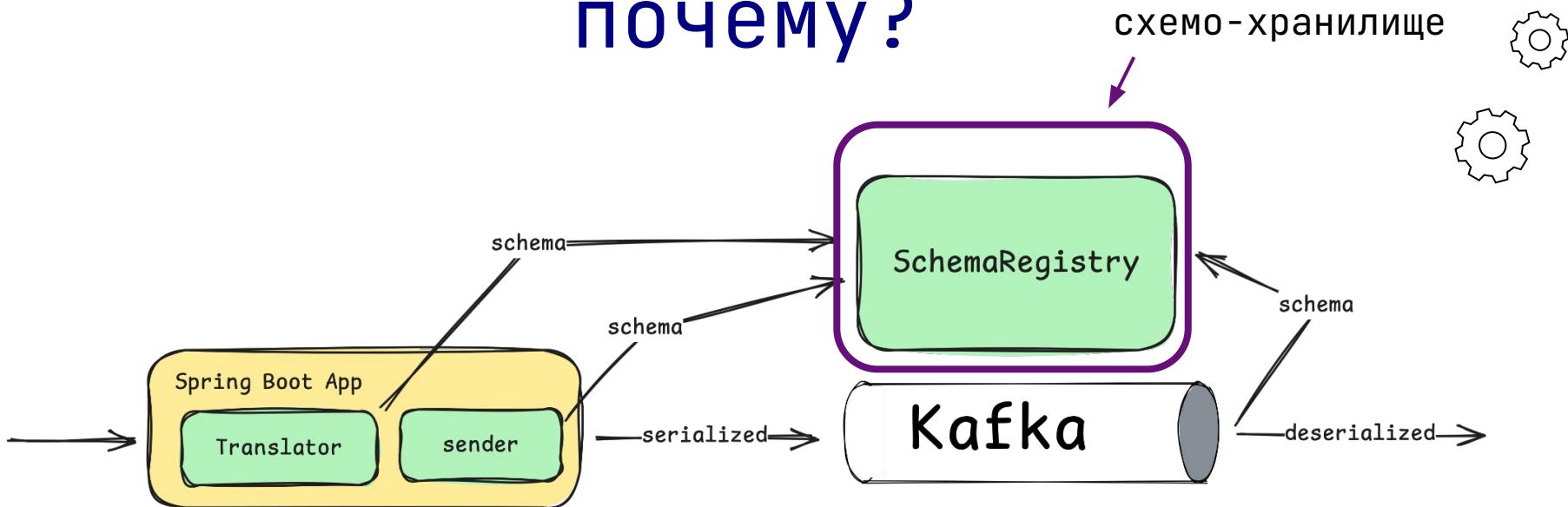
# Наш IR: AVRO

## почему?



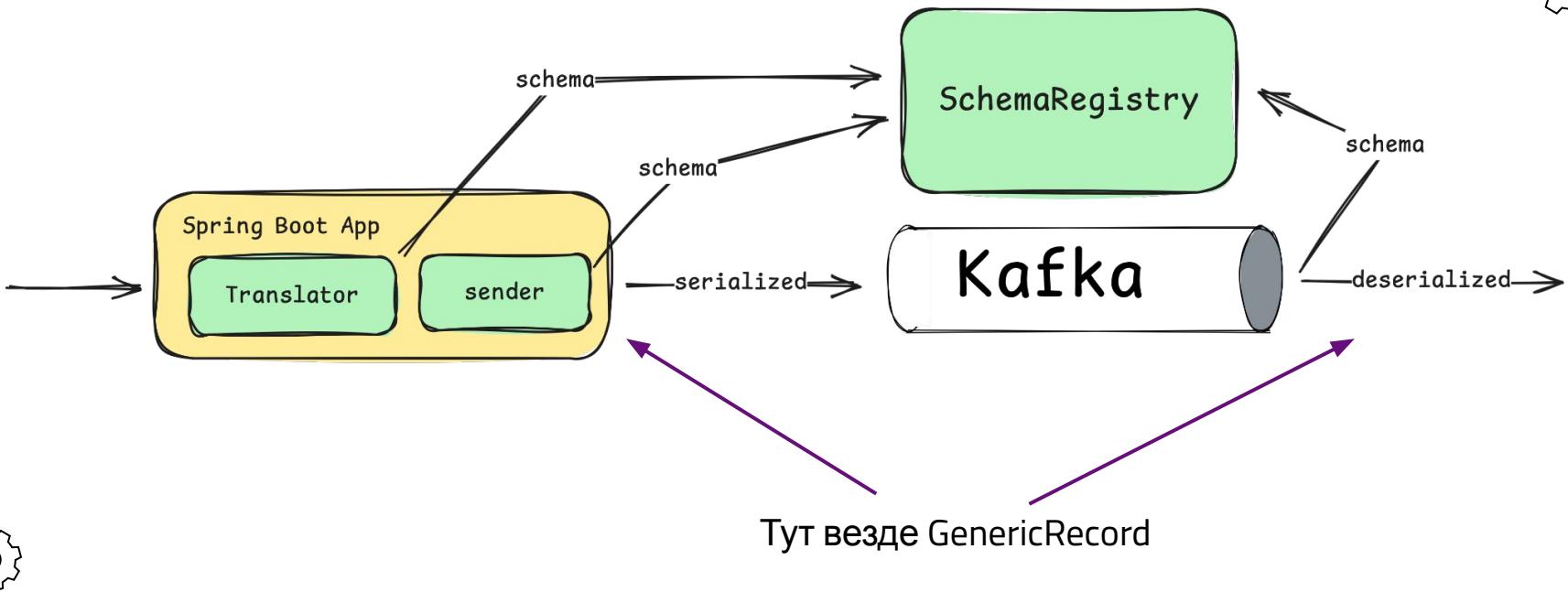
# Наш IR: AVRO

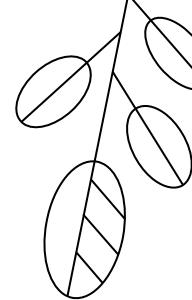
## почему?



# Наш IR: AVRO

## почему?





В чем разница?

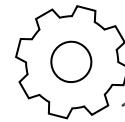
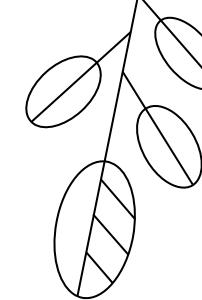
В схеме!



# Реальный сценарий

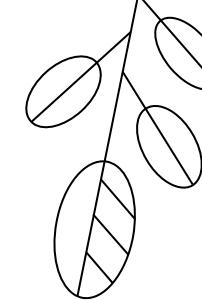
```
@PostMapping("/single")
fun postSingle(
    @RequestBody message: ByteArray,
    @RequestHeader(HttpHeaders.CONTENT_TYPE) contentType: MediaType,
    @RequestHeader(HttpHeaders.ACCEPT, required = false) accept: List<MediaType>?,
): ResponseEntity<Any> {

}
```



# Реальный сценарий

```
@PostMapping("/single")
fun postSingle(
    @RequestBody message: ByteArray,
    @RequestHeader(HttpHeaders.CONTENT_TYPE) contentType: MediaType,
    @RequestHeader(HttpHeaders.ACCEPT, required = false) accept: List<MediaType>?,
): ResponseEntity<Any> {
    // 1) используем определенную заранее схему из SchemaRegistry
}
```



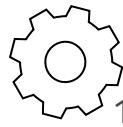
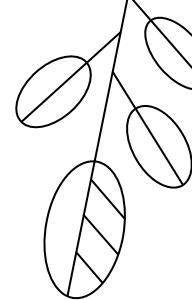
123

# Реальный сценарий

```
@PostMapping("/single")
fun postSingle(
    @RequestBody message: ByteArray,
    @RequestHeader(HttpHeaders.CONTENT_TYPE) contentType: MediaType,
    @RequestHeader(HttpHeaders.ACCEPT, required = false) accept: List<MediaType>?,
): ResponseEntity<Any> {
    // 1) используем определенную заранее схему из SchemaRegistry

    // 2) обманываем kotlinx.serialization,
    //     используя эту схему как замену для сгенерированного кода KSerializer и
    //     выигрывая за счет существующих Decoder'ы для работы с форматами

}
```

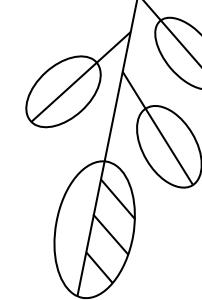


# Реальный сценарий

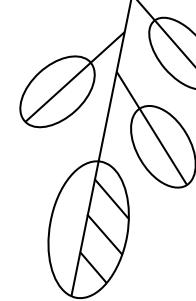
```
@PostMapping("/single")
fun postSingle(
    @RequestBody message: ByteArray,
    @RequestHeader(HttpHeaders.CONTENT_TYPE) contentType: MediaType,
    @RequestHeader(HttpHeaders.ACCEPT, required = false) accept: List<MediaType>?,
): ResponseEntity<Any> {
    // 1) используем определенную заранее схему из SchemaRegistry

    // 2) обманываем kotlinx.serialization,
    //     используя эту схему как замену для сгенерированного кода KSerializer и
    //     выигрывая за счет существующих Decoder'ы для работы с форматами

    // 3) отдаем ответ в выбранном пользователем формате
}
```



# “Обманываем” kotlinx

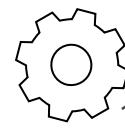
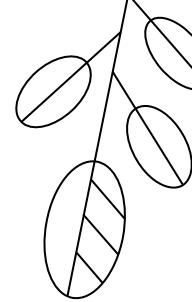


Передаем схему в `KSerializer`, замещаем  
**сгенерированный** код



# “Обманываем” kotlinx

```
class AvroSchemaDeserializer(  
    private val avroSchema: Schema,  
) : KSerializer<GenericRecord> {
```



# “Обманываем” kotlinx

```
class AvroSchemaDeserializer(  
    private val avroSchema: Schema,  
) : KSerializer<GenericRecord> {  
    ...  
  
    // (!) с помощью buildClassSerialDescriptor строим дескриптор  
    override val descriptor: SerialDescriptor = avroSchema.fields.toSerialDescriptor("Root")
```



# “Обманываем” kotlinx

```
class AvroSchemaDeserializer(  
    private val avroSchema: Schema,  
) : KSerializer<GenericRecord> {  
    ...  
  
    override fun deserialize(decoder: Decoder): GenericRecord {  
        val compositeDecoder = decoder.beginStructure(descriptor)  
        ...  
        // и начинаем крутиться в классическом цикле,  
        while (true) {  
            ...  
            val index = compositeDecoder.decodeElementIndex(descriptor)  
            if (index == CompositeDecoder.DECODE_DONE) break  
            ...  
        }  
    }
```



# “Обманываем” kotlinx

```
class AvroSchemaDeserializer(  
    private val avroSchema: Schema,  
) : KSerializer<GenericRecord> {  
    ...  
  
    override fun deserialize(decoder: Decoder): GenericRecord {  
        val compositeDecoder = decoder.beginStructure(descriptor)  
        ...  
        // и начинаем крутиться в классическом цикле,  
        while (true) {  
            ...  
            val index = compositeDecoder.decodeElementIndex(descriptor)  
            if (index == CompositeDecoder.DECODE_DONE) break  
            val key = avroSchema.fields[index]  
            val schema = fields[index].schema()  
            ...  
        }  
    }
```



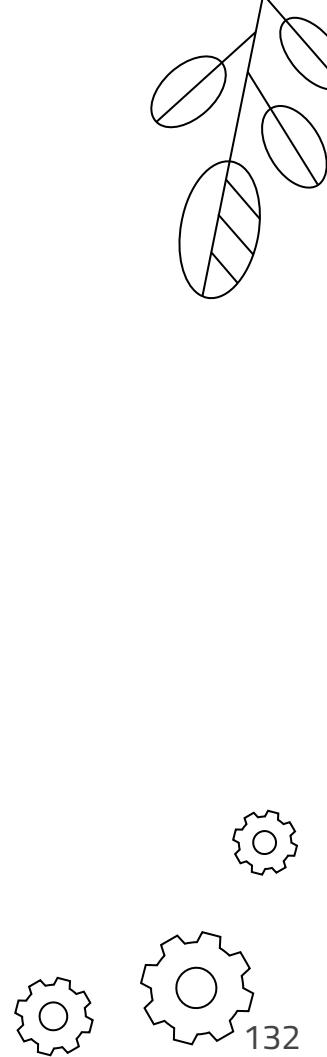
# “Обманываем” kotlinx

```
class AvroSchemaDeserializer(  
    private val avroSchema: Schema,  
) : KSerializer<GenericRecord> {  
    ...  
  
    override fun deserialize(decoder: Decoder): GenericRecord {  
        val compositeDecoder = decoder.beginStructure(descriptor)  
        ...  
        // и начинаем крутиться в классическом цикле,  
        while (true) {  
            ...  
            val value = compositeDecoder.decodeSerializableElement(  
                descriptor,  
                index,  
                schema.getDeserializer(errorListener)  
            )  
            ...  
        }  
    }
```

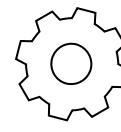


# “Обманываем” kotlinx

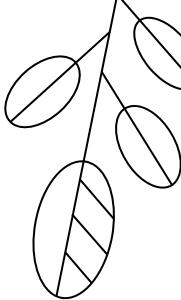
```
class AvroSchemaDeserializer(
    private val avroSchema: Schema,
) : KSerializer<GenericRecord> {
    ...
    ...
    override fun deserialize(decoder: Decoder): GenericRecord {
        val compositeDecoder = decoder.beginStructure(descriptor)
        ...
        // и начинаем крутиться в классическом цикле,
        while (true) {
            ...
            genericRecord.put(
                key.name(),
                value
            )
            ...
        }
        return genericRecord
    }
}
```



# Что мы сегодня поняли?



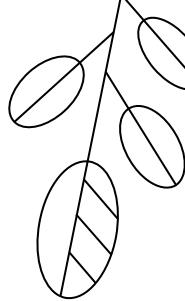
# Мы сегодня многое поняли



- Что `Koltinx.serialization` - продвинутый инструмент сериализации, особенно, если у вас мультиформатная работа



# Мы сегодня многое поняли

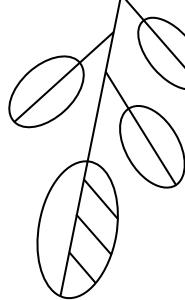


- Что `Koltinx.serialization` - продвинутый инструмент сериализации, особенно, если у вас мультиформатная работа
- Какой код `генерируется компилятором` и почему это важно знать



135

# Мы сегодня многое поняли

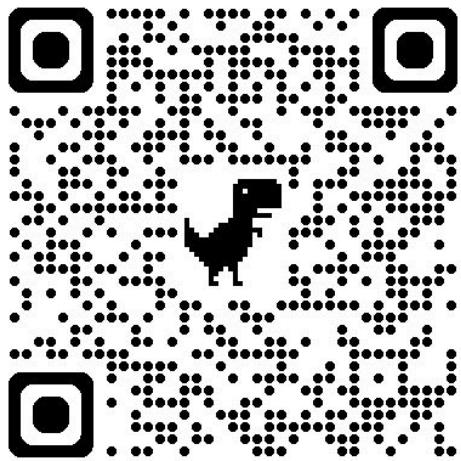


- Что `Koltinx.serialization` - продвинутый инструмент сериализации, особенно, если у вас мультиформатная работа
- Какой код `генерируется компилятором` и почему это важно знать
- Откуда что и как вызывается

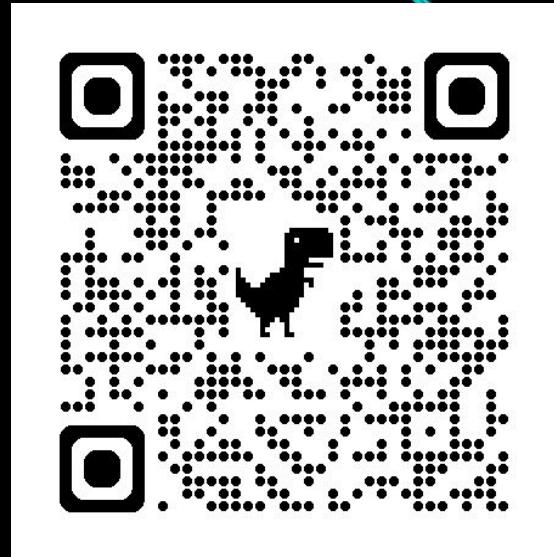




И спасибо, что продержались до конца!



[Telegram](#)



[Github](#)