CrossConf

Kotlinx.serialization: Сериализуют все!

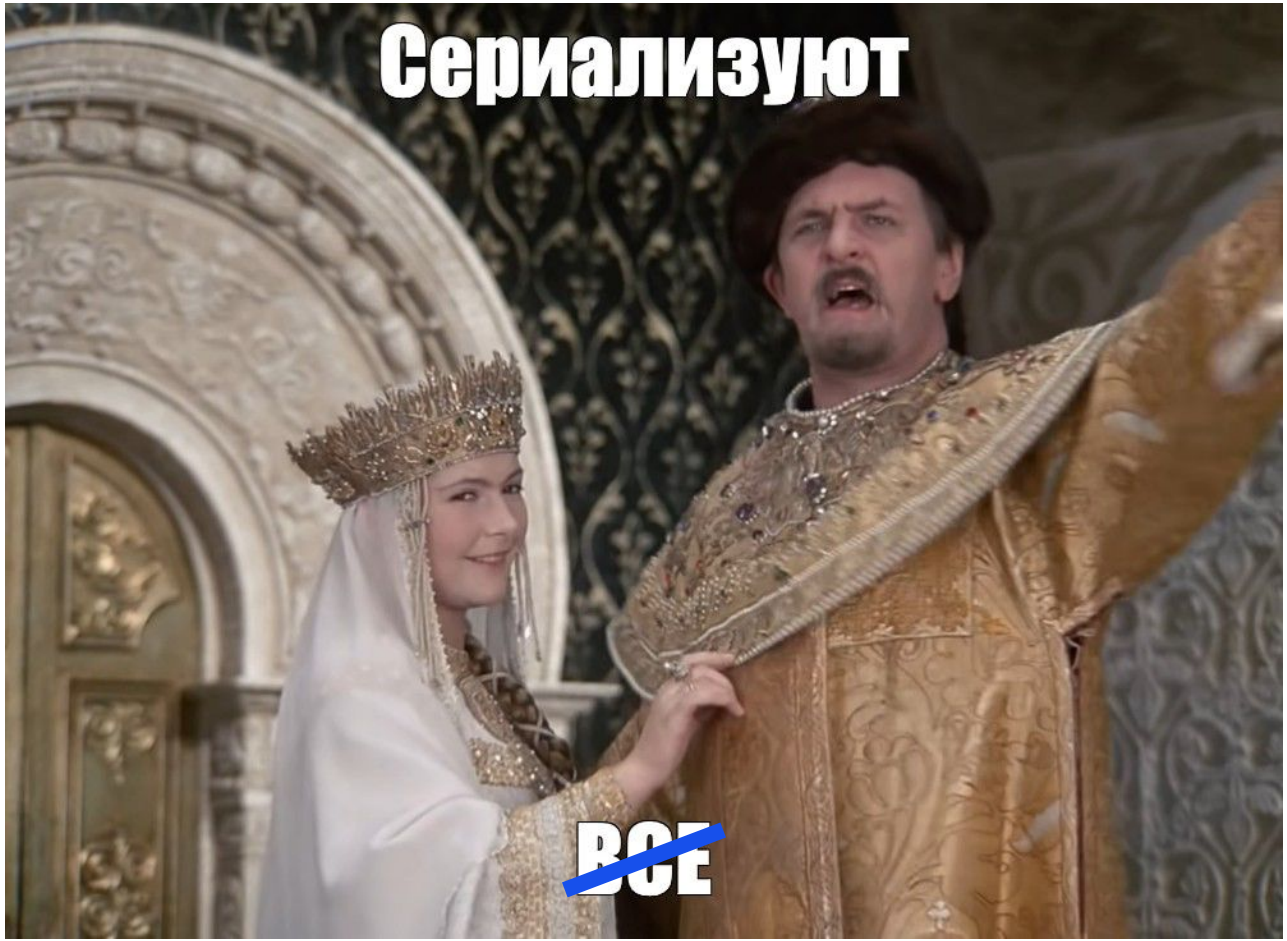
Кулешов Андрей

Positive Technologies

Руководитель разработки

8 ноября 2024 | crossconf.com


Сериализуют
ВСЕ

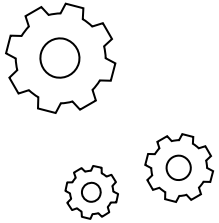
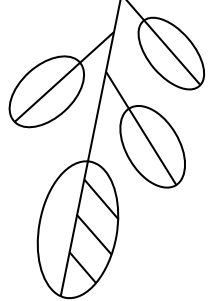

Сериализуют
все


Сериализуют
ВСЁ

$ whoami?

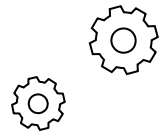
Андрей Кулешов

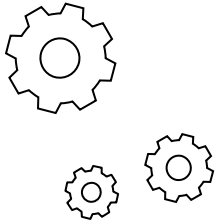
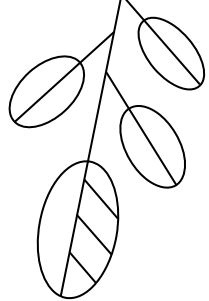
intel
Deutsche Bank
HUAWEI
pt
Стачка
phd 2
JPoint
SafeCode
Joker<?>
CrossConf
SIGA CONF
sysconf

О чем поговорим?

kotlinx.serialization

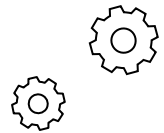



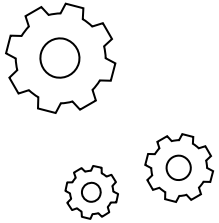



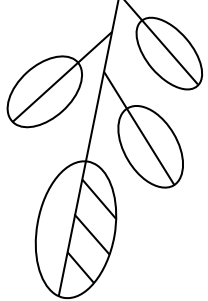
О чем поговорим?

kotlinx.serialization

что это за зверь



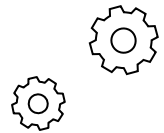


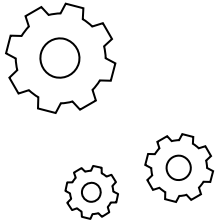
О чем поговорим?

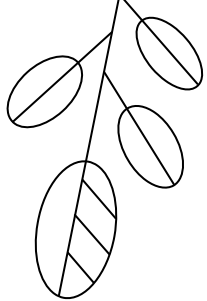
kotlinx.serialization

что это за зверь

что там под капотом







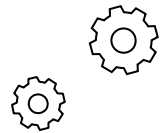
О чем поговорим?

kotlinx.serialization

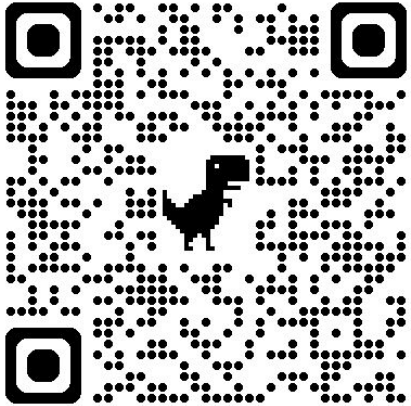
что это за зверь

что там под капотом

как это кастомизировать

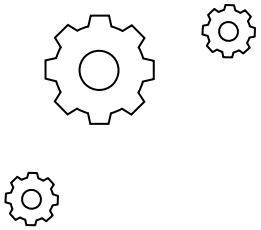


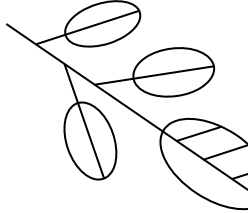
Почему я?



[KT]oml

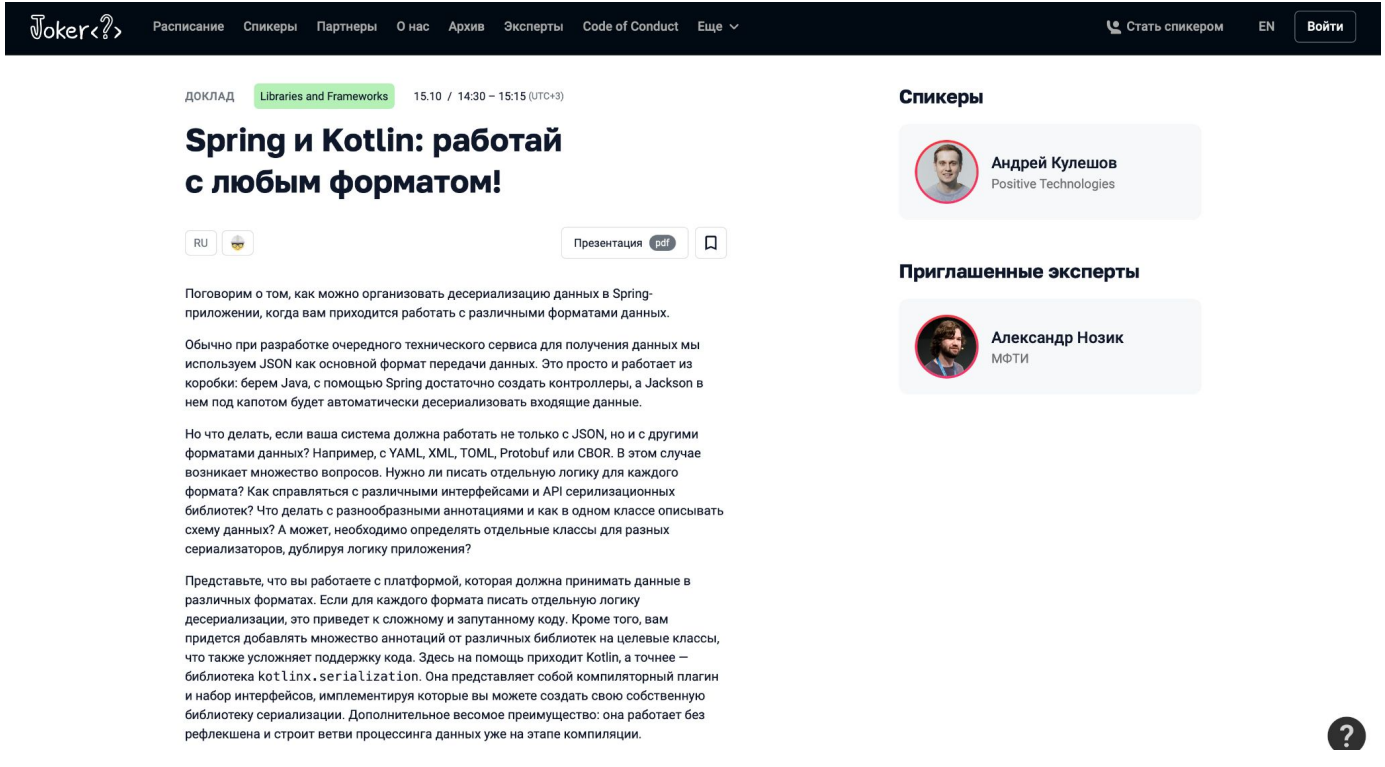
powered by kotlinx.serialization





Предыстория

Kotlinx в Spring-backend мире



Joker<?> Расписание Спикеры Партнеры О нас Архив Эксперты Code of Conduct Еще Стать спикером EN Войти

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

Spring и Kotlin: работай с любым форматом!

RU

Презентация pdf

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

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

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

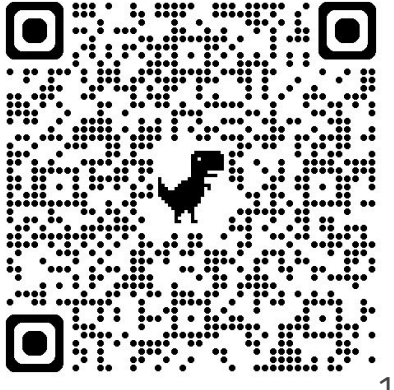
Представьте, что вы работаете с платформой, которая должна принимать данные в различных форматах. Если для каждого формата писать отдельную логику десериализации, это приведет к сложному и запутанному коду. Кроме того, вам придется добавлять множество аннотаций от различных библиотек на целевые классы,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

Спикеры

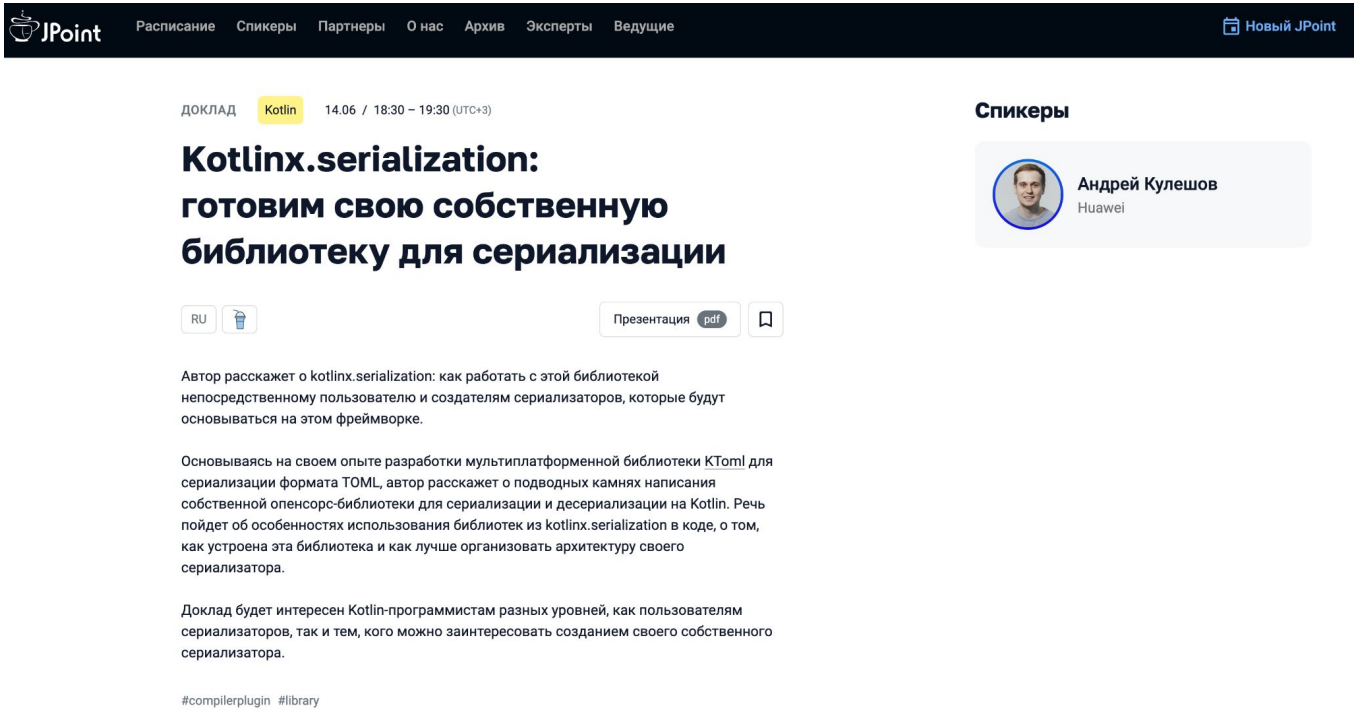
Андрей Кулешов
Positive Technologies

Приглашенные эксперты

Александр Нозик
МФТИ



Базовые вещи с со стороны разработчиков сериализаторов



Сегодня снова не будет 😭

@guai9632 1 year ago

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

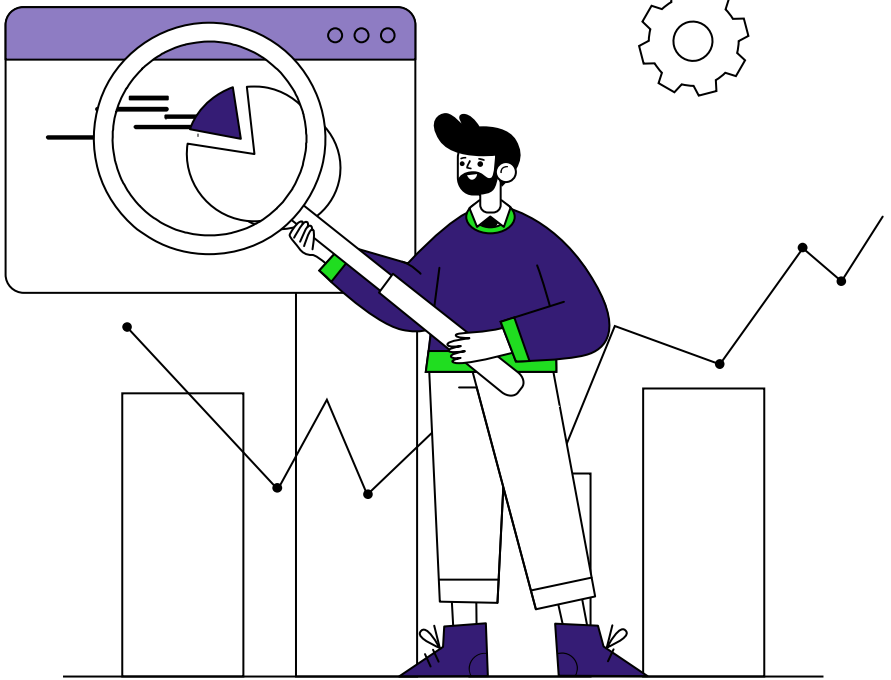
2 **Reply**

@telephon3208 1 year ago

бесценный доклад

1 **Reply**

01 Проблема



У вас есть данные

Текстовый формат

Бинарный формат

У данных есть схема

Имена

Типы

Формат

Надо данные разобрать
и работать с
ними внутри программы

Схема внутри кода

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

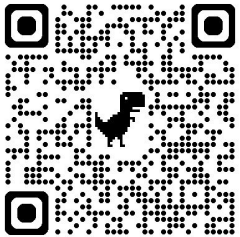


Схема внутри кода

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

типы

имена

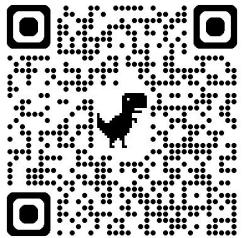


Схема внутри кода

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

типы

имена

Код как схема данных

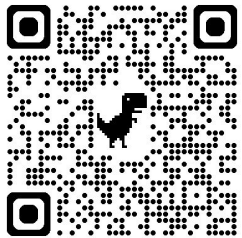
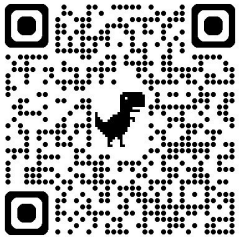
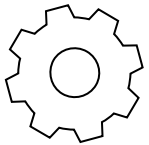


Схема внутри кода

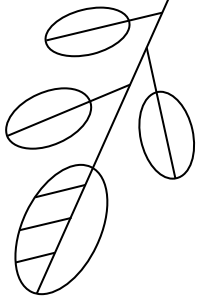
```
@Serializable
data class MyClass(
    val someBooleanProperty: Boolean,
    val table1: Table1,
    val table2: Table2,
    @SerialName("gradle-libs-like-property")
    val kotlinJvm: GradlePlugin,
    val myMap: Map<String, String>
)

...
```

Как это работает?

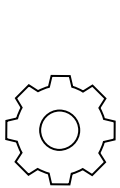


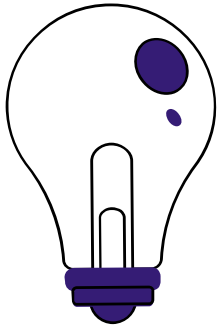



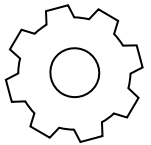
Легко, если есть возможность встраиваться в язык

02 kotlinx.serialization

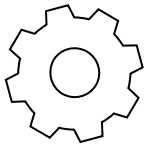




Но сначала терминология

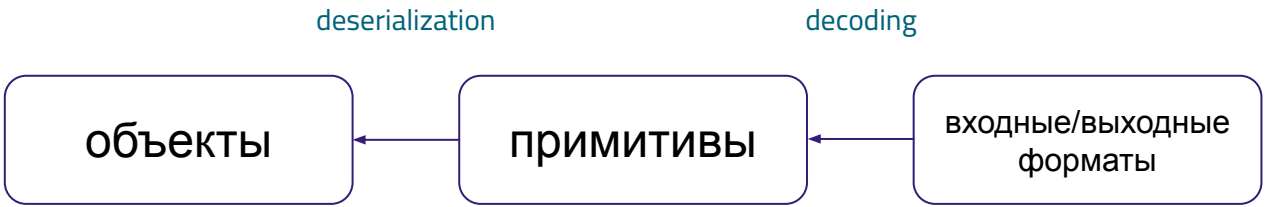


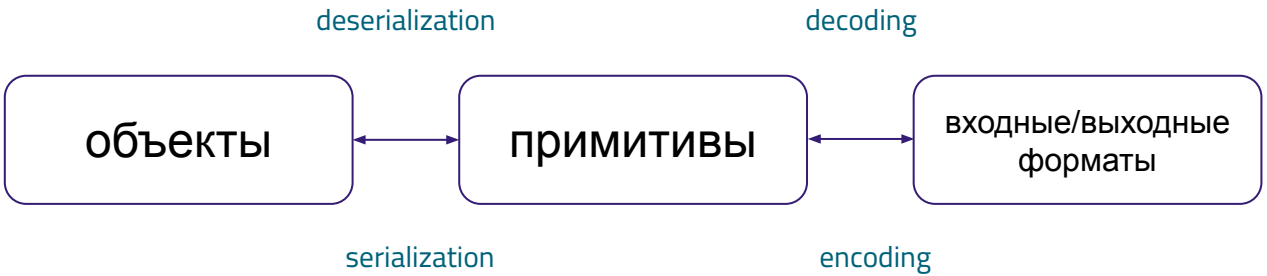


объекты

примитивы

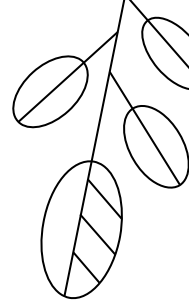
входные/выходные форматы

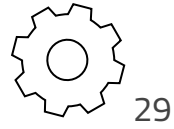


deserialization
decoding
объекты
примитивы
входные/выходные форматы


deserialization
decoding
объекты
примитивы
входные/выходные форматы
serialization
encoding

Вкратце

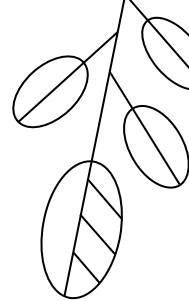
- Мультиплатформенная (**KMP**) библиотека

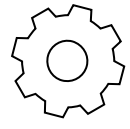




Вкратце

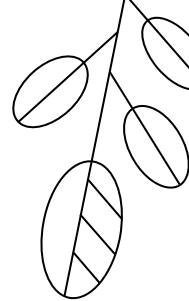
- Мультиплатформенная (**KMP**) библиотека
- Является **компиляторным плагином**

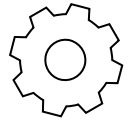




Вкратце

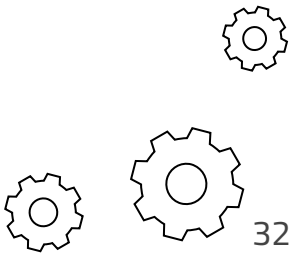
- Мультиплатформенная (**KMP**) библиотека
- Является **компиляторным плагином**
- **НЕ** использует reflection





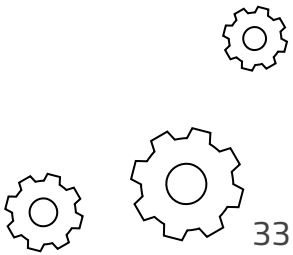
Вкратце

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



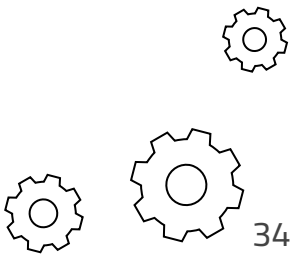
Вкратце

- 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



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



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

Еще более кратко

README | Code of conduct | Apache-2.0 license

Kotlin multiplatform / multi-format reflectionless serialization

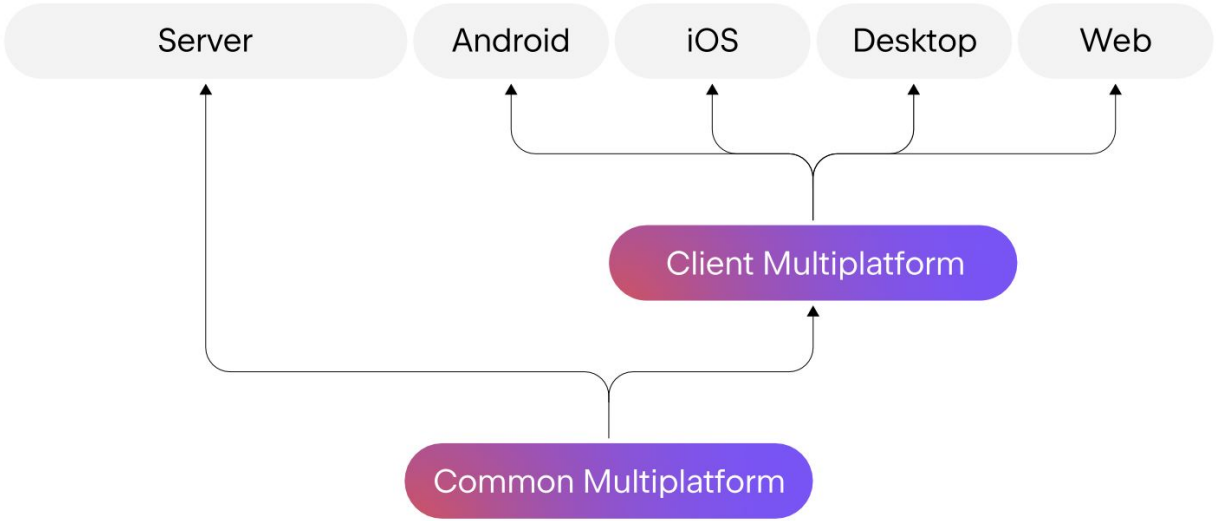
stable JetBrains official | license Apache License 2.0 | build passing | kotlin 2.0.20 | maven-central v1.7.3 | API reference KDoc | chat slack

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

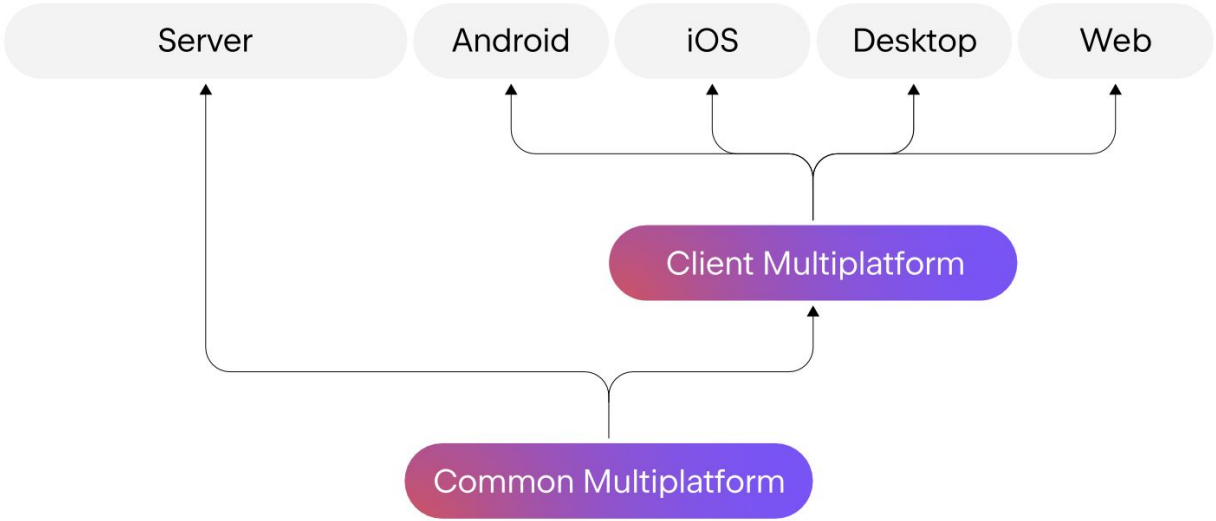
- Supports Kotlin classes marked as `@Serializable` and standard collections.
- Provides JSON, Protobuf, CBOR, Hocon and Properties formats.
- Complete multiplatform support: JVM, JS and Native.

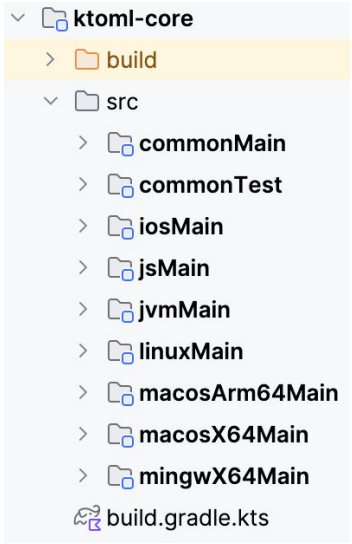
Table of contents

На всякий случай

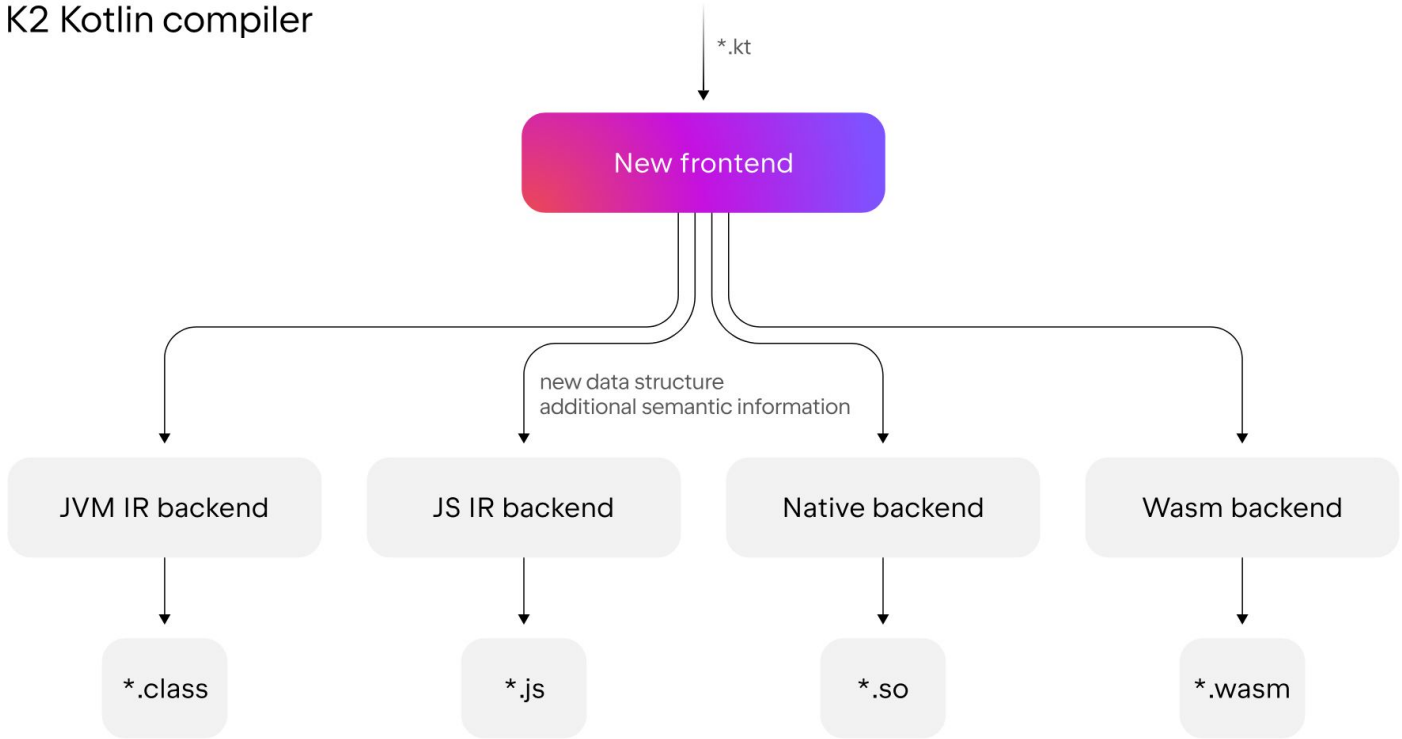


На всякий случай

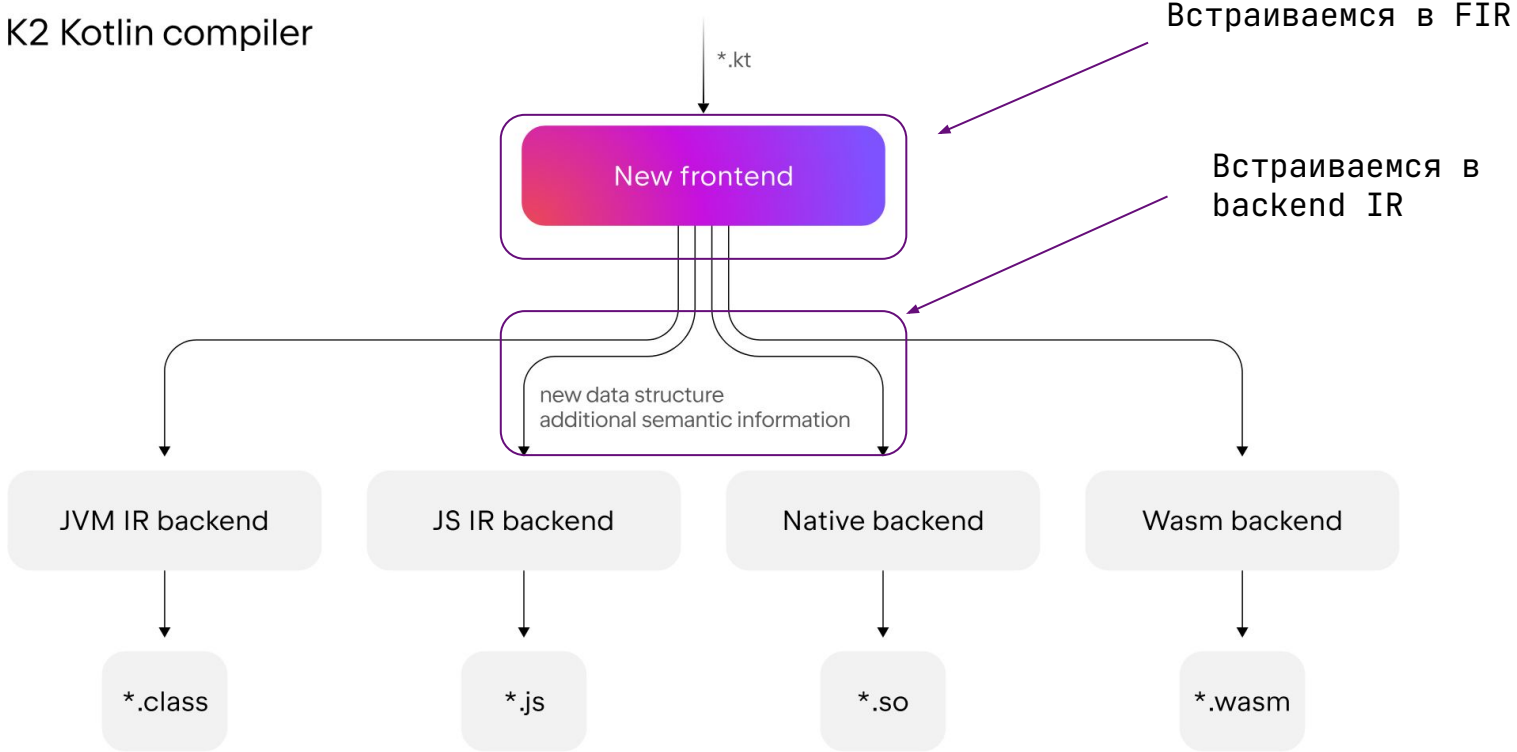




Компиляторные плагины



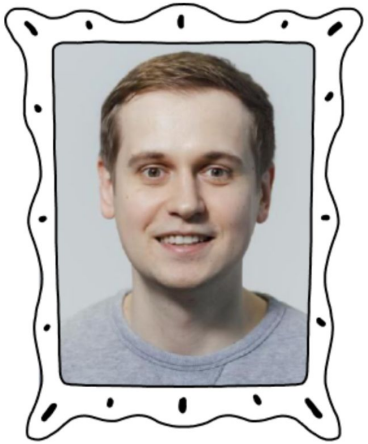
Компиляторные плагины



Компиляторные плагины

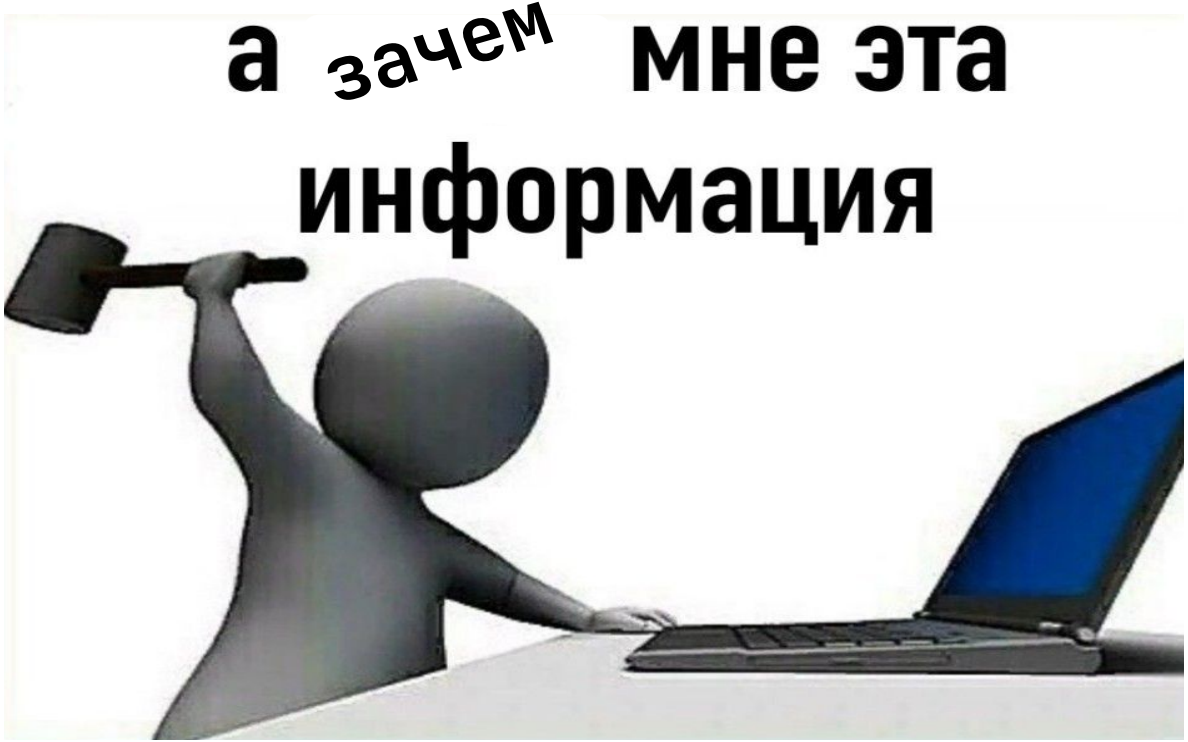
Компиляторные плагины в Котлин: почувствуй себя системным программистом

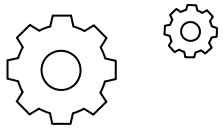
Компиляторные плагины — потрясающая фича, которая позволяет нам, обычным разработчикам, почувствовать себя в шкуре системных программистов без зазубривания теории компиляции, педантичной поддержки краевых случаев разных платформ и написания собственных лексеров с парсерами. Вместо этого мы можем сразу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



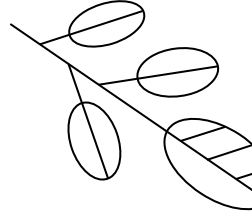
Андрей Кулешов,
Huawei

Компиляторные плагины





Пример использования

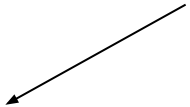


```
package com.akuleshov7



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.

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 c
9      va
10 )
11
12 fun ma
13     va
14     Js
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
)
```

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

```kotlin
package com.akuleshov7

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

@Serializable
data c
    va
)

fun ma
    va
    Js
}
```

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.

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

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

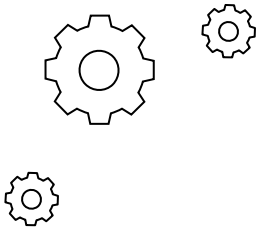
```
1  package com.akuleshov7
2
3  import kotlinx.serialization.Serializable
4  import kotlinx.serialization.encodeToString
5  import kotlinx.serialization.json.Json
6
7  @Serializable
8  data c
9      va
10 )
11
12 fun ma
13     va
14     Js
15 }
16
```

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.

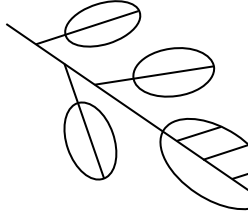
```
public constructor Serializable(
    val with: KClass<out KSerializer<*>> = KSerializer::class
)
```

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.



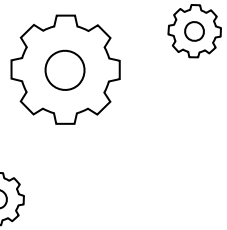
KSerializer



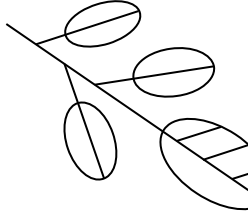
Как обнаруживается?

Зачем он вообще нужен?

Как генерируется?

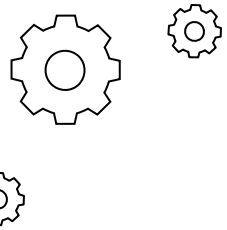


Поиск KSerializer

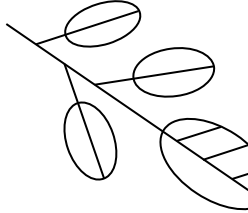


```
public fun SerializersModule.serializer(type: KType): KSerializer<Any?> =
    serializerByKTypeImpl(type, failOnMissingTypeArgSerializer = true) ?: type.kclass()
        .platformSpecificSerializerNotRegistered()
```

core/commonMain/src/kotlinx/serialization/**Serializers**.kt



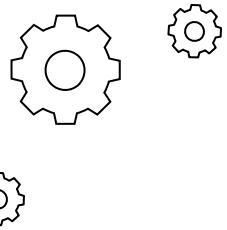
Поиск KSerializer



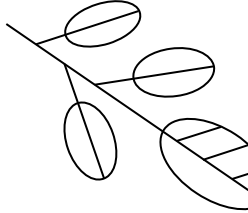
```
public fun SerializersModule.serializer(type: KType): KSerializer<Any?> =
    serializerByKTypeImpl(type, failOnMissingTypeArgSerializer = true) ?: type.kclass()
        .platformSpecificSerializerNotRegistered()
```

<… в конечном итоге приходим к **…>**

core/commonMain/src/kotlinx/serialization/**Serializers**.kt



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()
```

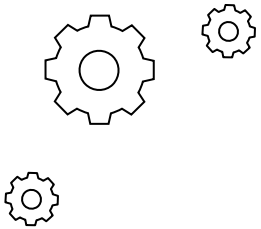
core/commonMain/src/kotlinx/serialization/**Serializers**.kt

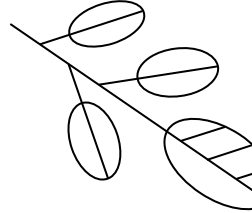
Встроенные KSerializer

```kotlin
@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())
      put(CharArray::class, CharArraySerializer())
      put(Double::class, Double.serializer())
      put(DoubleArray::class, DoubleArraySerializer())
      put(Float::class, Float.serializer())
      put(FloatArray::class, FloatArraySerializer())
```

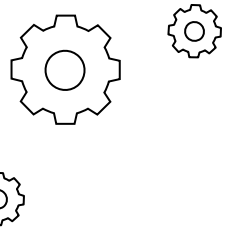
<...>

core/commonMain/src/kotlinx/serialization/**Serializers**.kt

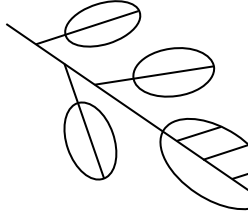




На пальцах пример для String



Пример для String



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

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

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

core/commonMain/src/kotlinx/serialization/**Serializers**.kt

Пример для String

Метаинформация (схема класса итд.)

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

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

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

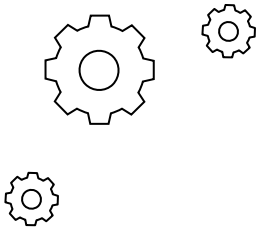
core/commonMain/src/kotlinx/serialization/**Serializers**.kt

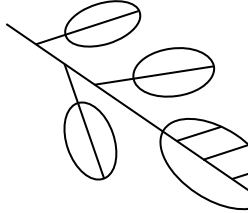
Пример для String

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))
}
```

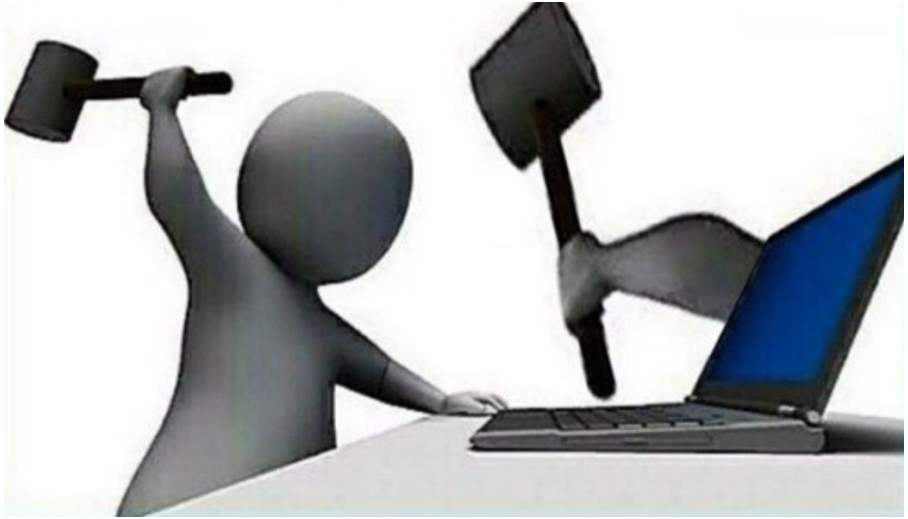
```
plugins {
    kotlin("plugin.serialization") version "2.0.20"
}
```

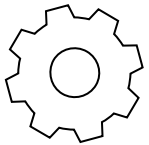
Все работает

```
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())
    println(Json.encodeToString(a))
}
```

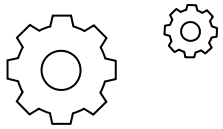
```
com.akuleshov7.A$$serializer@32d992b2
{"b":0}
```

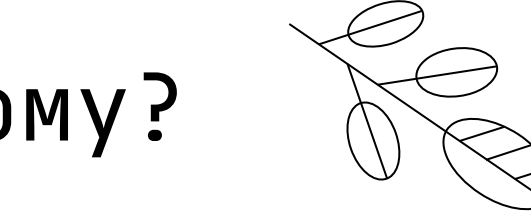
ПОЭТОМУ НУЖНО ПОНИМАТЬ ЧТО ПРОИСХОДИТ ПОД КАПОТОМ



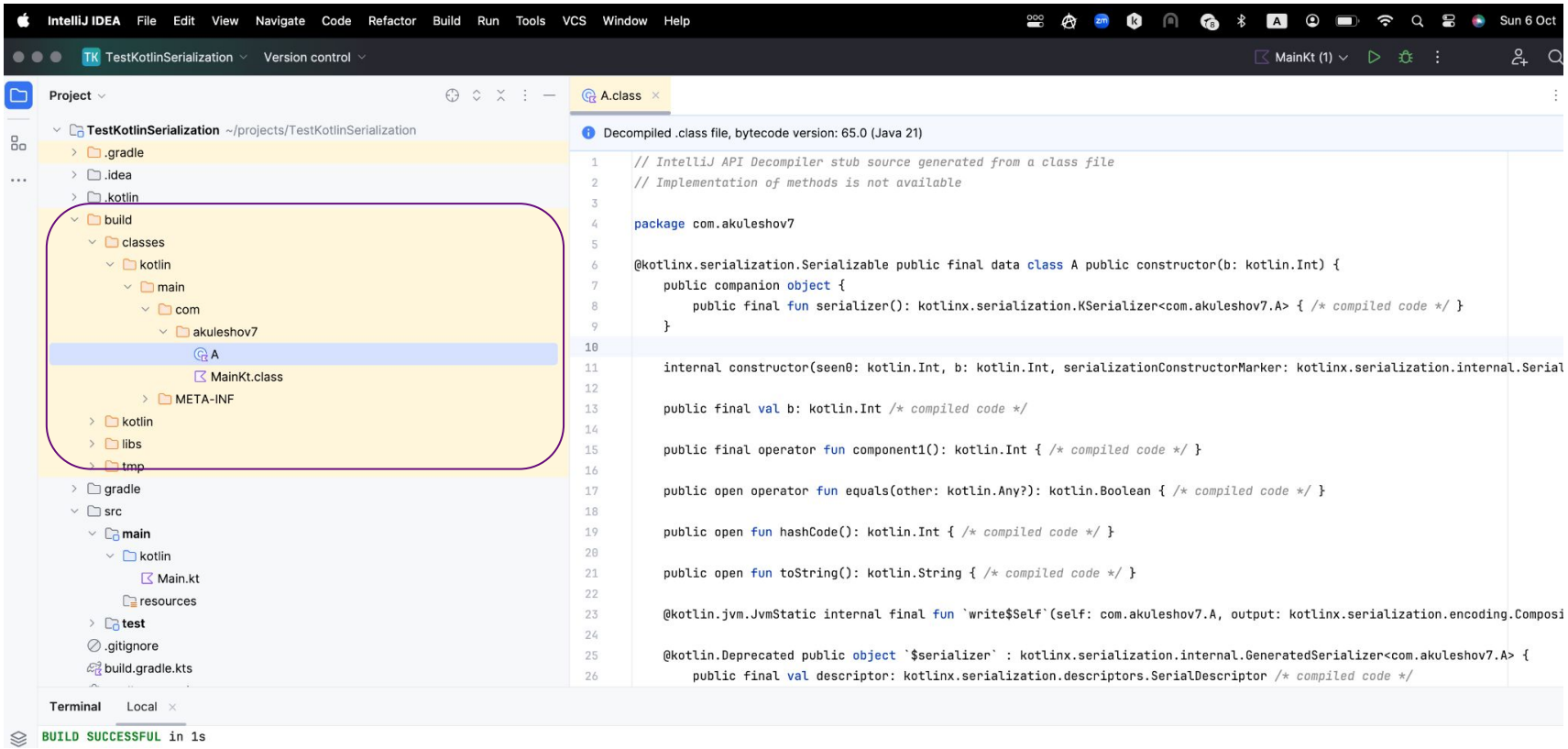
Посмотрим на сгенерированный код

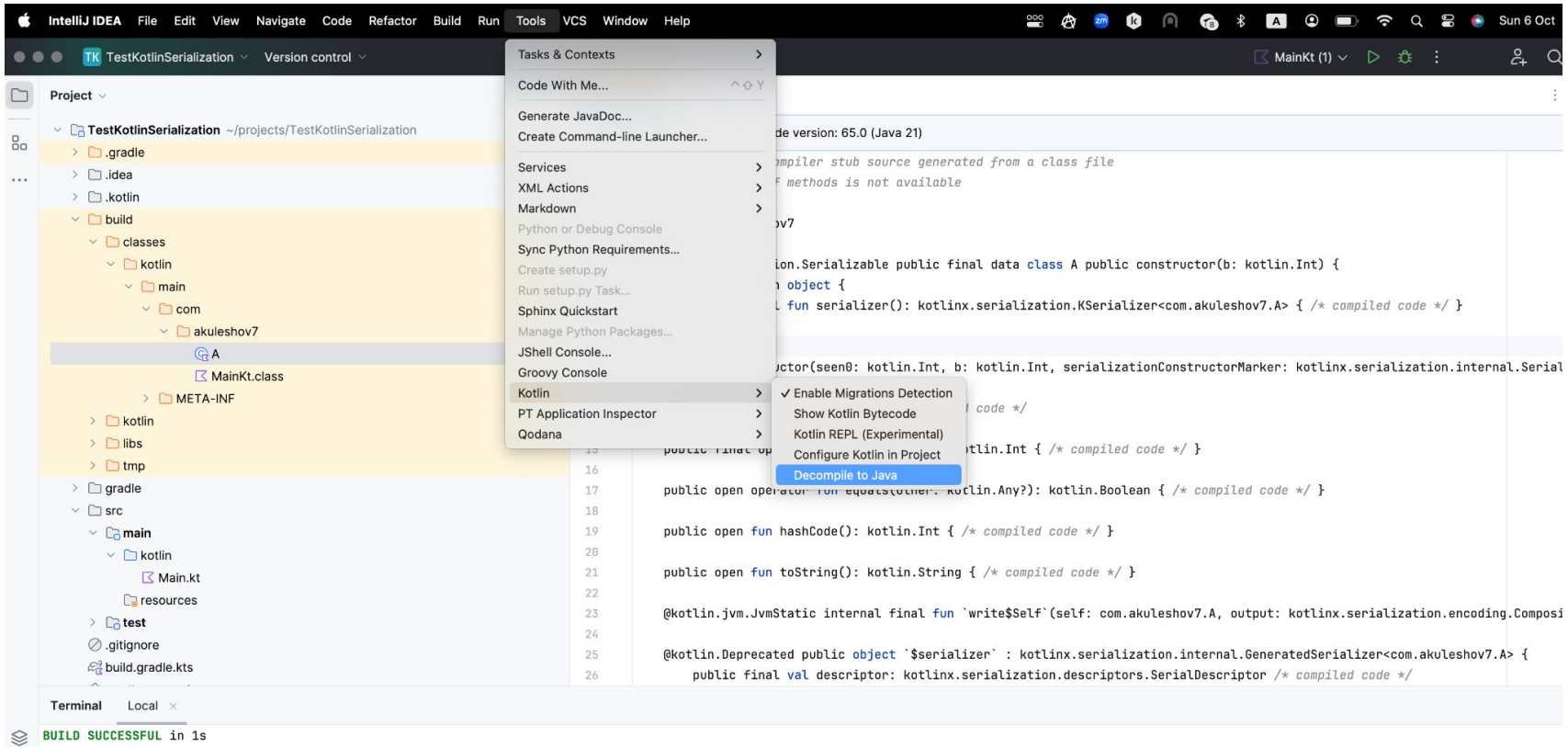



Как посмотреть самому?

Как посмотреть самому?

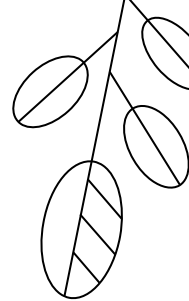


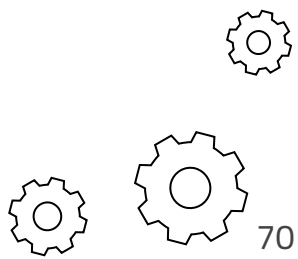
Как посмотреть самому?



Что будет сгенерировано?

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





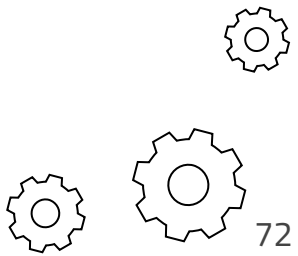
Что будет сгенерировано?

```
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

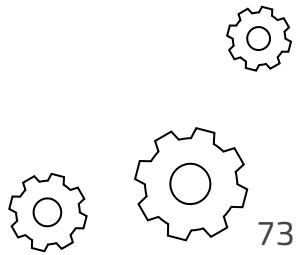
```
public class $serializer implements GeneratedSerializer {

    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

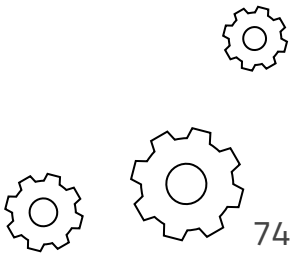
```
public class $serializer implements GeneratedSerializer {

    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



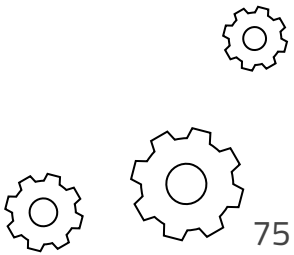
Сериализатор

```
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);
    }
```



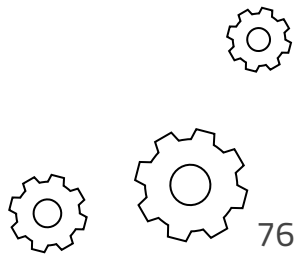
Сериализатор

```
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);
    }
```



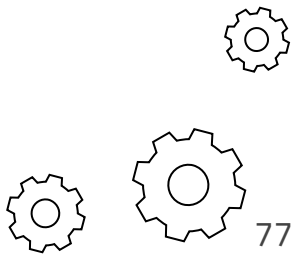
Сериализатор

```
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

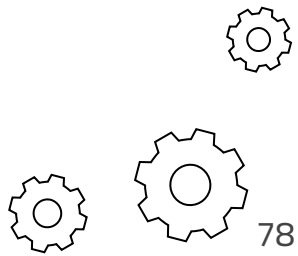
```
public class $serializer implements GeneratedSerializer {

    @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);
            }
        }
        ...
```

while (true)

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);
           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);
                    var5 |= 1;
                    break;
                default:
                    throw new UnknownFieldException(var4);
            }
        }
      ...
```

Не знаем индекс, мусор

Десериализатор

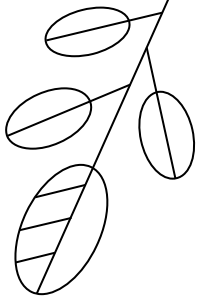
```
public class $serializer implements GeneratedSerializer {

    @NotNull
    public final A deserialize(@NotNull Decoder decoder) {

        ...

        var7.endStructure(var2);

        return new A(var5, var6, (SerializationConstructorMarker)null);
    }
```

Обработали структуру, все хорошо

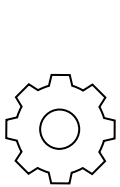
Десериализатор

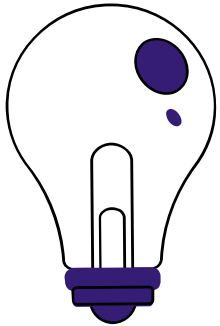
```
public class $serializer implements GeneratedSerializer {

    @NotNull
    public final A deserialize(@NotNull Decoder decoder) {

        ...

        var7.endStructure(var2);

        return new A(var5, var6, (SerializationConstructorMarker)null);
    }
```

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



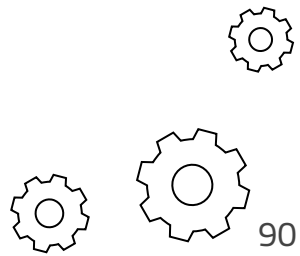
03 Склеиваем вместе





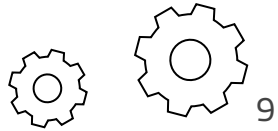
Цепочки вызовов

```
package com.akuleshov7


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

@Serializable
data class A(
    val b: Int
)

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



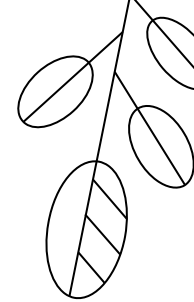
Цепочки вызовов

```
package com.akuleshov7


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

@Serializable
data class A(
    val b: Int
)

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



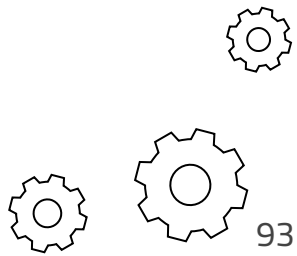
Цепочки вызовов

```
Json.decodeFromString(a)
```



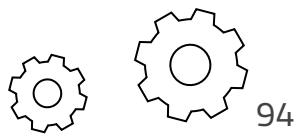
Цепочки вызовов

```
Json.decodeFromString(a)

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



Цепочки вызовов

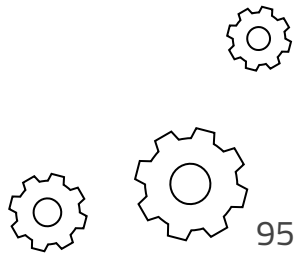
```
Json.decodeFromString(a)

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



Цепочки вызовов

```
Json.decodeFromString(a)

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

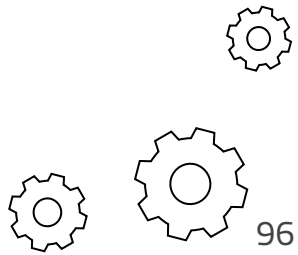
Помните, как мы искали сгенерированный по @Serializable аннотации код?



Цепочки вызовов

```
Json.decodeFromString(a)

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

Ответственный компилятор и авторы компиляторных плагинов



Цепочки вызовов

```
Json.decodeFromString(a)
```

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

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

Ответственный компилятор и авторы компиляторных плагинов

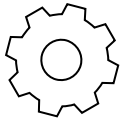
Цепочки вызовов

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

    }
}
```

Цепочки вызовов

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

    }
}
```

Верхнеуровневый API, чтобы
соответствовать kotlinx интерфейсам



Цепочки вызовов

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

    }
}
```

Верхнеуровневый API, чтобы соответствовать kotlinx интерфейсам

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

Цепочки вызовов

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

    }
}
```

Верхнеуровневый API, чтобы соответствовать kotlinx интерфейсам

Ответственный автор библиотеки для конкретного формата

Цепочки вызовов

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


    }
}
```

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

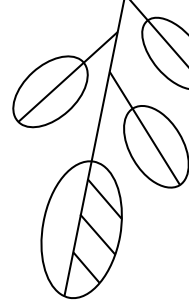
Цепочки вызовов

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

Внутри вызов того самого
deserializer.deserialize(),

Цепочки вызовов

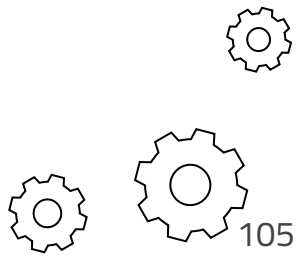
```
class MyDecoder: Decoder {

}
```



Цепочки вызовов

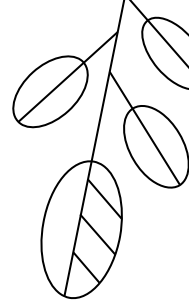
```
class MyDecoder: Decoder {
   public fun decodeInt(): Int
   public fun decodeString(): String

   ...
   public fun beginStructure(descriptor: SerialDescriptor): CompositeDecoder

   ...
   public fun decodeEnum(enumDescriptor: SerialDescriptor): Int
}
```

Ответственный автор библиотеки
для конкретного формата



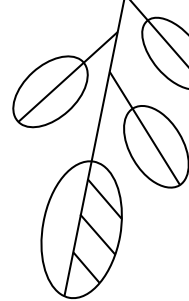
Цепочки вызовов

Мы	Компиляторный фреймворк	Либы
Не забываем @Serializable		
Вызываем decodeFrom… /encodeTo…		

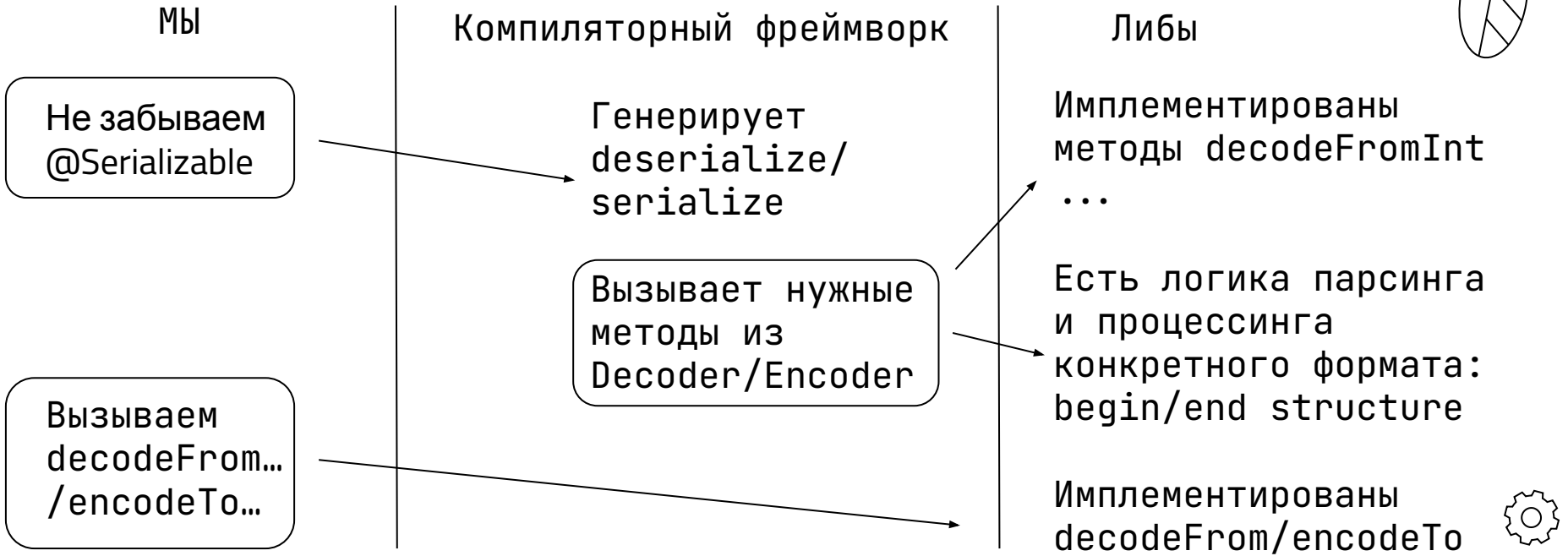


Цепочки вызовов

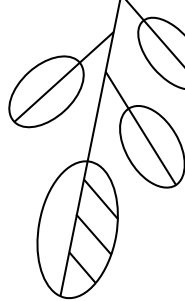
Мы	Компиляторный фреймворк	Либы
Не забываем @Serializable	Генерирует deserialize/ serialize Вызывает нужные методы из Decoder/Encoder	
Вызываем decodeFrom… /encodeTo…		



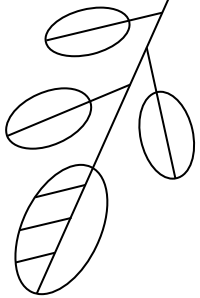
Цепочки вызовов



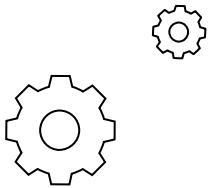
Цепочки вызовов

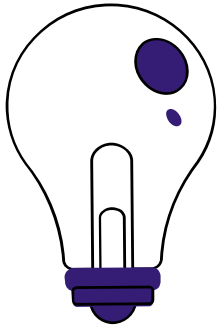


Мы	Компиляторный фреймворк	Либы
Не забываем @Serializable	Генерирует deserialize/ serialize	Имплементированы методы decodeFromInt ...
	Вызывает нужные методы из Decoder/Encoder	Есть логика парсинга и процессинга конкретного формата: begin/end structure
Вызываем decodeFrom… /encodeTo…		Имплементированы decodeFrom/encodeTo

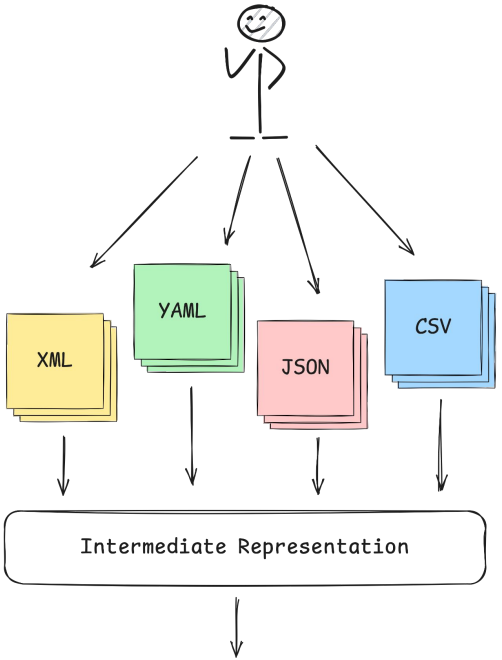


04 Хитрый сценарий для лучшего понимания

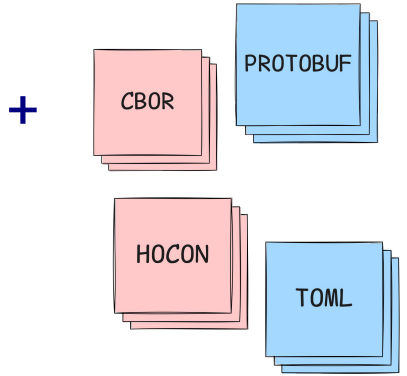




Наша проблема

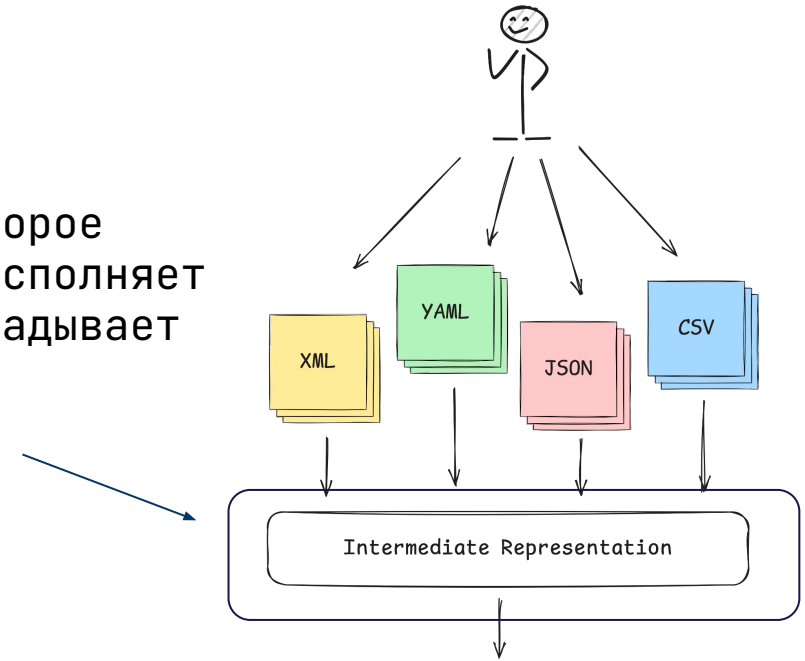


вероятно бинарные
и нестандартные
форматы



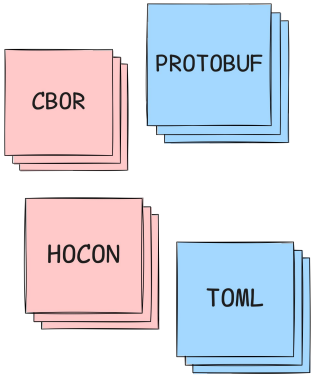
Наша проблема

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

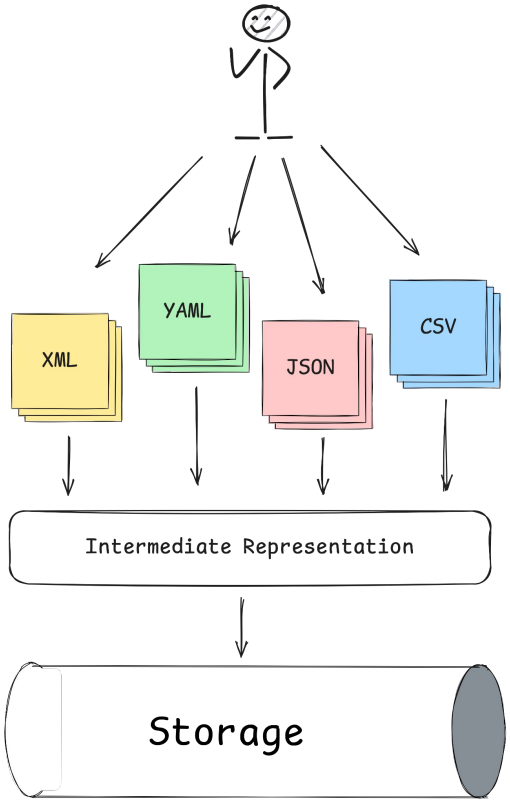


вероятно бинарные и нестандартные форматы

\+

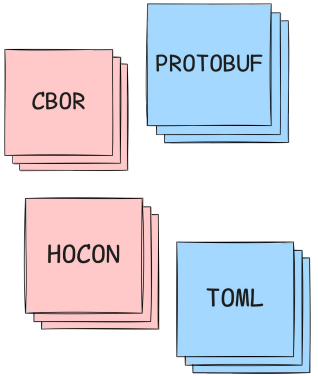


Наша проблема

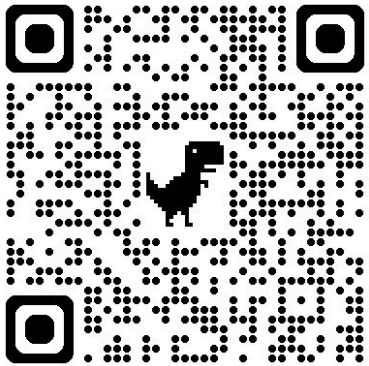


вероятно бинарные
и нестандартные
форматы

\+



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



Наш IR: AVRO schema

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

Наш IR: AVRO schema

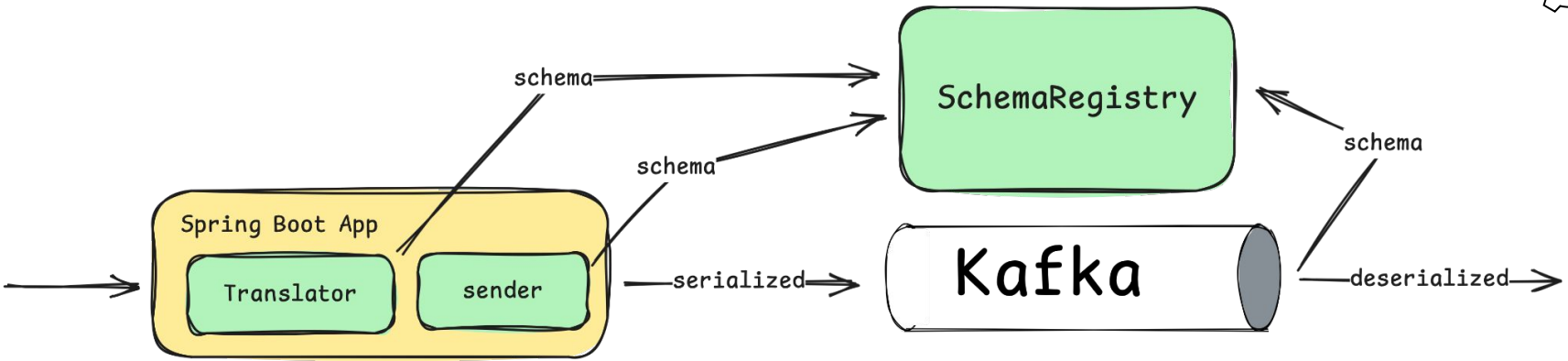
```
{
  "type" : "record",
  "name" : "Message",
  "namespace" : "schema.namespace",
  "fields" : [
     {
          "name" : "title",
          "type" : [ "null", "string" ],
          "default": null
     },
     {
          "name" : "message",
          "type" : "string"
     }
  ]
}
```

имена

типы

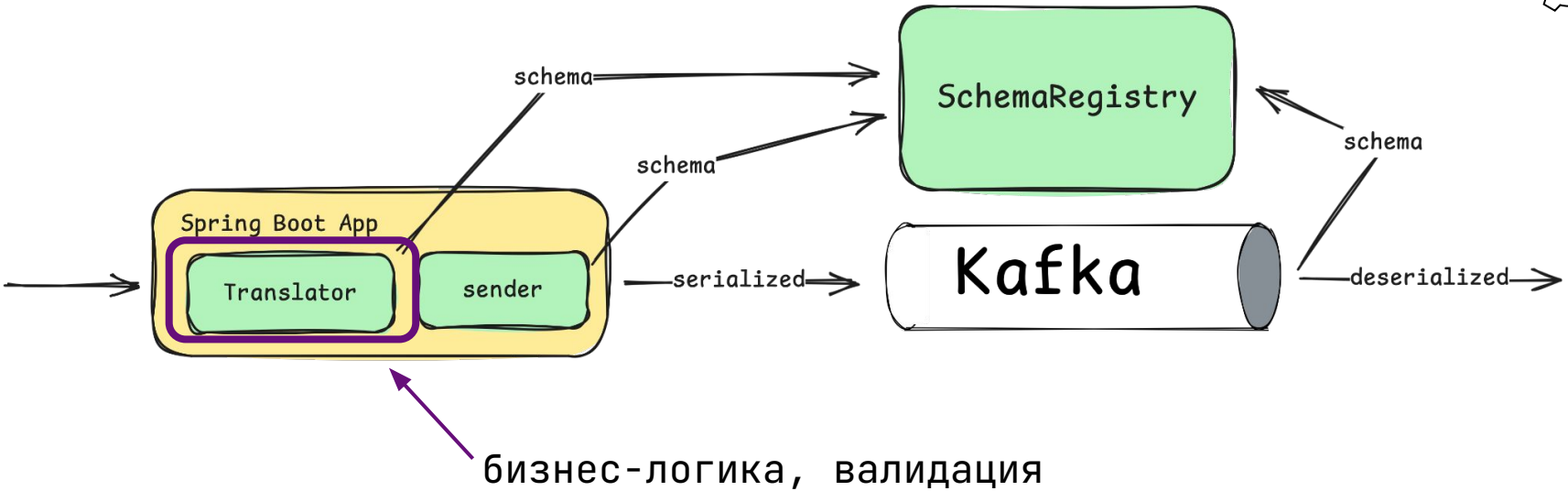
Наш IR: AVRO

почему?



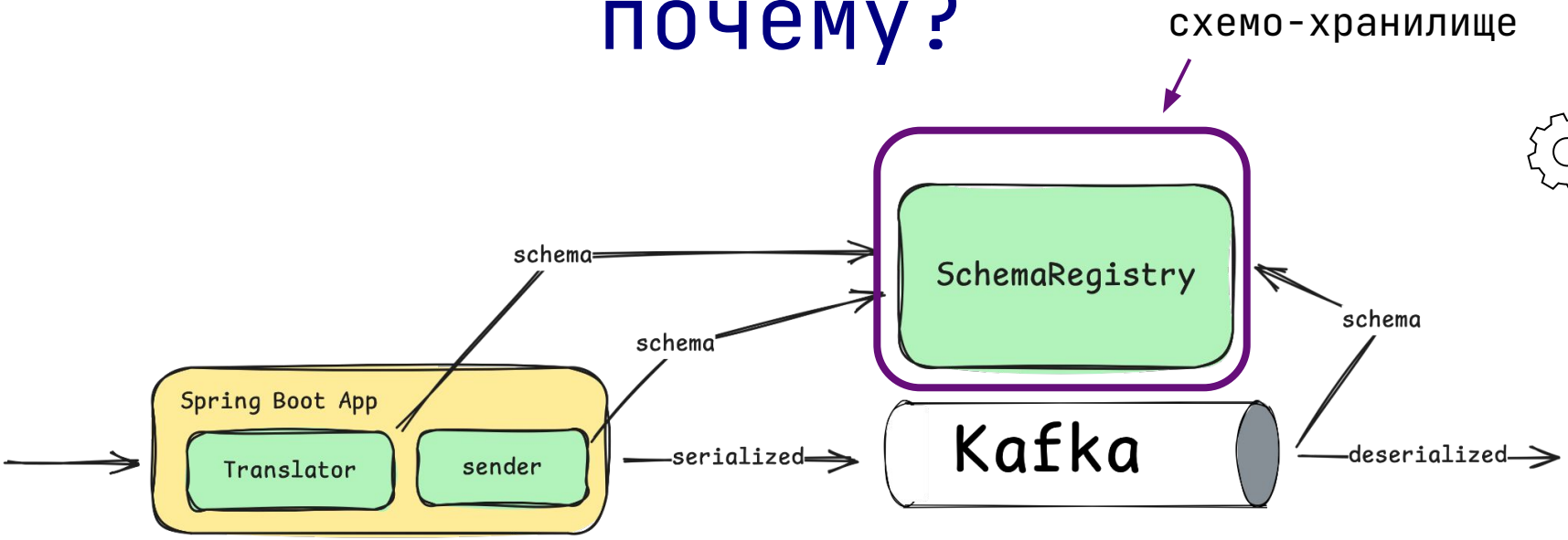
Наш IR: AVRO

почему?



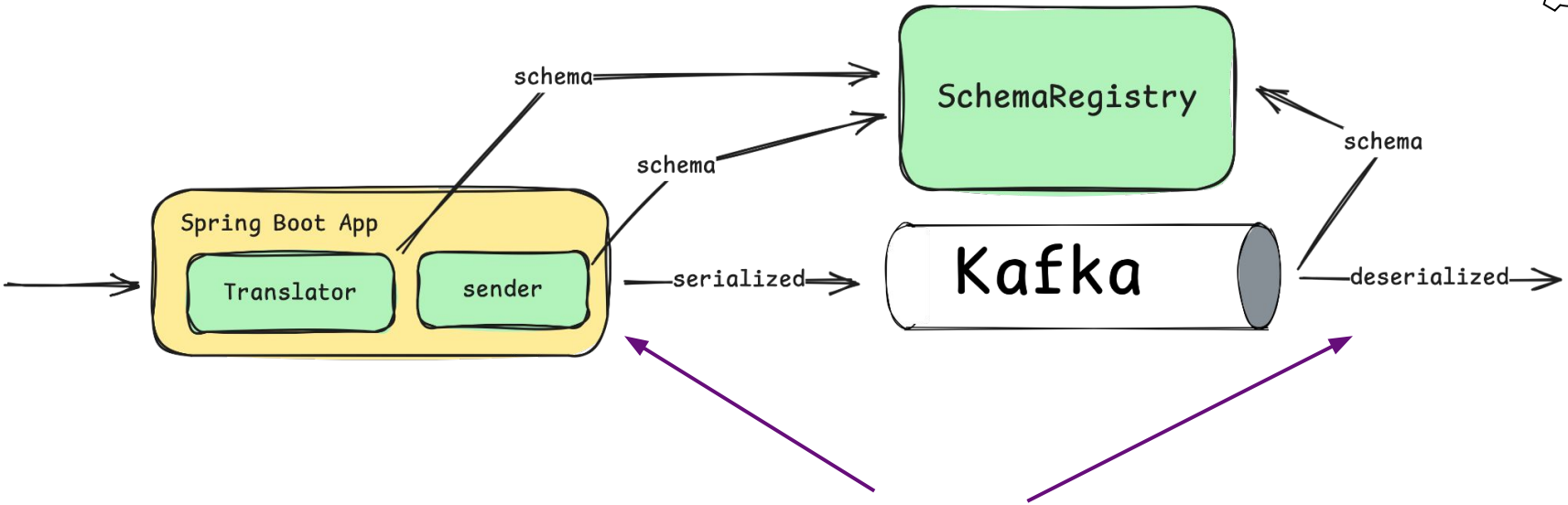
Наш IR: AVRO

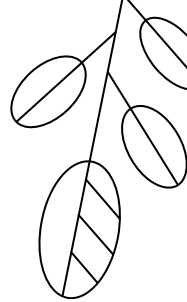
почему?



Наш IR: AVRO

почему?



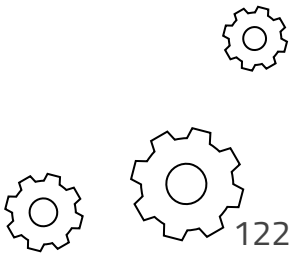


В чем разница?

В схеме!

Реальный сценарий

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









}
```



Реальный сценарий

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









}
```

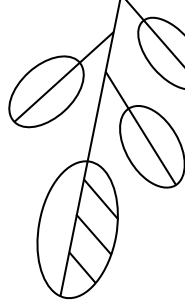
Реальный сценарий

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

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


}
```

Реальный сценарий

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

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

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

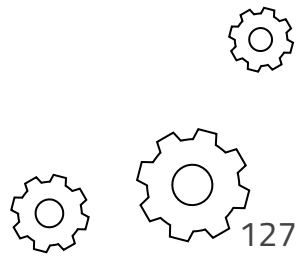
“Обманываем” kotlinx



Передаем схему в KSerializer, замещаем сгенерированный код

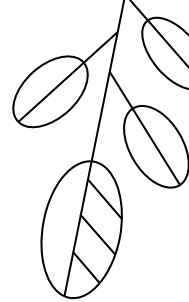
“Обманываем” kotlinx

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



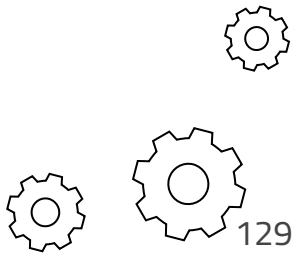
“Обманываем” kotlinx

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

// (!) с помощью buildClassSerialDescriptor строим дескриптор
override val descriptor: SerialDescriptor = avroSchema.fields.toSerialDescriptor("Root")
```



“Обманываем” kotlinx

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

override fun deserialize(decoder: Decoder): GenericRecord {
  val compositeDecoder = decoder.beginStructure(descriptor)
  ...
  // и начинаем крутиться в классическом цикле,
  while (true) {
  ...
       val index = compositeDecoder.decodeElementIndex(descriptor)
       if (index == CompositeDecoder.DECODE_DONE) break


  ...
  }
```

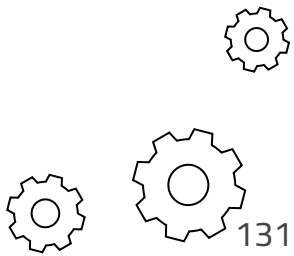


“Обманываем” kotlinx

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

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

“Обманываем” kotlinx

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

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



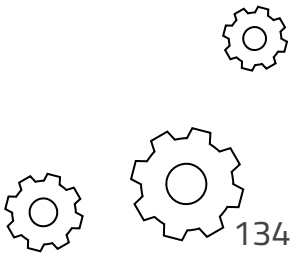
“Обманываем” kotlinx

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

override fun deserialize(decoder: Decoder): GenericRecord {
  val compositeDecoder = decoder.beginStructure(descriptor)
  ...
  // и начинаем крутиться в классическом цикле,
  while (true) {
  ...
     genericRecord.put(
        key.name(),
        value
     )
  ...
  }

  return genericRecord
```

Что мы сегодня поняли?

Мы сегодня многое поняли

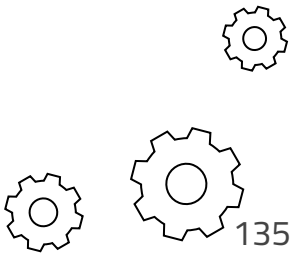
- Что Koltinx.serialization - продвинутый инструмент сериализации, особенно, если у вас мультиформатная работа



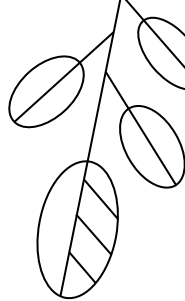
Мы сегодня многое поняли

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

- Какой код генерируется компилятором и почему это важно знать



Мы сегодня многое поняли



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

CrossConf

И спасибо, что продержались до конца!

Telegram

Github