

Spring Boot “fat” JAR

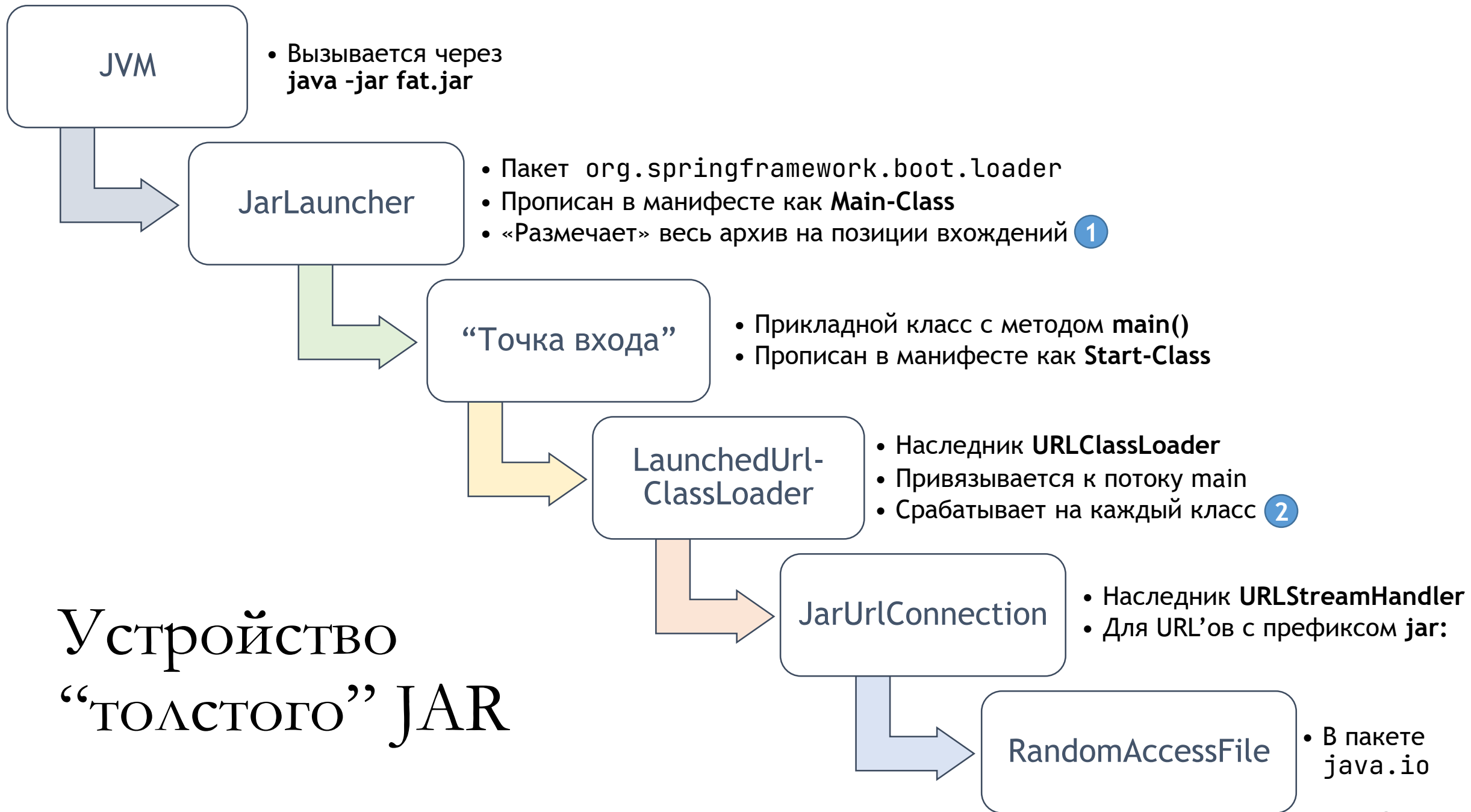
ТОНКИЕ части ТОЛСТОГО артефакта

Владимир Плизга
ЦФТ

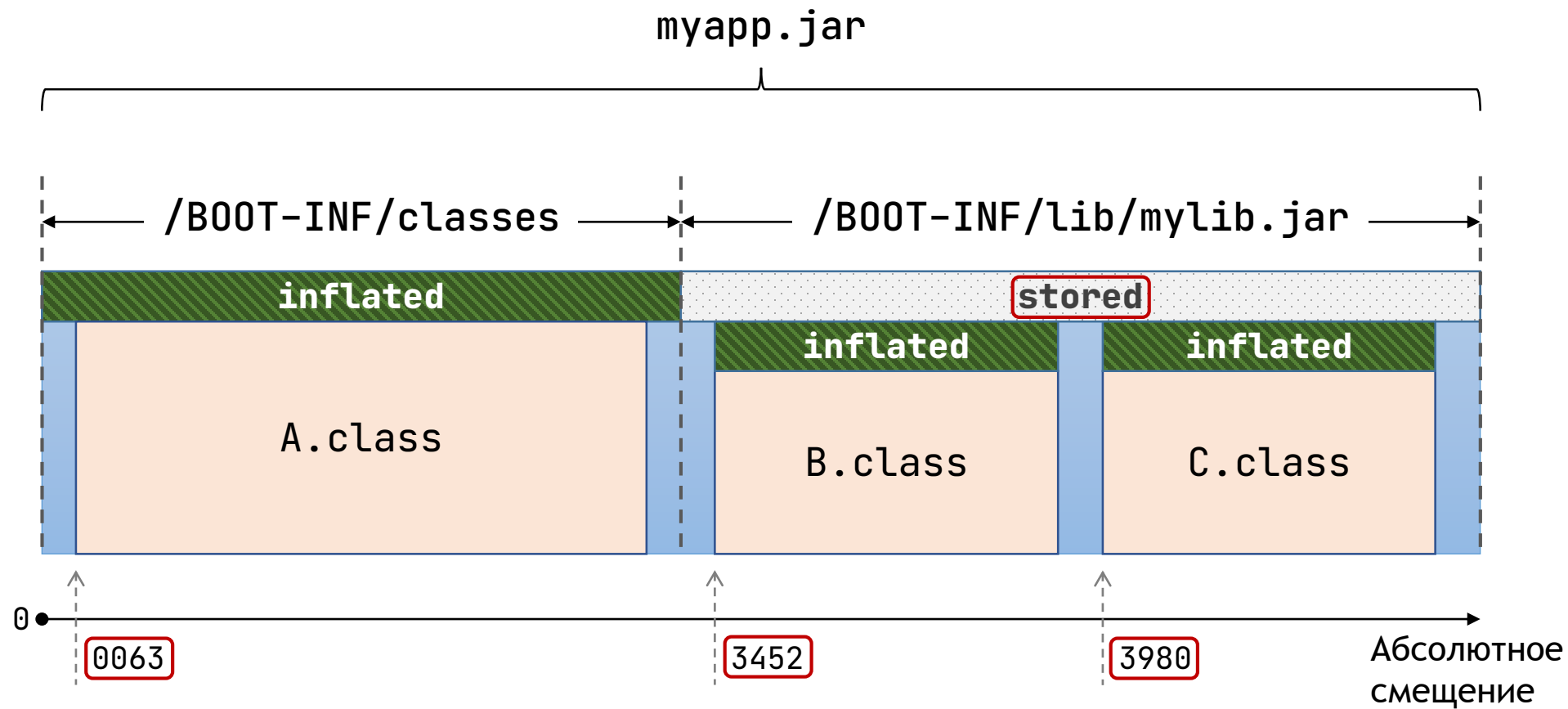
Происхождение “fat” JAR

- 1890 год
- Россия, Москва
- Художник Сергей Малютин
- В Spring Boot с версии 1.0 (2013)





1 «Разметка» внешнего архива



На основе [Appendix E: The Executable Jar Format](#)

2 Загрузка классов из архива

Пример пути в class-path:

jar:

URL-схема для Handler'a

/BOOT-INF/lib/slf4j-api-1.7.30.jar!

Путь к вложенному архиву

`jar:file:/C:/lang/samples/fatjar/build/libs/fat.jar!/BOOT-INF/lib/slf4j-api-1.7.30.jar!/org/slf4j/LoggerFactory.class`

file:/C:/lang/samples/fatjar/build/libs/fat.jar!

Полный путь (URL) к внешнему архиву

/org/slf4j/LoggerFactory.class

Путь к конечному классу

Устройство Spring Boot “fat” JAR (выжимка)

- Вложенные архивы **не** сжаты
- К потоку `main` привязан свой наследник `URLClassLoader`'а
- За обработку его URL'ов отвечает свой `Handler`
(видно в JVM-свойстве `java.protocol.handler.pkgs`)
- Загрузка классов сводится к чтению внешнего архива с нужной позиции через `RandomAccessFile`

Что мешает узнать больше?

```
Manifest-Version: 1.0
Implementation-Title: Spring Boot 'fat' JAR Sample
Implementation-Version: 0.0.1-SNAPSHOT
Implementation-Vendor: Toparvion
Main-Class: org.springframework.boot.loader.JarLauncher
Start-Class: pro.toparvion.sample.fatjar.FatjarApplication
Spring-Boot-Version: 2.4.0
Spring-Boot-Classes: BOOT-INF/classes/
Spring-Boot-Lib: BOOT-INF/lib/
Spring-Boot-Classpath-Index: BOOT-INF/classpath.idx
Spring-Boot-Layers-Index: BOOT-INF/layers.idx
```

spring-boot-loader
не доступен
в исходниках
библиотек

/META-INF/MANIFEST.MF

Как отлаживать загрузку “fat” JAR

1. Выкачать Spring Boot нужной версии (☕☕☕)

2. Поставить break point на

`org.springframework.boot.loader.JarLauncher#main`

3. Запустить “fat” JAR с отладчиком:

`-agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=*:5005`

4. Подключиться отладчиком из проекта Spring Boot

И это реально работает 😊



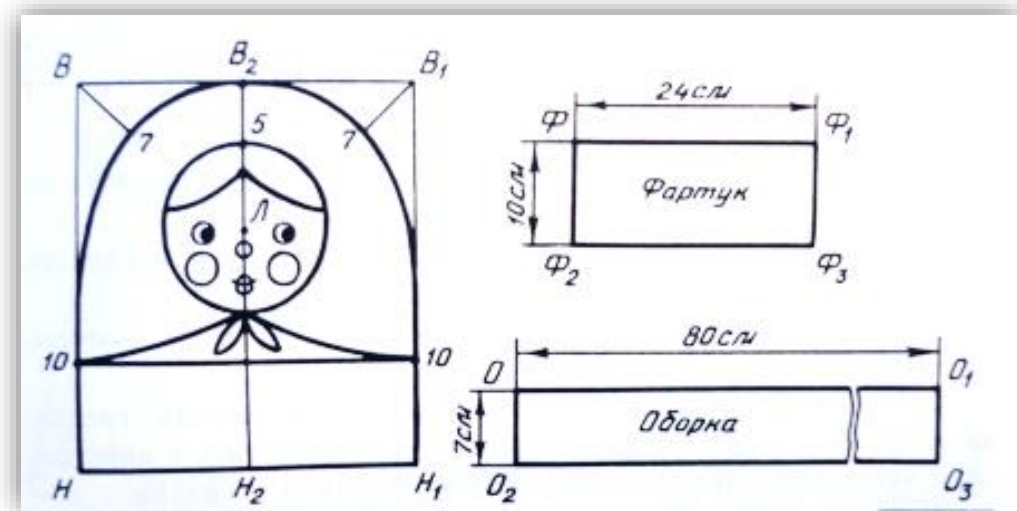
Попробуй переубедить! 😊

Запуск в IDE: резюме

- Порядки class-path'ов в IDE и fat JAR **могут отличаться**
- Это может приводить к **багам** типа “It works on my PC”
<https://github.com/spring-projects/spring-boot/issues/9128>
- Нужно проверять работу приложения в “fat” JAR
ещё на этапе разработки

Class Loading

И его спецэффекты



Основные проблемы с ClassLoader'ами

- Некоторые утилиты JDK не видят классы приложения
 - Например, **jshell** и **jdeps**
- Java-агенты не могут распознать class-path
 - Например, **jmint**
- Не работает Java Util Logging (JUL) и его производные
 - Например, **Oracle JDBC Diagnostic Driver**

Куда смотреть в последнюю очередь

Trying to load nested jar classes
with **ClassLoader.getSystemClassLoader()** **fails**.
java.util.Logging always uses the system classloader

<https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#executable-jar-restrictions>

И что делать?

- **Избегать** вызовов `ClassLoader.getSystemClassLoader()`
 - Например, через `jul-to-slf4j`
- **Оборачивать** `jshell` в [jshellw](#)
 - <https://youtu.be/fmLW7VkSuN8?t=3150>
- Распаковывать **весь** fat JAR
 - *(см. далее)*
- Распаковывать **отдельные** библиотеки*
 - [Extract Specific Libraries When an Executable Jar Runs](#)

*Как распаковать ОТДЕЛЬНЫЕ архивы

build.gradle

```
bootJar {  
    //...  
    requiresUnpack '**/jruby-complete-*.jar'  
}
```

pom.xml

```
<plugin>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-maven-plugin</artifactId>  
  <configuration>  
    <requiresUnpack>  
      <dependency>  
        <groupId>org.jruby</groupId>  
        <artifactId>jruby-complete</artifactId>  
      </dependency>  
    </requiresUnpack>  
  </configuration>  
</plugin>
```

```
$zipinfo -v build/libs/fat.jar BOOT-INF/lib/jruby-complete-9.2.13.0.jar
Archive:  build/libs/fat.jar
There is no zipfile comment.
...
Central directory entry #87:
-----

BOOT-INF/lib/jruby-complete-9.2.13.0.jar

offset of local header from start of archive: 118269
                                               (0000000000001CDFDh) bytes
...
compression method:                       none (stored)
file security status:                      not encrypted
...
Unix file attributes (100644 octal):       -rw-r--r--
MS-DOS file attributes (00 hex):          none

----- file comment begins -----
UNPACK:1e6de00e7bea5ff3c9d6086fd9e2610258c051ce
----- file comment ends -----
```


Попутное резюме

- Самобытный class-loading в «толстых» JAR создаёт **проблемы для некоторых** инструментов
- Большинство из типов проблем упомянуты **в документации**
 - и обходятся распаковкой архива 😊

Скорость запуска

И её нехватка



Fat JAR замедляет старт приложения (?)

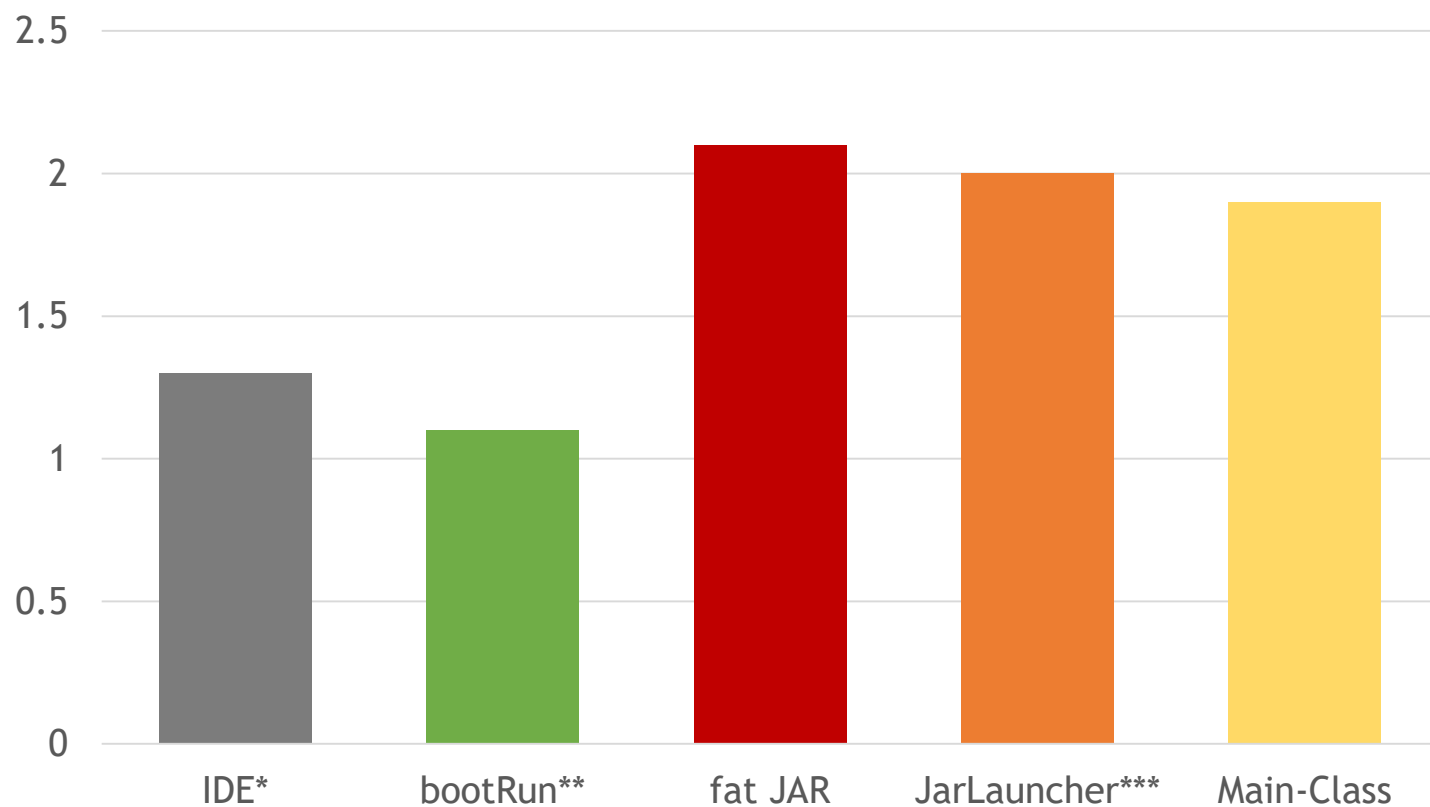
- Зависит от «толщины» архива
- Но в целом просад есть:

Benchmark	Mode	Cnt	Score	Error	Units
PetclinicLatestBenchmark.explodedJarMain	avgt	10	3.897	± 0.067	s/op
PetclinicLatestBenchmark.fatJar	avgt	10	<u>4.996</u>	± 0.032	s/op
PetclinicLatestBenchmark.noverify	avgt	10	4.399	± 0.029	s/op
PetclinicLatestBenchmark.explodedJarFlags	avgt	10	3.325	± 0.053	s/op

<https://github.com/dsyer/spring-boot-startup-bench#spring-boot-2x>

Эксперимент «на минималках»

Среднее время запуска, сек



* При включенных оптимизациях, но без JMX (а с ним ≈ 1.6 s)

** При активном Gradle Daemon

*** При распаковке больших JAR
разница должна быть больше

Как можно ускорить запуск

- Примеры рекомендаций разработчиков
 - **Опции JVM** (`-XX:TieredStopAtLevel=1 -noverify`)
 - **Фиксация** местонахождения конфигурации
 - **Обновление** Spring & Spring Boot
- Распаковать 😊
- Применить **AppCDS**:
 - Основы и примеры: <https://youtu.be/fmLW7VkSuN8?t=2492>
 - Dynamic CDS (JDK 13+): <https://habr.com/ru/post/472638/>

Попутное резюме

- Fat JAR вносит небольшой **overhead на старте** приложения
И ещё **меньше в runtime**
- Просад можно **скомпенсировать** другими мерами:
 - “*How do I make my app go faster?*”
<https://github.com/dsyer/spring-boot-allocations>
- Чем **тоньше** JAR, тем **лучше**

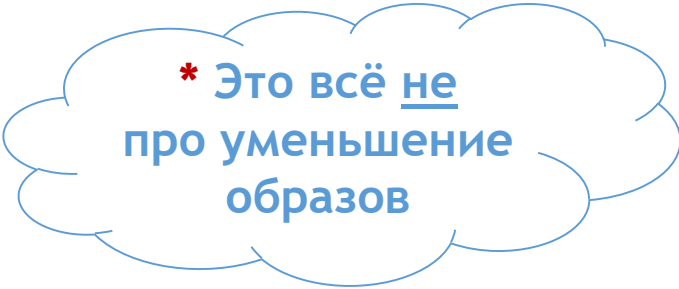
Контейнеризация

Вариант 1: анатомический



Суть оптимизации образов

- Docker строит образы из **упорядоченных слоёв**
- Каждый слой - это **diff** данных **с предыдущим** слоем
- Слой описывается **хэшем** от своих данных
- Если при сборке **хэш** нового слоя **совпал** со старым => **поор**
- При несовпадении хэша **предыдущие** слои **сохраняются**



* Это всё не
про уменьшение
образов

Начиная со Spring Boot 2.3

- Для “fat” JAR появился режим `-Djarmode=layertools`
- Позволяет пилить **толстый** архив на **тонкие** слои
- Тесно дружит с Maven/Gradle плагинами
Читает созданный ими файл `/BOOT-INF/layers.idx`

Layer tools: резюме

Плюсы:

- Максимальный контроль над сборкой
- Малый размер образа

Минусы:

- Усложнение Dockerfile
- Много ручных действий

Контейнеризация

Вариант 2: радужно-перспективный

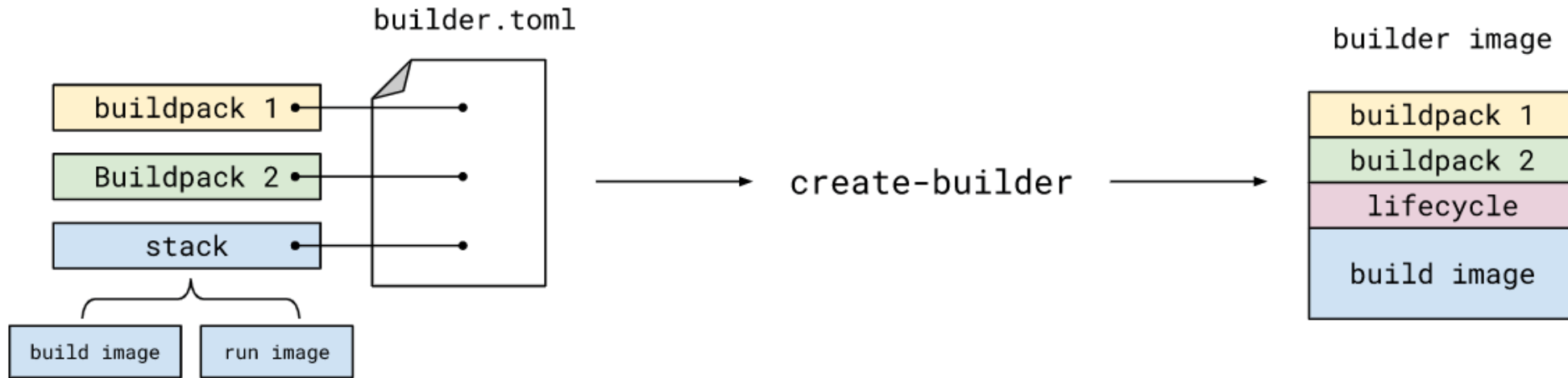


Минимальная терминология (1 / 2)

- **Buildpack** - набор действий для сборки и запуска приложения в контейнере
 - Проверяет сам себя на применимость (detection)
 - Не содержит в себе образов
 - Идея пришла из Heroku & CloudFoundry, теперь есть и в CNF

Минимальная терминология (2/2)

- **Builder** - образ, включающий buildpacks и другие образы для сборки и запуска приложения



- **Platform** - то, на чем запускается builder

А причем тут Spring Boot?

- Начиная с v2.3 можно собирать образы через buildpacks
 - Dockerfile больше **не нужен** 😊
 - Docker Daemon всё **ещё нужен** 😞
- Spring Boot Maven/Gradle плагины выступают **платформой**
- Они используют builder'ы и buildpack'и от [Paketo.io](https://paketo.io)
 - В том числе **Java Buildpack**
 - И можно настроить под себя

Buildpacks: резюме

Плюсы:

- Не нужен Dockerfile
- Многое достаётся из коробки

Минусы:

- Массивный образ (250 МБ)
- Зависимость от Docker Daemon

Контейнеризация

Вариант 3: альтернативный



Google Jib

- Может работать как Maven/Gradle **плагин**
- Умеет собирать образы **без Docker Daemon**
- Поддерживает разбиение **на слои**

Jib: резюме

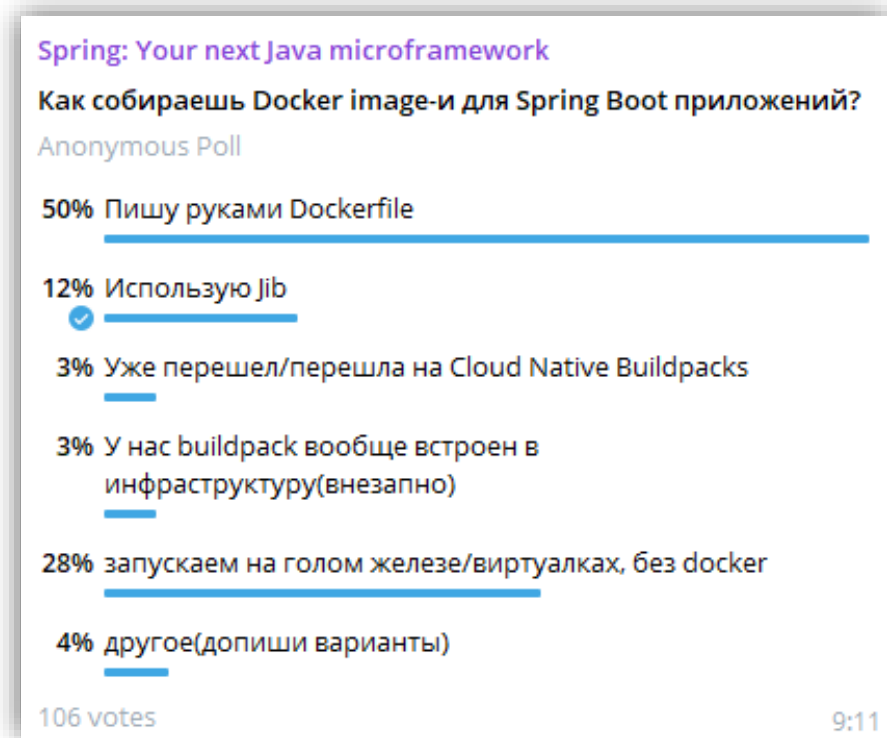
Плюсы:

- Не нужен Docker Daemon/Dockerfile
- Годится для любого приложения на Java

Минусы:

- Не учитывает специфику Spring Boot
- Сложновато управлять слоями


А как поставляете в production вы?



Joker<?>

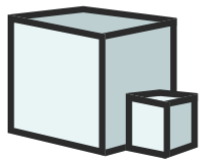
Алексей Нестеров
VMware

Spring: Your next Java micro-framework

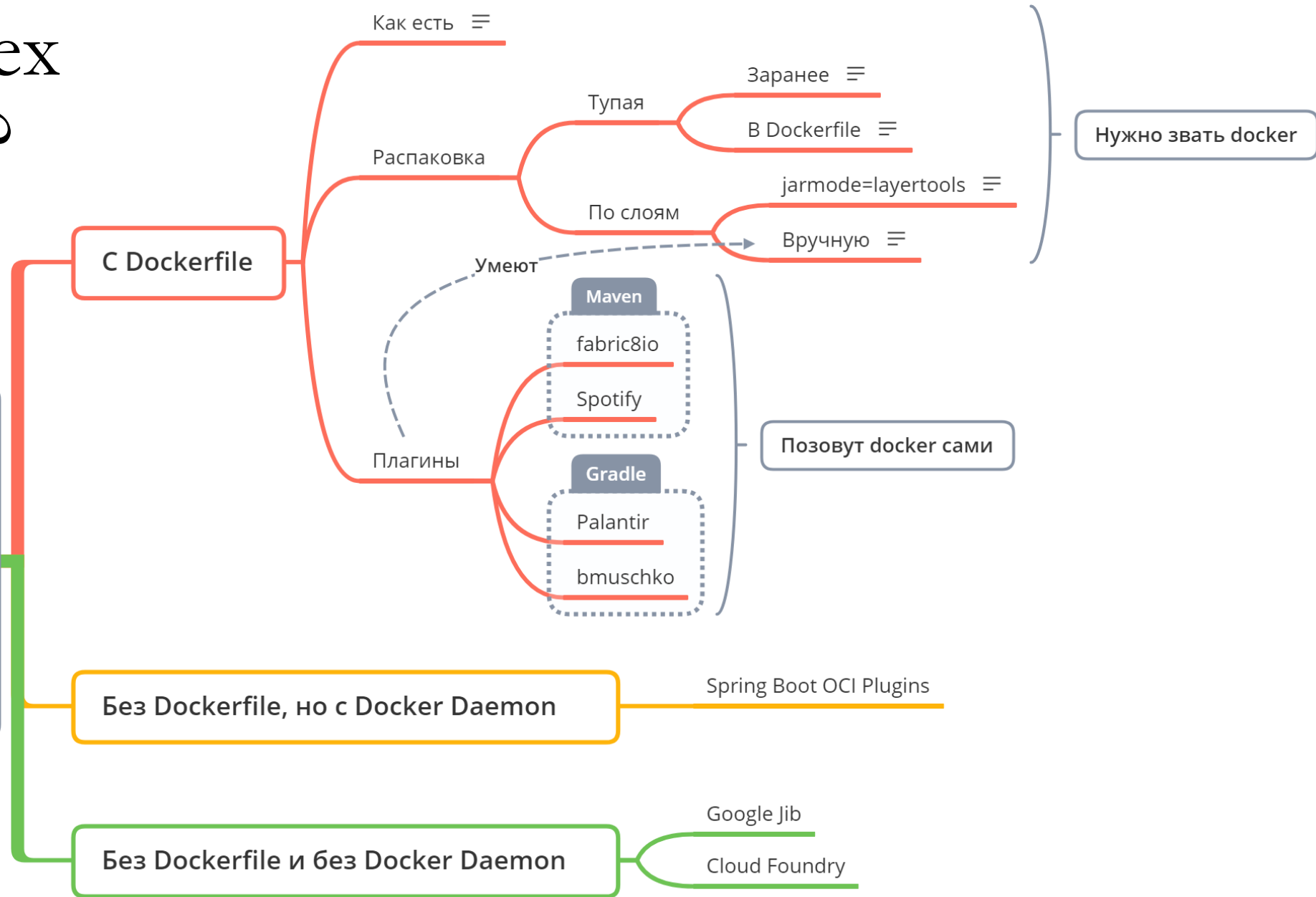


<https://jokerconf.com/2020/talks/5ruwqfah36hgcztljugsva/>

А можно всех посмотреть?



Контейнеризация Spring Boot



Сводка рассмотренных вариантов

	layertools	Buildpacks	Jib
Можно без Dockerfile	-	✓	✓
Можно без Docker Daemon	-	-	✓
Раскладка по слоям	✓	✓	✓
Фиксация class-path	✓	✓	-
“Автонастройка” опций JVM	-	✓	-
Reproducible builds	-	✓	✓
Размер образа по умолчанию*	≈ 140 MB	≈ 250 MB	≈ 140 MB

И как выбирать?

- Если нужен максимальный **контроль и лёгкость*** образа,
то **Layertools**
- Если нужно, чтобы всё работало **само**,
то **Buildpacks**
- Если надо обойтись **без Docker** или **нет Spring Boot 2.3**,
то **Jib**

* <http://jokerconf.com/2020/talks/7iu9r9lc8iorvvd0dqf6xu>

Распаковка

И как с ней правильно жить



Распакованный fat JAR можно запускать:

- Через прикладной класс (Main-Class):

```
java -cp BOOT-INF/classes:BOOT-INF/lib/* \  
pro.toparvion.sample.fatjar.FatjarApplication
```

- Через класс JarLauncher:

```
java org.springframework.boot.loader.JarLauncher
```


Какая разница?

Отличие	JarLauncher	Main-Class
Порядок в class-path	✓ Фиксирован в classpath.idx	— Как придётся
Скорость старта	— Ниже	✓ Выше
Имя стартового класса	✓ Фиксировано	— Зависит от приложения
Мета-данные из манифеста*	✓ Доступны	— Нет

Зачем могут быть нужны мета-данные?





Зачем ещё могут быть нужны мета-данные?

swagger

Select a spec default

RESToRun 1.26.0

[Base URL: <http://192.168.1.100:8080/>]

Точка входа

Schemes

HTTP

Распаковка: резюме

После распаковки запускайте приложение
через **JarLauncher**.

Итоги

И выводы



Что мы узнали?

- Общие **принципы** работы “fat” JAR
- Где и как узнать об этом **больше**
- Характер и примеры **проблем** при запуске из “fat” JAR
- **3 способа** развертывания в контейнерах **по слоям**
- Другие **варианты** исполняемого архива в Spring Boot

Что теперь делать?

- Проверьте class-path еще в IDE
- Обновитесь до Spring Boot 2.3+
- Распаковывайте JAR в целевом окружении
- Запускайте через JarLauncher (не через Main-Class)
- Используйте по возможности Cloud Native Buildpacks

Что почитать/посмотреть дальше?

- [Creating Efficient Docker Images with Spring Boot 2.3](#)
Блог пост от авторов Spring Boot про layertools & buildpacks
- [What's New in Spring Boot 2.3](#)
Screencast новых возможностей v2.3, в том числе этих же
- [Creating Optimized Docker Images for a Spring Boot Application](#)
Сравнение подходов к контейнеризации: Spring Boot vs Jib
- Просто примеры чужого опыта:
 - [Building Containers With Spring Boot 2.3](#)
 - [Поддержка Buildpacks в Spring Boot 2.3.0](#) (Хабр)



Q & A

Владимир Плизга
ЦФТ

 [@toparvion](https://twitter.com/toparvion)

 <https://github.com/Toparvion/fat-jar-sample>

 <https://toparvion.pro/event/2021/jugnsk-january>

 <https://snowone.ru>

