

# SPRING

и

# KOTLIN

Работай с  
любым форматом!



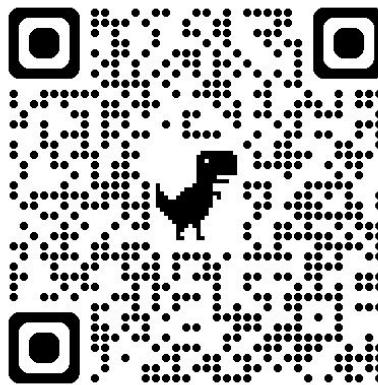
\$ whoami?



Андрей Кулешов

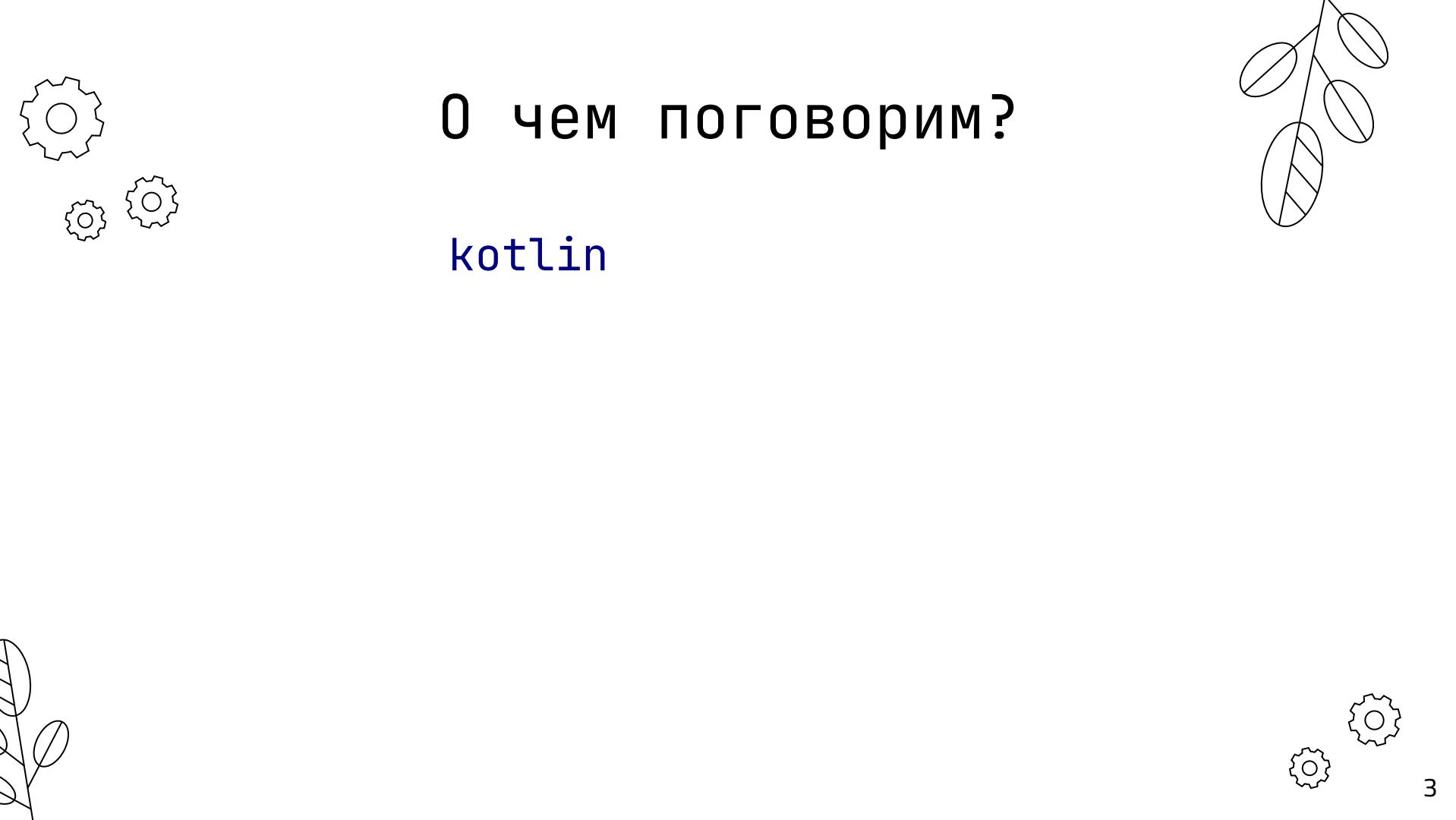


Стачка



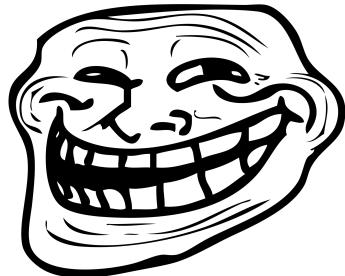
# О чём поговорим?

kotlin



# О чём поговорим?

`kotlinx.serialization`



# О чём поговорим?

`kotlinx.serialization`

**пользовательские** кастомизации

# О чём поговорим?

`kotlinx.serialization`

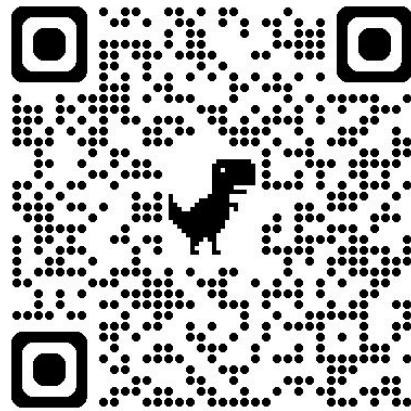
пользовательские кастомизации

реальная история  
десериализации в нашем  
`Spring Boot`'овом приложении

# О чём поговорим?



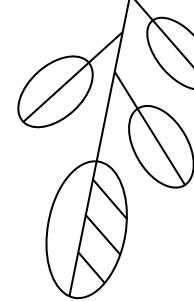
# Почему я?



[KT]oml

powered by kotlinx.serialization

# 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  

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 оненсорс-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.

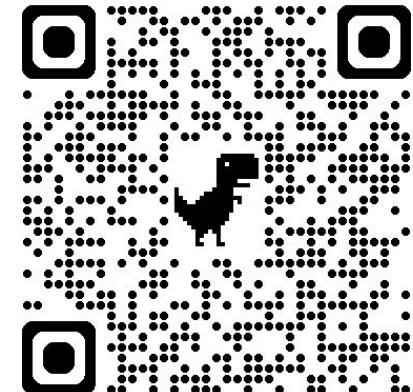
Доклад будет интересен Kotlin-программистам разных уровней, как пользователям сериализаторов, так и тем, кого можно заинтересовать созданием своего собственного сериализатора.

#compilerplugin #library

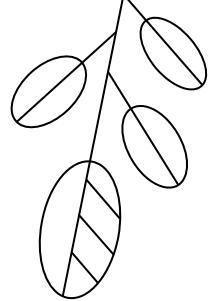
### Спикеры



Андрей Кулешов  
Huawei



# Сегодня снова не будет 😭



g

@guai9632 1 year ago

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

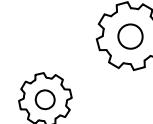
2 Reply



@telephon3208 1 year ago

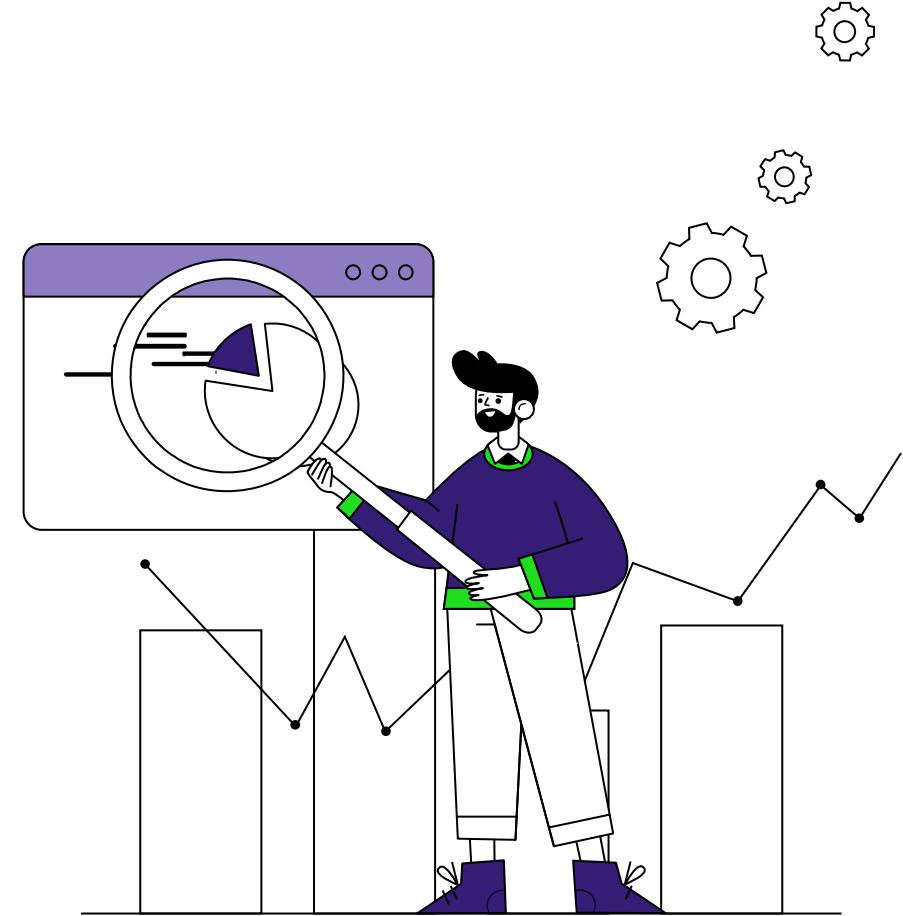
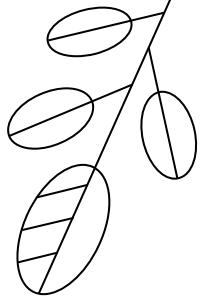
бесценный доклад

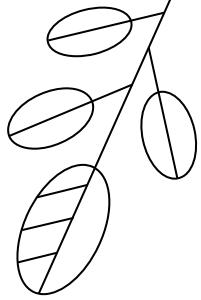
1 Reply

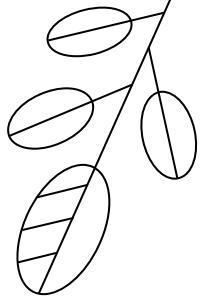


01

# Проблема

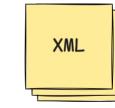
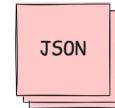
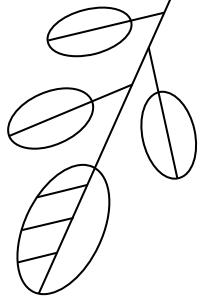






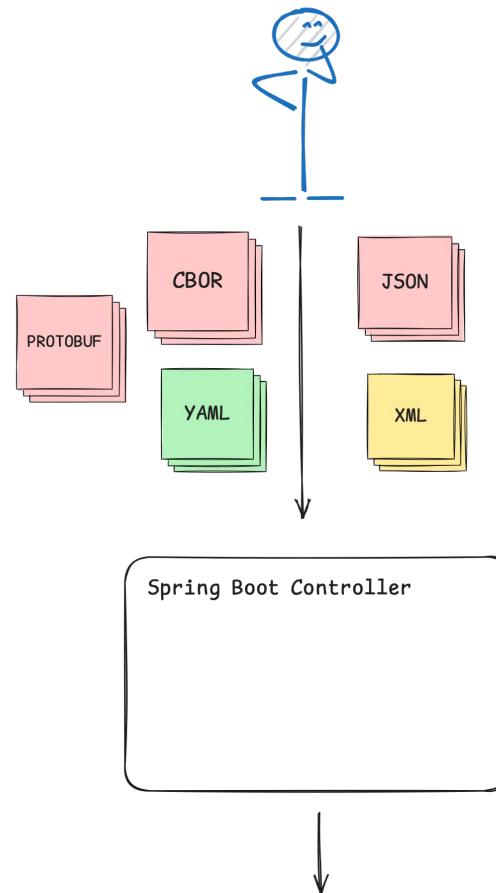
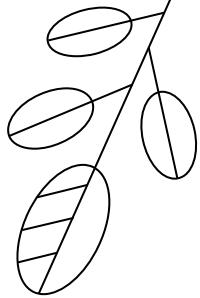
Spring Boot Controller

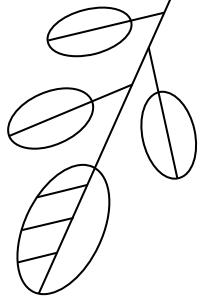




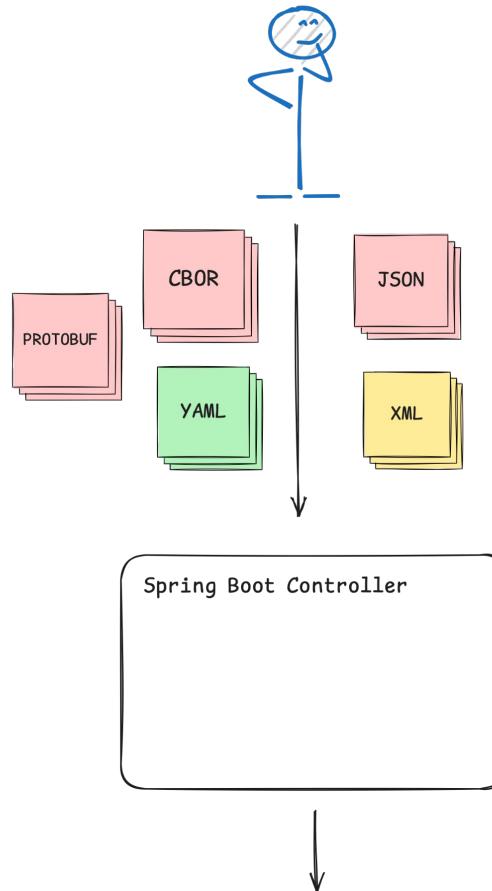
Spring Boot Controller

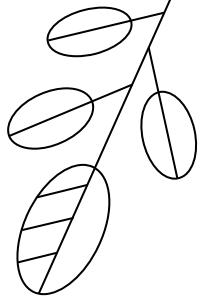




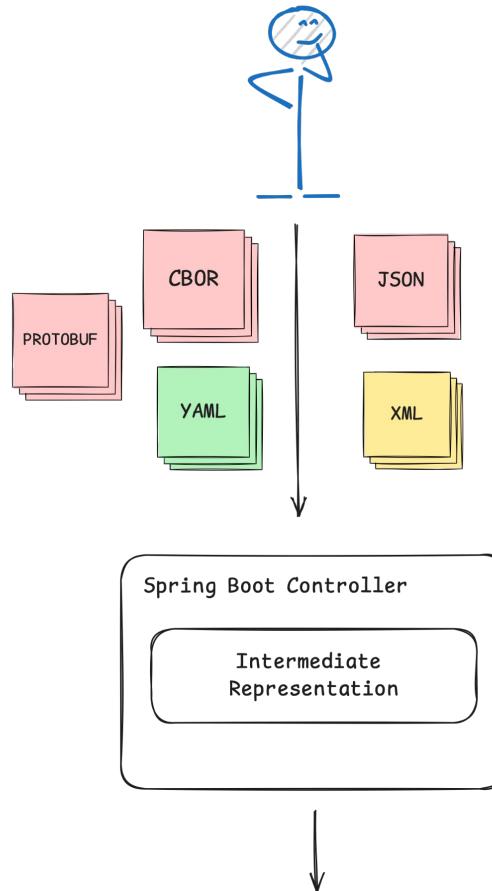


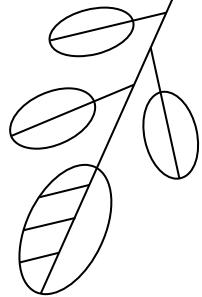
@RequestHeader(HttpHeaders.CONTENT\_TYPE)



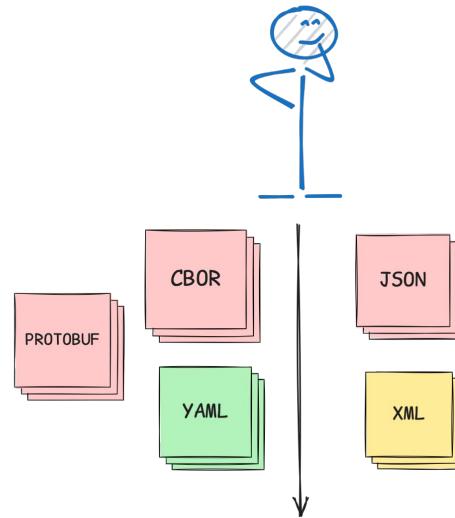


@RequestHeader(HttpHeaders.CONTENT\_TYPE)





@RequestHeader(HttpHeaders.CONTENT\_TYPE)



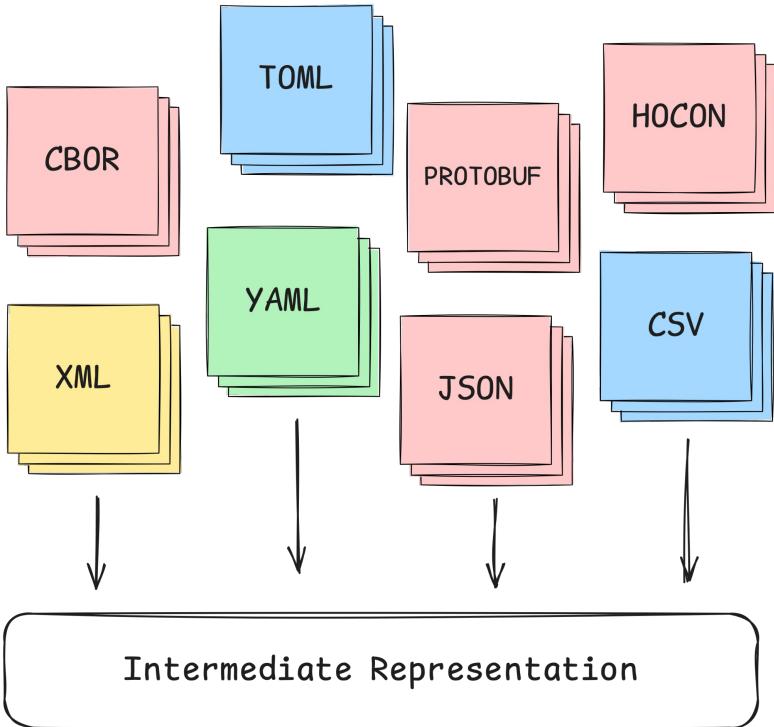
Spring Boot Controller

Intermediate  
Representation



@RequestHeader(HttpHeaders.ACCEPT)

# Много форматов - единое представление





# Какое представление?



# TOML

## How ktoml works: examples

! You can check how below examples work in [decoding ReadMeExampleTest](#) and [encoding ReadMeExampleTest](#).

### ▼ Deserialization

The following example:

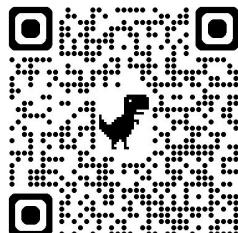
```
someBooleanProperty = true
# inline tables in gradle 'libs.versions.toml' notation
gradle-libs-like-property = { id = "org.jetbrains.kotlin.jvm", version.ref = "kotlin" }

[table1]
# null is prohibited by the TOML spec, but allowed in ktoml for nullable types
# so for 'property1' null value is ok. Use: property1 = null
property1 = 100
property2 = 6

[myMap]
a = "b"
c = "d"

[table2]
someNumber = 5
[table2."akuleshov7.com"]
name = 'this is a "literal" string'
# empty lists are also supported
configurationList = ["a", "b", "c"]

# such redeclaration of table2
# is prohibited in toml specification;
# but ktoml is allowing it in non-strict mode:
[table2]
otherNumber = 5.56
# use single quotes
charFromString = 'a'
charFromInteger = 123
```



# JSON

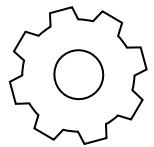
Translation of the example above to json-terminology:

```
{  
    "someBooleanProperty": true,  
  
    "gradle-libs-like-property": {  
        "id": "org.jetbrains.kotlin.jvm",  
        "version": {  
            "ref": "kotlin"  
        }  
    },  
  
    "table1": {  
        "property1": 100,  
        "property2": 5  
    },  
    "table2": {  
        "someNumber": 5,  
  
        "otherNumber": 5.56,  
        "akuleshov7.com": {  
            "name": "my name",  
            "configurationList": [  
                "a",  
                "b",  
                "c"  
            ]  
        }  
    }  
}
```

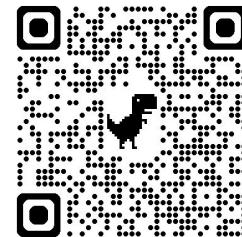




# Самый простой IR: со схемой внутри программы

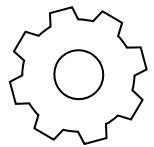


```
@Serializable  
data class MyClass(  
    val someBooleanProperty: Boolean,  
    val table1: Table1,  
    val table2: Table2,  
    @SerializedName("gradle-libs-like-property")  
    val kotlinJvm: GradlePlugin,  
    val myMap: Map<String, String>  
)
```





# Самый простой IR: со схемой внутри программы

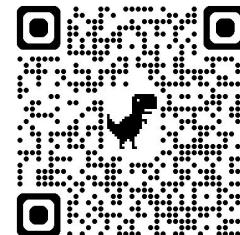


```
@Serializable  
data class MyClass(  
    val someBooleanProperty: Boolean,  
    val table1: Table1,  
    val table2: Table2,  
    @SerializedName("gradle-libs-like-property")  
    val kotlinJvm: GradlePlugin,  
    val myMap: Map<String, String>  
)
```

имена

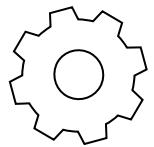
Boolean,

типы





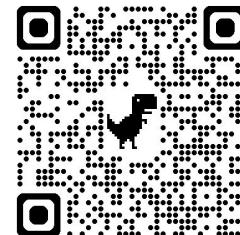
# Самый простой IR: со схемой внутри программы



```
@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

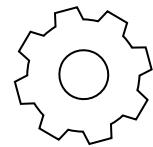


Код как схема данных





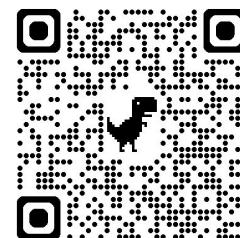
# Самый простой IR: со схемой внутри программы

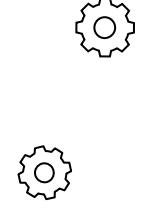
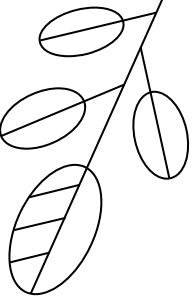


```
@Serializable  
data class MyClass(  
    val someBooleanProperty: Boolean,  
    val table1: Table1,  
    val table2: Table2,  
    @SerialName("gradle-libs-like-property")  
    val kotlinJvm: GradlePlugin,  
    val myMap: Map<String, String>  
)
```

...

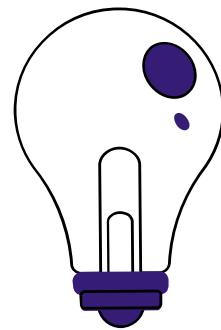
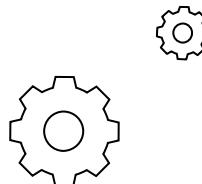
Как это работает?



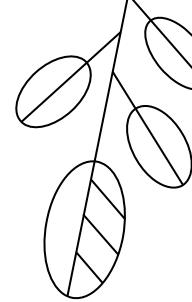


02

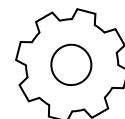
# kotlinx.serialization



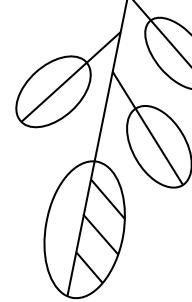
# Вкратце



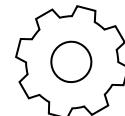
- Мультиплатформенная (**KMP**) библиотека



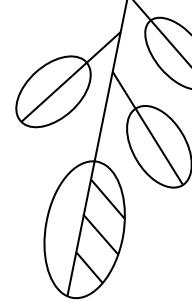
# Вкратце



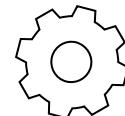
- Мультиплатформенная (КМР) библиотека
- Является **компиляторным плагином**



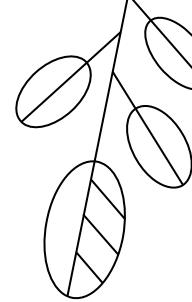
# Вкратце



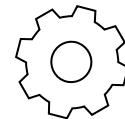
- Мультиплатформенная (КМР) библиотека
- Является **компиляторным плагином**
- **НЕ** использует reflection



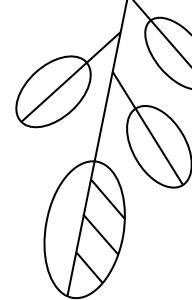
# Вкратце



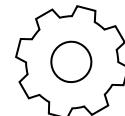
- Мультиплатформенная (КМР) библиотека
- Является **компиляторным плагином**
- **НЕ** использует reflection
- Встраивается в проект с помощью Maven/Gradle плагина



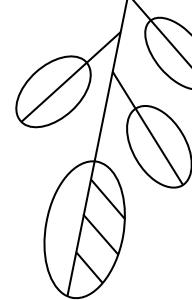
# Вкратце



- 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



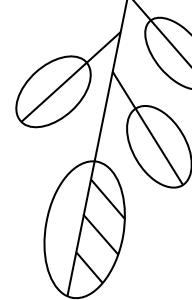
# Вкратце



- 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



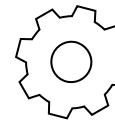
**sandwwraith** commented on 17 Jan

Member

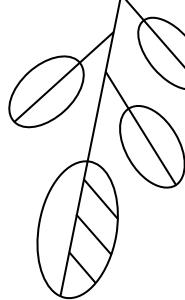


...

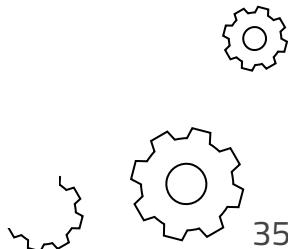
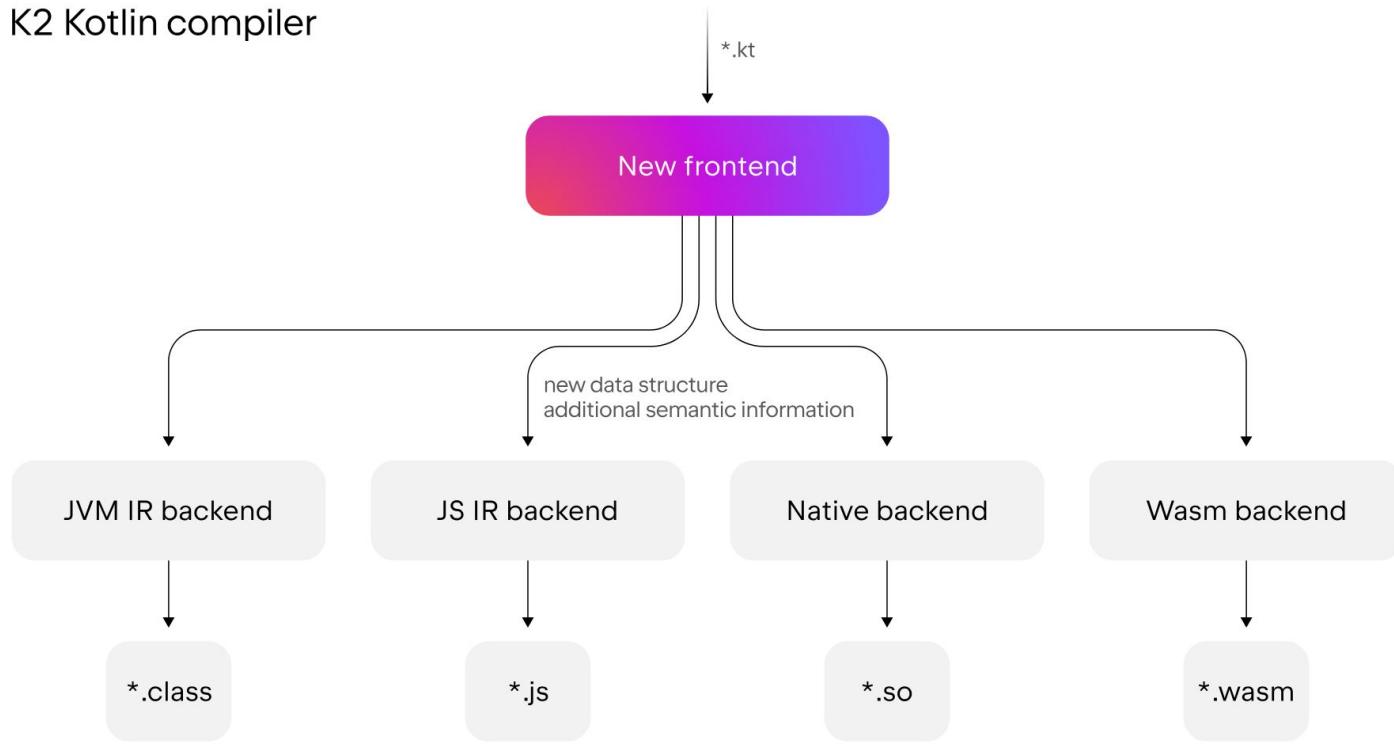
We don't have plans to merge more formats in this repository; I think the way it works now — with the list of community-supported formats — is the most optimal.



# Компиляторные плагины

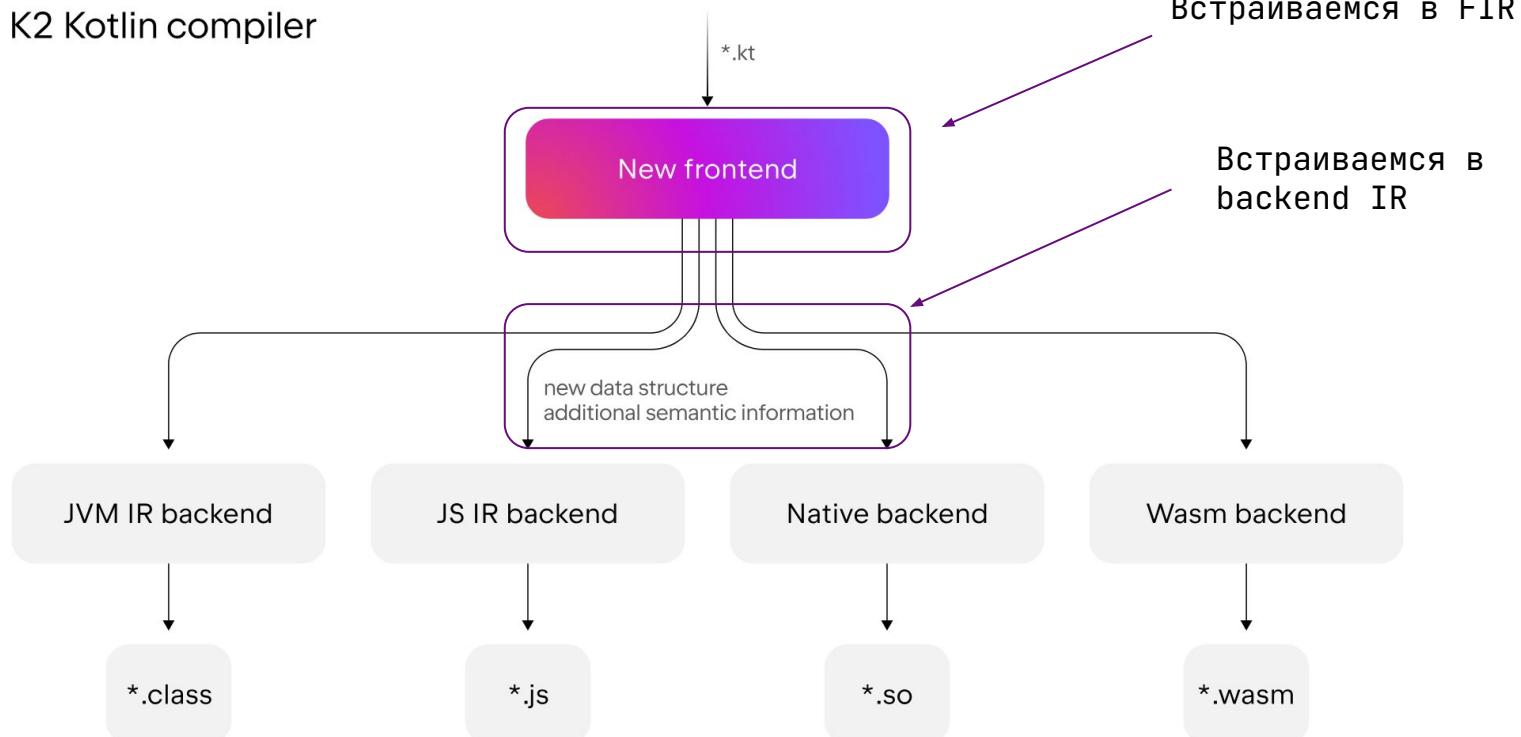


K2 Kotlin compiler

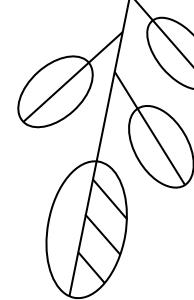


# Компиляторные плагины

K2 Kotlin compiler

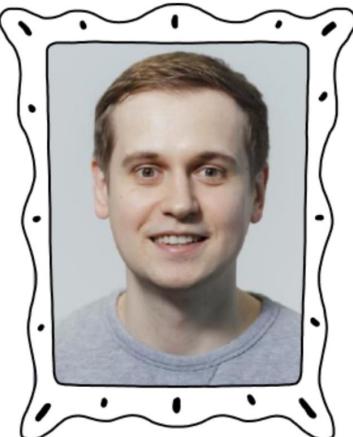


# Компиляторные плагины

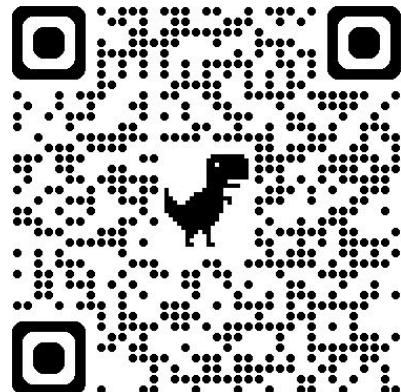
[Купить билет](#)[Спикеры](#)[Как это было в прошлом году](#)

## 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.

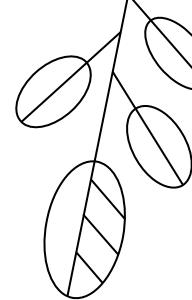


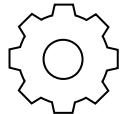
АНДРЕЙ КУЛЕШОВ,  
HUAWEI



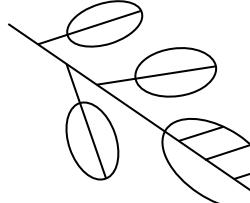
# Компиляторные плагины

а зачем мне эта  
информация





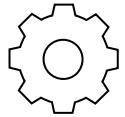
# Пример использования



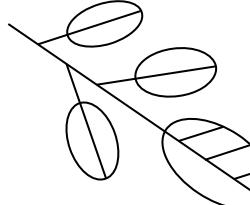
```
package com.akuleshov7
```

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

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



# Пример использования



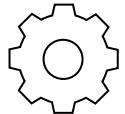
```
package com.akuleshov7
```

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

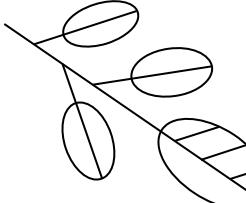
```
fun main() {  
    val a = A(0)  
    serializeToJsonSomehow(a)  
}
```

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





# Пример использования

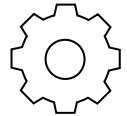


```
package com.akuleshov7
```

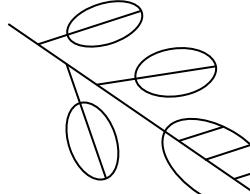
```
data class A(  
    val b: Int  
)
```

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

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



# Пример использования



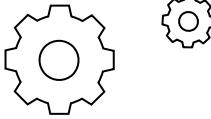
```
package com.akuleshov7

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

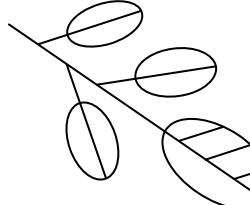
data class A(
    val b: Int
)

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

Build нормальный



# Ловим ошибку



```
package com.akuleshov7

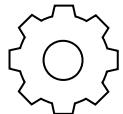
import kotlinx.serialization.encodeToString
import kotlinx.serialization.json.Json

data class A(
    val b: Int
)

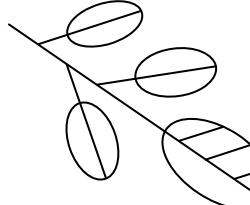
fun main() {
    val a = A(0)
    Json.encodeToString(a)
}
```

Build нормальный, а это на runtime

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



# Добавляем Serializable

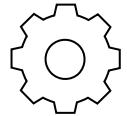


```
package com.akuleshov7

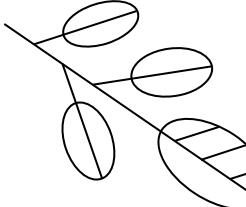
import kotlinx.serialization.Serializable
import kotlinx.serialization.encodeToString
import kotlinx.serialization.json.Json

@Serializable
data class A(
    val b: Int
)

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



# ЛОВИМ ОШИБКУ x2



```
package com.akuleshov7

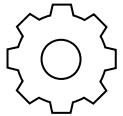
import kotlinx.serialization.Serializable
import kotlinx.serialization.encodeToString
import kotlinx.serialization.json.Json

@Serializable
data class A(
    val b: Int
)

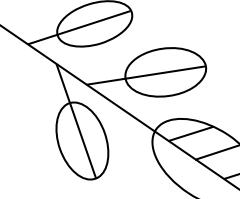
fun main() {
    val a = A(0)
    Json.encodeToString(a)
}
```



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



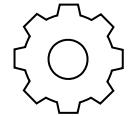
# Приисматриваемся



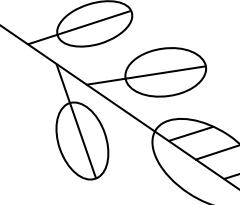
```
1 package com.akuleshov7
2
3 import kotlinx.serialization.Serializable
4 import kotlinx.serialization.encodeToString
5 import kotlinx.serialization.json.Json
6
7 @Serializable
8 data class User {
9     val name: String = "John Doe"
10 }
11
12 fun main() {
13     val user = User()
14     val json = Json.encodeToString(user)
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.



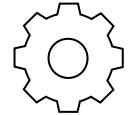
# Присматриваемся



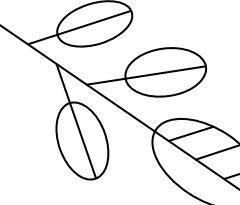
```
1 package com.akuleshov7
2
3 import kotlinx.serialization.Serializable
4 import kotlinx.serialization.encodeToString
5 import kotlinx.serialization.json.Json
6
7 @Serializable
8 data class Test {
9     val test = "test"
10 }
11
12 fun main() {
13     val json = Json.encodeToString(Test())
14 }
15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



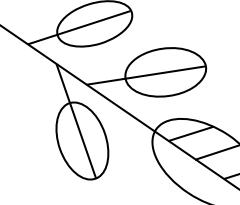
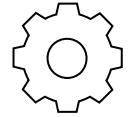
```
1 package com.akuleshov7
2
3 import kotlinx.serialization.Serializable
4 import kotlinx.serialization.encodeToString
5 import kotlinx.serialization.json.Json
6
7 @Serializable
8 data class Test {
9     val test = "test"
10 }
11
12 fun main() {
13     val json = Json.encodeToString(Test())
14    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

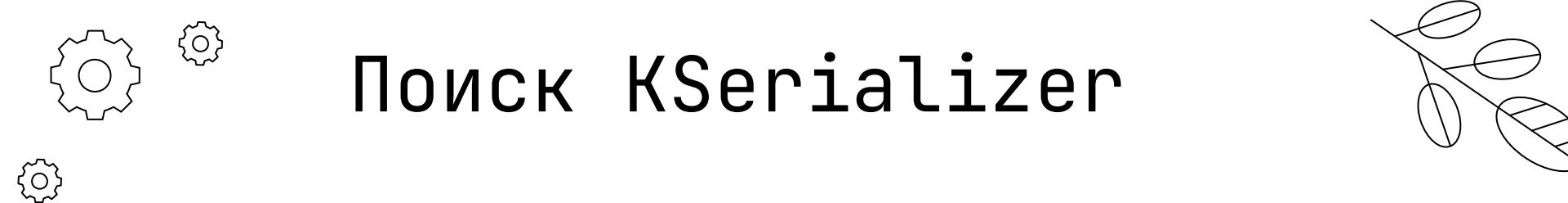


# KSerializer

Как обнаруживается?

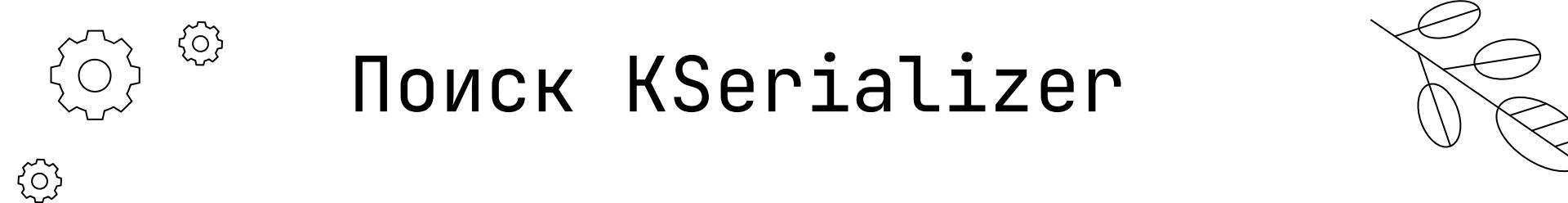
Зачем он вообще нужен?

Как генерируется?



# Поиск KSerializer

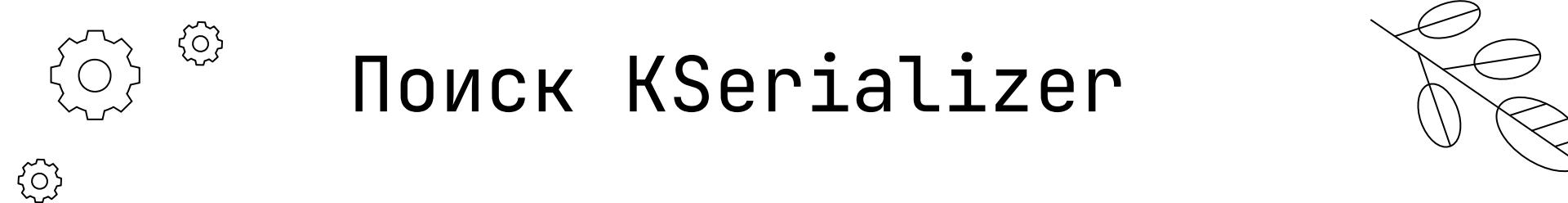
```
public fun SerializersModule.serializer(type: KType): KSerializer<Any?> =  
    serializerByKTypeImpl(type, failOnMissingTypeArgSerializer = true) ?: type.kclass()  
        .platformSpecificSerializerNotRegistered()
```



# Поиск KSerializer

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

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

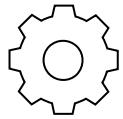


# Поиск KSerializer

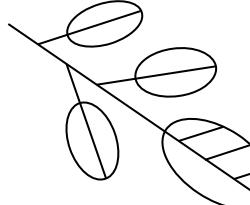
```
public fun SerializersModule.serializer(type: KType): KSerializer<Any?> =  
    serializerByKTypeImpl(type, failOnMissingTypeArgSerializer = true) ?: type.kclass()  
        .platformSpecificSerializerNotRegistered()
```

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

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

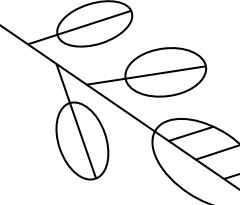
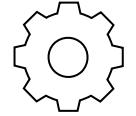


# Встроенные KSerializer



```
@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*

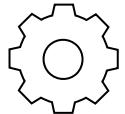


# Пример для String

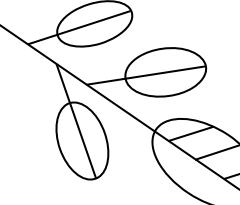
```
@PublishedApi
internal object StringSerializer : KSerializer<String> {
    override val descriptor: SerialDescriptor =
        PrimitiveSerialDescriptor("kotlin.String", PrimitiveKind.STRING)

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

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



# Пример для String

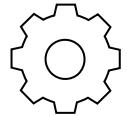


Метаинформация (схема класса итд.)

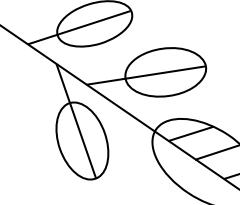
```
@PublishedApi
internal object StringSerializer : KSerializer<String> {
    override val descriptor: SerialDescriptor =
        PrimitiveSerialDescriptor("kotlin.String", PrimitiveKind.STRING)

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

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



# Пример для String



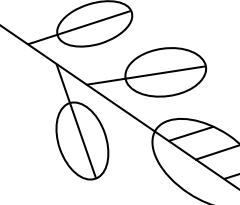
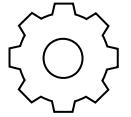
Логика сериализации для конкретного формата/библиотеки

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

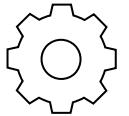
    override fun serialize(encoder: Encoder, value: String): Unit =
        encoder.encodeString(value)

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

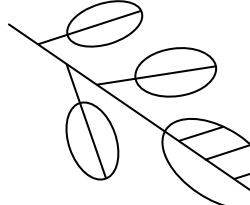
[core/commonMain/src/kotlinx/serialization/Serializers.kt](#)



# Пора добавить плагин



# Включаем плагин



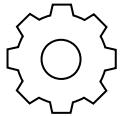
```
package com.akuleshov7

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

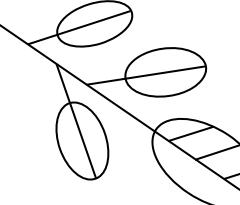
@Serializable
data class A(
    val b: Int
)

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}
}
```

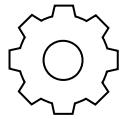
ПОЭТому НУЖНО ПОНИМАТЬ  
ЧТО ПРОИСХОДИТ ПОД  
КАПОТОМ



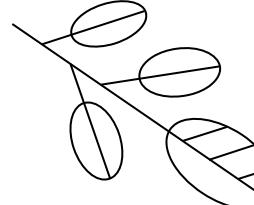


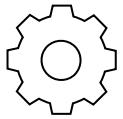
Посмотрим на  
сгенерированный код



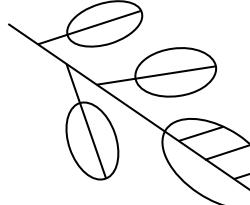


# Как посмотреть самому?





# Как посмотреть самому?



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

- Project View:** Shows the project structure for "TestKotlinSerialization". A yellow 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 */ }

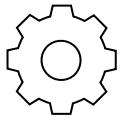
    public open fun hashCode(): kotlin.Int { /* compiled code */ }

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

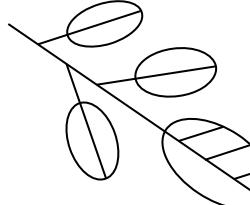
    @kotlin.jvm.JvmStatic internal final fun `write$Self`(self: com.akuleshov7.A, output: kotlinx.serialization.encoding.CompositeEncoder) { /* compiled code */ }

    @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, there is a status bar with the message "BUILD SUCCESSFUL in 1s".



# Как посмотреть самому?

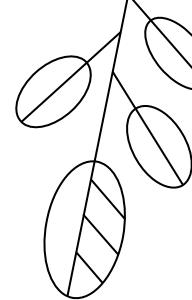


The screenshot shows the IntelliJ IDEA interface with the following details:

- Project View:** The left sidebar displays the project structure for "TestKotlinSerialization". It includes a .gradle folder, .idea, .kotlin, build (with classes, kotlin, main, com, akuleshov7), META-INF, kotlin, libs, tmp, gradle, src (main, test), .gitignore, and build.gradle.kts.
- Toolbars:** The top bar includes standard options like File, Edit, View, Navigate, Code, Refactor, Build, Run, Tools, VCS, Window, and Help.
- Code Editor:** The right side shows the code for the MainKt class. The code is annotated with comments indicating it is a compiler stub generated from a class file, noting that methods are not available.
- Contextual Menu:** A context menu is open over the MainKt class, with the "Decompile to Java" option highlighted.
- Bottom Status Bar:** The status bar at the bottom indicates "BUILD SUCCESSFUL in 1s".

# Что будет сгенерировано?

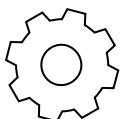
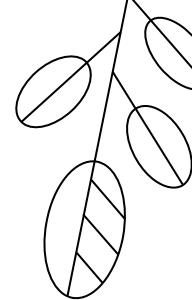
```
public class $serializer implements GeneratedSerializer
```



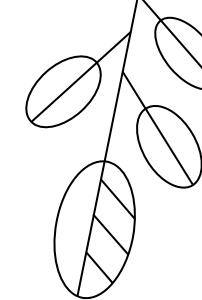
# Что будет сгенерировано?

```
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<*>>  
}
```



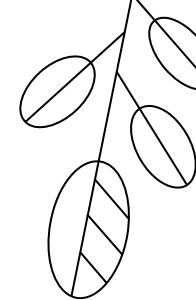
# Сериализатор



```
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

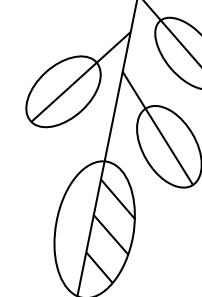


```
public class $serializer implements GeneratedSerializer {  
  
    public final void serialize(@NotNull Encoder encoder, @NotNull A value) {  
        Intrinsics.checkNotNullParameter(encoder, "encoder");  
        Intrinsics.checkNotNullParameter(value, "value");  
        SerialDescriptor var3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

“Движок”  
конкретного формата



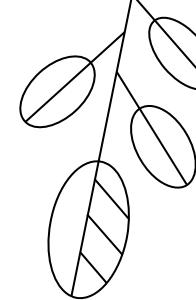
# Сериализатор



```
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);  
    }  
}
```



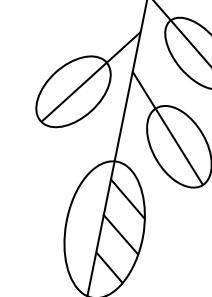
# Сериализатор



```
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

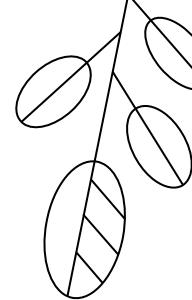


```
public class $serializer implements GeneratedSerializer {  
  
    public final void serialize(@NotNull Encoder encoder, @NotNull A value) {  
        Intrinsics.checkNotNullParameter(encoder, "encoder");  
        Intrinsics.checkNotNullParameter(value, "value");  
        SerialDescriptor var3 = 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);  
    }  
}
```

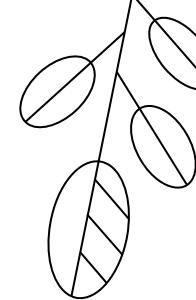


# Десериализатор

```
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);  
        ...  
    }  
}
```



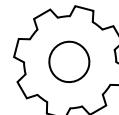
# Десериализатор



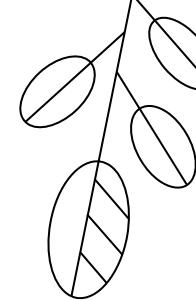
```
public class $serializer implements GeneratedSerializer {  
  
    @NotNull  
    public final A deserialize(@NotNull Decoder decoder) {  
        Intrinsics.checkNotNullParameter(decoder, "decoder");  
        SerialDescriptor var2 = descriptor;  
        boolean var3 = true;  
        int var5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

“Движок”

конкретного формата



# Десериализатор



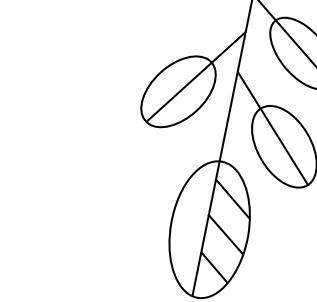
```
public class $serializer implements GeneratedSerializer {  
  
    @NotNull  
    public final A deserialize(@NotNull Decoder decoder) {  
        Intrinsics.checkNotNullParameter(decoder, "decoder");  
        SerialDescriptor var2 = descriptor;  
        boolean var3 = true;  
        int var5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

Начинаем обрабатывать структуру



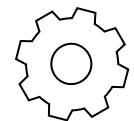
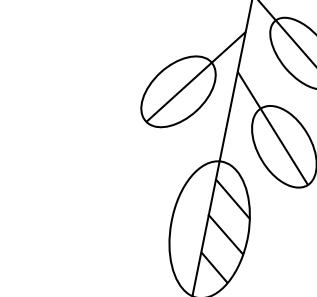
# Десериализатор

```
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);  
            }  
        }  
        ...  
    }  
}
```



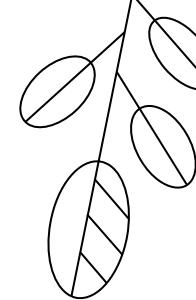
# Десериализатор

```
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

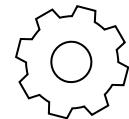


# Десериализатор

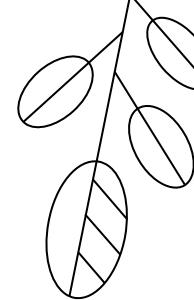


```
public class $serializer implements GeneratedSerializer {  
  
    @NotNull  
    public final A deserialize(@NotNull Decoder decoder) {  
        ...  
        while(var3) {  
            int var4 = var7.decodeElementIndex(var2); ← вытаскивается из дескриптора, который содержит эту метадату  
            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

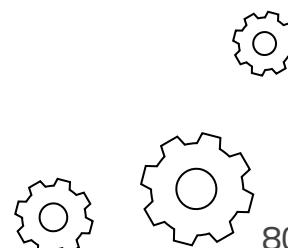


# Десериализатор

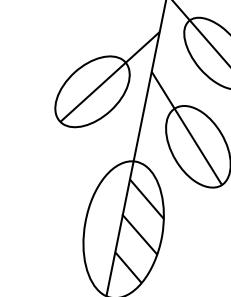


```
public class $serializer implements GeneratedSerializer {  
  
    @NotNull  
    public final A deserialize(@NotNull Decoder decoder) {  
        ...  
        while(var3) {  
            int var4 = var7.decodeElementIndex(var2);  
            switch (var4) {  
                case -1:  
                    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

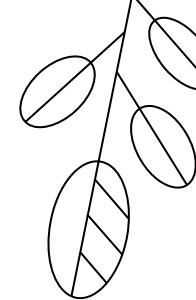


```
public class $serializer implements GeneratedSerializer {  
  
    @NotNull  
    public final A deserialize(@NotNull Decoder decoder) {  
        ...  
        while(var3) {  
            int var4 = var7.decodeElementIndex(var2);  
            switch (var4) {  
                case -1:  
      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

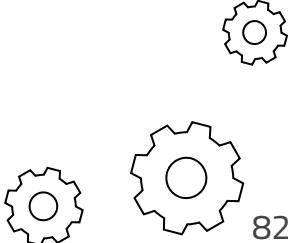
DECODE\_DONE → break@loop



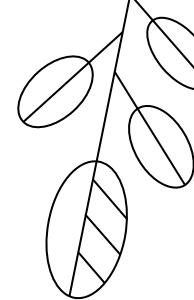
# Десериализатор



```
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);  
            }  
        }  
        ...  
    }  
}
```



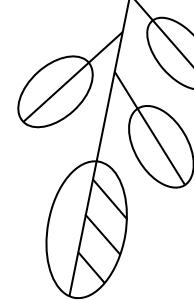
# Десериализатор



```
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

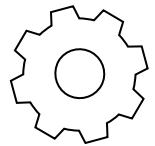


```
public class $serializer implements GeneratedSerializer {  
  
    @NotNull  
    public final A deserialize(@NotNull Decoder decoder) {  
  
        ...  
  
        var7.endStructure(var2);  
  
        return new A(var5, var6, (SerializationConstructorMarker)null);  
    }  
}
```



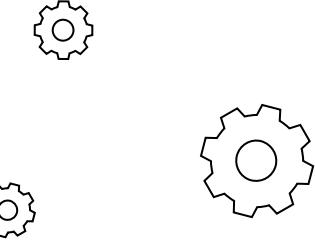
Хитрый синтетический конструктор для нашего класса, чтобы проверить то, что поля не забыты/потеряны. И присвоить значения





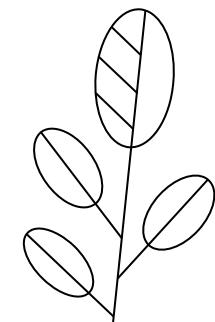
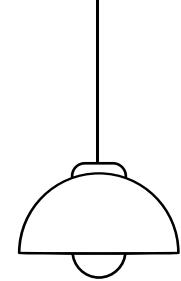
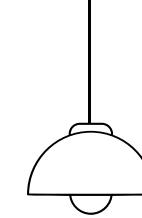
**Поняли, что компилятор  
генерирует, теперь обратно в  
реальность**





03

А что там в Spring?

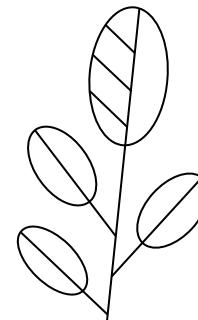




# Примитивный кейс

```
@PostMapping("/upload")
fun postData(@RequestBody request: A): ResponseEntity<String> {
    return ResponseEntity.ok("ok")
}

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



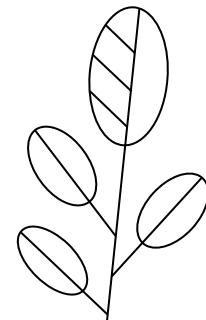


# Примитивный кейс

```
@PostMapping("/upload")
fun postData(@RequestBody request: A): ResponseEntity<String> {
    return ResponseEntity.ok("ok")
}

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

Будет работать?



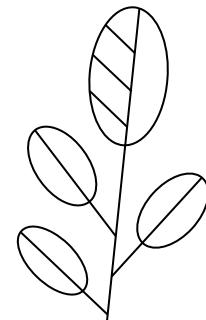


# Примитивный кейс

```
@PostMapping("/upload")
fun postData(@RequestBody request: A): ResponseEntity<String> {
    return ResponseEntity.ok("ok")
}

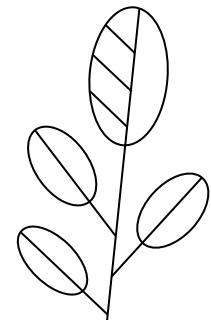
...
data class A(
    val b: Int
)
```

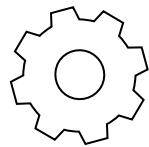
Конечно! Почему нет?





Ну мы вроде пишем на kotlin, но это же явно не `kotlinx.serialization`?





# Ну мы вроде пишем на kotlin, но это же явно не kotlinx.serialization?



org.springframework.http.converter.GenericHttpMessageConverter

The screenshot shows an IDE interface with the following details:

- Project Tree:** Shows the project structure under "org.springframework". The "converter" package is expanded, showing sub-packages like "cbor", "feed", "json", "protobuf", "smile", "support", and "xml". Several classes are listed under "converter": AbstractGenericHttpMessageConverter, AbstractHttpMessageConverter, AbstractKotlinSerializationHttpMessageConverter, BufferedImageHttpMessageConverter, ByteArrayHttpMessageConverter, FormHttpMessageConverter, GenericHttpMessageConverter, HttpMessageConversionException, HttpStatusMessageConverter, HttpStatusNotReadableException, HttpStatusNotWritableException, KotlinSerializationBinaryHttpMessageConverter, KotlinSerializationStringHttpMessageConverter, ObjectToStringHttpMessageConverter, package-info, ResourceHttpMessageConverter, ResourceRegionHttpMessageConverter, and StringHttpMessageConverter.
- Code Editor:** The file "GenericHttpMessageConverter.java" is open. The code defines a public interface "GenericHttpMessageConverter<T>" that extends "HttpMessageConverter<T>".

```
public interface GenericHttpMessageConverter<T> extends HttpMessageConverter<T> {
```
- Documentation:** A tooltip for the "canRead" method is displayed:

```
Indicates whether the given type can be read by this converter. This method should perform the same checks as HttpMessageConverter.canRead\(Class, MediaType\) with additional ones related to the generic type.
```

Params: `type` – the (potentially generic) type to test for readability  
`contextClass` – a context class for the target type, for example a class in which the target type appears in a method signature (can be `null`)  
`mediaType` – the media type to read, can be `null` if not specified. Typically, the value of a `Content-Type` header.  
Returns: `true` if readable; `false` otherwise

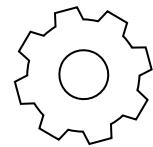
```
boolean canRead(Type type, @Nullable Class<?> contextClass, @Nullable MediaType mediaType);
```
- Method Documentation:** The "read" method is also documented:

```
T read(Type type, @Nullable Class<?> contextClass, HttpInputMessage inputMessage)  
throws IOException, HttpStatusNotReadableException;
```

Indicates whether the given class can be written by this converter.  
This method should perform the same checks as `HttpMessageConverter.canWrite(Class,`



# А там конечно Jackson



org.springframework.http.converter.json.AbstractJackson2HttpMessageConverter

The screenshot shows the IntelliJ IDEA IDE interface. On the left is the Project tool window, which lists the org.springframework module under the http converter package. The AbstractJackson2HttpMessageConverter.java file is selected and shown in the main editor area. The code implements the AbstractJackson2HttpMessageConverter class, which extends AbstractGenericHttpMessageConverter<Object>. It overrides the read and readInternal methods to handle Jackson2ObjectMapper instances. The code uses Java annotations like @Override and @NotNullable. The bottom of the screen shows the Debug toolbar, a list of running tests, and the Evaluate expression tool bar.

```
public abstract class AbstractJackson2HttpMessageConverter extends AbstractGenericHttpMessageConverter<Object> {
    @Override
    public Object read(Type type, @Nullable Class<?> contextClass, HttpInputMessage inputMessage) throws IOException, HttpMessageNotReadableException {
        JavaType javaType = getJavaType(type, contextClass);
        return readJavaType(javaType, inputMessage);
    }

    @Override
    protected Object readInternal(Class<?> clazz, HttpInputMessage inputMessage) throws IOException, HttpMessageNotReadableException {
        JavaType javaType = getJavaType(clazz, contextClass: null);
        return readJavaType(javaType, inputMessage);
    }

    private Object readJavaType(JavaType javaType, HttpInputMessage inputMessage) throws IOException {
        MediaType contentType = inputMessage.getHeaders().getContentType();
        Charset charset = getCharset(contentType);

        ObjectMapper objectMapper = selectObjectMapper(javaType.getRawClass(), contentType);
        Assert.state(objectMapper != null, () -> "No ObjectMapper for " + javaType);

        boolean isUnicode = ENCODINGS.containsKey(charset.name()) ||
            "UTF-16".equals(charset.name());
    }
}
```

Debug MyControllerTest.should return 200 when data is posted

Threads & Variables Console

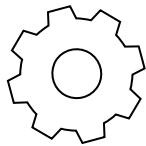
Test worker@1 in group "main": RUNNING

Evaluate expression (⌚) or add a watch (⌚⌚)

read:353, AbstractJackson2HttpMessageConverter (org.springframework.http.converter.json) > this = {MappingJackson2HttpMessageConverter@9033}



# Вроде использую Kotlin



Тогда где этот ваш `Kotlinx.serialization?`



Why Spring ▾ Learn ▾ Projects ▾ Academy ▾ Solutions ▾

› Web on Reactive Stack

› Integration

› Language Support

› Kotlin

Requirements

Extensions

Null-safety

Classes and Interfaces

Annotations

Bean Definition DSL

**Web**

Coroutines

Spring Projects in Kotlin

Getting Started

Resources

Apache Groovy

Dynamic Language Support

Spring Framework / Language Support / Kotlin / Web

```
}
```

See the [kotlin-script-template](#) example project for more details.

## Kotlin multiplatform serialization

**Kotlin multiplatform serialization** is supported in Spring MVC, Spring WebFlux and Spring Messaging (RSocket). The builtin support currently targets CBOR, JSON, and ProtoBuf formats.

To enable it, follow [those instructions](#) to add the related dependency and plugin. With Spring MVC and WebFlux, both Kotlin serialization and Jackson will be configured by default if they are in the classpath since Kotlin serialization is designed to serialize only Kotlin classes annotated with `@Serializable`. With Spring Messaging (RSocket), make sure that neither Jackson, GSON or JSONB are in the classpath if you want automatic configuration, if Jackson is needed configure `KotlinSerializationJsonMessageConverter` manually.

Prev

◀ [Bean Definition DSL](#)

Next

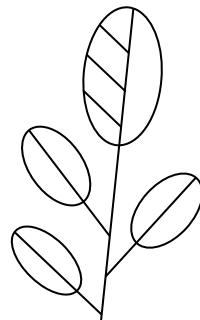
[Coroutines](#) ▶



# Добавляем plugin и зависимость

```
kotlin("plugin.serialization") version "2.0.20"  
  
implementation("org.jetbrains.kotlinx:kotlinx-serialization-json:1.7.3")
```

```
@PostMapping("/upload")  
fun postData(@RequestBody request: A): ResponseEntity<String> {  
    return ResponseEntity.ok("ok")  
}  
  
...  
  
data class A(  
    val b: Int  
)
```



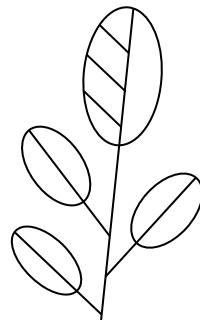


# Добавляем plugin и зависимость

```
kotlin("plugin.serialization") version "2.0.20"  
  
implementation("org.jetbrains.kotlinx:kotlinx-serialization-json:1.7.3")
```

```
@PostMapping("/upload")  
fun postData(@RequestBody request: A): ResponseEntity<String> {  
    return ResponseEntity.ok("ok")  
}  
  
...  
  
// без аннотации @Serializable  
data class A(  
    val b: Int  
)
```

Опять Jackson?



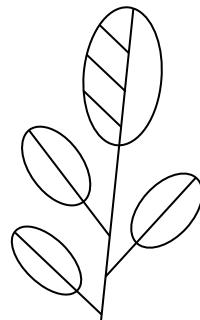


# Добавляем plugin и зависимость

```
kotlin("plugin.serialization") version "2.0.20"  
  
implementation("org.jetbrains.kotlinx:kotlinx-serialization-json:1.7.3")
```

```
@PostMapping("/upload")  
fun postData(@RequestBody request: A): ResponseEntity<String> {  
    return ResponseEntity.ok("ok")  
}  
  
...  
  
// без аннотации @Serializable  
data class A(  
    val b: Int  
)
```

ДА!

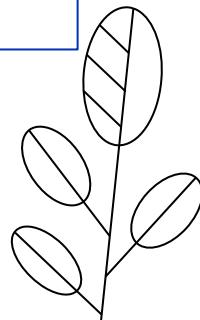




## Смотрим в код Spring'a: `messageConverters`

```
kotlinSerializationJsonPresent =  
    ClassUtils.isPresent("kotlinx.serialization.json.Json", classLoader);
```

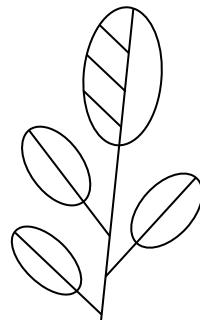
```
if (kotlinSerializationJsonPresent) {  
    this.messageConverters.add(new KotlinSerializationJsonHttpMessageConverter());  
}  
  
if (jackson2Present) {  
    this.messageConverters.add(new MappingJackson2HttpMessageConverter());  
}
```

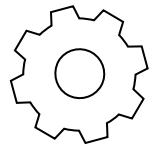




## Смотрим в код Spring'a: messageConverters

```
if (kotlinSerializationJsonPresent) {  
    this.messageConverters.add(new KotlinSerializationJsonHttpMessageConverter());  
}  
  
if (jackson2Present) {  
    this.messageConverters.add(new MappingJackson2HttpMessageConverter());  
}  
else if (gsonPresent) {  
    this.messageConverters.add(new GsonHttpMessageConverter());  
}  
else if (jsonbPresent) {  
    this.messageConverters.add(new JsonbHttpMessageConverter());  
}
```





Как этот список `messageConverters`  
используется?

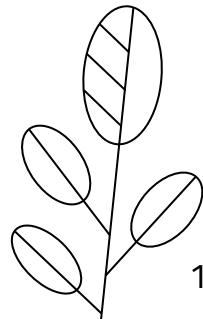




Смотрим в код Spring'a: **AbstractMessageConverterMethodArgumentResolver**

(!) Очень упрощенный псевдокод

```
<T> Object readWithMessageConverters(...) {  
    ...  
    return body;  
}
```

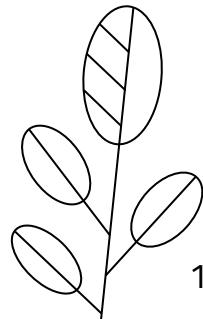




Смотрим в код Spring'a: `AbstractMessageConverterMethodArgumentResolver`

(!) Очень упрощенный псевдокод

```
<T> Object readWithMessageConverters(...) {  
  
    MediaType contentType = inputMessage.getHeaders().getContentType();  
  
    ...  
  
    return body;  
}
```

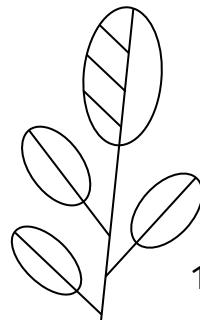




Смотрим в код Spring'a: `AbstractMessageConverterMethodArgumentResolver`

(!) Очень упрощенный псевдокод

```
<T> Object readWithMessageConverters(...) {  
  
    MediaType contentType = inputMessage.getHeaders().getContentType();  
  
    for (HttpMessageConverter<?> converter : this.messageConverters) {  
        if (converter.canRead(targetClass, contentType)) {  
  
            return body;  
    }  
}
```

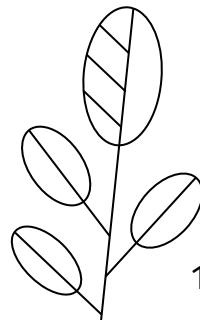




Смотрим в код Spring'a: `AbstractMessageConverterMethodArgumentResolver`

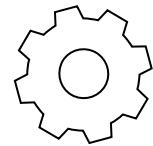
(!) Очень упрощенный псевдокод

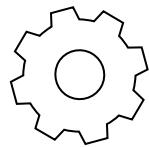
```
<T> Object readWithMessageConverters(...) {  
  
    MediaType contentType = inputMessage.getHeaders().getContentType();  
  
    for (HttpMessageConverter<?> converter : this.messageConverters) {  
        if (converter.canRead(targetClass, contentType)) {  
            body = converter.read(targetClass, msg);  
            break;  
        }  
    }  
  
    return body;  
}
```





Оффтоп: Как можно нам, **работягам**, добавить свой **converter**?





# Оффтоп: Как можно нам, **работягам**, добавить свой converter?



Why Spring ▾ Learn ▾ Projects ▾ Academy ▾ So



## HttpMessageConverter

Spring Boot ...

3.3.4

---

Search ⌘ + k

Overview

Documentation

Community

System Requirements

Installing Spring Boot

Upgrading Spring Boot

> Tutorials

▼ Reference

> Developing with Spring Boot

> Core Features

▼ Web

Servlet Web Applications

Reactive Web Applications

Graceful Shutdown

Spring Security

Spring Session

Spring Boot / Reference / Web / Servlet Web Applications

### HttpMessageConverters

Spring MVC uses the `HttpMessageConverter` interface to convert HTTP requests and responses. Sensible defaults are included out of the box. For example, objects can be automatically converted to JSON (by using the Jackson library) or XML (by using the Jackson XML extension, if available, or by using JAXB if the Jackson XML extension is not available). By default, strings are encoded in UTF-8 .

Any `HttpMessageConverter` bean that is present in the context is added to the list of converters. You can also override default converters in the same way.

If you need to add or customize converters, you can use Spring Boot's `HttpMessageConverters` class, as shown in the following listing:

Java    Kotlin

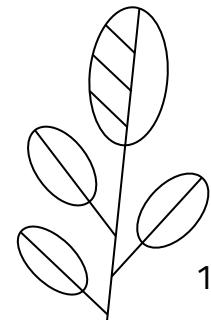
```
@Configuration(proxyBeanMethods = false)
public class MyHttpMessageConvertersConfiguration {

    @Bean
    public HttpMessageConverters customConverters() {
        HttpMessageConverter<?> additional = new AdditionalHttpMessageConverter();
        HttpMessageConverter<?> another = new AnotherHttpMessageConverter();
        return new HttpMessageConverters(additional, another);
    }
}
```

JAVA



Смотрим в код Spring'a: а как добавили kotlinx в Spring?

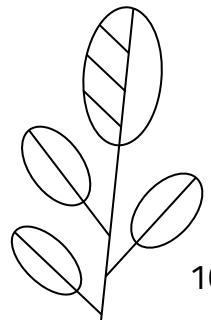




## Смотрим в код Spring'a: а как добавили kotlinx в Spring?

```
abstract class AbstractKotlinSerializationHttpMessageConverter
```

1. Имплементирован метод: `boolean canRead()`





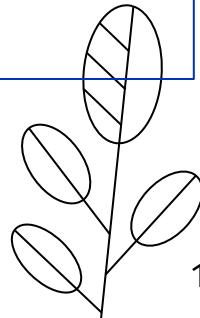
## Смотрим в код Spring'a: а как добавили kotlinx в Spring?

```
abstract class AbstractKotlinSerializationHttpMessageConverter
```

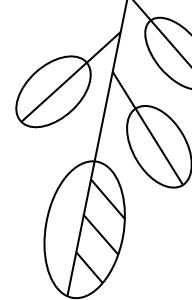
1. Имплементирован метод: `boolean canRead()`

```
KSerializer<Object> serializer = this.kTypeSerializerCache.get(type);  
...  
if (serializer == null) {  
    serializer = SerializersKt.serializerOrNull(this.format.getSerializersModule(),  
type);  
}  
...  
this.kTypeSerializerCache.put(type, serializer);
```

Наш знакомый из прошлой главы,  
который находит сгенерированный  
сериализатор



## Смотрим в код Spring'a: элегантность Kotlinx

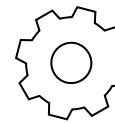


2. Имплементирован метод: `read()`

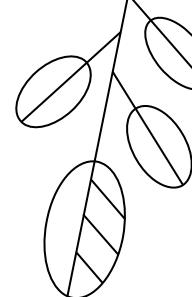
Abstract Kotlin Converter

Binary Converter

String Converter



# Смотрим в код Spring'a: элегантность Kotlinx



2. Имплементирован метод: `read()`

`Abstract Kotlin Converter`

`Binary Converter`

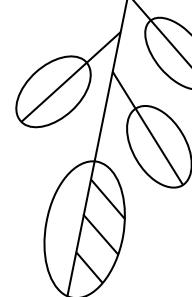
`String Converter`

```
format.decodeFromByteArray(  
    serializer,  
    StreamUtils.copyToByteArray(inputMessage.getBody())  
)
```

```
format.decodeFromString(  
    serializer,  
    StreamUtils.copyToString(inputMessage.getBody(), charset);  
)
```



# Смотрим в код Spring'a: элегантность Kotlinx



2. Имплементирован метод: `read()`

`Abstract Kotlin Converter`

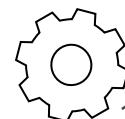
`Binary Converter`

`String Converter`

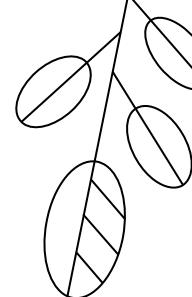
```
format.decodeFromByteArray(  
    serializer,  
    StreamUtils.copyToByteArray(inputMessage.getBody())  
)
```

```
format.decodeFromString(  
    serializer,  
    StreamUtils.copyToString(inputMessage.getBody(), charset);  
)
```

**format** - это отдельные библиотеки  
(CBOR, PROTOBUF, JSON) на базе общих интерфейсов kotlinx



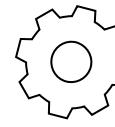
## Смотрим в код Spring'a: элегантность Kotlinx



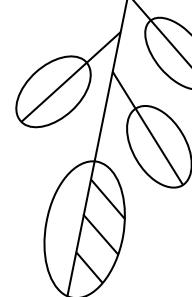
```
public KotlinSerializationProtobufHttpMessageConverter(ProtoBuf protobuf) {
    super(ProtoBuf.Default, MediaType.APPLICATION_PROTOBUF, MediaType.APPLICATION_OCTET_STREAM,
          new MediaType("application", "vnd.google.protobuf"));
}

public KotlinSerializationJsonHttpMessageConverter() {
    super(Json.Default, MediaType.APPLICATION_JSON, new MediaType("application", "*+json"));
}

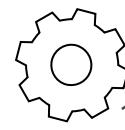
public KotlinSerializationCborHttpMessageConverter(Cbor cbor) {
    super(Cbor.Default, MediaType.APPLICATION_CBOR);
}
```



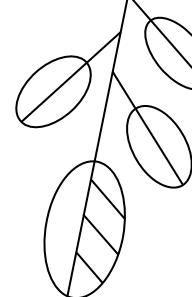
## Смотрим в код Spring'a: элегантность Kotlinx



```
public KotlinSerializationProtobufHttpMessageConverter(ProtoBuf protobuf) {  
    super(ProtoBuf.Default, MediaType.APPLICATION_PROTOBUF, MediaType.APPLICATION_OCTET_STREAM,  
          new MediaType("application", "vnd.google.protobuf"));  
}  
  
public KotlinSerializationJsonHttpMessageConverter() {  
    super(Json.Default, MediaType.APPLICATION_JSON, new MediaType("application", "*+json"));  
}  
  
public KotlinSerializationCborHttpMessageConverter(Cbor cbor) {  
    super(Cbor.Default, MediaType.APPLICATION_CBOR);  
}
```



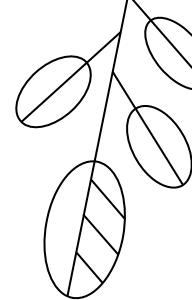
## Смотрим в код Spring'a: элегантность Kotlinx



```
public KotlinSerializationProtobufHttpMessageConverter(ProtoBuf protobuf) {  
    super(ProtoBuf.Default, MediaType.APPLICATION_PROTOBUF, MediaType.APPLICATION_OCTET_STREAM,  
          new MediaType("application", "vnd.google.protobuf"));  
}  
  
public KotlinSerializationJsonHttpMessageConverter() {  
    super(Json.Default, MediaType.APPLICATION_JSON, new MediaType("application", "*+json"));  
}  
  
public KotlinSerializationCborHttpMessageConverter(Cbor cbor) {  
    super(Cbor.Default, MediaType.APPLICATION_CBOR);  
}
```



# Откуда универсальность?



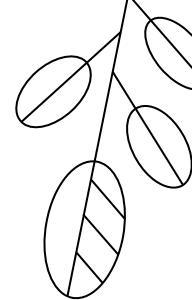
- Единые методы интерфейсов Format'ов:

```
public fun <T> encodeToString(serializer: SerializationStrategy<T>, value: T): String
```

```
public fun <T> decodeFromString(deserializer: DeserializationStrategy<T>, string: String): T
```



# Откуда универсальность?



- Единые методы интерфейсов Format'ов:

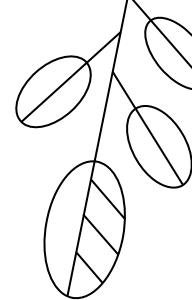
```
public fun <T> encodeToString(serializer: SerializationStrategy<T>, value: T): String
```

```
public fun <T> decodeFromString(deserializer: DeserializationStrategy<T>, string: String): T
```

- Единым образом генерируется код компилятором для KSerializer



# Откуда универсальность?

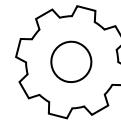


- Единые методы интерфейсов Format'ов:

```
public fun <T> encodeToString(serializer: SerializationStrategy<T>, value: T): String
```

```
public fun <T> decodeFromString(deserializer: DeserializationStrategy<T>, string: String): T
```

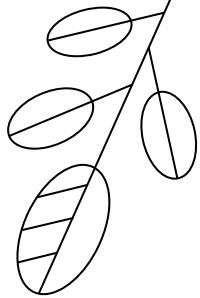
- Единым образом генерируется код компилятором для KSerializer
- Единые методы и организация логики Decoder/Encoder





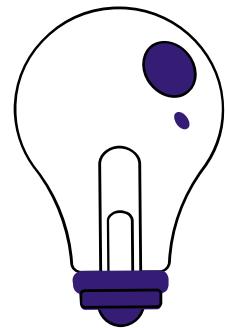
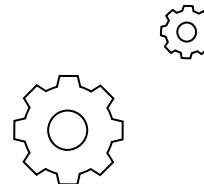
Как еще можно этим удобством  
можно воспользоваться?



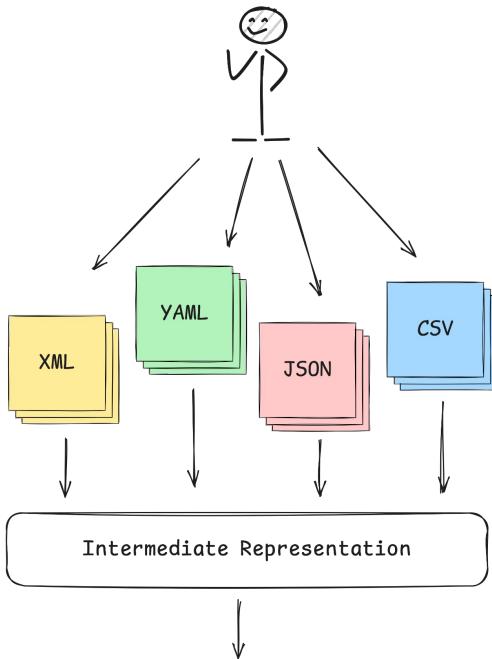


04

## Реальный сценарий

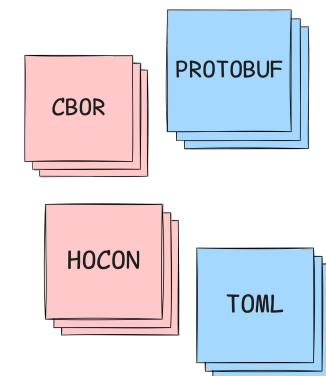


# Наша проблема



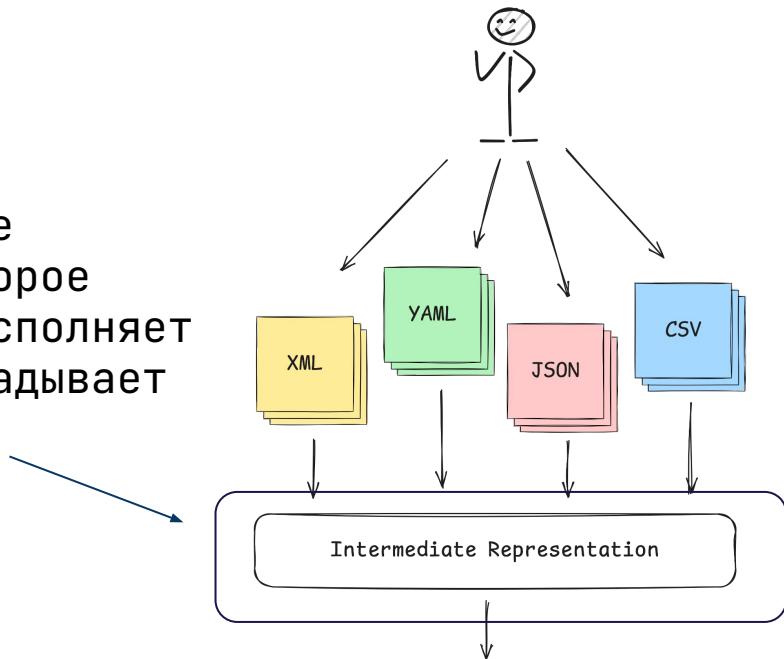
вероятно бинарные  
и нестандартные  
форматы

+



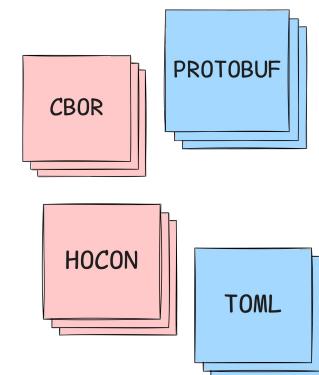
# Наша проблема

Spring Boot'овое  
приложение, которое  
десериализует, исполняет  
логику и перекладывает  
в другой формат

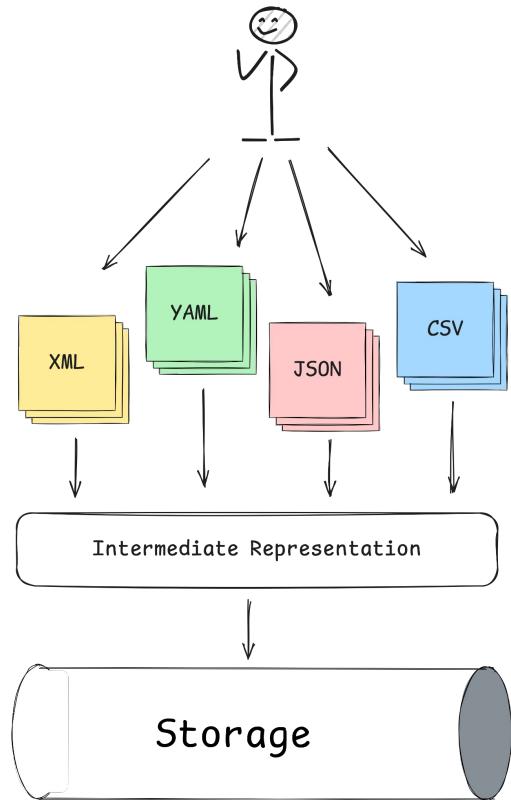


вероятно бинарные  
и нестандартные  
форматы

+

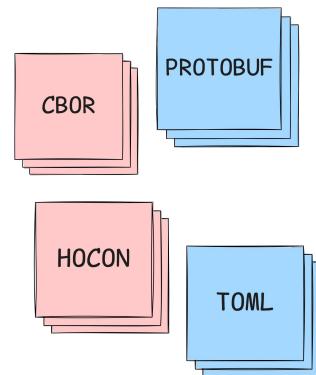


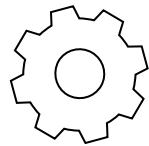
# Наша проблема



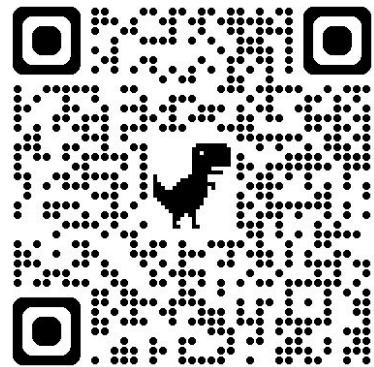
вероятно бинарные  
и нестандартные  
форматы

+



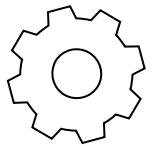


А вот в качестве IR у нас не просто  
схема из кода, а **AVRO**

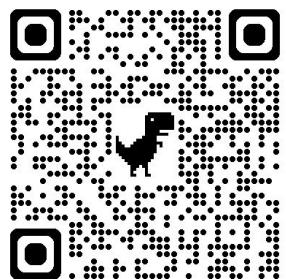




# Наш IR: AVRO schema

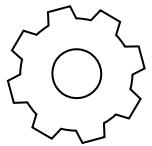


```
{  
    "type" : "record",  
    "name" : "Message",  
    "namespace" : "schema.namespace",  
    "fields" : [  
        {  
            "name" : "title",  
            "type" : [ "null", "string" ],  
            "default": null  
        },  
        {  
            "name" : "message",  
    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





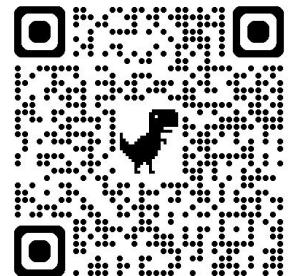
# Наш IR: AVRO schema



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

имена

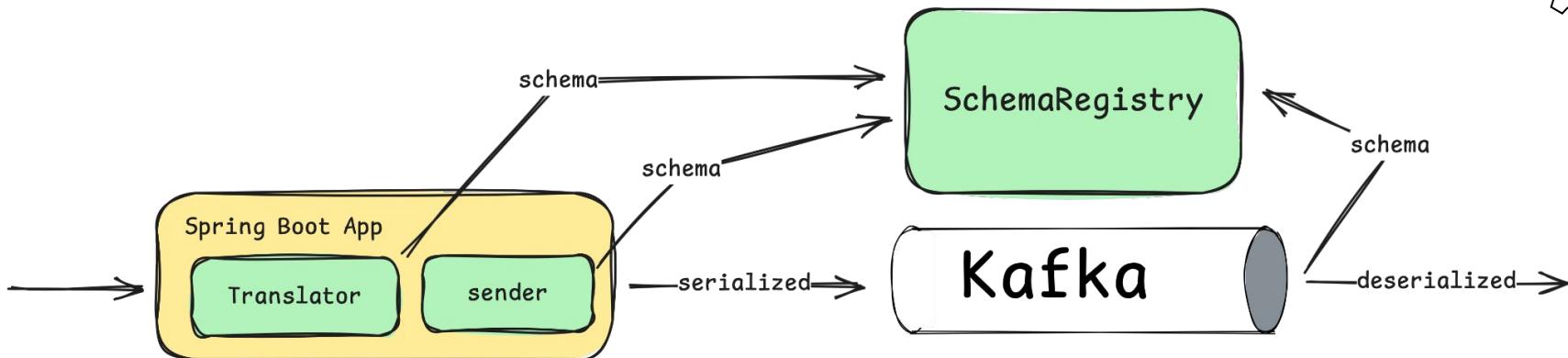
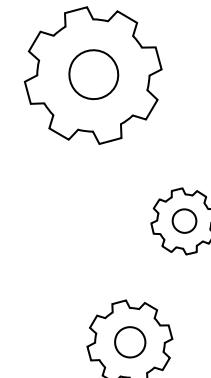
типы





# Наш IR: AVRO

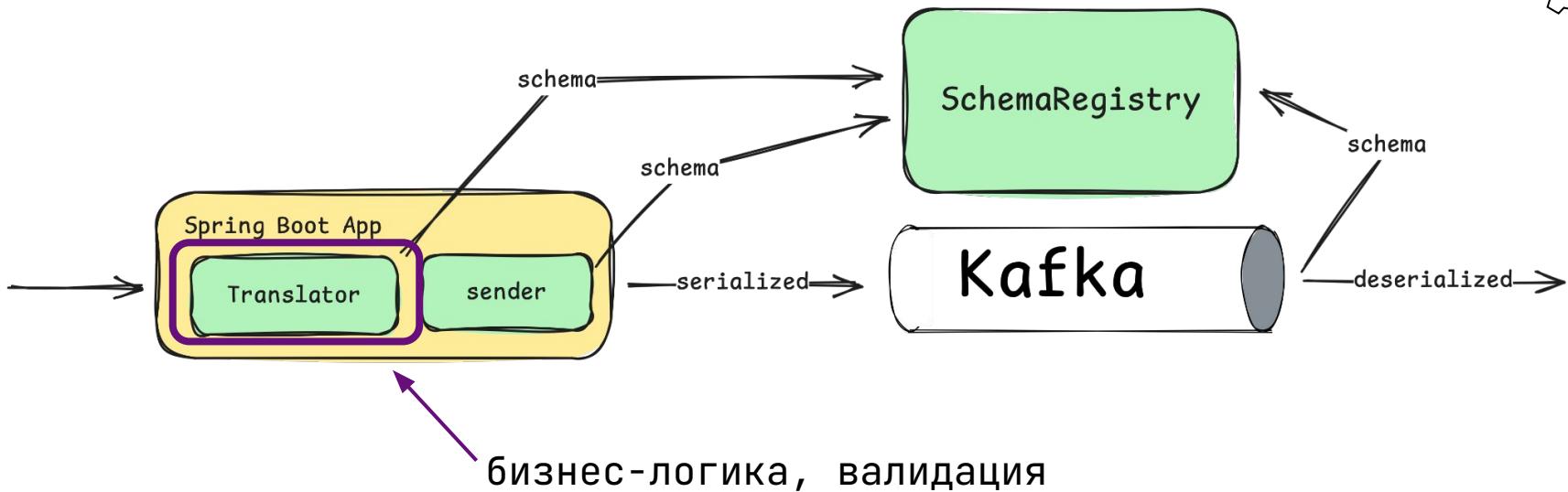
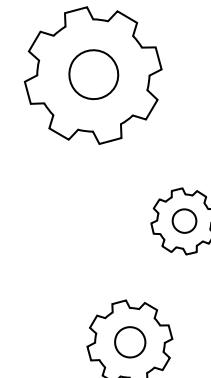
## почему?





# Наш IR: AVRO

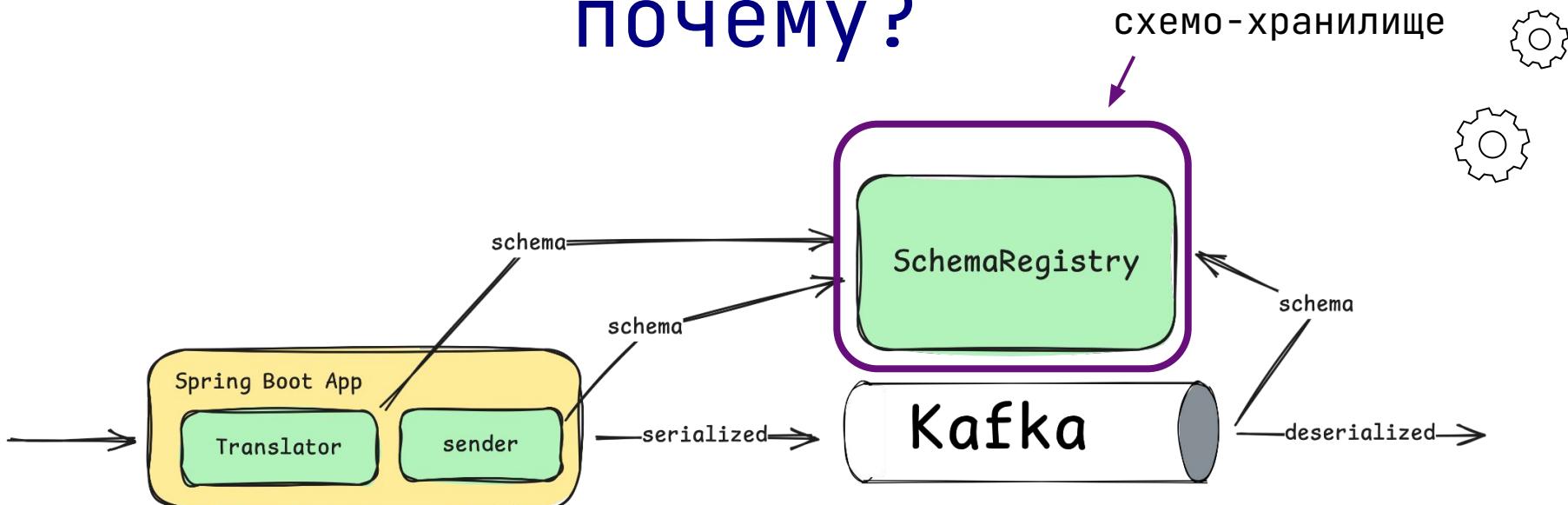
## почему?



бизнес-логика, валидация

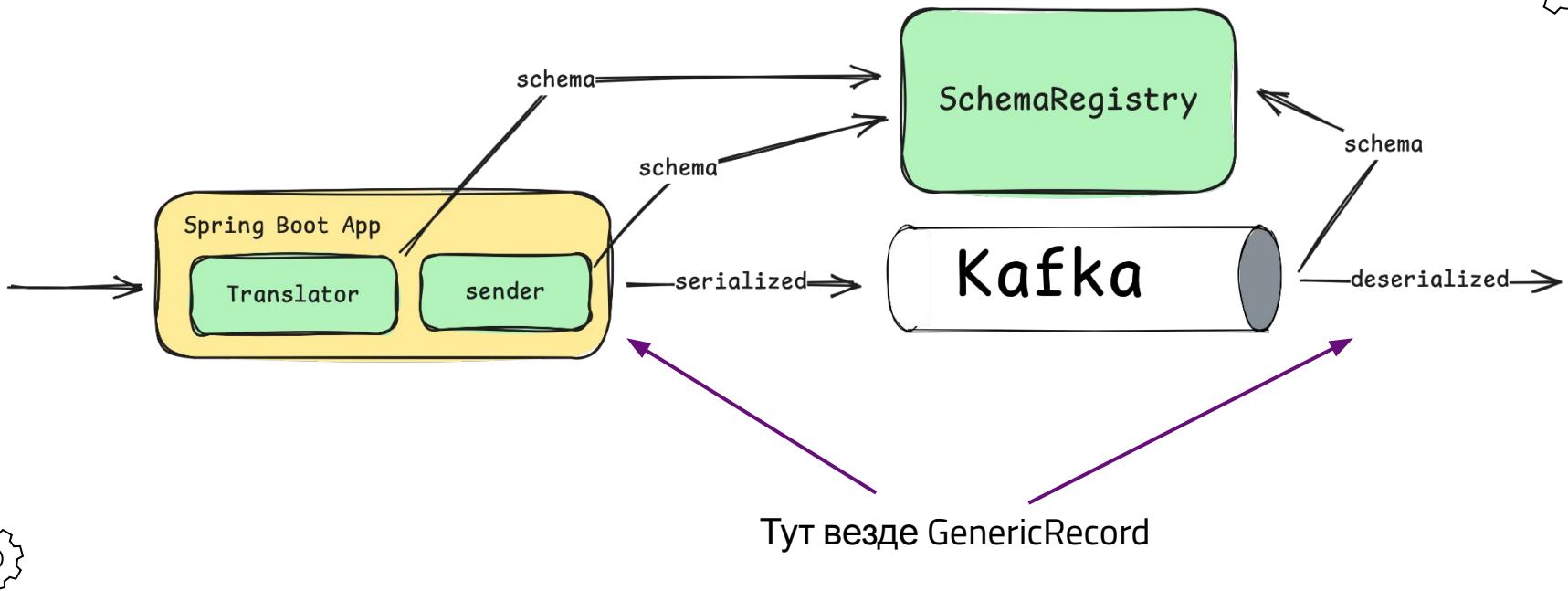
# Наш IR: AVRO

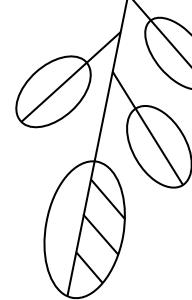
## почему?



# Наш IR: AVRO

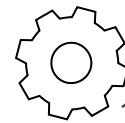
## почему?





В чем разница?

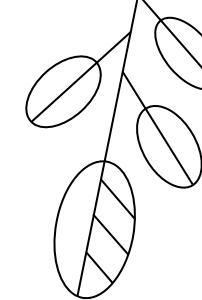
В схеме!



# Реальный сценарий

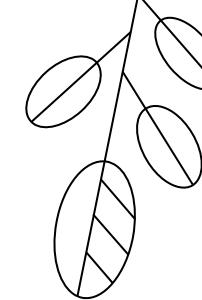
```
@PostMapping("/single")
fun postSingle(
    @RequestBody message: ByteArray,
    @RequestHeader(HttpHeaders.CONTENT_TYPE) contentType: MediaType,
    @RequestHeader(HttpHeaders.ACCEPT, required = false) accept: List<MediaType>?,
): ResponseEntity<Any> {

}
```



# 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
}
```

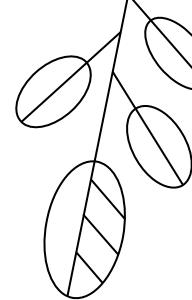


# Реальный сценарий

```
@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

}
```

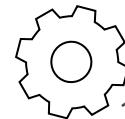
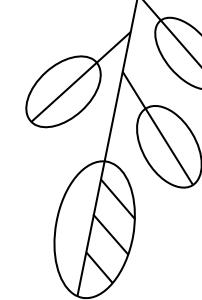


# Реальный сценарий

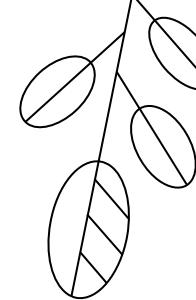
```
@PostMapping("/single")
fun postSingle(
    @RequestBody message: ByteArray,
    @RequestHeader(HttpHeaders.CONTENT_TYPE) contentType: MediaType,
    @RequestHeader(HttpHeaders.ACCEPT, required = false) accept: List<MediaType>?,
): ResponseEntity<Any> {
    // 1) используем определенную заранее схему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

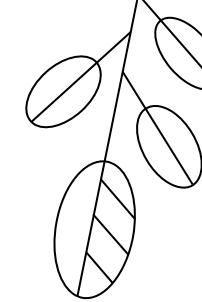


Передаем схему в `KSerializer`, замещаем  
**сгенерированный** код



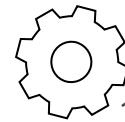
# “Обманываем” kotlinx

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



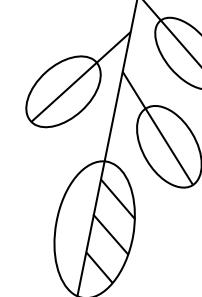
# “Обманываем” kotlinx

```
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()  
            ...  
        }  
    }
```



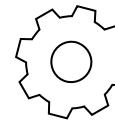
# “Обманываем” kotlinx

```
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)  
            )  
            ...  
        }  
    }
```



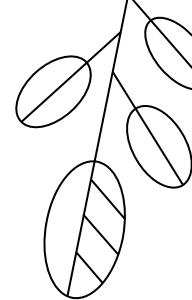
# “Обманываем” kotlinx

```
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
    }
}
```



```
serialFormat.decodeFromXXX(  
    AvroSchemaDeserializer(schema),  
    Input  
)
```

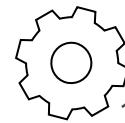




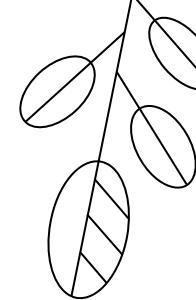
А форматы мапим примерно как в Spring



```
serialFormat.decodeFromXXX(  
    AvroSchemaDeserializer(schema),  
    Input  
)
```



# Какие вопросы остаются для отдельного обсуждения?



Производительность

Циклические графы

Работа с ошибками

Некоторые форматы  
с нестандартными  
подходами

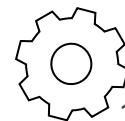
Безопасность

Конфигурация форматов

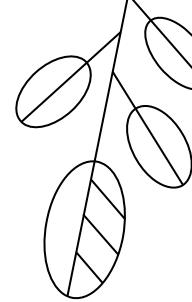
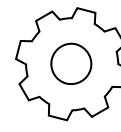
Встраивание напрямую в контроллер

Валидации

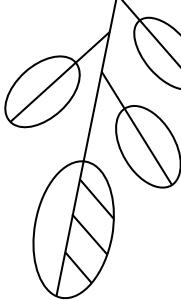
Логические типы



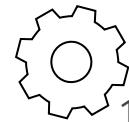
# Что мы сегодня поняли?



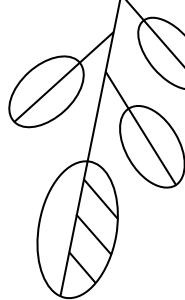
# Мы сегодня многое поняли



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



# Мы сегодня многое поняли

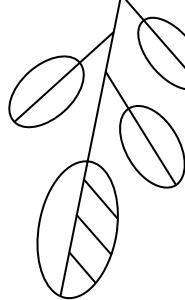


- Что `Koltinx.serialization` - продвинутый инструмент сериализации, особенно, если у вас мультиформатная работа
- Какой код `генерируется компилятором` и почему это важно знать



147

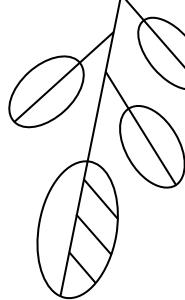
# Мы сегодня многое поняли



- Что `Kotlinx.serialization` - продвинутый инструмент сериализации, особенно, если у вас мультиформатная работа
- Какой код `генерируется компилятором` и почему это важно знать
- Что происходит в `Spring` с сериализацией и как туда добавлен `kotlinx`



# Мы сегодня многое поняли



- Что `Kotlinx.serialization` - продвинутый инструмент сериализации, особенно, если у вас мультиформатная работа
- Какой код `генерируется компилятором` и почему это важно знать
- Что происходит в `Spring` с сериализацией и как туда добавлен `kotlinx`
- Как можно сэкономить время за счет переиспользования экосистемы



# И спасибо, что продержались до конца!

Давайте дружить на Github:

