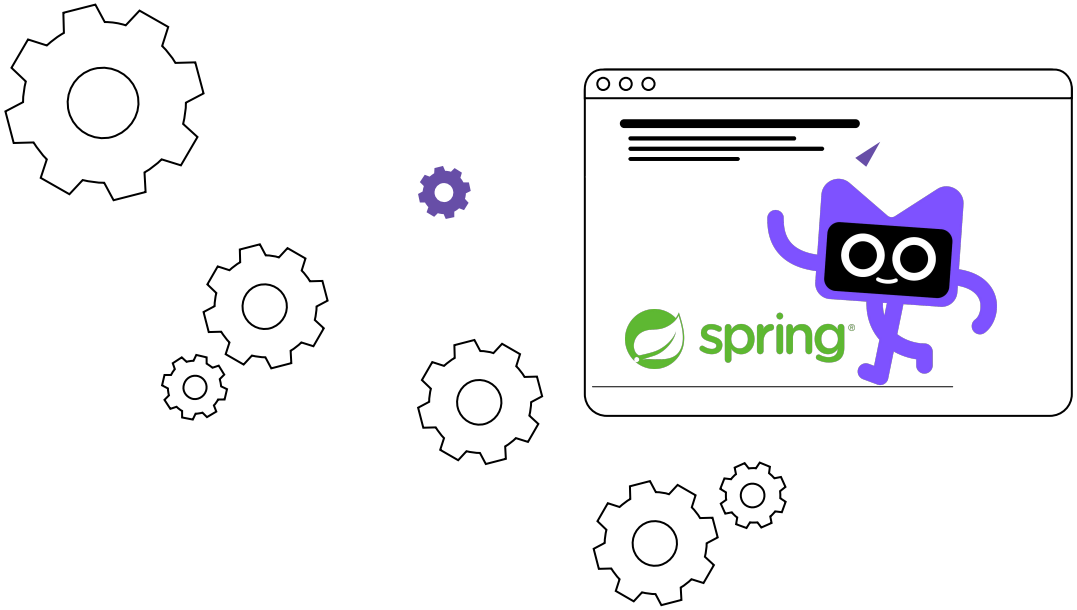
SPRING

И

KOTLIN

Работай с
любым форматом!

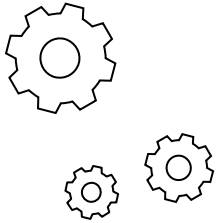


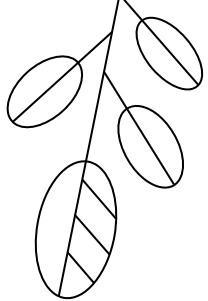
$ whoami?

Андрей Кулешов

intel. Deutsche Bank HUAWEI pt

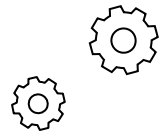
Стачка phd 2 SafeCode JPoint Joker<?> CrossConf GIGA CONF sysconf





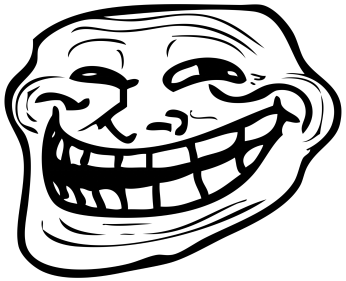
О чем поговорим?

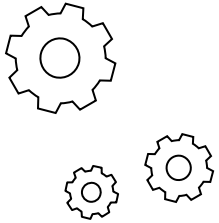
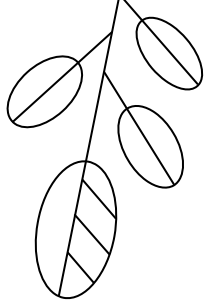
kotlin



О чем поговорим?

kotlinx.serialization

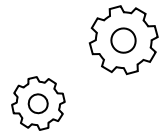


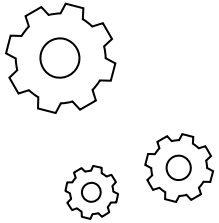
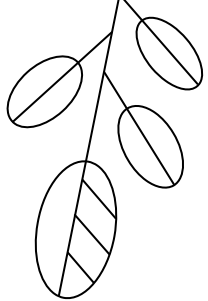



О чем поговорим?

kotlinx.serialization

пользовательские кастомизации

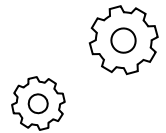



О чем поговорим?

kotlinx.serialization

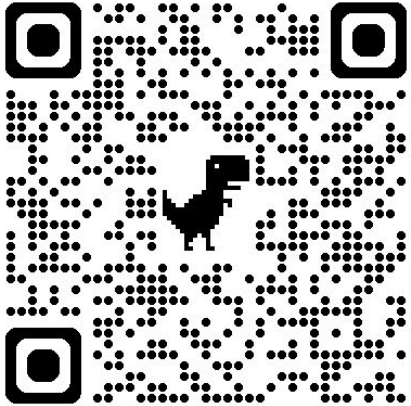
пользовательские кастомизации

реальная история
десериализации в нашем
Spring Boot’овом приложении

О чем поговорим?

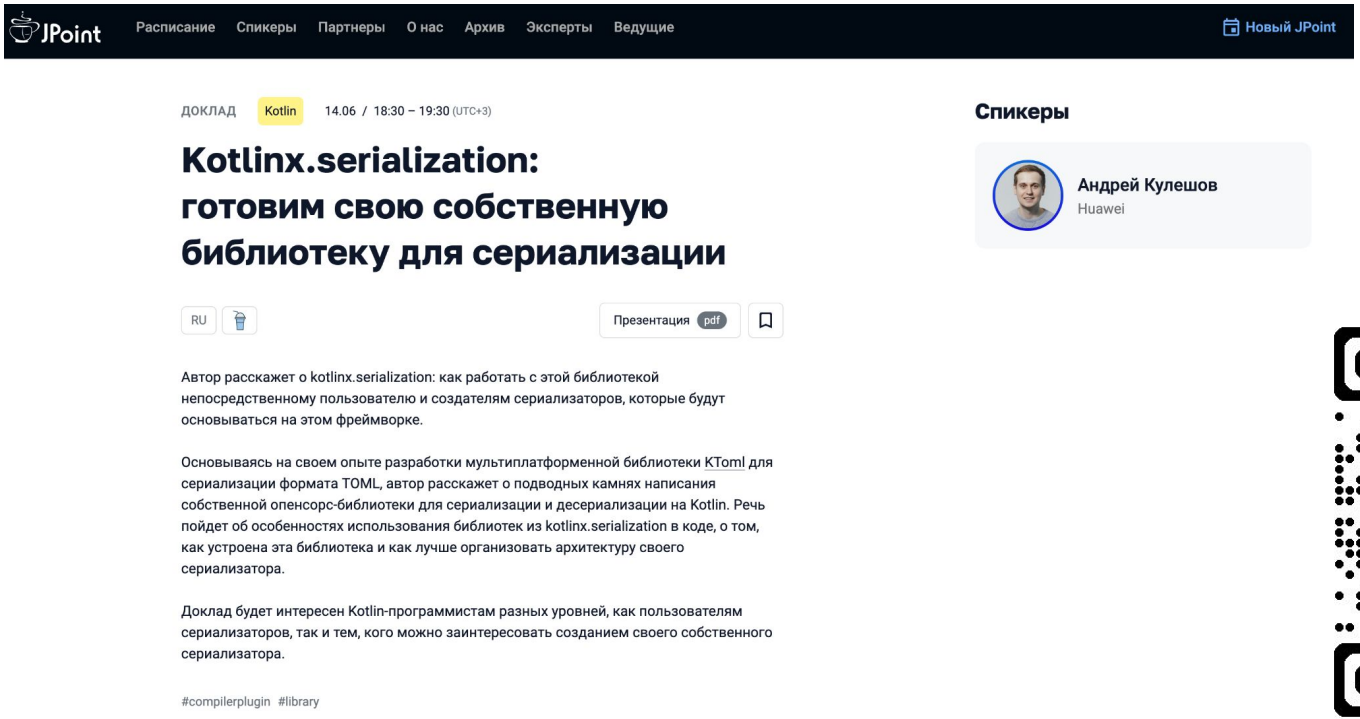
Почему я?



[KT]oml

powered by kotlinx.serialization

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

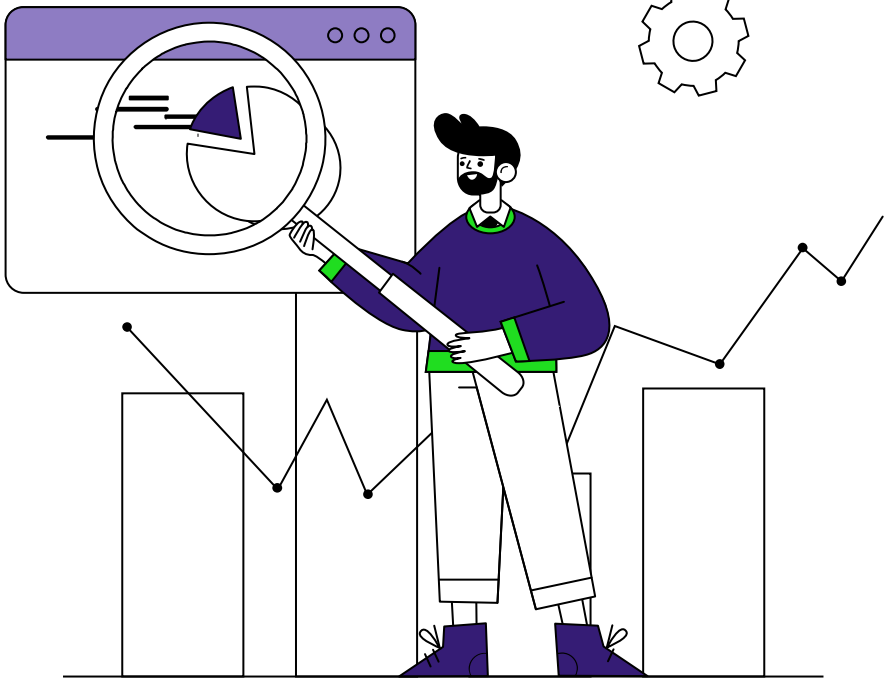
2 Reply

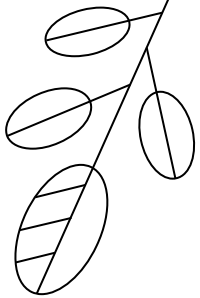
@telephon3208 1 year ago

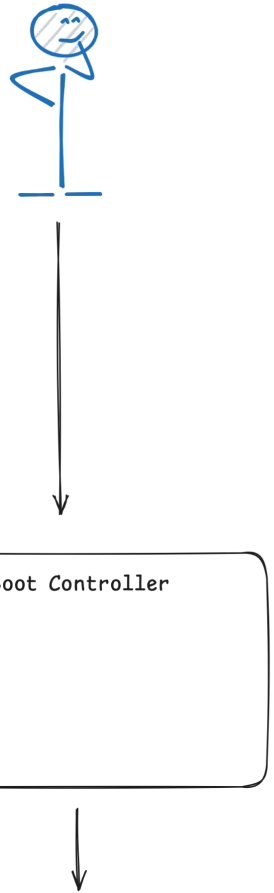
бесценный доклад

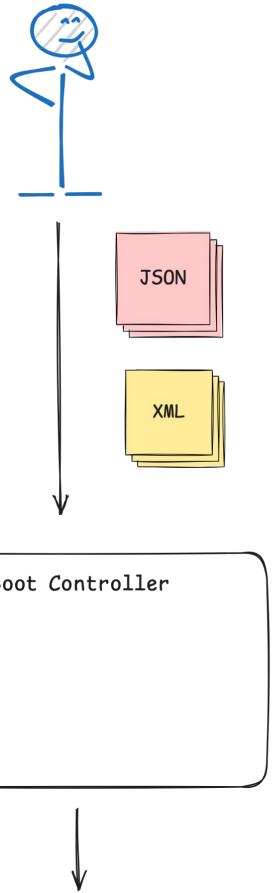
1 Reply

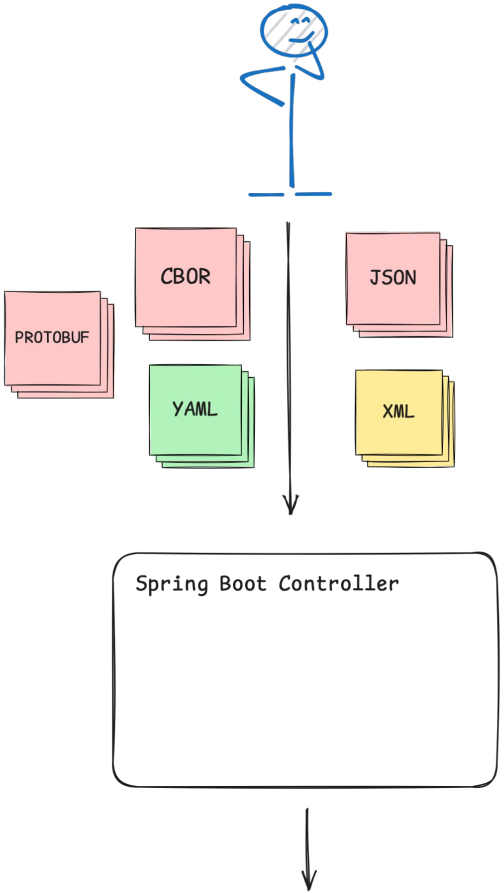
01 Проблема



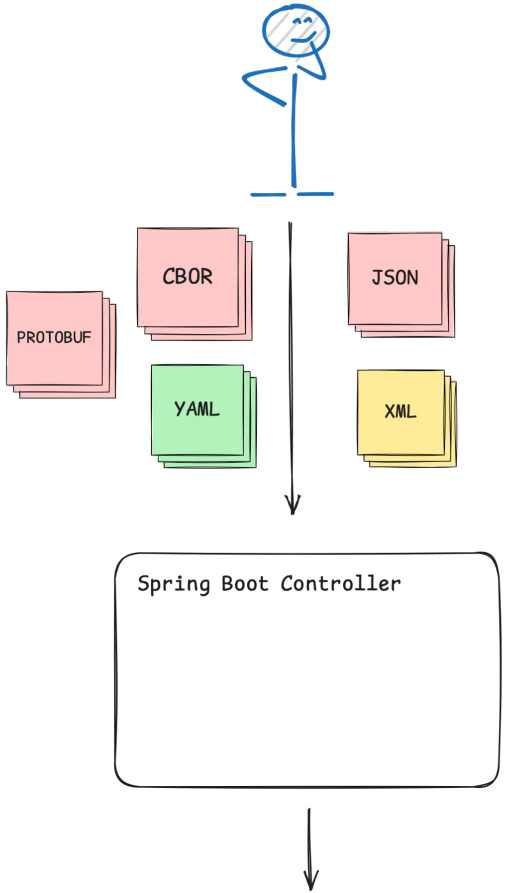



Spring Boot Controller

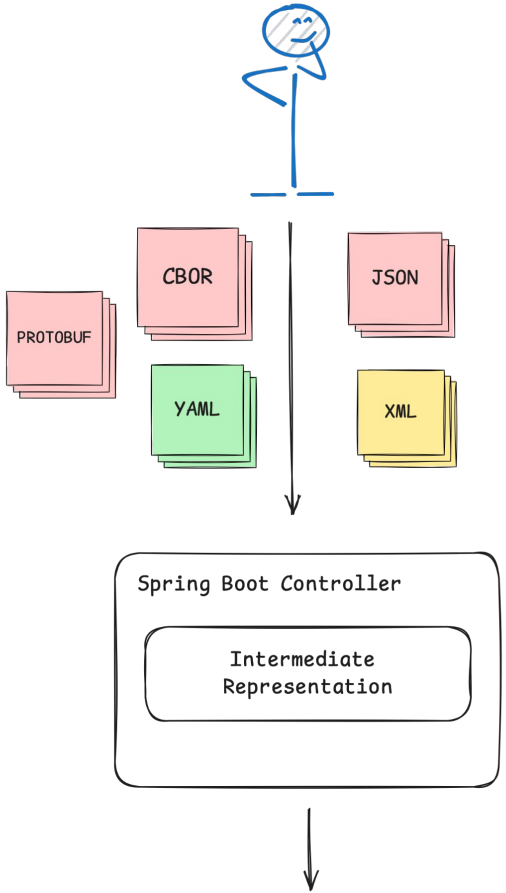

JSON
XML
Spring Boot Controller


CBOR
JSON
PROTOBUF
YAML
XML
Spring Boot Controller

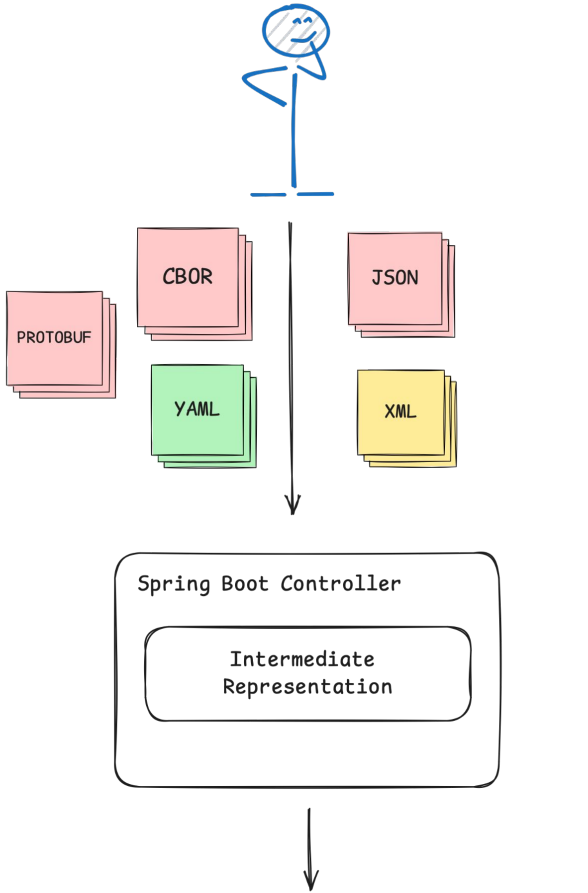
@RequestHeader(HttpHeaders.CONTENT_TYPE)



@RequestHeader(HttpHeaders.CONTENT_TYPE)

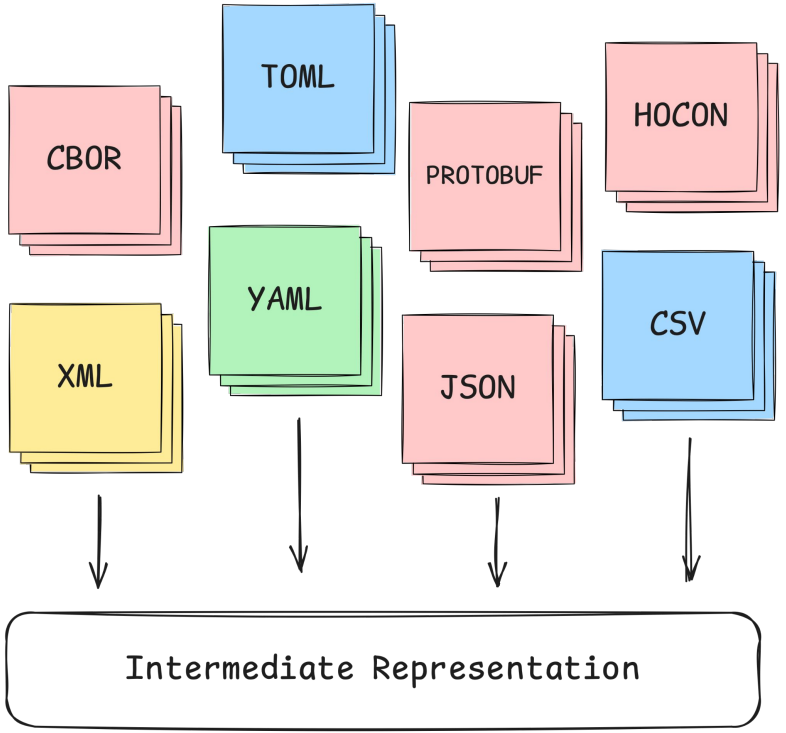


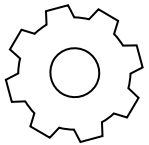
@RequestHeader(HttpHeaders.CONTENT_TYPE)



@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
```

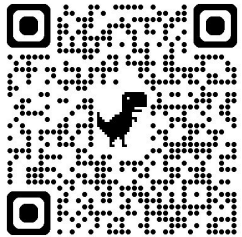
JSON

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,
    @SerialName("gradle-libs-like-property")
    val kotlinJvm: GradlePlugin,
    val myMap: Map<String, String>
)
```

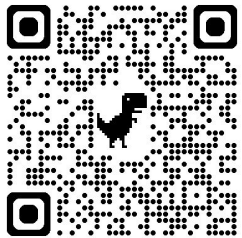


Самый простой IR: со схемой внутри программы

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

типы

имена



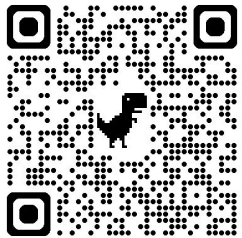
Самый простой IR: со схемой внутри программы

```
@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>
)
```

типы

имена

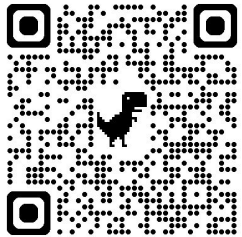
Код как схема данных

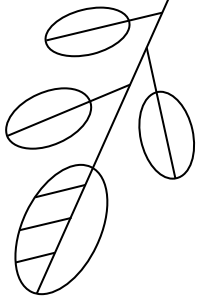


Самый простой IR: со схемой внутри программы

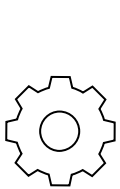
```
@Serializable
data class MyClass(
    val someBooleanProperty: Boolean,
    val table1: Table1,
    val table2: Table2,
    @SerialName("gradle-libs-like-property")
    val kotlinJvm: GradlePlugin,
    val myMap: Map<String, String>
)

...
```

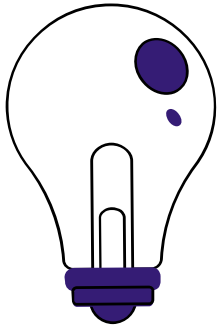
Как это работает?





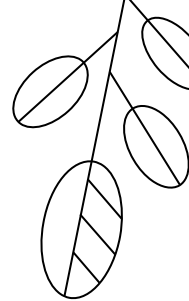
02 kotlinx.serialization

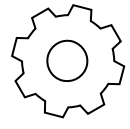




Вкратце

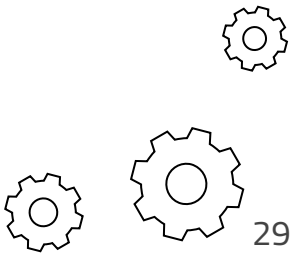
- Мультиплатформенная (**KMP**) библиотека





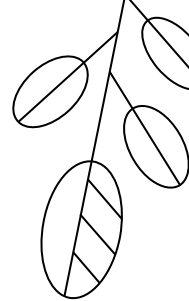
Вкратце

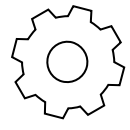
- Мультиплатформенная (**KMP**) библиотека
- Является **компиляторным плагином**



Вкратце

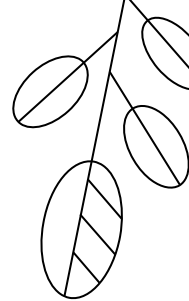
- Мультиплатформенная (**KMP**) библиотека
- Является **компиляторным плагином**
- **НЕ** использует reflection

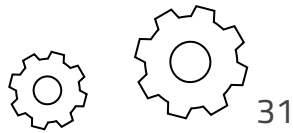




Вкратце

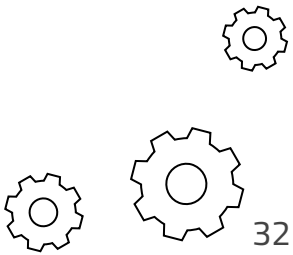
- Мультиплатформенная (**KMP**) библиотека
- Является **компиляторным плагином**
- **НЕ** использует reflection
- Встраивается в проект с помощью Maven/Gradle плагина





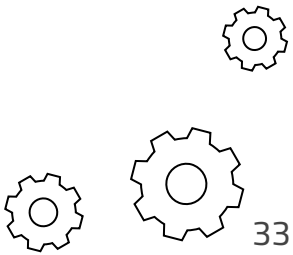
Вкратце

- 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

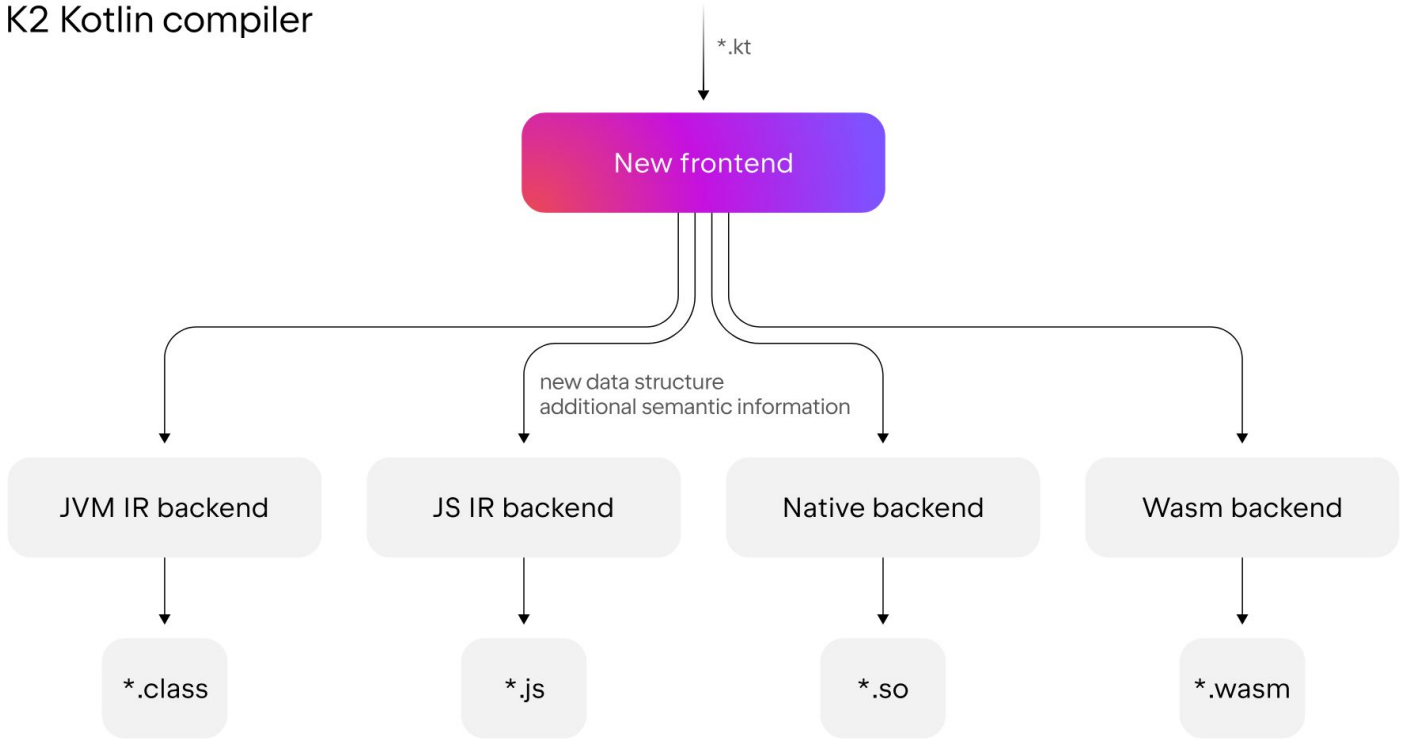
- Мультиплатформенная (**KMP**) библиотека
- Является **компиляторным плагином**
- **НЕ** использует reflection
- Встраивается в проект с помощью Maven/Gradle плагина
- **5** официальных форматов Json, Protobuf, CBOR, HOCON, Properties
- **19+** коммьюнити библиотек для других форматов



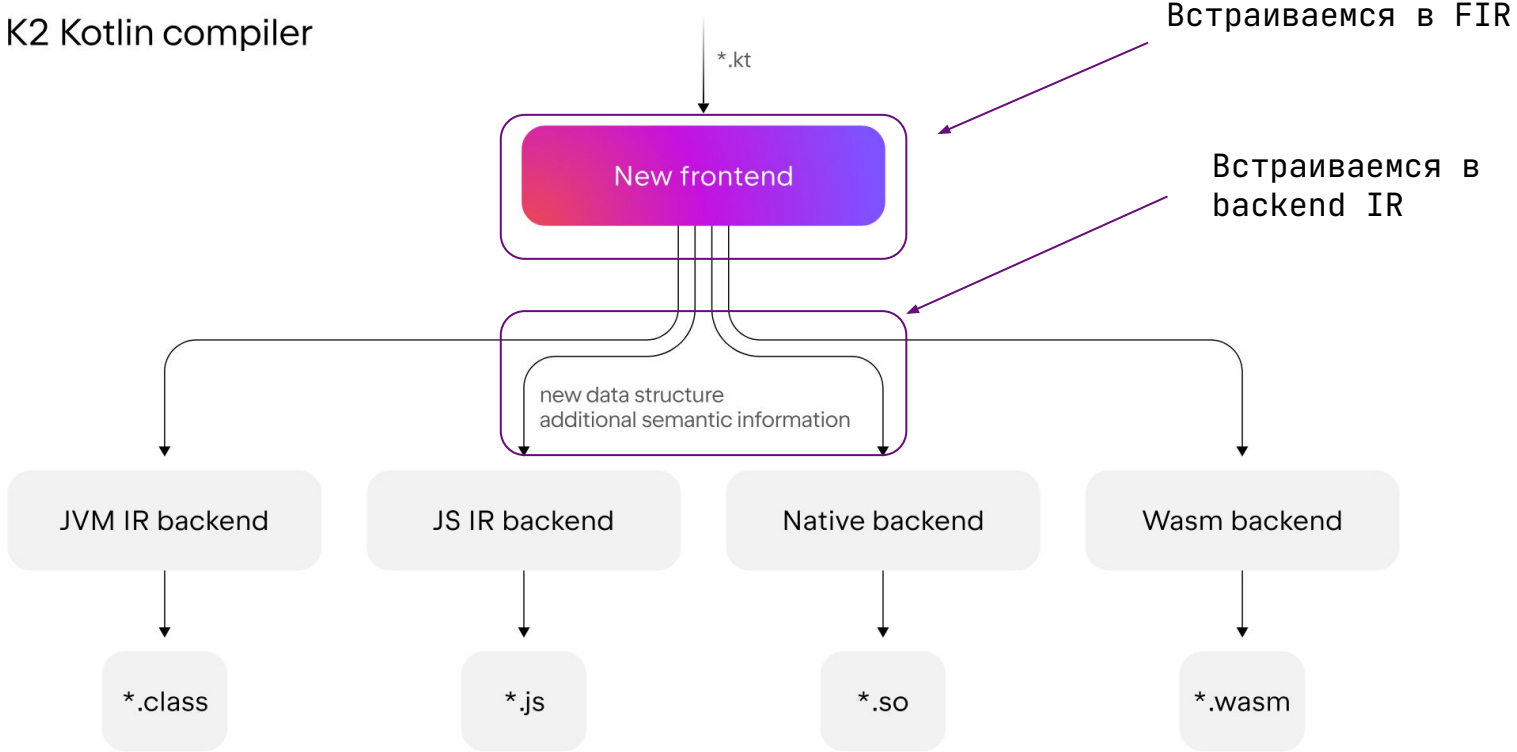
Вкратце

- 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

Компиляторные плагины



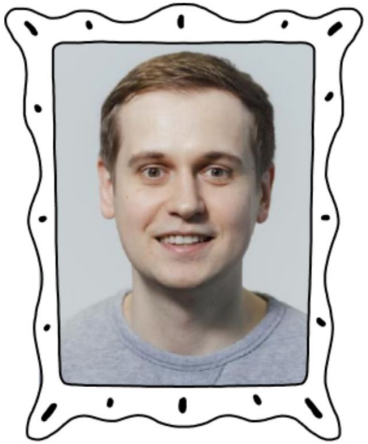
Компиляторные плагины



Компиляторные плагины

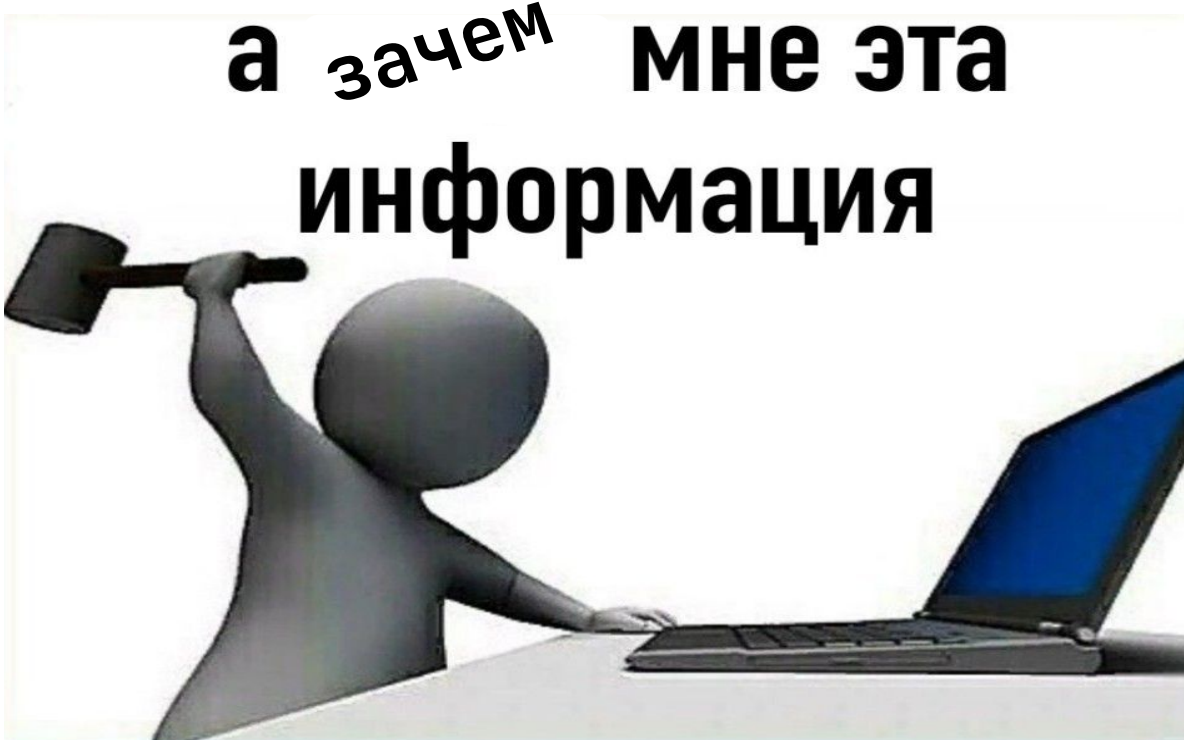
Компиляторные плагины в Котлин: почувствуй себя системным программистом

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

```
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

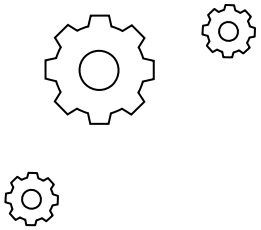
```
1  package com.akuleshov7
2
3  import kotlinx.serialization.Serializable
4  import kotlinx.serialization.encodeToString
5  import kotlinx.serialization.json.Json
6
7  @Serializable
8  data c
9      va
10 )
11
12 fun ma
13     va
14     Js
15 }
16
```

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.

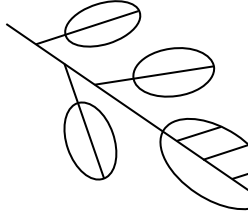
```
public constructor Serializable(
    val with: KClass<out KSerializer<*>> = KSerializer::class
)
```

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.



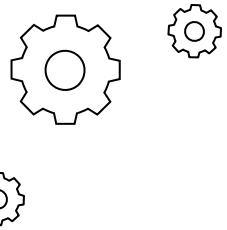
KSerializer



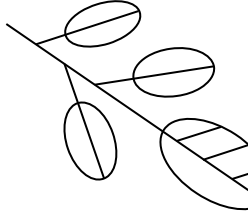
Как обнаруживается?

Зачем он вообще нужен?

Как генерируется?

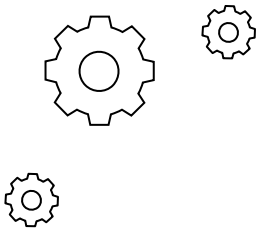


Поиск KSerializer

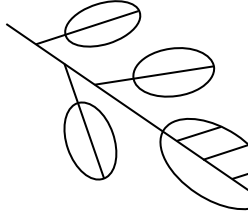


```
public fun SerializersModule.serializer(type: KType): KSerializer<Any?> =
    serializerByKTypeImpl(type, failOnMissingTypeArgSerializer = true) ?: type.kclass()
        .platformSpecificSerializerNotRegistered()
```

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()
```

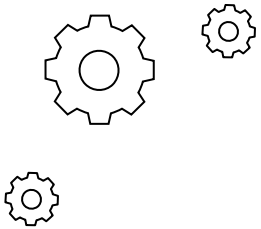
core/commonMain/src/kotlinx/serialization/**Serializers**.kt

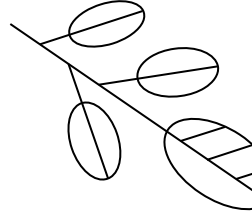
Встроенные KSerializer

```
@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())
```

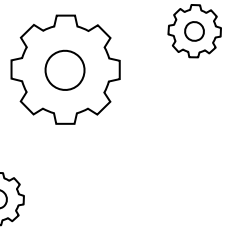
<...>

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

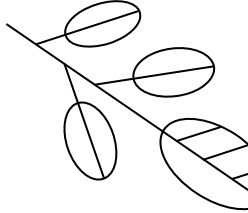




На пальцах пример для String



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()
}
```

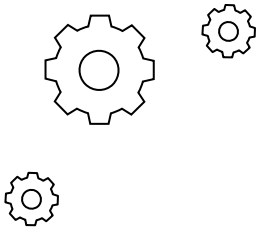
core/commonMain/src/kotlinx/serialization/**Serializers**.kt

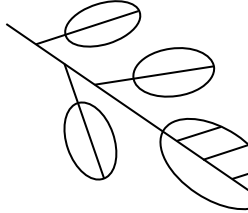
Пример для String

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))
}
```

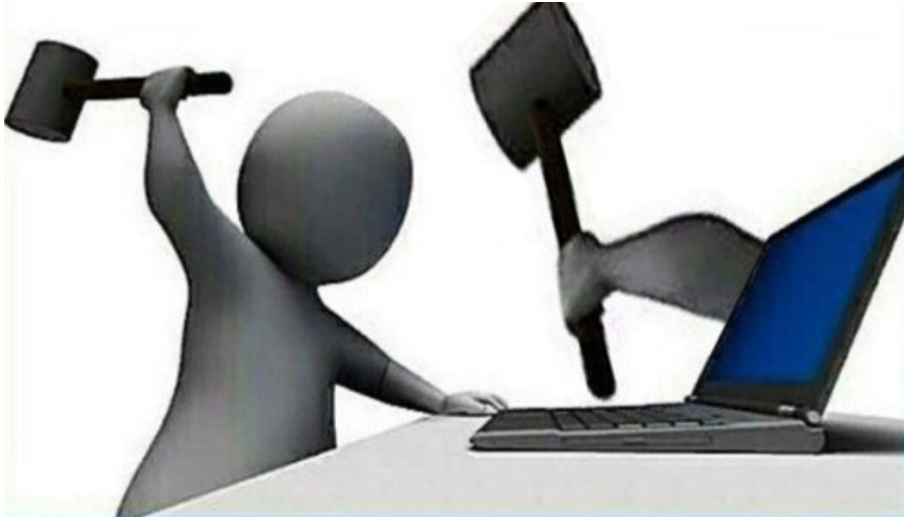
```
plugins {
    kotlin("plugin.serialization") version "2.0.20"
}
```

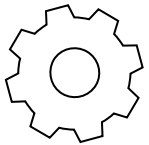
Все работает

```
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))
}
```

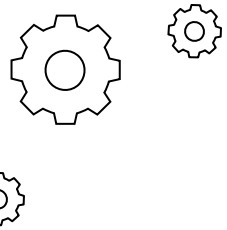
```
com.akuleshov7.A$$serializer@32d992b2
{"b":0}
```

ПОЭТОМУ НУЖНО ПОНИМАТЬ ЧТО ПРОИСХОДИТ ПОД КАПОТОМ

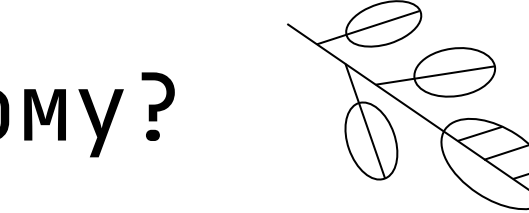


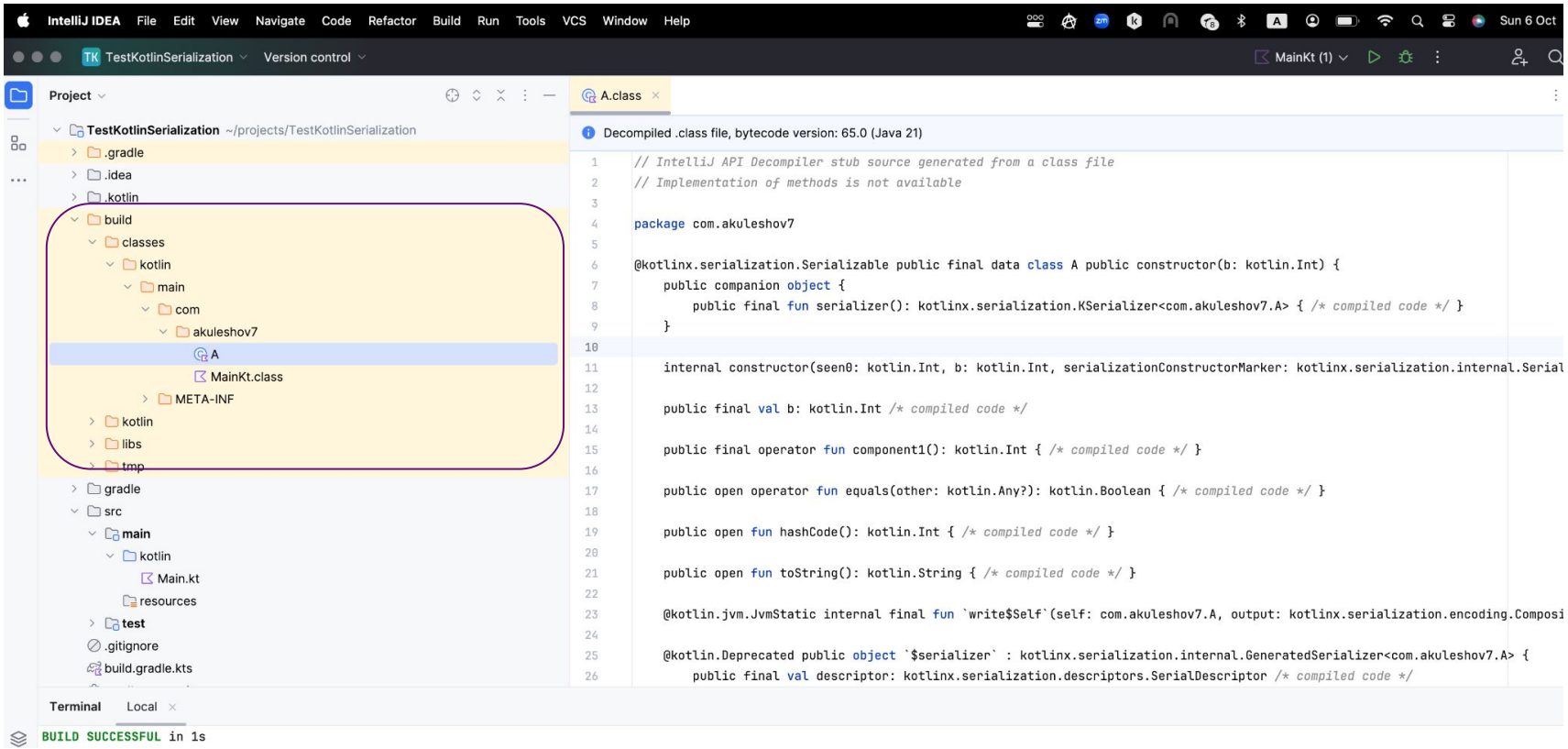
Посмотрим на сгенерированный код

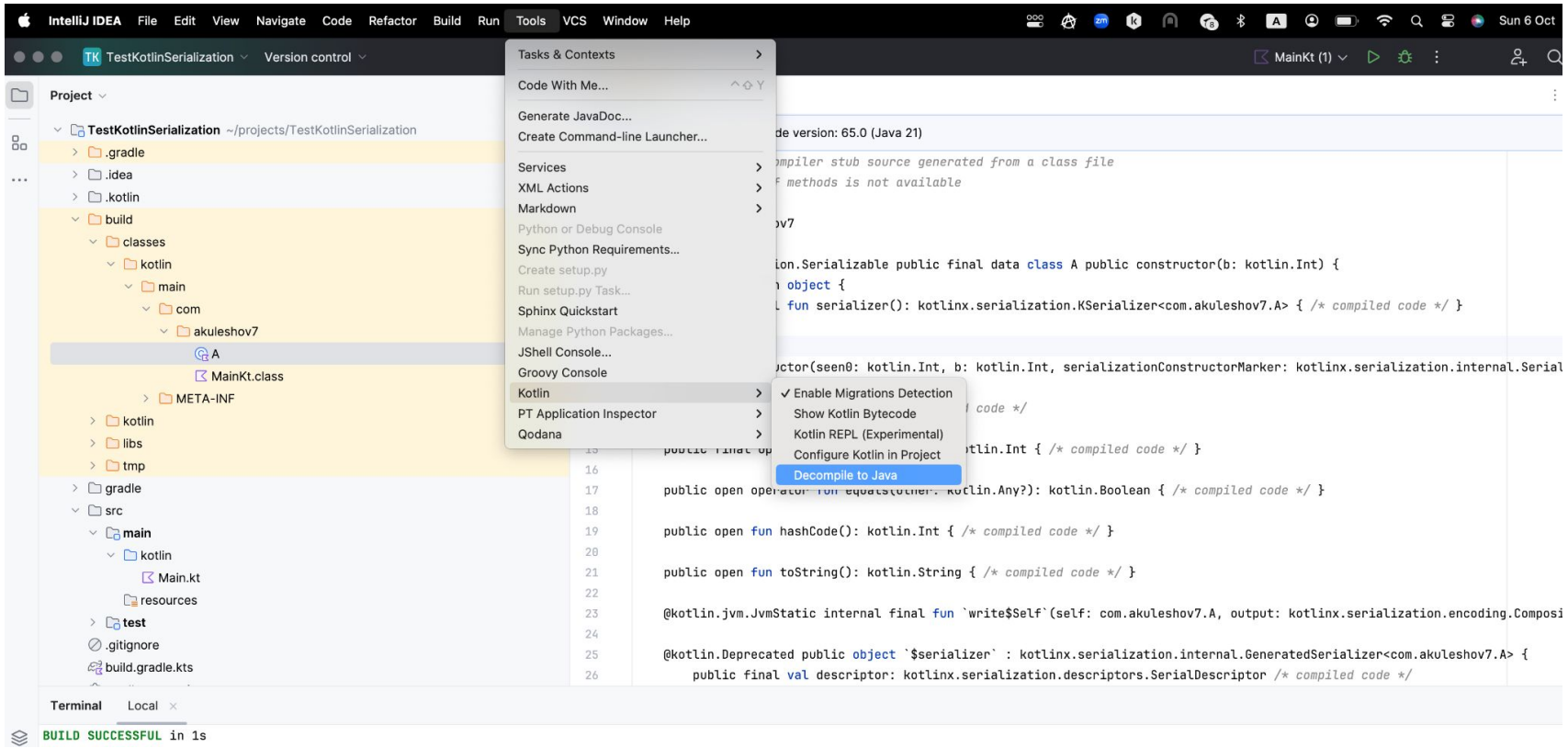
Как посмотреть самому?



Как посмотреть самому?

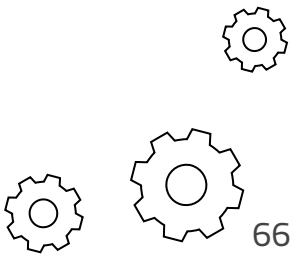


Как посмотреть самому?



Что будет сгенерировано?

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



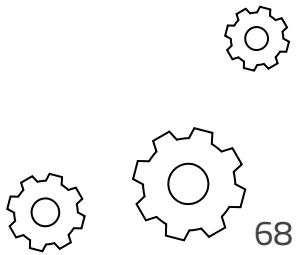
Что будет сгенерировано?

```
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

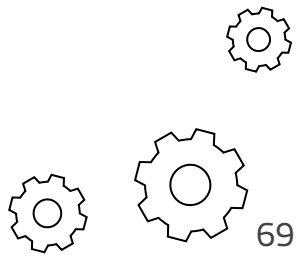
```
public class $serializer implements GeneratedSerializer {

    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

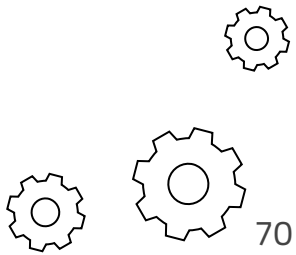
```
public class $serializer implements GeneratedSerializer {

    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



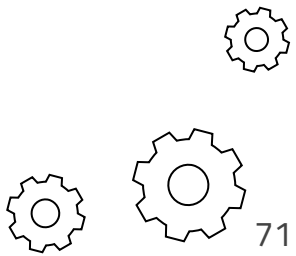
Сериализатор

```
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);
    }
```



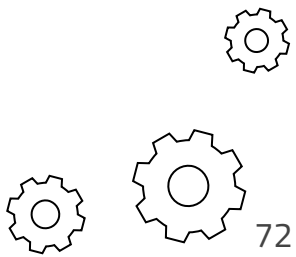
Сериализатор

```
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);
    }
```



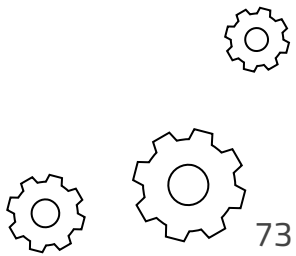
Сериализатор

```
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

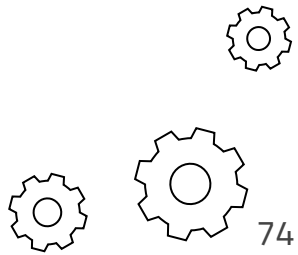
```
public class $serializer implements GeneratedSerializer {

    @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

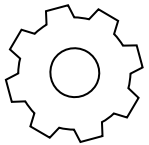
```
public class $serializer implements GeneratedSerializer {

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

        ...

        var7.endStructure(var2);

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

Обработали структуру, все хорошо

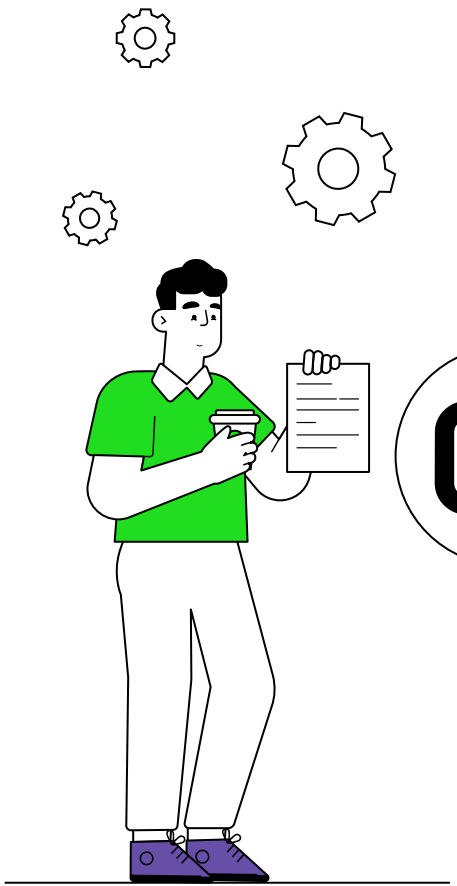
Десериализатор

```
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

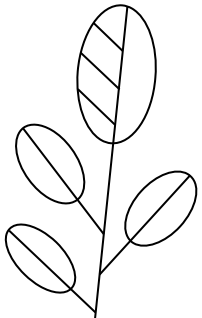
Поняли, что компилятор генерирует, теперь обратно в **реальность**

03 А что там в Spring?

Примитивный кейс

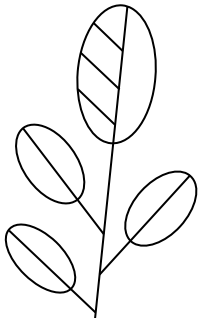
```
@PostMapping("/upload")
fun postData(@RequestBody request: A): ResponseEntity<String> {
    return ResponseEntity.ok("ok")
}

...

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
)
```

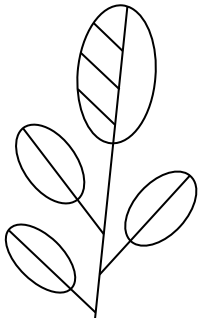
Будет работать?



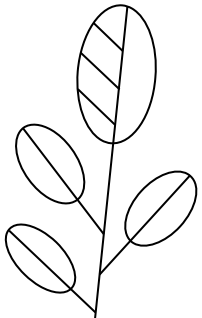
Примитивный кейс

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

...

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

Конечно! Почему нет?

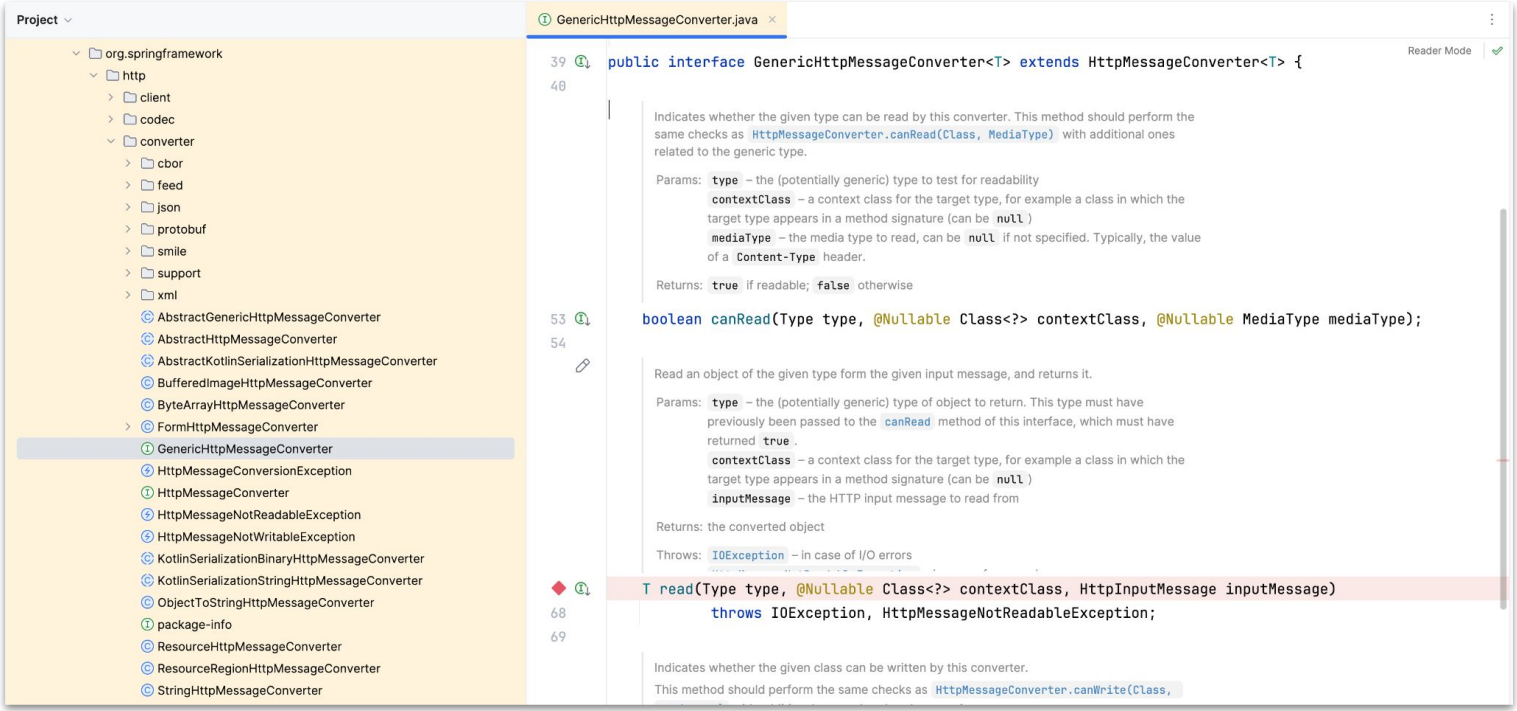


Ну мы вроде пишем на котлин, но это же явно не kotlinx.serialization?



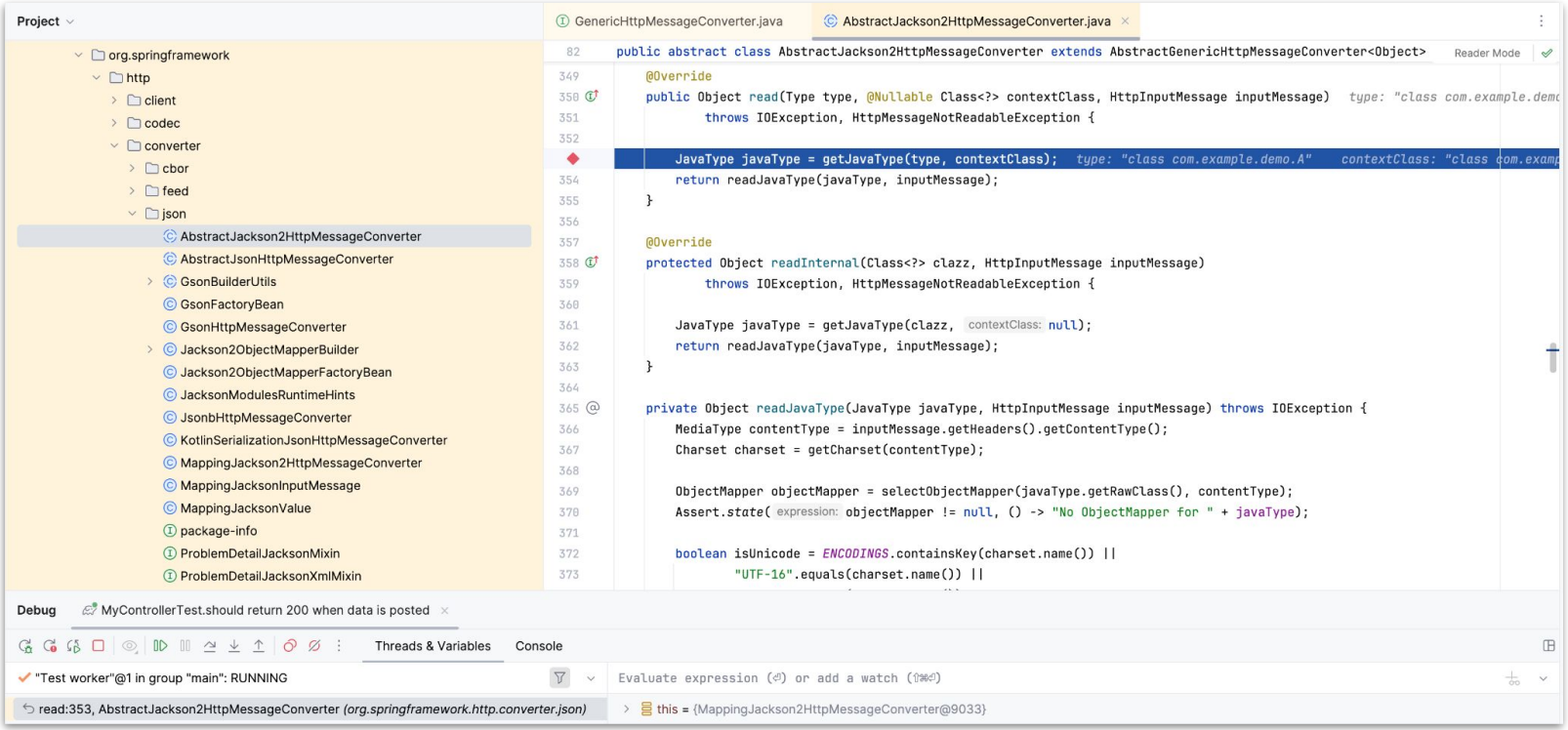
Ну мы вроде пишем на котлин, но это же явно не kotlinx.serialization?

`org.springframework.http.converter.`**`GenericHttpMessageConverter`**



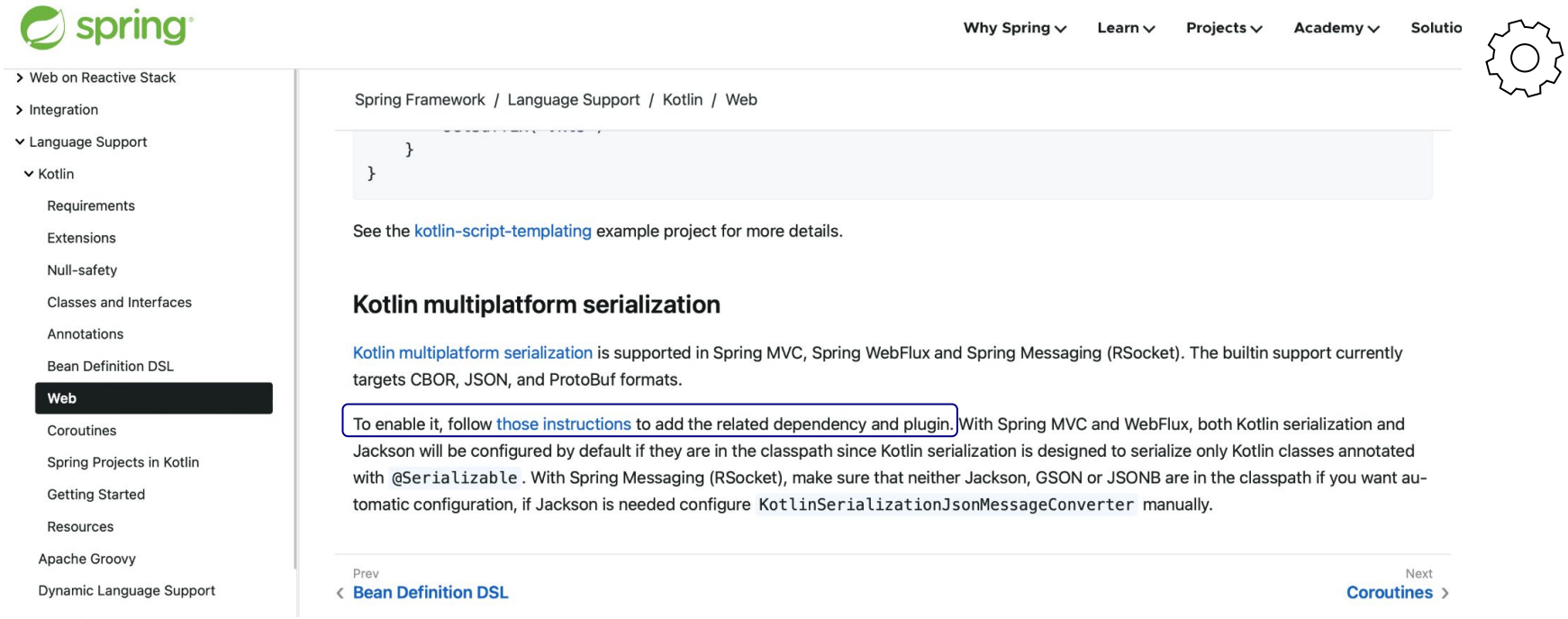
А там конечно Jackson

`org.springframework.http.converter.json.`**`AbstractJackson2HttpMessageConverter`**



Вроде использую Kotlin

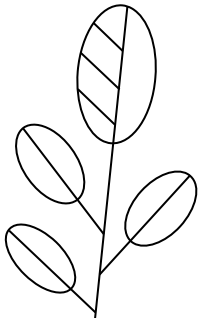
Тогда где этот **ваш** Kotlinx.serialization?



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
)
```

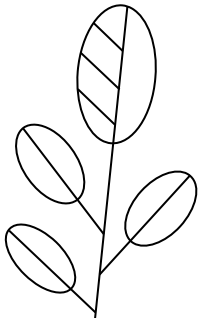


Добавляем plugin и зависимость

```
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?

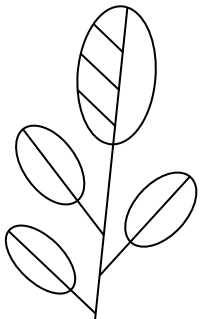


Добавляем plugin и зависимость

```
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
)
```

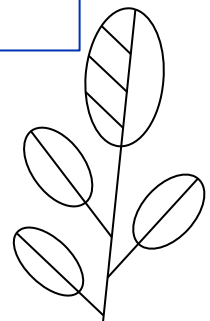
ДА!



Смотрим в код Spring'а: messageConverters

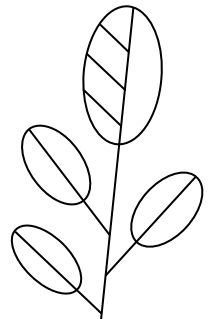
```
kotlinSerializationJsonPresent =
  ClassUtils.isPresent("kotlinx.serialization.json.Json", classLoader);
```

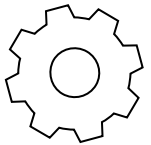
```
if (kotlinSerializationJsonPresent) {
   this.messageConverters.add(new KotlinSerializationJsonHttpMessageConverter());
}

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



Смотрим в код Spring'а: 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());
}
```



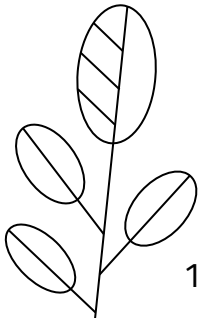



Как этот список **messageConverters** используется?


Смотрим в код Spring'а: AbstractMessageConverterMethodArgumentResolver

(!) Очень упрощенный псевдокод

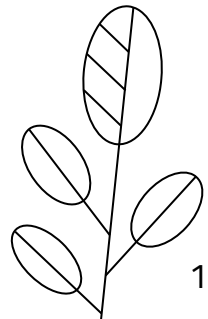
```
<T> Object readWithMessageConverters(...) {









    return body;
}
```



Смотрим в код Spring'а: AbstractMessageConverterMethodArgumentResolver

(!) Очень упрощенный псевдокод

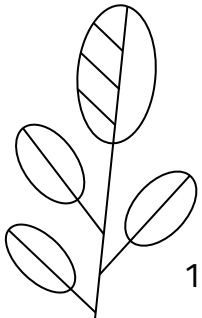
```
<T> Object readWithMessageConverters(...) {

    MediaType contentType = inputMessage.getHeaders().getContentType();









    return body;
}
```



Смотрим в код Spring'а: AbstractMessageConverterMethodArgumentResolver

(!) Очень упрощенный псевдокод

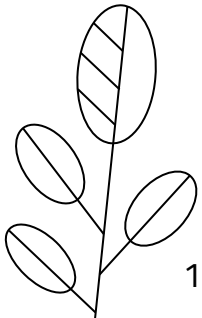
```
<T> Object readWithMessageConverters(...) {

    MediaType contentType = inputMessage.getHeaders().getContentType();

    for (HttpMessageConverter<?> converter : this.messageConverters) {
        if (converter.canRead(targetClass, contentType)) {




    return body;
}
```



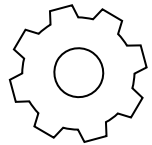
Смотрим в код Spring'а: AbstractMessageConverterMethodArgumentResolver

(!) 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;
}
```

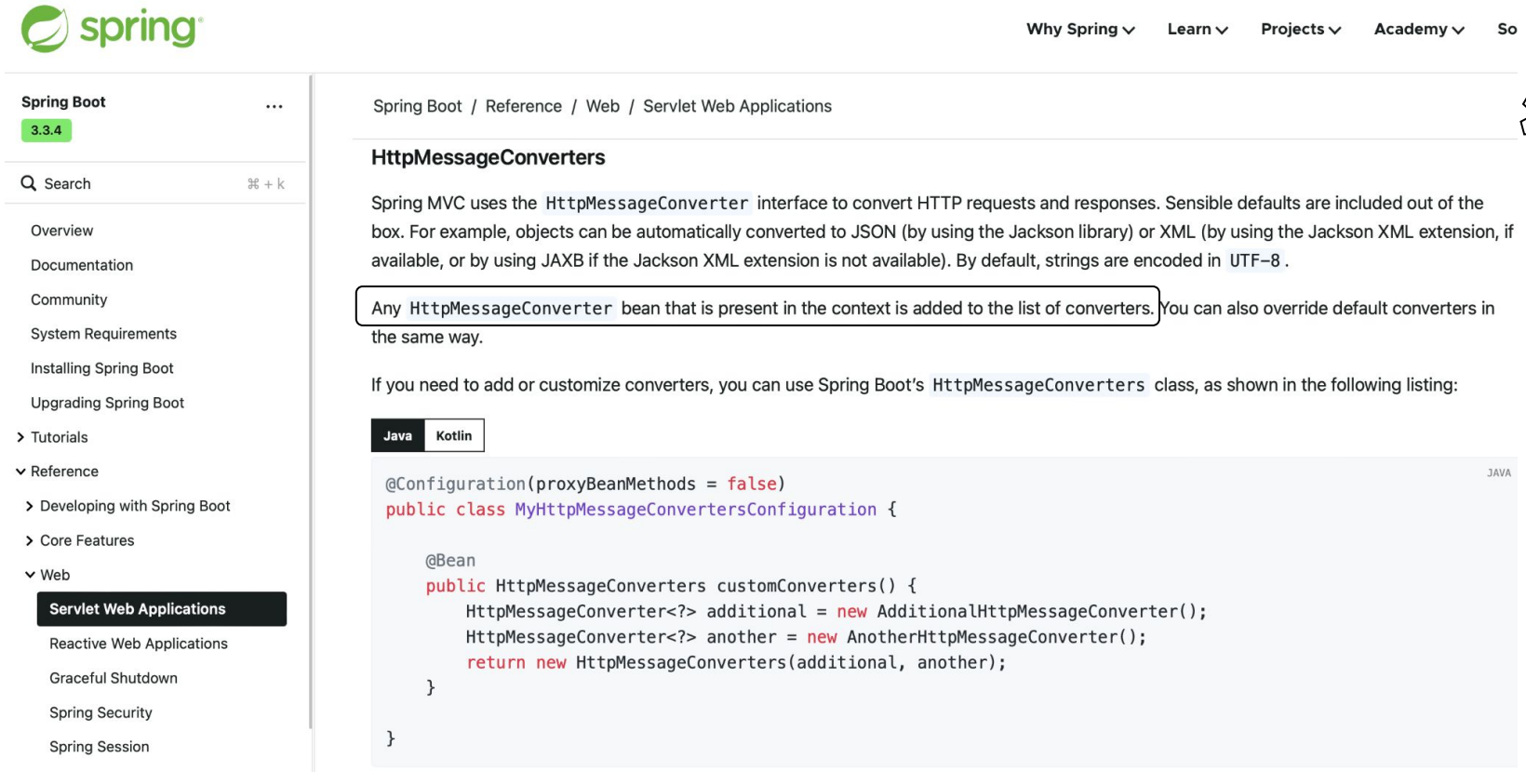


Оффтоп: Как можно нам, **работягам**, добавить свой converter?

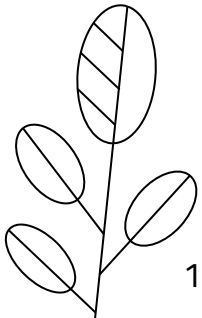


Оффтоп: Как можно нам, работягам, добавить свой converter?

HttpMessageConverter



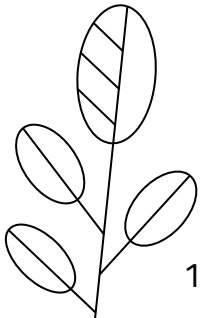
Смотрим в код Spring’а: **а как добавили kotlinx в Spring?**



Смотрим в код Spring’а: а как добавили kotlinx в Spring?

```
abstract class AbstractKotlinSerializationHttpMessageConverter
```

1. Имплементирован метод: `boolean canRead()`



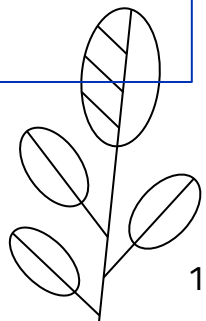
Смотрим в код Spring'а: а как добавили kotlinx в Spring?

`abstract class AbstractKotlinSerializationHttpMessageConverter`

1. Имплементирован метод: `boolean canRead()`

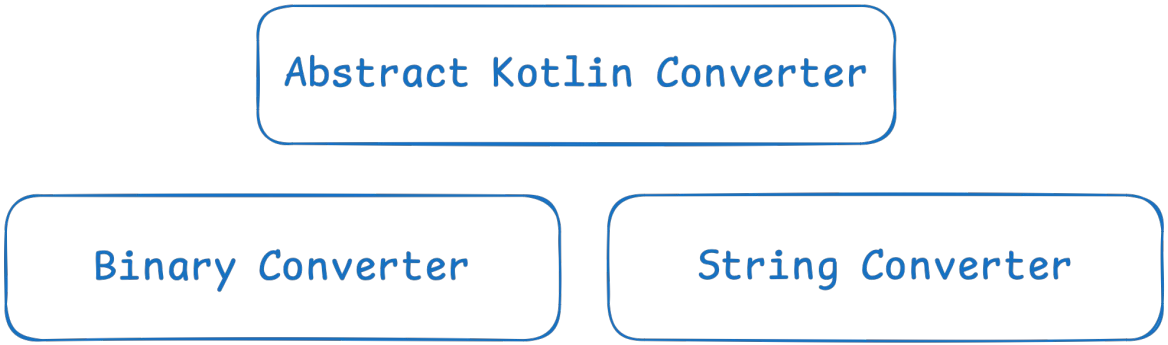
```
KSerializer<Object> serializer = this.kTypeSerializerCache.get(type);
...
if (serializer == null) {
    serializer = SerializersKt.serializerOrNull(this.format.getSerializersModule(),
type);
}
...
this.kTypeSerializerCache.put(type, serializer);
```

Наш знакомый из прошлой главы, который находит сгенерированный сериализатор



Смотрим в код Spring'а: элегантность Kotlinx

2. Имплементирован метод: **read()**



Смотрим в код Spring'а: элегантность Kotlinx

2. Имплементирован метод: **read()**


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

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

Смотрим в код Spring'а: 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'а: 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'а: 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);
}
```

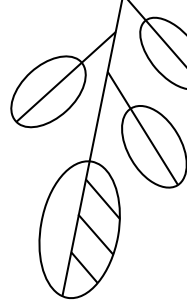
Смотрим в код Spring'а: элегантность Kotlinx

```
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"));
}
```

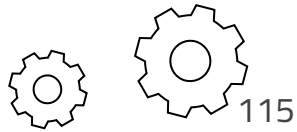
```
public KotlinSerializationCborHttpMessageConverter(Cbor cbor) {
   super(Cbor.Default, MediaType.APPLICATION_CBOR);
}
```

Откуда универсальность?

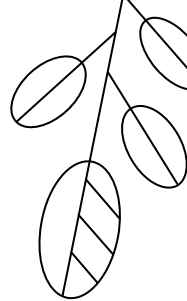


- Единые методы интерфейсов Format’ов:

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

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



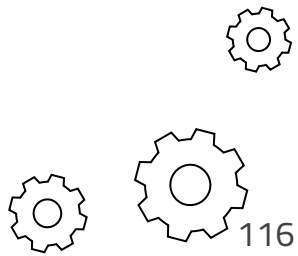
Откуда универсальность?



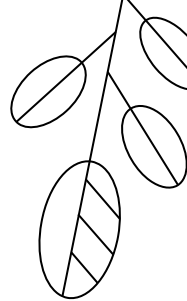
- Единые методы интерфейсов Format’ов:

```
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?

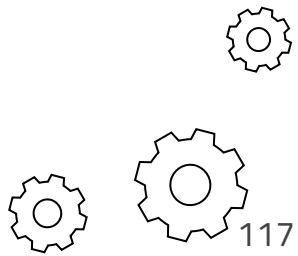


- Единые методы интерфейсов Format’ов:

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

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

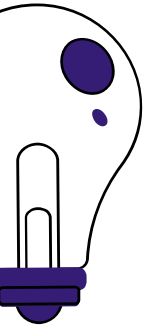
- Единым образом генерируется код компилятором для KSerializer

- Единые методы и организация логики Decoder/Encoder

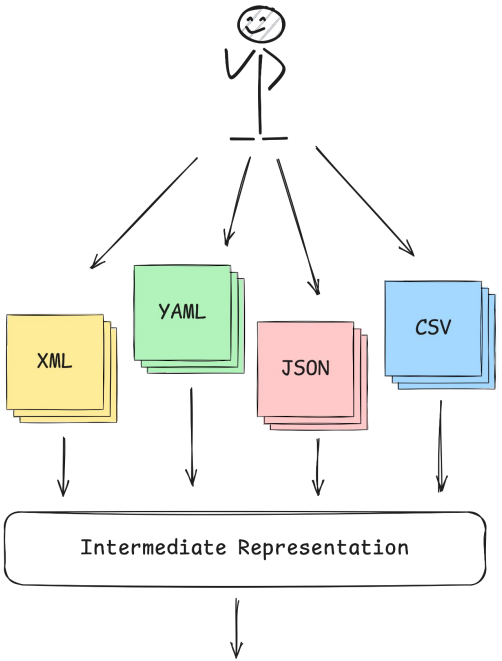


Как еще можно этим удобством можно воспользоваться?

04 Реальный сценарий

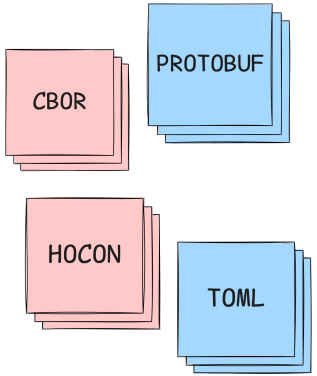


Наша проблема



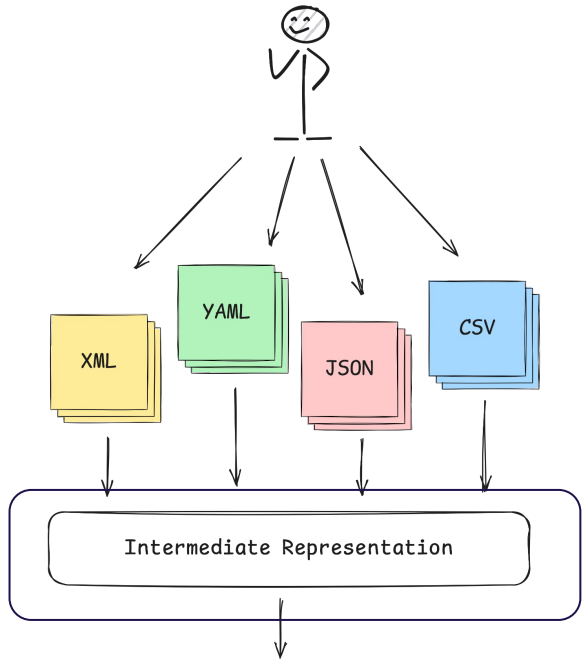
вероятно бинарные
и нестандартные
форматы

\+



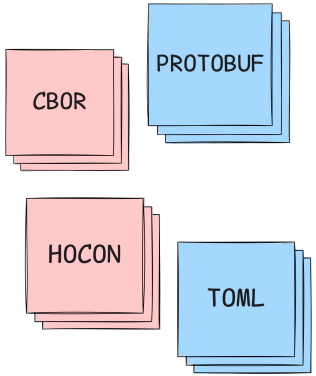
Наша проблема

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

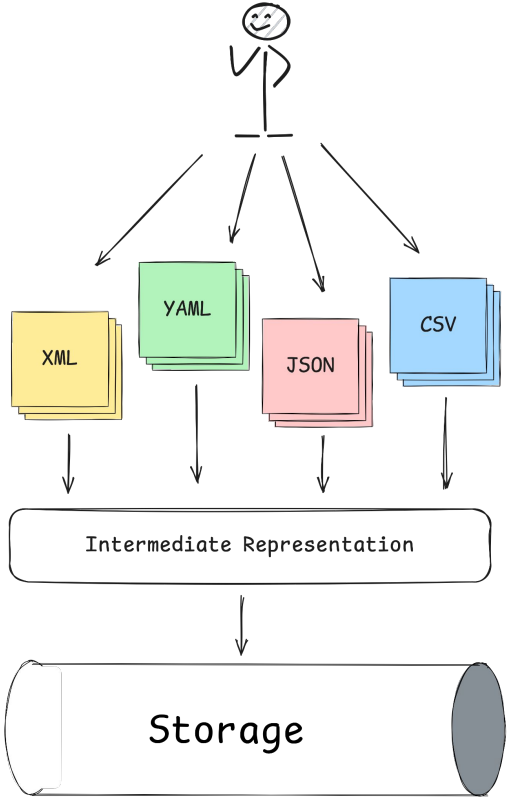


вероятно бинарные и нестандартные форматы

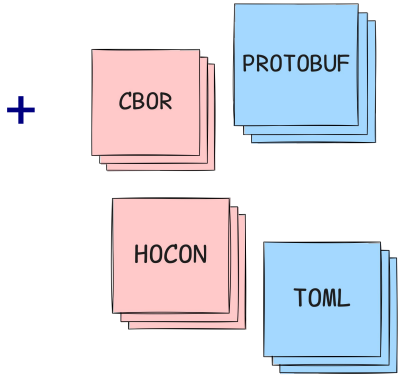
\+



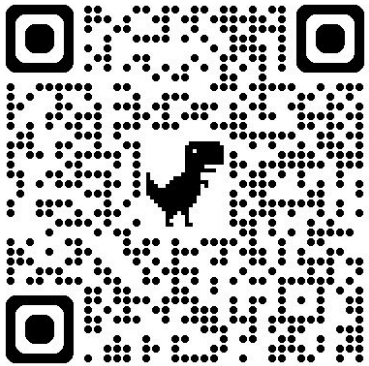
Наша проблема



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

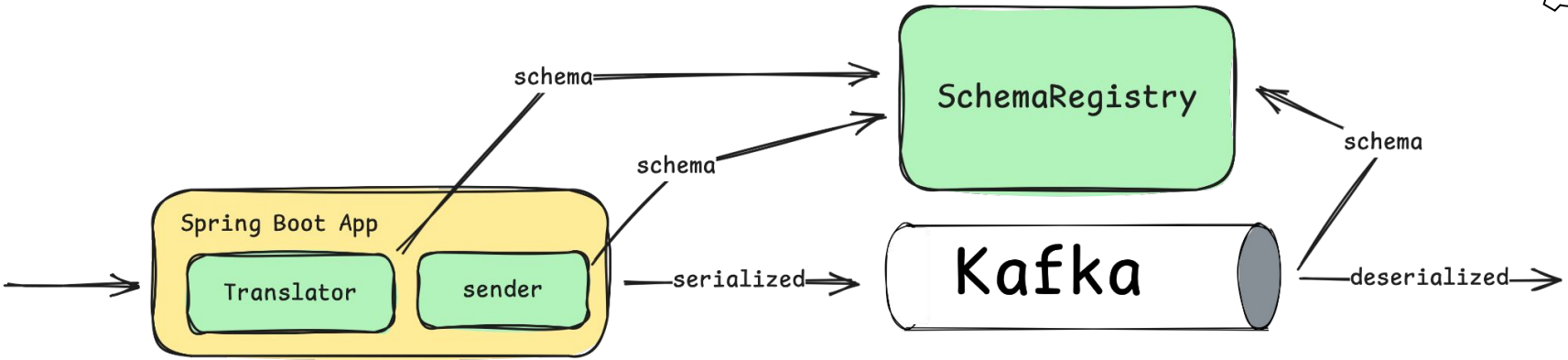
```
{
  "type" : "record",
  "name" : "Message",
  "namespace" : "schema.namespace",
  "fields" : [
     {
          "name" : "title",
          "type" : [ "null", "string" ],
          "default": null
     },
     {
          "name" : "message",
          "type" : "string"
     }
  ]
}
```

имена

типы

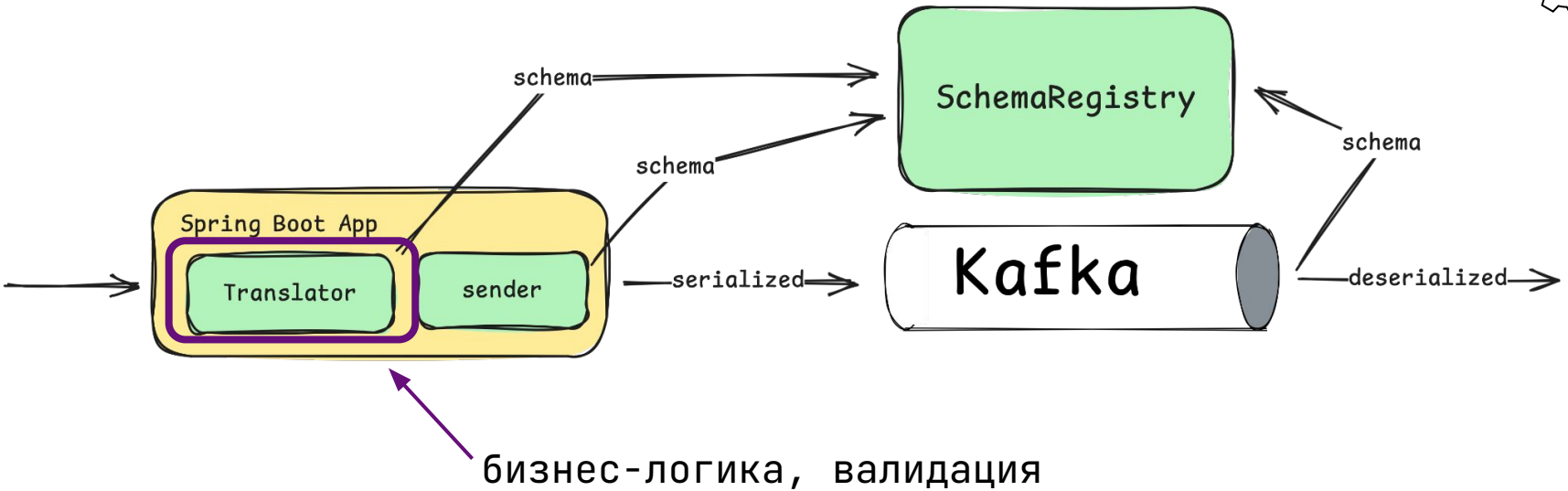
Наш IR: AVRO

почему?



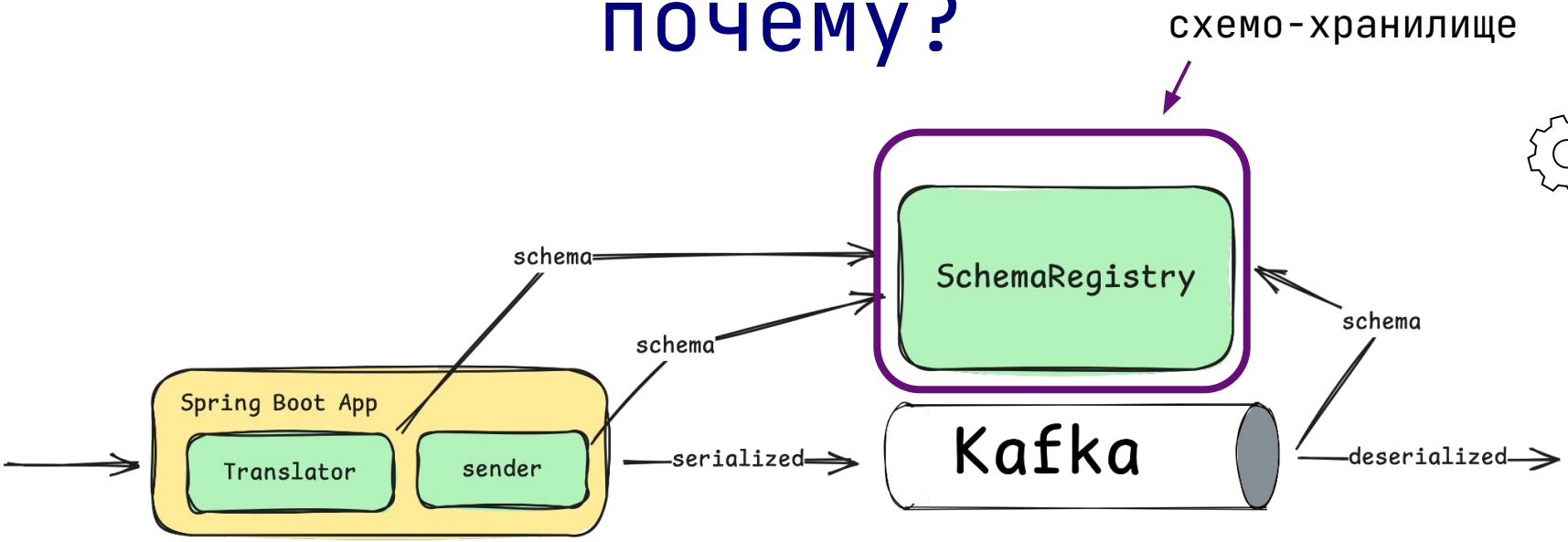
Наш IR: AVRO

почему?



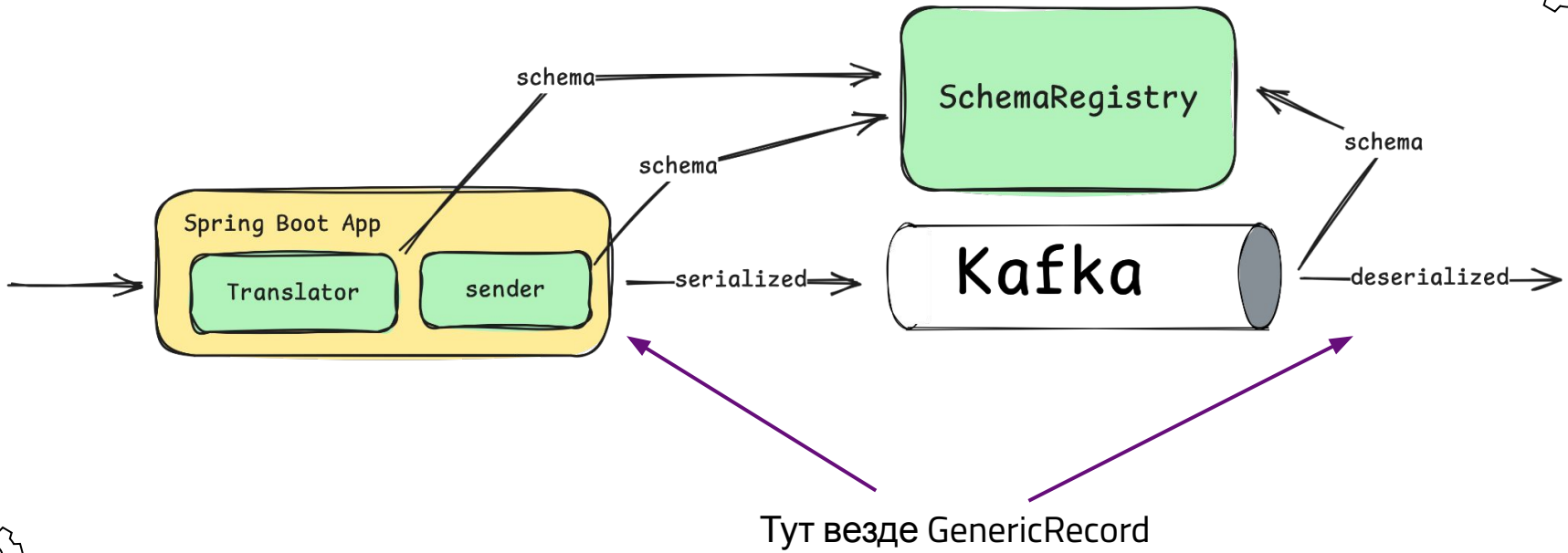
Наш IR: AVRO

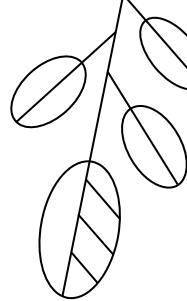
почему?



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




}
```

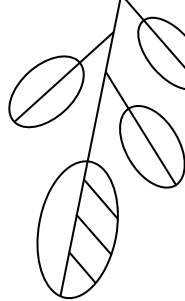
Реальный сценарий

```
@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


}
```

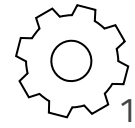
Реальный сценарий

```
@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

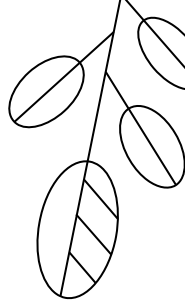


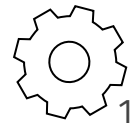
Передаем схему в KSerializer, замещаем сгенерированный код



“Обманываем” kotlinx

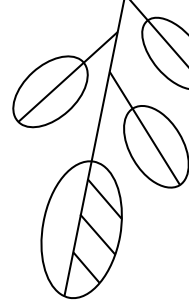
```
class AvroSchemaDeserializer(
    private val avroSchema: Schema,
) : KSerializer<GenericRecord> {
```

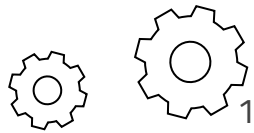




“Обманываем” kotlinx

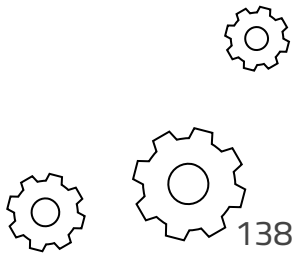
```
class AvroSchemaDeserializer(
   private val avroSchema: Schema,
) : KSerializer<GenericRecord> {
...

// (!) 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

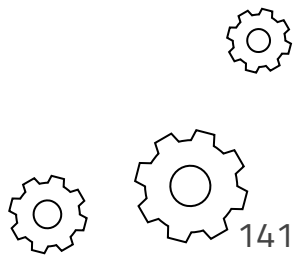
```
class AvroSchemaDeserializer(
   private val avroSchema: Schema,
) : KSerializer<GenericRecord> {
...

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()
  ...
  }
```

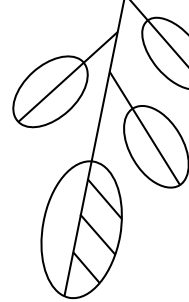
“Обманываем” kotlinx

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

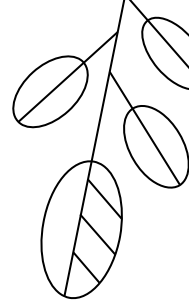
“Обманываем” kotlinx

```
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
```





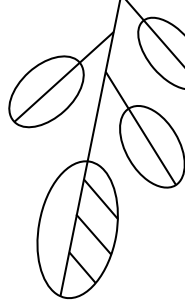
```
serialFormat.decodeFromXXX(
  AvroSchemaDeserializer(schema),
  Input
)
```



А форматы мапим примерно как в Spring

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

Какие вопросы остаются для отдельного обсуждения?



Производительность

Циклические графы

Работа с ошибками

Конфигурация форматов

Безопасность

Некоторые форматы с нестандартными подходами

Встраивание напрямую в контроллер

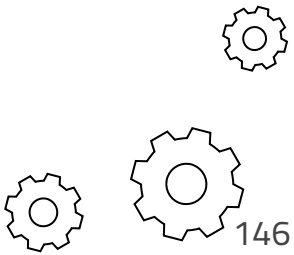
Валидации

Логические типы

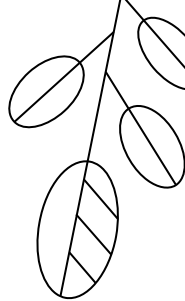
Что мы сегодня поняли?

Мы сегодня многое поняли

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



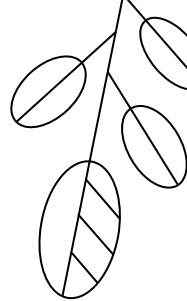
Мы сегодня многое поняли



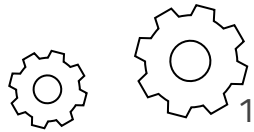
- Что Koltinx.serialization - продвинутый инструмент сериализации, особенно, если у вас мультиформатная работа

- Какой код генерируется компилятором и почему это важно знать

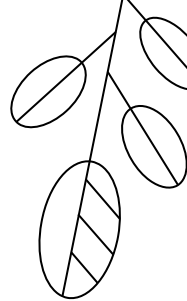
Мы сегодня многое поняли



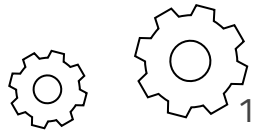
- Что Koltinx.serialization - продвинутый инструмент сериализации, особенно, если у вас мультиформатная работа
- Какой код генерируется компилятором и почему это важно знать
- Что происходит в Spring с сериализацией и как туда добавлен kotlinx



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
- Что происходит в Spring с сериализацией и как туда добавлен kotlinx
- Как можно сэкономить время за счет переиспользования эко-системы



И спасибо, что продержались до конца!

Давайте дружить на Github: